DPDK

DATA PLANE DEVELOPMENT KIT

DPDK documentation

Release 2.2.0

April 28, 2016

Contents

CHAPTER 1

Getting Started Guide for Linux

1.1 Introduction

This document contains instructions for installing and configuring the Intel® Data Plane Development Kit (DPDK) software. It is designed to get customers up and running quickly. The document describes how to compile and run a DPDK application in a Linux application (linuxapp) environment, without going deeply into detail.

1.1.1 Documentation Roadmap

The following is a list of DPDK documents in the suggested reading order:

- Release Notes: Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- Getting Started Guide (this document): Describes how to install and configure the DPDK; designed to get users up and running quickly with the software.
- Programmer's Guide: Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux application (linuxapp) environment
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

  A glossary of terms is also provided.
- API Reference: Provides detailed information about DPDK functions, data structures and other programming constructs.
- Sample Applications User Guide: Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

1.2 System Requirements

This chapter describes the packages required to compile the DPDK.

Note: If the DPDK is being used on an Intel® Communications Chipset 89xx Series platform, please consult the *Intel® Communications Chipset 89xx Series Software for Linux Getting Started Guide*.

1.2.1 BIOS Setting Prerequisite on x86

For the majority of platforms, no special BIOS settings are needed to use basic DPDK functionality. However, for additional HPET timer and power management functionality, and high performance of small packets on 40G NIC, BIOS setting changes may be needed. Consult the section on *Enabling Additional Functionality* for more information on the required changes.

1.2.2 Compilation of the DPDK

Required Tools:

Note: Testing has been performed using Fedora 18. The setup commands and installed packages needed on other systems may be different. For details on other Linux distributions and the versions tested, please consult the DPDK Release Notes.

- GNU `make`.
- coreutils: `cmp`, `sed`, `grep`, `arch`, etc.
- gcc: versions 4.5.x or later is recommended for `i686/x86_64`. Versions 4.8.x or later is recommended for `ppc_64` and `x86_x32` ABI. On some distributions, some specific compiler flags and linker flags are enabled by default and affect performance (`-fstack-protector`, for example). Please refer to the documentation of your distribution and to `gcc -dumpspecs`.
- libc headers (glibc-devel.i686 / libc6-dev-i386; glibc-devel.x86_64 for 64-bit compilation on Intel architecture; glibc-devel.ppc64 for 64 bit IBM Power architecture;)
- Linux kernel headers or sources required to build kernel modules. (kernel - devel.x86_64; kernel - devel.ppc64)
- Additional packages required for 32-bit compilation on 64-bit systems are:
  - glibc.i686, libgcc.i686, libstdc++.i686 and glibc-devel.i686 for Intel i686/x86_64;
  - glibc.ppc64, libgcc.ppc64, libstdc++.ppc64 and glibc-devel.ppc64 for IBM ppc_64;

Note: x86_x32 ABI is currently supported with distribution packages only on Ubuntu higher than 13.10 or recent Debian distribution. The only supported compiler is gcc 4.8+.

Note: Python, version 2.6 or 2.7, to use various helper scripts included in the DPDK package.

Optional Tools:

- Intel® C++ Compiler (icc). For installation, additional libraries may be required. See the icc Installation Guide found in the Documentation directory under the compiler installation.
- IBM® Advance ToolChain for Powerlinux. This is a set of open source development tools and runtime libraries which allows users to take leading edge advantage of IBM's latest POWER hardware features on Linux. To install it, see the IBM official installation document.
- libpcap headers and libraries (libpcap-devel) to compile and use the libpcap-based poll-mode driver. This driver is disabled by default and can be enabled by setting `CONFIG_RTE_LIBRTE_PMD_PCAP=y` in the build time config file.

1.2.3 Running DPDK Applications

To run an DPDK application, some customization may be required on the target machine.

System Software

Required:

- Kernel version >= 2.6.34

  The kernel version in use can be checked using the command:

  ```
  uname -r
  ```

- glibc >= 2.7 (for features related to cpuset)

  The version can be checked using the `ldd --version` command.

- Kernel configuration

  In the Fedora OS and other common distributions, such as Ubuntu, or Red Hat Enterprise Linux, the vendor supplied kernel configurations can be used to run most DPDK applications.

  For other kernel builds, options which should be enabled for DPDK include:

  - UIO support
  - HUGETLBFS
  - PROC_PAGE_MONITOR support
  - HPET and HPET_MMAP configuration options should also be enabled if HPET support is required. See the section on *High Precision Event Timer (HPET) Functionality* for more details.

Use of Hugepages in the Linux Environment

Hugepage support is required for the large memory pool allocation used for packet buffers (the HUGETLBFS option must be enabled in the running kernel as indicated the previous section). By using hugepage allocations, performance is increased since fewer pages are needed, and therefore less Translation Lookaside Buffers (TLBs, high speed translation caches), which reduce the time it takes to translate a virtual page address to a physical page address. Without hugepages, high TLB miss rates would occur with the standard 4k page size, slowing performance.

Reserving Hugepages for DPDK Use

The allocation of hugepages should be done at boot time or as soon as possible after system boot to prevent memory from being fragmented in physical memory. To reserve hugepages at boot time, a parameter is passed to the Linux kernel on the kernel command line.

For 2 MB pages, just pass the hugepages option to the kernel. For example, to reserve 1024 pages of 2 MB, use:

```
hugepages=1024
```

For other hugepage sizes, for example 1G pages, the size must be specified explicitly and can also be optionally set as the default hugepage size for the system. For example, to reserve 4G of hugepage memory in the form of four 1G pages, the following options should be passed to the kernel:

```
default_hugepagesz=1G hugepagesz=1G hugepages=4
```

Note: The hugepage sizes that a CPU supports can be determined from the CPU flags on Intel architecture. If pse exists, 2M hugepages are supported; if pdpe1gb exists, 1G hugepages are supported. On IBM Power architecture, the supported hugepage sizes are 16MB and 16GB.

Note: For 64-bit applications, it is recommended to use 1 GB hugepages if the platform supports them.

In the case of a dual-socket NUMA system, the number of hugepages reserved at boot time is generally divided equally between the two sockets (on the assumption that sufficient memory is present on both sockets).

See the Documentation/kernel-parameters.txt file in your Linux source tree for further details of these and other kernel options.

Alternative:

For 2 MB pages, there is also the option of allocating hugepages after the system has booted. This is done by echoing the number of hugepages required to a nr_hugepages file in the `/sys/devices/` directory. For a single-node system, the command to use is as follows (assuming that 1024 pages are required):

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

On a NUMA machine, pages should be allocated explicitly on separate nodes:

```
echo 1024 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
echo 1024 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
```

Note: For 1G pages, it is not possible to reserve the hugepage memory after the system has booted.

Using Hugepages with the DPDK

Once the hugepage memory is reserved, to make the memory available for DPDK use, perform the following steps:

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

The mount point can be made permanent across reboots, by adding the following line to the `/etc/fstab` file:

```
nodev /mnt/huge hugetlbfs defaults 0 0
```

For 1GB pages, the page size must be specified as a mount option:

```
nodev /mnt/huge_1GB hugetlbfs pagesize=1GB 0 0
```

Xen Domain0 Support in the Linux Environment

The existing memory management implementation is based on the Linux kernel hugepage mechanism. On the Xen hypervisor, hugepage support for DomainU (DomU) Guests means that DPDK applications work as normal for guests.

However, Domain0 (Dom0) does not support hugepages. To work around this limitation, a new kernel module rte_dom0_mm is added to facilitate the allocation and mapping of memory via **IOCTL** (allocation) and **MMAP** (mapping).

Enabling Xen Dom0 Mode in the DPDK

By default, Xen Dom0 mode is disabled in the DPDK build configuration files. To support Xen Dom0, the CONFIG_RTE_LIBRTE_XEN_DOM0 setting should be changed to “y”, which enables the Xen Dom0 mode at compile time.

Furthermore, the CONFIG_RTE_EAL_ALLOW_INV_SOCKET_ID setting should also be changed to “y” in the case of the wrong socket ID being received.

Loading the DPDK rte_dom0_mm Module

To run any DPDK application on Xen Dom0, the `rte_dom0_mm` module must be loaded into the running kernel with rsv_memsize option. The module is found in the kmod sub-directory of the DPDK target directory. This module should be loaded using the insmod command as shown below (assuming that the current directory is the DPDK target directory):

```
sudo insmod kmod/rte_dom0_mm.ko rsv_memsize=X
```

The value X cannot be greater than 4096(MB).

Configuring Memory for DPDK Use

After the rte_dom0_mm.ko kernel module has been loaded, the user must configure the memory size for DPDK usage. This is done by echoing the memory size to a memsize file in the /sys/devices/ directory. Use the following command (assuming that 2048 MB is required):

```
echo 2048 > /sys/kernel/mm/dom0-mm/memsize-mB/memsize
```

The user can also check how much memory has already been used:

```
cat /sys/kernel/mm/dom0-mm/memsize-mB/memsize_rsvd
```

Xen Domain0 does not support NUMA configuration, as a result the `--socket-mem` command line option is invalid for Xen Domain0.

Note: The memsize value cannot be greater than the rsv_memsize value.

Running the DPDK Application on Xen Domain0

To run the DPDK application on Xen Domain0, an extra command line option `--xen-dom0` is required.

1.3 Compiling the DPDK Target from Source

Note: Parts of this process can also be done using the setup script described in the *Quick Start Setup Script* section of this document.

1.3.1 Install the DPDK and Browse Sources

First, uncompress the archive and move to the uncompressed DPDK source directory:

```
unzip DPDK-<version>.zip
cd DPDK-<version>

ls
app/ config/ examples/ lib/ LICENSE.GPL LICENSE.LGPL Makefile
mk/ scripts/ tools/
```

The DPDK is composed of several directories:

- lib: Source code of DPDK libraries
- drivers: Source code of DPDK poll-mode drivers
- app: Source code of DPDK applications (automatic tests)
- examples: Source code of DPDK application examples
- config, tools, scripts, mk: Framework-related makefiles, scripts and configuration

1.3.2 Installation of DPDK Target Environments

The format of a DPDK target is:

```
ARCH-MACHINE-EXECENV-TOOLCHAIN
```

where:

- `ARCH` can be: `i686`, `x86_64`, `ppc_64`
- `MACHINE` can be: `native`, `ivshmem`, `power8`
- `EXECENV` can be: `linuxapp`, `bsdapp`
- `TOOLCHAIN` can be: `gcc`, `icc`

The targets to be installed depend on the 32-bit and/or 64-bit packages and compilers installed on the host. Available targets can be found in the DPDK/config directory. The defconfig_ prefix should not be used.

Note: Configuration files are provided with the `RTE_MACHINE` optimization level set. Within the configuration files, the `RTE_MACHINE` configuration value is set to native, which means that the compiled software is tuned for the platform on which it is built. For more information on this setting, and its possible values, see the *DPDK Programmers Guide*.

When using the Intel® C++ Compiler (icc), one of the following commands should be invoked for 64-bit or 32-bit use respectively. Notice that the shell scripts update the `$PATH` variable and therefore should not be performed in the same session. Also, verify the compiler's installation directory since the path may be different:

```
source /opt/intel/bin/iccvars.sh intel64
source /opt/intel/bin/iccvars.sh ia32
```

To install and make targets, use the `make install T=<target>` command in the top-level DPDK directory.

For example, to compile a 64-bit target using icc, run:

```
make install T=x86_64-native-linuxapp-icc
```

To compile a 32-bit build using gcc, the make command should be:

```
make install T=i686-native-linuxapp-gcc
```

To prepare a target without building it, for example, if the configuration changes need to be made before compilation, use the `make config T=<target>` command:

```
make config T=x86_64-native-linuxapp-gcc
```

Warning: Any kernel modules to be used, e.g. `igb_uio`, `kni`, must be compiled with the same kernel as the one running on the target. If the DPDK is not being built on the target machine, the `RTE_KERNELDIR` environment variable should be used to point the compilation at a copy of the kernel version to be used on the target machine.

Once the target environment is created, the user may move to the target environment directory and continue to make code changes and re-compile. The user may also make modifications to the compile-time DPDK configuration by editing the .config file in the build directory. (This is a build-local copy of the defconfig file from the top- level config directory).

```
cd x86_64-native-linuxapp-gcc
vi .config
make
```

In addition, the make clean command can be used to remove any existing compiled files for a subsequent full, clean rebuild of the code.

1.3.3 Browsing the Installed DPDK Environment Target

Once a target is created it contains all libraries, including poll-mode drivers, and header files for the DPDK environment that are required to build customer applications. In addition, the test and testpmd applications are built under the build/app directory, which may be used for testing. A kmod directory is also present that contains kernel modules which may be loaded if needed.

```
ls x86_64-native-linuxapp-gcc

app build hostapp include kmod lib Makefile
```

1.3.4 Loading Modules to Enable Userspace IO for DPDK

To run any DPDK application, a suitable uio module can be loaded into the running kernel. In many cases, the standard `uio_pci_generic` module included in the Linux kernel can provide the uio capability. This module can be loaded using the command

```
sudo modprobe uio_pci_generic
```

As an alternative to the `uio_pci_generic`, the DPDK also includes the igb_uio module which can be found in the kmod subdirectory referred to above. It can be loaded as shown below:

```
sudo modprobe uio
sudo insmod kmod/igb_uio.ko
```

Note: For some devices which lack support for legacy interrupts, e.g. virtual function (VF) devices, the `igb_uio` module may be needed in place of `uio_pci_generic`.

Since DPDK release 1.7 onward provides VFIO support, use of UIO is optional for platforms that support using VFIO.

1.3.5 Loading VFIO Module

To run an DPDK application and make use of VFIO, the `vfio-pci` module must be loaded:

```
sudo modprobe vfio-pci
```

Note that in order to use VFIO, your kernel must support it. VFIO kernel modules have been included in the Linux kernel since version 3.6.0 and are usually present by default, however please consult your distributions documentation to make sure that is the case.

Also, to use VFIO, both kernel and BIOS must support and be configured to use IO virtualization (such as Intel® VT-d).

For proper operation of VFIO when running DPDK applications as a non-privileged user, correct permissions should also be set up. This can be done by using the DPDK setup script (called setup.sh and located in the tools directory).

1.3.6 Binding and Unbinding Network Ports to/from the Kernel Modules

As of release 1.4, DPDK applications no longer automatically unbind all supported network ports from the kernel driver in use. Instead, all ports that are to be used by an DPDK application must be bound to the `uio_pci_generic`, `igb_uio` or `vfio-pci` module before the application is run. Any network ports under Linux* control will be ignored by the DPDK poll-mode drivers and cannot be used by the application.

> **Warning:** The DPDK will, by default, no longer automatically unbind network ports from the kernel driver at startup. Any ports to be used by an DPDK application must be unbound from Linux* control and bound to the `uio_pci_generic`, `igb_uio` or `vfio-pci` module before the application is run.

To bind ports to the `uio_pci_generic`, `igb_uio` or `vfio-pci` module for DPDK use, and then subsequently return ports to Linux* control, a utility script called dpdk_nic _bind.py is provided in the tools subdirectory. This utility can be used to provide a view of the current state of the network ports on the system, and to bind and unbind those ports from the different kernel modules, including the uio and vfio modules. The following are some examples of how the script can be used. A full description of the script and its parameters can be obtained by calling the script with the `--help` or `--usage` options. Note that the uio or vfio kernel modules to be used, should be loaded into the kernel before running the `dpdk_nic_bind.py` script.

> **Warning:** Due to the way VFIO works, there are certain limitations to which devices can be used with VFIO. Mainly it comes down to how IOMMU groups work. Any Virtual Function device can be used with VFIO on its own, but physical devices will require either all ports bound to VFIO, or some of them bound to VFIO while others not being bound to anything at all.
> If your device is behind a PCI-to-PCI bridge, the bridge will then be part of the IOMMU group in which your device is in. Therefore, the bridge driver should also be unbound from the bridge PCI device for VFIO to work with devices behind the bridge.

> **Warning:** While any user can run the dpdk_nic_bind.py script to view the status of the network ports, binding or unbinding network ports requires root privileges.

To see the status of all network ports on the system:

```
./tools/dpdk_nic_bind.py --status

Network devices using DPDK-compatible driver
============================================
0000:82:00.0 '82599EB 10-GbE NIC' drv=uio_pci_generic unused=ixgbe
0000:82:00.1 '82599EB 10-GbE NIC' drv=uio_pci_generic unused=ixgbe

Network devices using kernel driver
===================================
0000:04:00.0 'I350 1-GbE NIC' if=em0  drv=igb unused=uio_pci_generic *Active*
0000:04:00.1 'I350 1-GbE NIC' if=eth1 drv=igb unused=uio_pci_generic
0000:04:00.2 'I350 1-GbE NIC' if=eth2 drv=igb unused=uio_pci_generic
0000:04:00.3 'I350 1-GbE NIC' if=eth3 drv=igb unused=uio_pci_generic

Other network devices
=====================
<none>
```

To bind device `eth1`,“04:00.1“, to the `uio_pci_generic` driver:

```
./tools/dpdk_nic_bind.py --bind=uio_pci_generic 04:00.1
```

or, alternatively,

```
./tools/dpdk_nic_bind.py --bind=uio_pci_generic eth1
```

To restore device `82:00.0` to its original kernel binding:

```
./tools/dpdk_nic_bind.py --bind=ixgbe 82:00.0
```

1.4 Compiling and Running Sample Applications

The chapter describes how to compile and run applications in an DPDK environment. It also provides a pointer to where sample applications are stored.

Note: Parts of this process can also be done using the setup script described the *Quick Start Setup Script* section of this document.

1.4.1 Compiling a Sample Application

Once an DPDK target environment directory has been created (such as `x86_64-native-linuxapp-gcc`), it contains all libraries and header files required to build an application.

When compiling an application in the Linux* environment on the DPDK, the following variables must be exported:

- `RTE_SDK` - Points to the DPDK installation directory.
- `RTE_TARGET` - Points to the DPDK target environment directory.

The following is an example of creating the `helloworld` application, which runs in the DPDK Linux environment. This example may be found in the `${RTE_SDK}/examples` directory.

The directory contains the `main.c` file. This file, when combined with the libraries in the DPDK target environment, calls the various functions to initialize the DPDK environment, then launches an entry point (dispatch application) for each core to be utilized. By default, the binary is generated in the build directory.

```
cd examples/helloworld/
export RTE_SDK=$HOME/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc

make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map

ls build/app
    helloworld helloworld.map
```

Note: In the above example, `helloworld` was in the directory structure of the DPDK. However, it could have been located outside the directory structure to keep the DPDK structure intact. In the following case, the `helloworld` application is copied to a new directory as a new starting point.

```
export RTE_SDK=/home/user/DPDK
cp -r $(RTE_SDK)/examples/helloworld my_rte_app
cd my_rte_app/
export RTE_TARGET=x86_64-native-linuxapp-gcc

make
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map
```

1.4.2 Running a Sample Application

Warning: The UIO drivers and hugepages must be setup prior to running an application.

Warning: Any ports to be used by the application must be already bound to an appropriate kernel module, as described in *Binding and Unbinding Network Ports to/from the Kernel Modules*, prior to running the application.

The application is linked with the DPDK target environment's Environmental Abstraction Layer (EAL) library, which provides some options that are generic to every DPDK application.

The following is the list of options that can be given to the EAL:

```
./rte-app -c COREMASK [-n NUM] [-b <domain:bus:devid.func>] \
          [--socket-mem=MB,...] [-m MB] [-r NUM] [-v] [--file-prefix] \
          [--proc-type <primary|secondary|auto>] [-- xen-dom0]
```

The EAL options are as follows:

- `-c COREMASK`: An hexadecimal bit mask of the cores to run on. Note that core numbering can change between platforms and should be determined beforehand.
- `-n NUM`: Number of memory channels per processor socket.
- `-b <domain:bus:devid.func>`: Blacklisting of ports; prevent EAL from using specified PCI device (multiple `-b` options are allowed).
- `--use-device`: use the specified Ethernet device(s) only. Use comma-separate `[domain:]bus:devid.func` values. Cannot be used with `-b` option.
- `--socket-mem`: Memory to allocate from hugepages on specific sockets.
- `-m MB`: Memory to allocate from hugepages, regardless of processor socket. It is recommended that `--socket-mem` be used instead of this option.
- `-r NUM`: Number of memory ranks.
- `-v`: Display version information on startup.
- `--huge-dir`: The directory where hugetlbfs is mounted.
- `--file-prefix`: The prefix text used for hugepage filenames.
- `--proc-type`: The type of process instance.

- `--xen-dom0`: Support application running on Xen Domain0 without hugetlbfs.
- `--vmware-tsc-map`: Use VMware TSC map instead of native RDTSC.
- `--base-virtaddr`: Specify base virtual address.
- `--vfio-intr`: Specify interrupt type to be used by VFIO (has no effect if VFIO is not used).

The `-c` and option is mandatory; the others are optional.

Copy the DPDK application binary to your target, then run the application as follows (assuming the platform has four memory channels per processor socket, and that cores 0-3 are present and are to be used for running the application):

```
./helloworld -c f -n 4
```

Note: The `--proc-type` and `--file-prefix` EAL options are used for running multiple DPDK processes. See the “Multi-process Sample Application” chapter in the *DPDK Sample Applications User Guide* and the *DPDK Programmers Guide* for more details.

Logical Core Use by Applications

The coremask parameter is always mandatory for DPDK applications. Each bit of the mask corresponds to the equivalent logical core number as reported by Linux. Since these logical core numbers, and their mapping to specific cores on specific NUMA sockets, can vary from platform to platform, it is recommended that the core layout for each platform be considered when choosing the coremask to use in each case.

On initialization of the EAL layer by an DPDK application, the logical cores to be used and their socket location are displayed. This information can also be determined for all cores on the system by examining the `/proc/cpuinfo` file, for example, by running cat `/proc/cpuinfo`. The physical id attribute listed for each processor indicates the CPU socket to which it belongs. This can be useful when using other processors to understand the mapping of the logical cores to the sockets.

Note: A more graphical view of the logical core layout may be obtained using the `lstopo` Linux utility. On Fedora Linux, this may be installed and run using the following command:

```
sudo yum install hwloc
./lstopo
```

Warning: The logical core layout can change between different board layouts and should be checked before selecting an application coremask.

Hugepage Memory Use by Applications

When running an application, it is recommended to use the same amount of memory as that allocated for hugepages. This is done automatically by the DPDK application at startup, if no `-m` or `--socket-mem` parameter is passed to it when run.

If more memory is requested by explicitly passing a `-m` or `--socket-mem` value, the application fails. However, the application itself can also fail if the user requests less memory than the reserved amount of hugepage-memory, particularly if using the `-m` option. The reason is as follows. Suppose the system has 1024 reserved 2 MB pages in socket 0 and 1024 in socket 1. If the user requests 128 MB of memory, the 64 pages may not match the constraints:

- The hugepage memory by be given to the application by the kernel in socket 1 only. In this case, if the application attempts to create an object, such as a ring or memory pool in socket 0, it fails. To avoid this issue, it is recommended that the `--socket-mem` option be used instead of the `-m` option.
- These pages can be located anywhere in physical memory, and, although the DPDK EAL will attempt to allocate memory in contiguous blocks, it is possible that the pages will not be contiguous. In this case, the application is not able to allocate big memory pools.

The socket-mem option can be used to request specific amounts of memory for specific sockets. This is accomplished by supplying the `--socket-mem` flag followed by amounts of memory requested on each socket, for example, supply `--socket-mem=0,512` to try and reserve 512 MB for socket 1 only. Similarly, on a four socket system, to allocate 1 GB memory on each of sockets 0 and 2 only, the parameter `--socket-mem=1024,0,1024` can be used. No memory will be reserved on any CPU socket that is not explicitly referenced, for example, socket 3 in this case. If the DPDK cannot allocate enough memory on each socket, the EAL initialization fails.

1.4.3 Additional Sample Applications

Additional sample applications are included in the ${RTE_SDK}/examples directory. These sample applications may be built and run in a manner similar to that described in earlier sections in this manual. In addition, see the *DPDK Sample Applications User Guide* for a description of the application, specific instructions on compilation and execution and some explanation of the code.

1.4.4 Additional Test Applications

In addition, there are two other applications that are built when the libraries are created. The source files for these are in the DPDK/app directory and are called test and testpmd. Once the libraries are created, they can be found in the build/app directory.

- The test application provides a variety of specific tests for the various functions in the DPDK.
- The testpmd application provides a number of different packet throughput tests and examples of features such as how to use the Flow Director found in the Intel® 82599 10 Gigabit Ethernet Controller.

1.5 Enabling Additional Functionality

1.5.1 High Precision Event Timer HPET) Functionality

BIOS Support

The High Precision Timer (HPET) must be enabled in the platform BIOS if the HPET is to be used. Otherwise, the Time Stamp Counter (TSC) is used by default. The BIOS is typically accessed by pressing F2 while the platform is starting up. The user can then navigate to the HPET option. On the Crystal Forest platform BIOS, the path is: **Advanced -> PCH-IO Configuration -> High Precision Timer ->** (Change from Disabled to Enabled if necessary).

On a system that has already booted, the following command can be issued to check if HPET is enabled:

```
grep hpet /proc/timer_list
```

If no entries are returned, HPET must be enabled in the BIOS (as per the instructions above) and the system rebooted.

Linux Kernel Support

The DPDK makes use of the platform HPET timer by mapping the timer counter into the process address space, and as such, requires that the `HPET_MMAP` kernel configuration option be enabled.

Warning: On Fedora, and other common distributions such as Ubuntu, the `HPET_MMAP` kernel option is not enabled by default. To recompile the Linux kernel with this option enabled, please consult the distributions documentation for the relevant instructions.

Enabling HPET in the DPDK

By default, HPET support is disabled in the DPDK build configuration files. To use HPET, the `CONFIG_RTE_LIBEAL_USE_HPET` setting should be changed to `y`, which will enable the HPET settings at compile time.

For an application to use the `rte_get_hpet_cycles()` and `rte_get_hpet_hz()` API calls, and optionally to make the HPET the default time source for the rte_timer library, the new `rte_eal_hpet_init()` API call should be called at application initialization. This API call will ensure that the HPET is accessible, returning an error to the application if it is not, for example, if `HPET_MMAP` is not enabled in the kernel. The application can then determine what action to take, if any, if the HPET is not available at run-time.

Note: For applications that require timing APIs, but not the HPET timer specifically, it is recommended that the `rte_get_timer_cycles()` and `rte_get_timer_hz()` API calls be used instead of the HPET-specific APIs. These generic APIs can work with either TSC or HPET time sources, depending on what is requested by an application call to `rte_eal_hpet_init()`, if any, and on what is available on the system at runtime.

1.5.2 Running DPDK Applications Without Root Privileges

Although applications using the DPDK use network ports and other hardware resources directly, with a number of small permission adjustments it is possible to run these applications as a user other than "root". To do so, the ownership, or permissions, on the following Linux file system objects should be adjusted to ensure that the Linux user account being used to run the DPDK application has access to them:

- All directories which serve as hugepage mount points, for example, `/mnt/huge`
- The userspace-io device files in `/dev`, for example, `/dev/uio0`, `/dev/uio1`, and so on
- The userspace-io sysfs config and resource files, for example for `uio0`:

  ```
  /sys/class/uio/uio0/device/config
  /sys/class/uio/uio0/device/resource*
  ```

- If the HPET is to be used, `/dev/hpet`

Note: On some Linux installations, `/dev/hugepages` is also a hugepage mount point created by default.

1.5.3 Power Management and Power Saving Functionality

Enhanced Intel SpeedStep® Technology must be enabled in the platform BIOS if the power management feature of DPDK is to be used. Otherwise, the sys file folder `/sys/devices/system/cpu/cpu0/cpufreq` will not exist, and the CPU frequency- based power management cannot be used. Consult the relevant BIOS documentation to determine how these settings can be accessed.

For example, on some Intel reference platform BIOS variants, the path to Enhanced Intel SpeedStep® Technology is:

```
Advanced
  -> Processor Configuration
  -> Enhanced Intel SpeedStep® Tech
```

In addition, C3 and C6 should be enabled as well for power management. The path of C3 and C6 on the same platform BIOS is:

```
Advanced
  -> Processor Configuration
  -> Processor C3 Advanced
  -> Processor Configuration
  -> Processor C6
```

1.5.4 Using Linux Core Isolation to Reduce Context Switches

While the threads used by an DPDK application are pinned to logical cores on the system, it is possible for the Linux scheduler to run other tasks on those cores also. To help prevent additional workloads from running on those cores, it is possible to use the `isolcpus` Linux kernel parameter to isolate them from the general Linux scheduler.

For example, if DPDK applications are to run on logical cores 2, 4 and 6, the following should be added to the kernel parameter list:

```
isolcpus=2,4,6
```

1.5.5 Loading the DPDK KNI Kernel Module

To run the DPDK Kernel NIC Interface (KNI) sample application, an extra kernel module (the kni module) must be loaded into the running kernel. The module is found in the kmod sub-directory of the DPDK target directory. Similar to the loading of the `igb_uio` module, this module should be loaded using the insmod command as shown below (assuming that the current directory is the DPDK target directory):

```
insmod kmod/rte_kni.ko
```

Note: See the "Kernel NIC Interface Sample Application" chapter in the *DPDK Sample Applications User Guide* for more details.

1.5.6 Using Linux IOMMU Pass-Through to Run DPDK with Intel® VT-d

To enable Intel® VT-d in a Linux kernel, a number of kernel configuration options must be set. These include:

- `IOMMU_SUPPORT`
- `IOMMU_API`
- `INTEL_IOMMU`

In addition, to run the DPDK with Intel® VT-d, the `iommu=pt` kernel parameter must be used when using `igb_uio` driver. This results in pass-through of the DMAR (DMA Remapping) lookup in the host. Also, if `INTEL_IOMMU_DEFAULT_ON` is not set in the kernel, the `intel_iommu=on` kernel parameter must be used too. This ensures that the Intel IOMMU is being initialized as expected.

Please note that while using `iommu=pt` is compulsory for `igb_uio driver`, the `vfio-pci` driver can actually work with both `iommu=pt` and `iommu=on`.

1.5.7 High Performance of Small Packets on 40G NIC

As there might be firmware fixes for performance enhancement in latest version of firmware image, the firmware update might be needed for getting high performance. Check with the local Intel's Network Division application engineers for firmware updates. The base driver to support firmware version of FVL3E will be integrated in the next DPDK release, so currently the validated firmware version is 4.2.6.

Enabling Extended Tag and Setting Max Read Request Size

PCI configurations of `extended_tag` and max _read_requ st_size have big impacts on performance of small packets on 40G NIC. Enabling extended_tag and setting `max_read_request_size` to small size such as 128 bytes provide great helps to high performance of small packets.

- These can be done in some BIOS implementations.

- For other BIOS implementations, PCI configurations can be changed by using command of `setpci`, or special configurations in DPDK config file of `common_linux`.
  - Bits 7:5 at address of 0xA8 of each PCI device is used for setting the max_read_request_size, and bit 8 of 0xA8 of each PCI device is used for enabling/disabling the extended_tag. lspci and setpci can be used to read the values of 0xA8 and then write it back after being changed.
  - In config file of common_linux, below three configurations can be changed for the same purpose.

    `CONFIG_RTE_PCI_CONFIG`

    `CONFIG_RTE_PCI_EXTENDED_TAG`

    `CONFIG_RTE_PCI_MAX_READ_REQUEST_SIZE`

Use 16 Bytes RX Descriptor Size

As i40e PMD supports both 16 and 32 bytes RX descriptor sizes, and 16 bytes size can provide helps to high performance of small packets. Configuration of `CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC` in config files can be changed to use 16 bytes size RX descriptors.

High Performance and per Packet Latency Tradeoff

Due to the hardware design, the interrupt signal inside NIC is needed for per packet descriptor write-back. The minimum interval of interrupts could be set at compile time by `CONFIG_RTE_LIBRTE_I40E_ITR_INTERVAL` in configuration files. Though there is a default configuration, the interval could be tuned by the users with that configuration item depends on what the user cares about more, performance or per packet latency.

1.6 Quick Start Setup Script

The setup.sh script, found in the tools subdirectory, allows the user to perform the following tasks:

- Build the DPDK libraries
- Insert and remove the DPDK IGB_UIO kernel module
- Insert and remove VFIO kernel modules
- Insert and remove the DPDK KNI kernel module
- Create and delete hugepages for NUMA and non-NUMA cases
- View network port status and reserve ports for DPDK application use
- Set up permissions for using VFIO as a non-privileged user
- Run the test and testpmd applications
- Look at hugepages in the meminfo
- List hugepages in `/mnt/huge`

- Remove built DPDK libraries

Once these steps have been completed for one of the EAL targets, the user may compile their own application that links in the EAL libraries to create the DPDK image.

1.6.1 Script Organization

The setup.sh script is logically organized into a series of steps that a user performs in sequence. Each step provides a number of options that guide the user to completing the desired task. The following is a brief synopsis of each step.

Step 1: Build DPDK Libraries

Initially, the user must select a DPDK target to choose the correct target type and compiler options to use when building the libraries.

The user must have all libraries, modules, updates and compilers installed in the system prior to this, as described in the earlier chapters in this Getting Started Guide.

Step 2: Setup Environment

The user configures the Linux* environment to support the running of DPDK applications. Hugepages can be set up for NUMA or non-NUMA systems. Any existing hugepages will be removed. The DPDK kernel module that is needed can also be inserted in this step, and network ports may be bound to this module for DPDK application use.

Step 3: Run an Application

The user may run the test application once the other steps have been performed. The test application allows the user to run a series of functional tests for the DPDK. The testpmd application, which supports the receiving and sending of packets, can also be run.

Step 4: Examining the System

This step provides some tools for examining the status of hugepage mappings.

Step 5: System Cleanup

The final step has options for restoring the system to its original state.

1.6.2 Use Cases

The following are some example of how to use the setup.sh script. The script should be run using the source command. Some options in the script prompt the user for further data before proceeding.

Warning: The setup.sh script should be run with root privileges.

```
source tools/setup.sh

------------------------------------------------------------------------

RTE_SDK exported as /home/user/rte

------------------------------------------------------------------------

Step 1: Select the DPDK environment to build
```

```
----------------------------------------------------------------------

[1] i686-native-linuxapp-gcc

[2] i686-native-linuxapp-icc

[3] ppc_64-power8-linuxapp-gcc

[4] x86_64-ivshmem-linuxapp-gcc

[5] x86_64-ivshmem-linuxapp-icc

[6] x86_64-native-bsdapp-clang

[7] x86_64-native-bsdapp-gcc

[8] x86_64-native-linuxapp-clang

[9] x86_64-native-linuxapp-gcc

[10] x86_64-native-linuxapp-icc

----------------------------------------------------------------------

Step 2: Setup linuxapp environment

----------------------------------------------------------------------

[11] Insert IGB UIO module

[12] Insert VFIO module

[13] Insert KNI module

[14] Setup hugepage mappings for non-NUMA systems

[15] Setup hugepage mappings for NUMA systems

[16] Display current Ethernet device settings

[17] Bind Ethernet device to IGB UIO module

[18] Bind Ethernet device to VFIO module

[19] Setup VFIO permissions

----------------------------------------------------------------------

Step 3: Run test application for linuxapp environment

----------------------------------------------------------------------

[20] Run test application ($RTE_TARGET/app/test)

[21] Run testpmd application in interactive mode ($RTE_TARGET/app/testpmd)

----------------------------------------------------------------------

Step 4: Other tools

----------------------------------------------------------------------
```

```
[22] List hugepage info from /proc/meminfo

----------------------------------------------------------------------

Step 5: Uninstall and system cleanup

----------------------------------------------------------------------

[23] Uninstall all targets

[24] Unbind NICs from IGB UIO driver

[25] Remove IGB UIO module

[26] Remove VFIO module

[27] Remove KNI module

[28] Remove hugepage mappings

[29] Exit Script
```

Option:

The following selection demonstrates the creation of the `x86_64-native-linuxapp-gcc` DPDK library.

```
Option: 9

================== Installing x86_64-native-linuxapp-gcc

Configuration done
== Build lib
...
Build complete
RTE_TARGET exported as x86_64-native-linuxapp-gcc
```

The following selection demonstrates the starting of the DPDK UIO driver.

```
Option: 25

Unloading any existing DPDK UIO module
Loading DPDK UIO module
```

The following selection demonstrates the creation of hugepages in a NUMA system. 1024 2 MByte pages are assigned to each node. The result is that the application should use -m 4096 for starting the application to access both memory areas (this is done automatically if the -m option is not provided).

Note: If prompts are displayed to remove temporary files, type 'y'.

```
Option: 15

Removing currently reserved hugepages
mounting /mnt/huge and removing directory
Input the number of 2MB pages for each node
Example: to have 128MB of hugepages available per node,
enter '64' to reserve 64 * 2MB pages on each node
Number of pages for node0: 1024
Number of pages for node1: 1024
Reserving hugepages
Creating /mnt/huge and mounting as hugetlbfs
```

The following selection demonstrates the launch of the test application to run on a single core.

```
Option: 20

Enter hex bitmask of cores to execute test app on
Example: to execute app on cores 0 to 7, enter 0xff
bitmask: 0x01
Launching app
EAL: coremask set to 1
EAL: Detected lcore 0 on socket 0
...
EAL: Master core 0 is ready (tid=1b2ad720)
RTE>>
```

1.6.3 Applications

Once the user has run the setup.sh script, built one of the EAL targets and set up hugepages (if using one of the Linux EAL targets), the user can then move on to building and running their application or one of the examples provided.

The examples in the /examples directory provide a good starting point to gain an understanding of the operation of the DPDK. The following command sequence shows how the helloworld sample application is built and run. As recommended in Section 4.2.1 , “Logical Core Use by Applications”, the logical core layout of the platform should be determined when selecting a core mask to use for an application.

```
cd helloworld/
make
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map

sudo ./build/app/helloworld -c 0xf -n 3
[sudo] password for rte:

EAL: coremask set to f
EAL: Detected lcore 0 as core 0 on socket 0
EAL: Detected lcore 1 as core 0 on socket 1
EAL: Detected lcore 2 as core 1 on socket 0
EAL: Detected lcore 3 as core 1 on socket 1
EAL: Setting up hugepage memory...
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0add800000 (size = 0x200000)
EAL: Ask a virtual area of 0x3d400000 bytes
EAL: Virtual area found at 0x7f0aa0200000 (size = 0x3d400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9fc00000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9f600000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9f000000 (size = 0x400000)
EAL: Ask a virtual area of 0x800000 bytes
EAL: Virtual area found at 0x7f0a9e600000 (size = 0x800000)
EAL: Ask a virtual area of 0x800000 bytes
EAL: Virtual area found at 0x7f0a9dc00000 (size = 0x800000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9d600000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
EAL: Virtual area found at 0x7f0a9d000000 (size = 0x400000)
EAL: Ask a virtual area of 0x400000 bytes
```

```
EAL: Virtual area found at 0x7f0a9ca00000 (size = 0x400000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a9c600000 (size = 0x200000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a9c200000 (size = 0x200000)
EAL: Ask a virtual area of 0x3fc00000 bytes
EAL: Virtual area found at 0x7f0a5c400000 (size = 0x3fc00000)
EAL: Ask a virtual area of 0x200000 bytes
EAL: Virtual area found at 0x7f0a5c000000 (size = 0x200000)
EAL: Requesting 1024 pages of size 2MB from socket 0
EAL: Requesting 1024 pages of size 2MB from socket 1
EAL: Master core 0 is ready (tid=de25b700)
EAL: Core 1 is ready (tid=5b7fe700)
EAL: Core 3 is ready (tid=5a7fc700)
EAL: Core 2 is ready (tid=5affd700)
hello from core 1
hello from core 2
hello from core 3
hello from core 0
```

1.7 How to get best performance with NICs on Intel platforms

This document is a step-by-step guide for getting high performance from DPDK applications on Intel platforms.

1.7.1 Hardware and Memory Requirements

For best performance use an Intel Xeon class server system such as Ivy Bridge, Haswell or newer.

Ensure that each memory channel has at least one memory DIMM inserted, and that the memory size for each is at least 4GB. **Note**: this has one of the most direct effects on performance.

You can check the memory configuration using `dmidecode` as follows:

```
dmidecode -t memory | grep Locator

Locator: DIMM_A1
Bank Locator: NODE 1
Locator: DIMM_A2
Bank Locator: NODE 1
Locator: DIMM_B1
Bank Locator: NODE 1
Locator: DIMM_B2
Bank Locator: NODE 1
...
Locator: DIMM_G1
Bank Locator: NODE 2
Locator: DIMM_G2
Bank Locator: NODE 2
Locator: DIMM_H1
Bank Locator: NODE 2
Locator: DIMM_H2
Bank Locator: NODE 2
```

The sample output above shows a total of 8 channels, from `A` to `H`, where each channel has 2 DIMMs.

You can also use `dmidecode` to determine the memory frequency:

```
dmidecode -t memory | grep Speed

Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
...
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
Speed: 2133 MHz
Configured Clock Speed: 2134 MHz
Speed: Unknown
Configured Clock Speed: Unknown
```

The output shows a speed of 2133 MHz (DDR4) and Unknown (not existing). This aligns with the previous output which showed that each channel has one memory bar.

Network Interface Card Requirements

Use a DPDK supported high end NIC such as the Intel XL710 40GbE.

Make sure each NIC has been flashed the latest version of NVM/firmware.

Use PCIe Gen3 slots, such as Gen3 `x8` or Gen3 `x16` because PCIe Gen2 slots don't provide enough bandwidth for 2 x 10GbE and above. You can use `lspci` to check the speed of a PCI slot using something like the following:

```
lspci -s 03:00.1 -vv | grep LnkSta

LnkSta: Speed 8GT/s, Width x8, TrErr- Train- SlotClk+ DLActive- ...
LnkSta2: Current De-emphasis Level: -6dB, EqualizationComplete+ ...
```

When inserting NICs into PCI slots always check the caption, such as CPU0 or CPU1 to indicate which socket it is connected to.

Care should be take with NUMA. If you are using 2 or more ports from different NICs, it is best to ensure that these NICs are on the same CPU socket. An example of how to determine this is shown further below.

BIOS Settings

The following are some recommendations on BIOS settings. Different platforms will have different BIOS naming so the following is mainly for reference:

1. Before starting consider resetting all BIOS settings to their default.
2. Disable all power saving options such as: Power performance tuning, CPU P-State, CPU C3 Report and CPU C6 Report.
3. Select **Performance** as the CPU Power and Performance policy.
4. Disable Turbo Boost to ensure the performance scaling increases with the number of cores.

5. Set memory frequency to the highest available number, NOT auto.
6. Disable all virtualization options when you test the physical function of the NIC, and turn on `VT-d` if you wants to use VFIO.

Linux boot command line

The following are some recommendations on GRUB boot settings:

1. Use the default grub file as a starting point.
2. Reserve 1G huge pages via grub configurations. For example to reserve 8 huge pages of 1G size:

   ```
   default_hugepagesz=1G hugepagesz=1G hugepages=8
   ```

3. Isolate CPU cores which will be used for DPDK. For example:

   ```
   isolcpus=2,3,4,5,6,7,8
   ```

4. If it wants to use VFIO, use the following additional grub parameters:

   ```
   iommu=pt intel_iommu=on
   ```

1.7.2 Configurations before running DPDK

1. Build the DPDK target and reserve huge pages. See the earlier section on *Use of Hugepages in the Linux Environment* for more details.

   The following shell commands may help with building and configuration:

   ```
   # Build DPDK target.
   cd dpdk_folder
   make install T=x86_64-native-linuxapp-gcc -j

   # Get the hugepage size.
   awk '/Hugepagesize/ {print $2}' /proc/meminfo

   # Get the total huge page numbers.
   awk '/HugePages_Total/ {print $2} ' /proc/meminfo

   # Unmount the hugepages.
   umount `awk '/hugetlbfs/ {print $2}' /proc/mounts`

   # Create the hugepage mount folder.
   mkdir -p /mnt/huge

   # Mount to the specific folder.
   mount -t hugetlbfs nodev /mnt/huge
   ```

2. Check the CPU layout using using the DPDK `cpu_layout` utility:

   ```
   cd dpdk_folder

   tools/cpu_layout.py
   ```

   Or run `lscpu` to check the the cores on each socket.

3. Check your NIC id and related socket id:

   ```
   # List all the NICs with PCI address and device IDs.
   lspci -nn | grep Eth
   ```

For example suppose your output was as follows:

```
82:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
82:00.1 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.1 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
```

Check the PCI device related numa node id:

```
cat /sys/bus/pci/devices/0000\:xx\:00.x/numa_node
```

Usually `0x:00.x` is on socket 0 and `8x:00.x` is on socket 1. **Note**: To get the best performance, ensure that the core and NICs are in the same socket. In the example above `85:00.0` is on socket 1 and should be used by cores on socket 1 for the best performance.

4. Bind the test ports to DPDK compatible drivers, such as igb_uio. For example bind two ports to a DPDK compatible driver and check the status:

   ```
   # Bind ports 82:00.0 and 85:00.0 to dpdk driver
   ./dpdk_folder/tools/dpdk_nic_bind.py -b igb_uio 82:00.0 85:00.0

   # Check the port driver status
   ./dpdk_folder/tools/dpdk_nic_bind.py --status
   ```

   See `dpdk_nic_bind.py --help` for more details.

More details about DPDK setup and Linux kernel requirements see linux_gsg_compiling_dpdk.

1.7.3 Example of getting best performance for an Intel NIC

The following is an example of running the DPDK `l3fwd` sample application to get high performance with an Intel server platform and Intel XL710 NICs. For specific 40G NIC configuration please refer to the i40e NIC guide.

The example scenario is to get best performance with two Intel XL710 40GbE ports. See Fig. 1.1 for the performance test setup.

Fig. 1.1: Performance Test Setup

1. Add two Intel XL710 NICs to the platform, and use one port per card to get best performance. The reason for using two NICs is to overcome a PCIe Gen3's limitation since it cannot provide 80G bandwidth for two 40G ports, but two different PCIe Gen3 x8 slot can. Refer to the sample NICs output above, then we can select `82:00.0` and `85:00.0` as test ports:

   ```
   82:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
   85:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
   ```

2. Connect the ports to the traffic generator. For high speed testing, it's best to use a hardware traffic generator.

3. Check the PCI devices numa node (socket id) and get the cores number on the exact socket id. In this case, `82:00.0` and `85:00.0` are both in socket 1, and the cores on socket 1 in the referenced platform are 18-35 and 54-71. Note: Don't use 2 logical cores on the same core (e.g core18 has 2 logical cores, core18 and core54), instead, use 2 logical cores from different cores (e.g core18 and core19).

4. Bind these two ports to igb_uio.

5. As to XL710 40G port, we need at least two queue pairs to achieve best performance, then two queues per port will be required, and each queue pair will need a dedicated CPU core for receiving/transmitting packets.
6. The DPDK sample application `l3fwd` will be used for performance testing, with using two ports for bi-directional forwarding. Compile the `l3fwd sample` with the default lpm mode.
7. The command line of running l3fwd would be something like the followings:

   ```
   ./l3fwd -c 0x3c0000 -n 4 -w 82:00.0 -w 85:00.0 \
           -- -p 0x3 --config '(0,0,18),(0,1,19),(1,0,20),(1,1,21)'
   ```

   This means that the application uses core 18 for port 0, queue pair 0 forwarding, core 19 for port 0, queue pair 1 forwarding, core 20 for port 1, queue pair 0 forwarding, and core 21 for port 1, queue pair 1 forwarding.
8. Configure the traffic at a traffic generator.
   - Start creating a stream on packet generator.
   - Set the Ethernet II type to 0x0800.

CHAPTER 2

Getting Started Guide for FreeBSD

2.1 Introduction

This document contains instructions for installing and configuring the Data Plane Development Kit (DPDK) software. It is designed to get customers up and running quickly and describes how to compile and run a DPDK application in a FreeBSD application (bsdapp) environment, without going deeply into detail.

For a comprehensive guide to installing and using FreeBSD, the following handbook is available from the FreeBSD Documentation Project: FreeBSD Handbook.

Note: The DPDK is now available as part of the FreeBSD ports collection. Installing via the ports collection infrastructure is now the recommended way to install the DPDK on FreeBSD, and is documented in the next chapter, *Installing DPDK from the Ports Collection*.

2.1.1 Documentation Roadmap

The following is a list of DPDK documents in the suggested reading order:

- **Release Notes** : Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guide** (this document): Describes how to install and configure the DPDK; designed to get users up and running quickly with the software.
- **Programmer's Guide**: Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linuxapp) environment
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

  A glossary of terms is also provided.

- **API Reference**: Provides detailed information about DPDK functions, data structures and other programming constructs.
- **Sample Applications User Guide**: Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

2.2 Installing DPDK from the Ports Collection

The easiest way to get up and running with the DPDK on FreeBSD is to install it from the ports collection. Details of getting and using the ports collection are documented in the FreeBSD Handbook.

Note: Testing has been performed using FreeBSD 10.0-RELEASE (x86_64) and requires the installation of the kernel sources, which should be included during the installation of FreeBSD.

2.2.1 Installing the DPDK FreeBSD Port

On a system with the ports collection installed in `/usr/ports`, the DPDK can be installed using the commands:

```
cd /usr/ports/net/dpdk

make install
```

After the installation of the DPDK port, instructions will be printed on how to install the kernel modules required to use the DPDK. A more complete version of these instructions can be found in the sections *Loading the DPDK contigmem Module* and *Loading the DPDK nic_uio Module*. Normally, lines like those below would be added to the file `/boot/loader.conf`.

```
# Reserve 2 x 1G blocks of contiguous memory using contigmem driver:
hw.contigmem.num_buffers=2
hw.contigmem.buffer_size=1073741824
contigmem_load="YES"

# Identify NIC devices for DPDK apps to use and load nic_uio driver:
hw.nic_uio.bdfs="2:0:0,2:0:1"
nic_uio_load="YES"
```

2.2.2 Compiling and Running the Example Applications

When the DPDK has been installed from the ports collection it installs its example applications in `/usr/local/share/dpdk/examples` - also accessible via symlink as `/usr/local/share/examples/dpdk`. These examples can be compiled and run as described in *Compiling and Running Sample Applications*. In this case, the required environmental variables should be set as below:

- `RTE_SDK=/usr/local/share/dpdk`
- `RTE_TARGET=x86_64-native-bsdapp-clang`

Note: To install a copy of the DPDK compiled using gcc, please download the official DPDK package from http://dpdk.org/ and install manually using the instructions given in the next chapter, *Compiling the DPDK Target from Source*

An example application can therefore be copied to a user's home directory and compiled and run as below:

```
export RTE_SDK=/usr/local/share/dpdk

export RTE_TARGET=x86_64-native-bsdapp-clang

cp -r /usr/local/share/dpdk/examples/helloworld .

cd helloworld/

gmake
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map

sudo ./build/helloworld -c F -n 2

EAL: Contigmem driver has 2 buffers, each of size 1GB
EAL: Sysctl reports 8 cpus
EAL: Detected lcore 0
EAL: Detected lcore 1
EAL: Detected lcore 2
EAL: Detected lcore 3
EAL: Support maximum 64 logical core(s) by configuration.
EAL: Detected 4 lcore(s)
EAL: Setting up physically contiguous memory...
EAL: Mapped memory segment 1 @ 0x802400000: len 1073741824
EAL: Mapped memory segment 2 @ 0x842400000: len 1073741824
EAL: WARNING: clock_gettime cannot use CLOCK_MONOTONIC_RAW and HPET
     is not available - clock timings may be less accurate.
EAL: TSC frequency is ~3569023 KHz
EAL: PCI scan found 24 devices
EAL: Master core 0 is ready (tid=0x802006400)
EAL: Core 1 is ready (tid=0x802006800)
EAL: Core 3 is ready (tid=0x802007000)
EAL: Core 2 is ready (tid=0x802006c00)
EAL: PCI device 0000:01:00.0 on NUMA socket 0
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x80074a000
EAL:   PCI memory mapped at 0x8007ca000
EAL: PCI device 0000:01:00.1 on NUMA socket 0
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x8007ce000
EAL:   PCI memory mapped at 0x80084e000
EAL: PCI device 0000:02:00.0 on NUMA socket 0
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x800852000
EAL:   PCI memory mapped at 0x8008d2000
EAL: PCI device 0000:02:00.1 on NUMA socket 0
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x801b3f000
EAL:   PCI memory mapped at 0x8008d6000
hello from core 1
hello from core 2
hello from core 3
hello from core 0
```

Note: To run a DPDK process as a non-root user, adjust the permissions on the `/dev/contigmem` and `/dev/uio device` nodes as described in section *Running DPDK Applications Without Root Privileges*

Note: For an explanation of the command-line parameters that can be passed to an DPDK application, see section *Running a Sample Application*.

2.3 Compiling the DPDK Target from Source

2.3.1 System Requirements

The DPDK and its applications require the GNU make system (gmake) to build on FreeBSD. Optionally, gcc may also be used in place of clang to build the DPDK, in which case it too must be installed prior to compiling the DPDK. The installation of these tools is covered in this section.

Compiling the DPDK requires the FreeBSD kernel sources, which should be included during the installation of FreeBSD on the development platform. The DPDK also requires the use of FreeBSD ports to compile and function.

To use the FreeBSD ports system, it is required to update and extract the FreeBSD ports tree by issuing the following commands:

```
portsnap fetch
portsnap extract
```

If the environment requires proxies for external communication, these can be set using:

```
setenv http_proxy <my_proxy_host>:<port>
setenv ftp_proxy <my_proxy_host>:<port>
```

The FreeBSD ports below need to be installed prior to building the DPDK. In general these can be installed using the following set of commands:

```
cd /usr/ports/<port_location>

make config-recursive

make install

make clean
```

Each port location can be found using:

```
whereis <port_name>
```

The ports required and their locations are as follows:

- dialog4ports: `/usr/ports/ports-mgmt/dialog4ports`
- GNU make(gmake): `/usr/ports/devel/gmake`
- coreutils: `/usr/ports/sysutils/coreutils`

For compiling and using the DPDK with gcc, the compiler must be installed from the ports collection:

- gcc: version 4.8 is recommended `/usr/ports/lang/gcc48`. Ensure that `CPU_OPTS` is selected (default is OFF).

When running the make config-recursive command, a dialog may be presented to the user. For the installation of the DPDK, the default options were used.

Note: To avoid multiple dialogs being presented to the user during make install, it is advisable before running the make install command to re-run the make config-recursive command until no more dialogs are seen.

2.3.2 Install the DPDK and Browse Sources

First, uncompress the archive and move to the DPDK source directory:

```
unzip DPDK-<version>.zip
cd DPDK-<version>

ls
app/ config/ examples/ lib/ LICENSE.GPL LICENSE.LGPL Makefile
mk/ scripts/ tools/
```

The DPDK is composed of several directories:

- lib: Source code of DPDK libraries
- app: Source code of DPDK applications (automatic tests)
- examples: Source code of DPDK applications
- config, tools, scripts, mk: Framework-related makefiles, scripts and configuration

2.3.3 Installation of the DPDK Target Environments

The format of a DPDK target is:

```
ARCH-MACHINE-EXECENV-TOOLCHAIN
```

Where:

- `ARCH` is: `x86_64`
- `MACHINE` is: `native`
- `EXECENV` is: `bsdapp`
- `TOOLCHAIN` is: `gcc` | `clang`

The configuration files for the DPDK targets can be found in the DPDK/config directory in the form of:

```
defconfig_ARCH-MACHINE-EXECENV-TOOLCHAIN
```

Note: Configuration files are provided with the `RTE_MACHINE` optimization level set. Within the configuration files, the `RTE_MACHINE` configuration value is set to native, which means that the compiled software is tuned for the platform on which it is built. For more information on this setting, and its possible values, see the *DPDK Programmers Guide*.

To make the target, use `gmake install T=<target>`.

For example to compile for FreeBSD use:

```
gmake install T=x86_64-native-bsdapp-clang
```

Note: If the compiler binary to be used does not correspond to that given in the TOOLCHAIN part of the target, the compiler command may need to be explicitly specified. For example, if compiling for gcc, where the gcc binary is called gcc4.8, the command would need to be `gmake install T=<target> CC=gcc4.8`.

2.3.4 Browsing the Installed DPDK Environment Target

Once a target is created, it contains all the libraries and header files for the DPDK environment that are required to build customer applications. In addition, the test and testpmd applications are built under the build/app directory, which may be used for testing. A kmod directory is also present that contains the kernel modules to install:

```
ls x86_64-native-bsdapp-gcc

app build hostapp include kmod lib Makefile
```

2.3.5 Loading the DPDK contigmem Module

To run a DPDK application, physically contiguous memory is required. In the absence of non-transparent superpages, the included sources for the contigmem kernel module provides the ability to present contiguous blocks of memory for the DPDK to use. The contigmem module must be loaded into the running kernel before any DPDK is run. The module is found in the kmod sub-directory of the DPDK target directory.

The amount of physically contiguous memory along with the number of physically contiguous blocks to be reserved by the module can be set at runtime prior to module loading using:

```
kenv hw.contigmem.num_buffers=n
kenv hw.contigmem.buffer_size=m
```

The kernel environment variables can also be specified during boot by placing the following in `/boot/loader.conf`:

```
hw.contigmem.num_buffers=n hw.contigmem.buffer_size=m
```

The variables can be inspected using the following command:

```
sysctl -a hw.contigmem
```

Where n is the number of blocks and m is the size in bytes of each area of contiguous memory. A default of two buffers of size 1073741824 bytes (1 Gigabyte) each is set during module load if they are not specified in the environment.

The module can then be loaded using kldload (assuming that the current directory is the DPDK target directory):

```
kldload ./kmod/contigmem.ko
```

It is advisable to include the loading of the contigmem module during the boot process to avoid issues with potential memory fragmentation during later system up time. This can be achieved by copying the module to the `/boot/kernel/` directory and placing the following into `/boot/loader.conf`:

```
contigmem_load="YES"
```

Note: The contigmem_load directive should be placed after any definitions of `hw.contigmem.num_buffers` and `hw.contigmem.buffer_size` if the default values are not to be used.

An error such as:

```
kldload: can't load ./x86_64-native-bsdapp-gcc/kmod/contigmem.ko:
         Exec format error
```

is generally attributed to not having enough contiguous memory available and can be verified via dmesg or `/var/log/messages`:

```
kernel: contigmalloc failed for buffer <n>
```

To avoid this error, reduce the number of buffers or the buffer size.

2.3.6 Loading the DPDK nic_uio Module

After loading the contigmem module, the `nic_uio must` also be loaded into the running kernel prior to running any DPDK application. This module must be loaded using the kldload command as shown below (assuming that the current directory is the DPDK target directory).

```
kldload ./kmod/nic_uio.ko
```

Note: If the ports to be used are currently bound to a existing kernel driver then the `hw.nic_uio.bdfs sysctl` value will need to be set before loading the module. Setting this value is described in the next section below.

Currently loaded modules can be seen by using the `kldstat` command and a module can be removed from the running kernel by using `kldunload <module_name>`.

To load the module during boot, copy the `nic_uio` module to `/boot/kernel` and place the following into `/boot/loader.conf`:

```
nic_uio_load="YES"
```

Note: `nic_uio_load="YES"` must appear after the contigmem_load directive, if it exists.

By default, the `nic_uio` module will take ownership of network ports if they are recognized DPDK devices and are not owned by another module. However, since the FreeBSD kernel includes support, either built-in, or via a separate driver module, for most network card devices, it is likely that the ports to be used are already bound to a driver other than `nic_uio`. The following sub-section describe how to query and modify the device ownership of the ports to be used by DPDK applications.

Binding Network Ports to the nic_uio Module

Device ownership can be viewed using the pciconf -l command. The example below shows four Intel® 82599 network ports under `if_ixgbe` module ownership.

```
pciconf -l
ix0@pci0:1:0:0: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
ix1@pci0:1:0:1: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
ix2@pci0:2:0:0: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
ix3@pci0:2:0:1: class=0x020000 card=0x00038086 chip=0x10fb8086 rev=0x01 hdr=0x00
```

The first column constitutes three components:

1. Device name: `ixN`
2. Unit name: `pci0`
3. Selector (Bus:Device:Function): `1:0:0`

Where no driver is associated with a device, the device name will be `none`.

By default, the FreeBSD kernel will include built-in drivers for the most common devices; a kernel rebuild would normally be required to either remove the drivers or configure them as loadable modules.

To avoid building a custom kernel, the `nic_uio` module can detach a network port from its current device driver. This is achieved by setting the `hw.nic_uio.bdfs` kernel environment variable prior to loading `nic_uio`, as follows:

```
hw.nic_uio.bdfs="b:d:f,b:d:f,..."
```

Where a comma separated list of selectors is set, the list must not contain any whitespace.

For example to re-bind `ix2@pci0:2:0:0` and `ix3@pci0:2:0:1` to the `nic_uio` module upon loading, use the following command:

```
kenv hw.nic_uio.bdfs="2:0:0,2:0:1"
```

The variable can also be specified during boot by placing the following into `/boot/loader.conf`, before the previously-described `nic_uio_load` line - as shown:

```
hw.nic_uio.bdfs="2:0:0,2:0:1"
nic_uio_load="YES"
```

Binding Network Ports Back to their Original Kernel Driver

If the original driver for a network port has been compiled into the kernel, it is necessary to reboot FreeBSD to restore the original device binding. Before doing so, update or remove the `hw.nic_uio.bdfs` in `/boot/loader.conf`.

If rebinding to a driver that is a loadable module, the network port binding can be reset without rebooting. To do so, unload both the target kernel module and the `nic_uio` module, modify or clear the `hw.nic_uio.bdfs` kernel environment (kenv) value, and reload the two drivers - first the original kernel driver, and then the `nic_uio driver`. Note: the latter does not need to be reloaded unless there are ports that are still to be bound to it.

Example commands to perform these steps are shown below:

```
kldunload nic_uio
kldunload <original_driver>

# To clear the value completely:
```

```
kenv -u hw.nic_uio.bdfs

# To update the list of ports to bind:
kenv hw.nic_uio.bdfs="b:d:f,b:d:f,..."

kldload <original_driver>

kldload nic_uio  # optional
```

2.4 Compiling and Running Sample Applications

The chapter describes how to compile and run applications in a DPDK environment. It also provides a pointer to where sample applications are stored.

2.4.1 Compiling a Sample Application

Once a DPDK target environment directory has been created (such as `x86_64-native-bsdapp-clang`), it contains all libraries and header files required to build an application.

When compiling an application in the FreeBSD environment on the DPDK, the following variables must be exported:

- `RTE_SDK` - Points to the DPDK installation directory.
- `RTE_TARGET` - Points to the DPDK target environment directory. For FreeBSD, this is the `x86_64-native-bsdapp-clang` or `x86_64-native-bsdapp-gcc` directory.

The following is an example of creating the `helloworld` application, which runs in the DPDK FreeBSD environment. While the example demonstrates compiling using gcc version 4.8, compiling with clang will be similar, except that the `CC=` parameter can probably be omitted. The `helloworld` example may be found in the `${RTE_SDK}/examples` directory.

The directory contains the `main.c` file. This file, when combined with the libraries in the DPDK target environment, calls the various functions to initialize the DPDK environment, then launches an entry point (dispatch application) for each core to be utilized. By default, the binary is generated in the build directory.

```
setenv RTE_SDK /home/user/DPDK
cd $(RTE_SDK)
cd examples/helloworld/
setenv RTE_SDK $HOME/DPDK
setenv RTE_TARGET x86_64-native-bsdapp-gcc

gmake CC=gcc48
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map

ls build/app
  helloworld helloworld.map
```

Note: In the above example, `helloworld` was in the directory structure of the DPDK. However, it could have been located outside the directory structure to keep the DPDK structure intact. In the following case, the `helloworld` application is copied to a new directory as a new starting point.

```
setenv RTE_SDK /home/user/DPDK
cp -r $(RTE_SDK)/examples/helloworld my_rte_app
cd my_rte_app/
setenv RTE_TARGET x86_64-native-bsdapp-gcc

gmake CC=gcc48
  CC main.o
  LD helloworld
  INSTALL-APP helloworld
  INSTALL-MAP helloworld.map
```

2.4.2 Running a Sample Application

1. The `contigmem` and `nic_uio` modules must be set up prior to running an application.
2. Any ports to be used by the application must be already bound to the `nic_uio` module, as described in section *Binding Network Ports to the nic_uio Module*, prior to running the application. The application is linked with the DPDK target environment's Environment Abstraction Layer (EAL) library, which provides some options that are generic to every DPDK application.

The following is the list of options that can be given to the EAL:

```
./rte-app -c COREMASK [-n NUM] [-b <domain:bus:devid.func>] \
          [-r NUM] [-v] [--proc-type <primary|secondary|auto>]
```

Note: EAL has a common interface between all operating systems and is based on the Linux notation for PCI devices. For example, a FreeBSD device selector of `pci0:2:0:1` is referred to as `02:00.1` in EAL.

The EAL options for FreeBSD are as follows:

- `-c COREMASK`: A hexadecimal bit mask of the cores to run on. Note that core numbering can change between platforms and should be determined beforehand.
- `-n NUM`: Number of memory channels per processor socket.
- `-b <domain:bus:devid.func>`: Blacklisting of ports; prevent EAL from using specified PCI device (multiple `-b` options are allowed).
- `--use-device`: Use the specified Ethernet device(s) only. Use comma-separate `[domain:]bus:devid.func` values. Cannot be used with `-b` option.
- `-r NUM`: Number of memory ranks.
- `-v`: Display version information on startup.
- `--proc-type`: The type of process instance.

Other options, specific to Linux and are not supported under FreeBSD are as follows:

- `socket-mem`: Memory to allocate from hugepages on specific sockets.
- `--huge-dir`: The directory where hugetlbfs is mounted.

- `--file-prefix`: The prefix text used for hugepage filenames.
- `-m MB`: Memory to allocate from hugepages, regardless of processor socket. It is recommended that `--socket-mem` be used instead of this option.

The `-c` option is mandatory; the others are optional.

Copy the DPDK application binary to your target, then run the application as follows (assuming the platform has four memory channels, and that cores 0-3 are present and are to be used for running the application):

```
./helloworld -c f -n 4
```

Note: The `--proc-type` and `--file-prefix` EAL options are used for running multiple DPDK processes. See the "Multi-process Sample Application" chapter in the *DPDK Sample Applications User Guide and the DPDK Programmers Guide* for more details.

2.4.3 Running DPDK Applications Without Root Privileges

Although applications using the DPDK use network ports and other hardware resources directly, with a number of small permission adjustments, it is possible to run these applications as a user other than "root". To do so, the ownership, or permissions, on the following file system objects should be adjusted to ensure that the user account being used to run the DPDK application has access to them:

- The userspace-io device files in `/dev`, for example, `/dev/uio0`, `/dev/uio1`, and so on
- The userspace contiguous memory device: `/dev/contigmem`

Note: Please refer to the DPDK Release Notes for supported applications.

CHAPTER 3

Xen Guide

3.1 DPDK Xen Based Packet-Switching Solution

3.1.1 Introduction

DPDK provides a para-virtualization packet switching solution, based on the Xen hypervisor's Grant Table, Note 1, which provides simple and fast packet switching capability between guest domains and host domain based on MAC address or VLAN tag.

This solution is comprised of two components; a Poll Mode Driver (PMD) as the front end in the guest domain and a switching back end in the host domain. XenStore is used to exchange configure information between the PMD front end and switching back end, including grant reference IDs for shared Virtio RX/TX rings, MAC address, device state, and so on. XenStore is an information storage space shared between domains, see further information on XenStore below.

The front end PMD can be found in the DPDK directory lib/ librte_pmd_xenvirt and back end example in examples/vhost_xen.

The PMD front end and switching back end use shared Virtio RX/TX rings as para- virtualized interface. The Virtio ring is created by the front end, and Grant table references for the ring are passed to host. The switching back end maps those grant table references and creates shared rings in a mapped address space.

The following diagram describes the functionality of the DPDK Xen Packet- Switching Solution.

Note 1 The Xen hypervisor uses a mechanism called a Grant Table to share memory between domains (http://wiki.xen.org/wiki/Grant Table).

A diagram of the design is shown below, where "gva" is the Guest Virtual Address, which is the data pointer of the mbuf, and "hva" is the Host Virtual Address:

In this design, a Virtio ring is used as a para-virtualized interface for better performance over a Xen private ring when packet switching to and from a VM. The additional performance is gained by avoiding a system call and memory map in each memory copy with a XEN private ring.

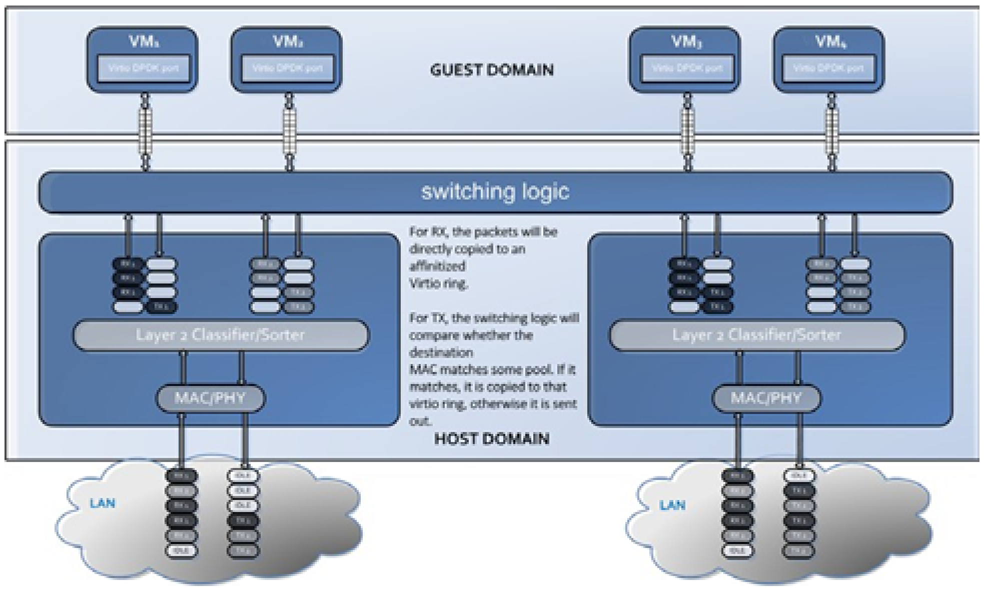

VM₁
Virtio DPDK port
VM₂
Virtio DPDK port
GUEST DOMAIN
VM₃
Virtio DPDK port
VM₄
Virtio DPDK port
switching logic
For RX, the packets will be directly copied to an affinitized Virtio ring.
For TX, the switching logic will compare whether the destination MAC matches some pool. If it matches, it is copied to that virtio ring, otherwise it is sent out.
Layer 2 Classifier/Sorter
MAC/PHY
Layer 2 Classifier/Sorter
MAC/PHY
HOST DOMAIN
LAN
LAN

Fig. 3.1: Functionality of the DPDK Xen Packet Switching Solution.

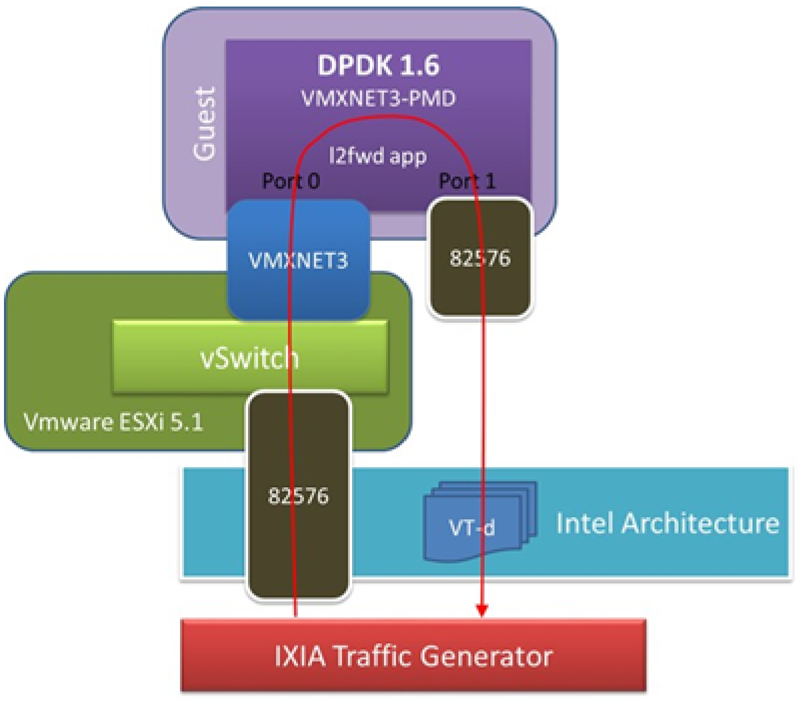


Fig. 3.2: DPDK Xen Layout

3.1.2 Device Creation

Poll Mode Driver Front End

- Mbuf pool allocation:

  To use a Xen switching solution, the DPDK application should use rte_mempool_gntalloc_create() to reserve mbuf pools during initialization. rte_mempool_gntalloc_create() creates a mempool with objects from memory allocated and managed via gntalloc/gntdev.

  The DPDK now supports construction of mempools from allocated virtual memory through the rte_mempool_xmem_create() API.

  This front end constructs mempools based on memory allocated through the xen_gntalloc driver. rte_mempool_gntalloc_create() allocates Grant pages, maps them to continuous virtual address space, and calls rte_mempool_xmem_create() to build mempools. The Grant IDs for all Grant pages are passed to the host through XenStore.

- Virtio Ring Creation:

  The Virtio queue size is defined as 256 by default in the VQ_DESC_NUM macro. Using the queue setup function, Grant pages are allocated based on ring size and are mapped to continuous virtual address space to form the Virtio ring. Normally, one ring is comprised of several pages. Their Grant IDs are passed to the host through XenStore.

  There is no requirement that this memory be physically continuous.

- Interrupt and Kick:

  There are no interrupts in DPDK Xen Switching as both front and back ends work in polling mode. There is no requirement for notification.

- Feature Negotiation:

  Currently, feature negotiation through XenStore is not supported.

- Packet Reception & Transmission:

  With mempools and Virtio rings created, the front end can operate Virtio devices, as it does in Virtio PMD for KVM Virtio devices with the exception that the host does not require notifications or deal with interrupts.

XenStore is a database that stores guest and host information in the form of (key, value) pairs. The following is an example of the information generated during the startup of the front end PMD in a guest VM (domain ID 1):

```
xenstore -ls /local/domain/1/control/dpdk
0_mempool_gref="3042,3043,3044,3045"
0_mempool_va="0x7fcbc6881000"
0_tx_vring_gref="3049"
0_rx_vring_gref="3053"
0_ether_addr="4e:0b:d0:4e:aa:f1"
0_vring_flag="3054"
...
```

Multiple mempools and multiple Virtios may exist in the guest domain, the first number is the index, starting from zero.

The idx#_mempool_va stores the guest virtual address for mempool idx#.

The idx#_ether_adder stores the MAC address of the guest Virtio device.

For idx#_rx_ring_gref, idx#_tx_ring_gref, and idx#_mempool_gref, the value is a list of Grant references. Take idx#_mempool_gref node for example, the host maps those Grant references to a continuous virtual address space. The real Grant reference information is stored in this virtual address space, where (gref, pfn) pairs follow each other with -1 as the terminator.

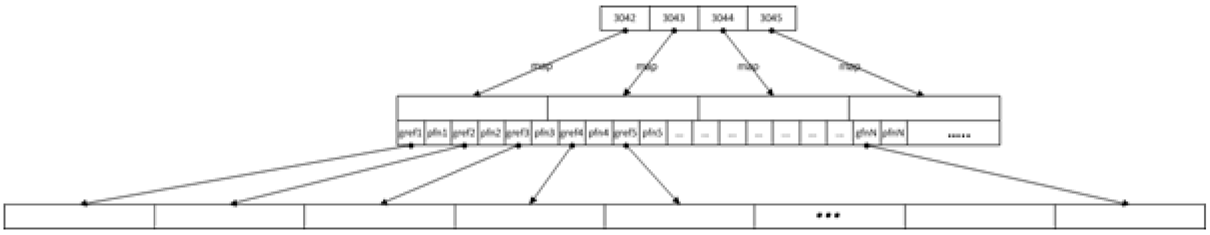


Fig. 3.3: Mapping Grant references to a continuous virtual address space

After all gref# IDs are retrieved, the host maps them to a continuous virtual address space. With the guest mempool virtual address, the host establishes 1:1 address mapping. With multiple guest mempools, the host establishes multiple address translation regions.

Switching Back End

The switching back end monitors changes in XenStore. When the back end detects that a new Virtio device has been created in a guest domain, it will:

1. Retrieve Grant and configuration information from XenStore.
2. Map and create a Virtio ring.
3. Map mempools in the host and establish address translation between the guest address and host address.
4. Select a free VMDQ pool, set its affinity with the Virtio device, and set the MAC/ VLAN filter.

Packet Reception

When packets arrive from an external network, the MAC?VLAN filter classifies packets into queues in one VMDQ pool. As each pool is bonded to a Virtio device in some guest domain, the switching back end will:

1. Fetch an available entry from the Virtio RX ring.
2. Get gva, and translate it to hva.
3. Copy the contents of the packet to the memory buffer pointed to by gva.

The DPDK application in the guest domain, based on the PMD front end, is polling the shared Virtio RX ring for available packets and receives them on arrival.

Packet Transmission

When a Virtio device in one guest domain is to transmit a packet, it puts the virtual address of the packet's data area into the shared Virtio TX ring.

The packet switching back end is continuously polling the Virtio TX ring. When new packets are available for transmission from a guest, it will:

1. Fetch an available entry from the Virtio TX ring.
2. Get gva, and translate it to hva.
3. Copy the packet from hva to the host mbuf's data area.
4. Compare the destination MAC address with all the MAC addresses of the Virtio devices it manages. If a match exists, it directly copies the packet to the matched VIrtio RX ring. Otherwise, it sends the packet out through hardware.

Note: The packet switching back end is for demonstration purposes only. The user could implement their switching logic based on this example. In this example, only one physical port on the host is supported. Multiple segments are not supported. The biggest mbuf supported is 4KB. When the back end is restarted, all front ends must also be restarted.

3.1.3 Running the Application

The following describes the steps required to run the application.

Validated Environment

Host:

Xen-hypervisor: 4.2.2

Distribution: Fedora release 18

Kernel: 3.10.0

Xen development package (including Xen, Xen-libs, xen-devel): 4.2.3

Guest:

Distribution: Fedora 16 and 18

Kernel: 3.6.11

Xen Host Prerequisites

Note that the following commands might not be the same on different Linux* distributions.

- Install xen-devel package:

  ```
  yum install xen-devel.x86_64
  ```

- Start xend if not already started:

  ```
  /etc/init.d/xend start
  ```

- Mount xenfs if not already mounted:

  ```
  mount -t xenfs none /proc/xen
  ```

- Enlarge the limit for xen_gntdev driver:

```
modprobe -r xen_gntdev
modprobe xen_gntdev limit=1000000
```

Note: The default limit for earlier versions of the xen_gntdev driver is 1024. That is insufficient to support the mapping of multiple Virtio devices into multiple VMs, so it is necessary to enlarge the limit by reloading this module. The default limit of recent versions of xen_gntdev is 1048576. The rough calculation of this limit is:

> limit=nb_mbuf# * VM#.
>
> In DPDK examples, nb_mbuf# is normally 8192.

Building and Running the Switching Backend

1. Edit config/common_linuxapp, and change the default configuration value for the following two items:

   ```
   CONFIG_RTE_LIBRTE_XEN_DOM0=y
   CONFIG RTE_LIBRTE_PMD_XENVIRT=n
   ```

2. Build the target:

   ```
   make install T=x86_64-native-linuxapp-gcc
   ```

3. Ensure that RTE_SDK and RTE_TARGET are correctly set. Build the switching example:

   ```
   make -C examples/vhost_xen/
   ```

4. Load the Xen DPDK memory management module and preallocate memory:

   ```
   insmod ./x86_64-native-linuxapp-gcc/build/lib/librte_eal/linuxapp/xen_dom0/rte_dom0_mm.ko
   echo 2048> /sys/kernel/mm/dom0-mm/memsize-mB/memsize
   ```

   Note: On Xen Dom0, there is no hugepage support. Under Xen Dom0, the DPDK uses a special memory management kernel module to allocate chunks of physically continuous memory. Refer to the *DPDK Getting Started Guide* for more information on memory management in the DPDK. In the above command, 4 GB memory is reserved (2048 of 2 MB pages) for DPDK.

5. Load uio_pci_generic and bind one Intel NIC controller to it:

   ```
   modprobe uio_pci_generic
   python tools/dpdk_nic_bind.py -b uio_pci_generic 0000:09:00:00.0
   ```

   In this case, 0000:09:00.0 is the PCI address for the NIC controller.

6. Run the switching back end example:

   ```
   examples/vhost_xen/build/vhost-switch -c f -n 3 --xen-dom0 -- -p1
   ```

Note: The -xen-dom0 option instructs the DPDK to use the Xen kernel module to allocate memory.

Other Parameters:

- -vm2vm

  The vm2vm parameter enables/disables packet switching in software. Disabling vm2vm implies that on a VM packet transmission will always go to the Ethernet port and will not be switched to another VM

- -Stats

  The Stats parameter controls the printing of Virtio-net device statistics. The parameter specifies the interval (in seconds) at which to print statistics, an interval of 0 seconds will disable printing statistics.

Xen PMD Frontend Prerequisites

1. Install xen-devel package for accessing XenStore:

   ```
   yum install xen-devel.x86_64
   ```

2. Mount xenfs, if it is not already mounted:

   ```
   mount -t xenfs none /proc/xen
   ```

3. Enlarge the default limit for xen_gntalloc driver:

   ```
   modprobe -r xen_gntalloc
   modprobe xen_gntalloc limit=6000
   ```

Note: Before the Linux kernel version 3.8-rc5, Jan 15th 2013, a critical defect occurs when a guest is heavily allocating Grant pages. The Grant driver allocates fewer pages than expected which causes kernel memory corruption. This happens, for example, when a guest uses the v1 format of a Grant table entry and allocates more than 8192 Grant pages (this number might be different on different hypervisor versions). To work around this issue, set the limit for gntalloc driver to 6000. (The kernel normally allocates hundreds of Grant pages with one Xen front end per virtualized device). If the kernel allocates a lot of Grant pages, for example, if the user uses multiple net front devices, it is best to upgrade the Grant alloc driver. This defect has been fixed in kernel version 3.8-rc5 and later.

Building and Running the Front End

1. Edit config/common_linuxapp, and change the default configuration value:

   ```
   CONFIG_RTE_LIBRTE_XEN_DOM0=n
   CONFIG_RTE_LIBRTE_PMD_XENVIRT=y
   ```

2. Build the package:

   ```
   make install T=x86_64-native-linuxapp-gcc
   ```

3. Enable hugepages. Refer to the *DPDK Getting Started Guide* for instructions on how to use hugepages in the DPDK.

4. Run TestPMD. Refer to *DPDK TestPMD Application User Guide* for detailed parameter usage.

   ```
   ./x86_64-native-linuxapp-gcc/app/testpmd -c f -n 4 --vdev="eth_xenvirt0,mac=00:00:00:00:00
   testpmd>set fwd mac
   testpmd>start
   ```

As an example to run two TestPMD instances over 2 Xen Virtio devices:

```
--vdev="eth_xenvirt0,mac=00:00:00:00:00:11" --vdev="eth_xenvirt1;mac=00:00:00:00:00:22"
```

Usage Examples: Injecting a Packet Stream Using a Packet Generator

Loopback Mode

Run TestPMD in a guest VM:

```
./x86_64-native-linuxapp-gcc/app/testpmd -c f -n 4 --vdev="eth_xenvirt0,mac=00:00:00:00:00:11"
testpmd> set fwd mac
testpmd> start
```

Example output of the vhost_switch would be:

```
DATA:(0) MAC_ADDRESS 00:00:00:00:00:11 and VLAN_TAG 1000 registered.
```

The above message indicates that device 0 has been registered with MAC address 00:00:00:00:00:11 and VLAN tag 1000. Any packets received on the NIC with these values is placed on the device's receive queue.

Configure a packet stream in the packet generator, set the destination MAC address to 00:00:00:00:00:11, and VLAN to 1000, the guest Virtio receives these packets and sends them out with destination MAC address 00:00:00:00:00:22.

Inter-VM Mode

Run TestPMD in guest VM1:

```
./x86_64-native-linuxapp-gcc/app/testpmd -c f -n 4 --vdev="eth_xenvirt0,mac=00:00:00:00:00:11"
```

Run TestPMD in guest VM2:

```
./x86_64-native-linuxapp-gcc/app/testpmd -c f -n 4 --vdev="eth_xenvirt0,mac=00:00:00:00:00:22"
```

Configure a packet stream in the packet generator, and set the destination MAC address to 00:00:00:00:00:11 and VLAN to 1000. The packets received in Virtio in guest VM1 will be forwarded to Virtio in guest VM2 and then sent out through hardware with destination MAC address 00:00:00:00:00:33.

The packet flow is:

packet generator->Virtio in guest VM1->switching backend->Virtio in guest VM2->switching backend->wire

CHAPTER 4

Programmer's Guide

4.1 Introduction

This document provides software architecture information, development environment information and optimization guidelines.

For programming examples and for instructions on compiling and running each sample application, see the *DPDK Sample Applications User Guide* for details.

For general information on compiling and running applications, see the *DPDK Getting Started Guide*.

4.1.1 Documentation Roadmap

The following is a list of DPDK documents in the suggested reading order:

- **Release Notes** (this document): Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guide** : Describes how to install and configure the DPDK software; designed to get users up and running quickly with the software.
- **FreeBSD* Getting Started Guide** : A document describing the use of the DPDK with FreeBSD* has been added in DPDK Release 1.6.0. Refer to this guide for installation and configuration instructions to get started using the DPDK with FreeBSD*.
- **Programmer's Guide** (this document): Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linuxapp) environment
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

  A glossary of terms is also provided.
- **API Reference** : Provides detailed information about DPDK functions, data structures and other programming constructs.

- **Sample Applications User Guide**: Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

4.1.2 Related Publications

The following documents provide information that is relevant to the development of applications using the DPDK:

- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide

Part 1: Architecture Overview

4.2 Overview

This section gives a global overview of the architecture of Data Plane Development Kit (DPDK).

The main goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. Users may use the code to understand some of the techniques employed, to build upon for prototyping or to add their own protocol stacks. Alternative ecosystem options that use the DPDK are available.

The framework creates a set of libraries for specific environments through the creation of an Environment Abstraction Layer (EAL), which may be specific to a mode of the Intel® architecture (32-bit or 64-bit), Linux* user space compilers or a specific platform. These environments are created through the use of make files and configuration files. Once the EAL library is created, the user may link with the library to create their own applications. Other libraries, outside of EAL, including the Hash, Longest Prefix Match (LPM) and rings libraries are also provided. Sample applications are provided to help show the user how to use various features of the DPDK.

The DPDK implements a run to completion model for packet processing, where all resources must be allocated prior to calling Data Plane applications, running as execution units on logical processing cores. The model does not support a scheduler and all devices are accessed by polling. The primary reason for not using interrupts is the performance overhead imposed by interrupt processing.

In addition to the run-to-completion model, a pipeline model may also be used by passing packets or messages between cores via the rings. This allows work to be performed in stages and may allow more efficient use of code on cores.

4.2.1 Development Environment

The DPDK project installation requires Linux and the associated toolchain, such as one or more compilers, assembler, make utility, editor and various libraries to create the DPDK components and libraries.

Once these libraries are created for the specific environment and architecture, they may then be used to create the user's data plane application.

When creating applications for the Linux user space, the glibc library is used. For DPDK applications, two environmental variables (RTE_SDK and RTE_TARGET) must be configured before compiling the applications. The following are examples of how the variables can be set:

```
export RTE_SDK=/home/user/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started Guide* for information on setting up the development environment.

4.2.2 Environment Abstraction Layer

The Environment Abstraction Layer (EAL) provides a generic interface that hides the environment specifics from the applications and libraries. The services provided by the EAL are:

- DPDK loading and launching
- Support for multi-process and multi-thread execution types
- Core affinity/assignment procedures
- System memory allocation/de-allocation
- Atomic/lock operations
- Time reference
- PCI bus access
- Trace and debug functions
- CPU feature identification
- Interrupt handling
- Alarm operations
- Memory management (malloc)

The EAL is fully described in *Environment Abstraction Layer*.

4.2.3 Core Components

The *core components* are a set of libraries that provide all the elements needed for high-performance packet processing applications.

Fig. 4.1: Core Components Architecture

Ring Manager (librte_ring)

The ring structure provides a lockless multi-producer, multi-consumer FIFO API in a finite size table. It has some advantages over lockless queues; easier to implement, adapted to bulk operations and faster. A ring is used by the *Memory Pool Manager (librte_mempool)* and may be used as a general communication mechanism between cores and/or execution blocks connected together on a logical core.

This ring buffer and its usage are fully described in *Ring Library*.

Memory Pool Manager (librte_mempool)

The Memory Pool Manager is responsible for allocating pools of objects in memory. A pool is identified by name and uses a ring to store free objects. It provides some other optional services, such as a per-core object cache and an alignment helper to ensure that objects are padded to spread them equally on all RAM channels.

This memory pool allocator is described in *Mempool Library*.

Network Packet Buffer Management (librte_mbuf)

The mbuf library provides the facility to create and destroy buffers that may be used by the DPDK application to store message buffers. The message buffers are created at startup time and stored in a mempool, using the DPDK mempool library.

This library provide an API to allocate/free mbufs, manipulate control message buffers (ctrlmbuf) which are generic message buffers, and packet buffers (pktmbuf) which are used to carry network packets.

Network Packet Buffer Management is described in *Mbuf Library*.

Timer Manager (librte_timer)

This library provides a timer service to DPDK execution units, providing the ability to execute a function asynchronously. It can be periodic function calls, or just a one-shot call. It uses the timer interface provided by the Environment Abstraction Layer (EAL) to get a precise time reference and can be initiated on a per-core basis as required.

The library documentation is available in *Timer Library*.

4.2.4 Ethernet* Poll Mode Driver Architecture

The DPDK includes Poll Mode Drivers (PMDs) for 1 GbE, 10 GbE and 40GbE, and para virtualized virtio Ethernet controllers which are designed to work without asynchronous, interrupt-based signaling mechanisms.

See *Poll Mode Driver*.

4.2.5 Packet Forwarding Algorithm Support

The DPDK includes Hash (librte_hash) and Longest Prefix Match (LPM,librte_lpm) libraries to support the corresponding packet forwarding algorithms.

See *Hash Library* and *LPM Library* for more information.

4.2.6 librte_net

The librte_net library is a collection of IP protocol definitions and convenience macros. It is based on code from the FreeBSD* IP stack and contains protocol numbers (for use in IP headers), IP-related macros, IPv4/IPv6 header structures and TCP, UDP and SCTP header structures.

4.3 Environment Abstraction Layer

The Environment Abstraction Layer (EAL) is responsible for gaining access to low-level resources such as hardware and memory space. It provides a generic interface that hides the environment specifics from the applications and libraries. It is the responsibility of the initialization routine to decide how to allocate these resources (that is, memory space, PCI devices, timers, consoles, and so on).

Typical services expected from the EAL are:

- DPDK Loading and Launching: The DPDK and its application are linked as a single application and must be loaded by some means.
- Core Affinity/Assignment Procedures: The EAL provides mechanisms for assigning execution units to specific cores as well as creating execution instances.
- System Memory Reservation: The EAL facilitates the reservation of different memory zones, for example, physical memory areas for device interactions.
- PCI Address Abstraction: The EAL provides an interface to access PCI address space.
- Trace and Debug Functions: Logs, dump_stack, panic and so on.
- Utility Functions: Spinlocks and atomic counters that are not provided in libc.
- CPU Feature Identification: Determine at runtime if a particular feature, for example, Intel® AVX is supported. Determine if the current CPU supports the feature set that the binary was compiled for.
- Interrupt Handling: Interfaces to register/unregister callbacks to specific interrupt sources.
- Alarm Functions: Interfaces to set/remove callbacks to be run at a specific time.

4.3.1 EAL in a Linux-userland Execution Environment

In a Linux user space environment, the DPDK application runs as a user-space application using the pthread library. PCI information about devices and address space is discovered through the /sys kernel interface and through kernel modules such as uio_pci_generic, or igb_uio. Refer to the UIO: User-space drivers documentation in the Linux kernel. This memory is mmap'd in the application.

The EAL performs physical memory allocation using mmap() in hugetlbfs (using huge page sizes to increase performance). This memory is exposed to DPDK service layers such as the *Mempool Library*.

At this point, the DPDK services layer will be initialized, then through pthread setaffinity calls, each execution unit will be assigned to a specific logical core to run as a user-level thread.

The time reference is provided by the CPU Time-Stamp Counter (TSC) or by the HPET kernel API through a mmap() call.

Initialization and Core Launching

Part of the initialization is done by the start function of glibc. A check is also performed at initialization time to ensure that the micro architecture type chosen in the config file is supported by the CPU. Then, the main() function is called. The core initialization and launch is done in rte_eal_init() (see the API documentation). It consist of calls to the pthread library (more specifically, pthread_self(), pthread_create(), and pthread_setaffinity_np()).

Fig. 4.2: EAL Initialization in a Linux Application Environment

---

Note: Initialization of objects, such as memory zones, rings, memory pools, lpm tables and hash tables, should be done as part of the overall application initialization on the master lcore. The creation and initialization functions for these objects are not multi-thread safe. However, once initialized, the objects themselves can safely be used in multiple threads simultaneously.

Multi-process Support

The Linuxapp EAL allows a multi-process as well as a multi-threaded (pthread) deployment model. See chapter 2.20 *Multi-process Support* for more details.

Memory Mapping Discovery and Memory Reservation

The allocation of large contiguous physical memory is done using the hugetlbfs kernel filesystem. The EAL provides an API to reserve named memory zones in this contiguous memory. The physical address of the reserved memory for that memory zone is also returned to the user by the memory zone reservation API.

Note: Memory reservations done using the APIs provided by rte_malloc are also backed by pages from the hugetlbfs filesystem.

Xen Dom0 support without hugetbls

The existing memory management implementation is based on the Linux kernel hugepage mechanism. However, Xen Dom0 does not support hugepages, so a new Linux kernel module rte_dom0_mm is added to workaround this limitation.

The EAL uses IOCTL interface to notify the Linux kernel module rte_dom0_mm to allocate memory of specified size, and get all memory segments information from the module, and the EAL uses MMAP interface to map the allocated memory. For each memory segment, the physical addresses are contiguous within it but actual hardware addresses are contiguous within 2MB.

PCI Access

The EAL uses the /sys/bus/pci utilities provided by the kernel to scan the content on the PCI bus. To access PCI memory, a kernel module called uio_pci_generic provides a /dev/uioX device file and resource files in /sys that can be mmap'd to obtain access to PCI address space from the application. The DPDK-specific igb_uio module can also be used for this. Both drivers use the uio kernel feature (userland driver).

Per-lcore and Shared Variables

Note: lcore refers to a logical execution unit of the processor, sometimes called a hardware *thread*.

Shared variables are the default behavior. Per-lcore variables are implemented using *Thread Local Storage* (TLS) to provide per-thread local storage.

Logs

A logging API is provided by EAL. By default, in a Linux application, logs are sent to syslog and also to the console. However, the log function can be overridden by the user to use a different logging mechanism.

Trace and Debug Functions

There are some debug functions to dump the stack in glibc. The rte_panic() function can voluntarily provoke a SIG_ABORT, which can trigger the generation of a core file, readable by gdb.

CPU Feature Identification

The EAL can query the CPU at runtime (using the rte_cpu_get_feature() function) to determine which CPU features are available.

User Space Interrupt Event

- User Space Interrupt and Alarm Handling in Host Thread

The EAL creates a host thread to poll the UIO device file descriptors to detect the interrupts. Callbacks can be registered or unregistered by the EAL functions for a specific interrupt event and are called in the host thread asynchronously. The EAL also allows timed callbacks to be used in the same way as for NIC interrupts.

Note: In DPDK PMD, the only interrupts handled by the dedicated host thread are those for link status change, i.e. link up and link down notification.

- RX Interrupt Event

The receive and transmit routines provided by each PMD don't limit themselves to execute in polling thread mode. To ease the idle polling with tiny throughput, it's useful to pause the polling and wait until the wake-up event happens. The RX interrupt is the first choice to be such kind of wake-up event, but probably won't be the only one.

EAL provides the event APIs for this event-driven thread mode. Taking linuxapp as an example, the implementation relies on epoll. Each thread can monitor an epoll instance in which all the wake-up events' file descriptors are added. The event file descriptors are created and mapped to the interrupt vectors according to the UIO/VFIO spec. From bsdapp's perspective, kqueue is the alternative way, but not implemented yet.

EAL initializes the mapping between event file descriptors and interrupt vectors, while each device initializes the mapping between interrupt vectors and queues. In this way, EAL actually is unaware of the interrupt cause on the specific vector. The eth_dev driver takes responsibility to program the latter mapping.

Note: Per queue RX interrupt event is only allowed in VFIO which supports multiple MSI-X vector. In UIO, the RX interrupt together with other interrupt causes shares the same vector. In this case, when RX interrupt and LSC(link status change) interrupt are both enabled(intr_conf.lsc == 1 && intr_conf.rxq == 1), only the former is capable.

The RX interrupt are controlled/enabled/disabled by ethdev APIs - 'rte_eth_dev_rx_intr_*'. They return failure if the PMD hasn't support them yet. The intr_conf.rxq flag is used to turn on the capability of RX interrupt per device.

Blacklisting

The EAL PCI device blacklist functionality can be used to mark certain NIC ports as blacklisted, so they are ignored by the DPDK. The ports to be blacklisted are identified using the PCIe* description (Domain:Bus:Device.Function).

Misc Functions

Locks and atomic operations are per-architecture (i686 and x86_64).

4.3.2 Memory Segments and Memory Zones (memzone)

The mapping of physical memory is provided by this feature in the EAL. As physical memory can have gaps, the memory is described in a table of descriptors, and each descriptor (called rte_memseg) describes a contiguous portion of memory.

On top of this, the memzone allocator's role is to reserve contiguous portions of physical memory. These zones are identified by a unique name when the memory is reserved.

The rte_memzone descriptors are also located in the configuration structure. This structure is accessed using rte_eal_get_configuration(). The lookup (by name) of a memory zone returns a descriptor containing the physical address of the memory zone.

Memory zones can be reserved with specific start address alignment by supplying the align parameter (by default, they are aligned to cache line size). The alignment value should be a power of two and not less than the cache line size (64 bytes). Memory zones can also be reserved from either 2 MB or 1 GB hugepages, provided that both are available on the system.

4.3.3 Multiple pthread

DPDK usually pins one pthread per core to avoid the overhead of task switching. This allows for significant performance gains, but lacks flexibility and is not always efficient.

Power management helps to improve the CPU efficiency by limiting the CPU runtime frequency. However, alternately it is possible to utilize the idle cycles available to take advantage of the full capability of the CPU.

By taking advantage of cgroup, the CPU utilization quota can be simply assigned. This gives another way to improve the CPU efficiency, however, there is a prerequisite; DPDK must handle the context switching between multiple pthreads per core.

For further flexibility, it is useful to set pthread affinity not only to a CPU but to a CPU set.

EAL pthread and lcore Affinity

The term "lcore" refers to an EAL thread, which is really a Linux/FreeBSD pthread. "EAL pthreads" are created and managed by EAL and execute the tasks issued by *remote_launch*. In each EAL pthread, there is a TLS (Thread Local Storage) called *_lcore_id* for unique identification. As EAL pthreads usually bind 1:1 to the physical CPU, the *_lcore_id* is typically equal to the CPU ID.

When using multiple pthreads, however, the binding is no longer always 1:1 between an EAL pthread and a specified physical CPU. The EAL pthread may have affinity to a CPU set, and as such the *_lcore_id* will not be the same as the CPU ID. For this reason, there is an EAL long option '–lcores' defined to assign the CPU affinity of lcores. For a specified lcore ID or ID group, the option allows setting the CPU set for that EAL pthread.

The format pattern: –lcores='<lcore_set>[@cpu_set][,<lcore_set>[@cpu_set],...]'

'lcore_set' and 'cpu_set' can be a single number, range or a group.

A number is a "digit([0-9]+)"; a range is "<number>-<number>"; a group is "(<number|range>[,<number|range>,...])".

If a '@cpu_set' value is not supplied, the value of 'cpu_set' will default to the value of 'lcore_set'.

```
For example, "--lcores='1,2@(5-7),(3-5)@(0,2),(0,6),7-8'" which means start 9 EAL thread;
    lcore 0 runs on cpuset 0x41 (cpu 0,6);
    lcore 1 runs on cpuset 0x2 (cpu 1);
    lcore 2 runs on cpuset 0xe0 (cpu 5,6,7);
    lcore 3,4,5 runs on cpuset 0x5 (cpu 0,2);
    lcore 6 runs on cpuset 0x41 (cpu 0,6);
    lcore 7 runs on cpuset 0x80 (cpu 7);
    lcore 8 runs on cpuset 0x100 (cpu 8).
```

Using this option, for each given lcore ID, the associated CPUs can be assigned. It's also compatible with the pattern of corelist('-l') option.

non-EAL pthread support

It is possible to use the DPDK execution context with any user pthread (aka. Non-EAL pthreads). In a non-EAL pthread, the *_lcore_id* is always LCORE_ID_ANY which identifies that it is not an EAL thread with a valid, unique, *_lcore_id*. Some libraries will use an alternative unique ID (e.g. TID), some will not be impacted at all, and some will work but with limitations (e.g. timer and mempool libraries).

All these impacts are mentioned in *Known Issues* section.

Public Thread API

There are two public APIs `rte_thread_set_affinity()` and `rte_pthread_get_affinity()` introduced for threads. When they're used in any pthread context, the Thread Local Storage(TLS) will be set/get.

Those TLS include *_cpuset* and *_socket_id*:

- *_cpuset* stores the CPUs bitmap to which the pthread is affinitized.
- *_socket_id* stores the NUMA node of the CPU set. If the CPUs in CPU set belong to different NUMA node, the *_socket_id* will be set to SOCKET_ID_ANY.

Known Issues

- rte_mempool

  The rte_mempool uses a per-lcore cache inside the mempool. For non-EAL pthreads, `rte_lcore_id()` will not return a valid number. So for now, when rte_mempool is used with non-EAL pthreads, the put/get operations will bypass the mempool cache and there is a performance penalty because of this bypass. Support for non-EAL mempool cache is currently being enabled.

- rte_ring

  rte_ring supports multi-producer enqueue and multi-consumer dequeue. However, it is non-preemptive, this has a knock on effect of making rte_mempool non-preemptable.

  Note: The "non-preemptive" constraint means:

  – a pthread doing multi-producers enqueues on a given ring must not be preempted by another pthread doing a multi-producer enqueue on the same ring.

  – a pthread doing multi-consumers dequeues on a given ring must not be preempted by another pthread doing a multi-consumer dequeue on the same ring.

  Bypassing this constraint it may cause the 2nd pthread to spin until the 1st one is scheduled again. Moreover, if the 1st pthread is preempted by a context that has an higher priority, it may even cause a dead lock.

  This does not mean it cannot be used, simply, there is a need to narrow down the situation when it is used by multi-pthread on the same core.

  1. It CAN be used for any single-producer or single-consumer situation.
  2. It MAY be used by multi-producer/consumer pthread whose scheduling policy are all SCHED_OTHER(cfs). User SHOULD be aware of the performance penalty before using it.
  3. It MUST not be used by multi-producer/consumer pthreads, whose scheduling policies are SCHED_FIFO or SCHED_RR.

  `RTE_RING_PAUSE_REP_COUNT` is defined for rte_ring to reduce contention. It's mainly for case 2, a yield is issued after number of times pause repeat.

It adds a sched_yield() syscall if the thread spins for too long while waiting on the other thread to finish its operations on the ring. This gives the preempted thread a chance to proceed and finish with the ring enqueue/dequeue operation.

- rte_timer

  Running `rte_timer_manager()` on a non-EAL pthread is not allowed. However, re-setting/stopping the timer from a non-EAL pthread is allowed.

- rte_log

  In non-EAL pthreads, there is no per thread loglevel and logtype, global loglevels are used.

- misc

  The debug statistics of rte_ring, rte_mempool and rte_timer are not supported in a non-EAL pthread.

cgroup control

The following is a simple example of cgroup control usage, there are two pthreads(t0 and t1) doing packet I/O on the same core ($CPU). We expect only 50% of CPU spend on packet IO.

```
mkdir /sys/fs/cgroup/cpu/pkt_io
mkdir /sys/fs/cgroup/cpuset/pkt_io

echo $cpu > /sys/fs/cgroup/cpuset/cpuset.cpus

echo $t0 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t0 > /sys/fs/cgroup/cpuset/pkt_io/tasks

echo $t1 > /sys/fs/cgroup/cpu/pkt_io/tasks
echo $t1 > /sys/fs/cgroup/cpuset/pkt_io/tasks

cd /sys/fs/cgroup/cpu/pkt_io
echo 100000 > pkt_io/cpu.cfs_period_us
echo  50000 > pkt_io/cpu.cfs_quota_us
```

4.3.4 Malloc

The EAL provides a malloc API to allocate any-sized memory.

The objective of this API is to provide malloc-like functions to allow allocation from hugepage memory and to facilitate application porting. The *DPDK API Reference* manual describes the available functions.

Typically, these kinds of allocations should not be done in data plane processing because they are slower than pool-based allocation and make use of locks within the allocation and free paths. However, they can be used in configuration code.

Refer to the rte_malloc() function description in the *DPDK API Reference* manual for more information.

Cookies

When CONFIG_RTE_MALLOC_DEBUG is enabled, the allocated memory contains overwrite protection fields to help identify buffer overflows.

Alignment and NUMA Constraints

The rte_malloc() takes an align argument that can be used to request a memory area that is aligned on a multiple of this value (which must be a power of two).

On systems with NUMA support, a call to the rte_malloc() function will return memory that has been allocated on the NUMA socket of the core which made the call. A set of APIs is also provided, to allow memory to be explicitly allocated on a NUMA socket directly, or by allocated on the NUMA socket where another core is located, in the case where the memory is to be used by a logical core other than on the one doing the memory allocation.

Use Cases

This API is meant to be used by an application that requires malloc-like functions at initialization time.

For allocating/freeing data at runtime, in the fast-path of an application, the memory pool library should be used instead.

Internal Implementation

Data Structures

There are two data structure types used internally in the malloc library:

- struct malloc_heap - used to track free space on a per-socket basis
- struct malloc_elem - the basic element of allocation and free-space tracking inside the library.

Structure: malloc_heap The malloc_heap structure is used to manage free space on a per-socket basis. Internally, there is one heap structure per NUMA node, which allows us to allocate memory to a thread based on the NUMA node on which this thread runs. While this does not guarantee that the memory will be used on that NUMA node, it is no worse than a scheme where the memory is always allocated on a fixed or random node.

The key fields of the heap structure and their function are described below (see also diagram above):

- lock - the lock field is needed to synchronize access to the heap. Given that the free space in the heap is tracked using a linked list, we need a lock to prevent two threads manipulating the list at the same time.
- free_head - this points to the first element in the list of free nodes for this malloc heap.

Note: The malloc_heap structure does not keep track of in-use blocks of memory, since these are never touched except when they are to be freed again - at which point the pointer to the block is an input to the free() function.

Fig. 4.3: Example of a malloc heap and malloc elements within the malloc library

Structure: malloc_elem The malloc_elem structure is used as a generic header structure for various blocks of memory. It is used in three different ways - all shown in the diagram above:

1. As a header on a block of free or allocated memory - normal case
2. As a padding header inside a block of memory
3. As an end-of-memseg marker

The most important fields in the structure and how they are used are described below.

Note: If the usage of a particular field in one of the above three usages is not described, the field can be assumed to have an undefined value in that situation, for example, for padding headers only the "state" and "pad" fields have valid values.

- heap - this pointer is a reference back to the heap structure from which this block was allocated. It is used for normal memory blocks when they are being freed, to add the newly-freed block to the heap's free-list.
- prev - this pointer points to the header element/block in the memseg immediately behind the current one. When freeing a block, this pointer is used to reference the previous block to check if that block is also free. If so, then the two free blocks are merged to form a single larger block.
- next_free - this pointer is used to chain the free-list of unallocated memory blocks together. It is only used in normal memory blocks; on `malloc()` to find a suitable free block to allocate and on `free()` to add the newly freed element to the free-list.
- state - This field can have one of three values: `FREE`, `BUSY` or `PAD`. The former two are to indicate the allocation state of a normal memory block and the latter is to indicate that the element structure is a dummy structure at the end of the start-of-block padding, i.e. where the start of the data within a block is not at the start of the block itself, due to alignment constraints. In that case, the pad header is used to locate the actual malloc element header for the block. For the end-of-memseg structure, this is always a `BUSY` value, which ensures that no element, on being freed, searches beyond the end of the memseg for other blocks to merge with into a larger free area.
- pad - this holds the length of the padding present at the start of the block. In the case of a normal block header, it is added to the address of the end of the header to give the address of the start of the data area, i.e. the value passed back to the application on a malloc. Within a dummy header inside the padding, this same value is stored, and is subtracted from the address of the dummy header to yield the address of the actual block header.
- size - the size of the data block, including the header itself. For end-of-memseg structures, this size is given as zero, though it is never actually checked. For normal blocks

which are being freed, this size value is used in place of a "next" pointer to identify the location of the next block of memory that in the case of being FREE, the two free blocks can be merged into one.

Memory Allocation

On EAL initialization, all memsegs are setup as part of the malloc heap. This setup involves placing a dummy structure at the end with BUSY state, which may contain a sentinel value if CONFIG_RTE_MALLOC_DEBUG is enabled, and a proper *element header* with FREE at the start for each memseg. The FREE element is then added to the free_list for the malloc heap.

When an application makes a call to a malloc-like function, the malloc function will first index the lcore_config structure for the calling thread, and determine the NUMA node of that thread. The NUMA node is used to index the array of malloc_heap structures which is passed as a parameter to the heap_alloc() function, along with the requested size, type, alignment and boundary parameters.

The heap_alloc() function will scan the free_list of the heap, and attempt to find a free block suitable for storing data of the requested size, with the requested alignment and boundary constraints.

When a suitable free element has been identified, the pointer to be returned to the user is calculated. The cache-line of memory immediately preceding this pointer is filled with a struct malloc_elem header. Because of alignment and boundary constraints, there could be free space at the start and/or end of the element, resulting in the following behavior:

1. Check for trailing space. If the trailing space is big enough, i.e. > 128 bytes, then the free element is split. If it is not, then we just ignore it (wasted space).
2. Check for space at the start of the element. If the space at the start is small, i.e. <=128 bytes, then a pad header is used, and the remaining space is wasted. If, however, the remaining space is greater, then the free element is split.

The advantage of allocating the memory from the end of the existing element is that no adjustment of the free list needs to take place - the existing element on the free list just has its size pointer adjusted, and the following element has its "prev" pointer redirected to the newly created element.

Freeing Memory

To free an area of memory, the pointer to the start of the data area is passed to the free function. The size of the malloc_elem structure is subtracted from this pointer to get the element header for the block. If this header is of type PAD then the pad length is further subtracted from the pointer to get the proper element header for the entire block.

From this element header, we get pointers to the heap from which the block was allocated and to where it must be freed, as well as the pointer to the previous element, and via the size field, we can calculate the pointer to the next element. These next and previous elements are then checked to see if they are also FREE, and if so, they are merged with the current element. This means that we can never have two FREE memory blocks adjacent to one another, as they are always merged into a single block.

4.4 Ring Library

The ring allows the management of queues. Instead of having a linked list of infinite size, the rte_ring has the following properties:

- FIFO
- Maximum size is fixed, the pointers are stored in a table
- Lockless implementation
- Multi-consumer or single-consumer dequeue
- Multi-producer or single-producer enqueue
- Bulk dequeue - Dequeues the specified count of objects if successful; otherwise fails
- Bulk enqueue - Enqueues the specified count of objects if successful; otherwise fails
- Burst dequeue - Dequeue the maximum available objects if the specified count cannot be fulfilled
- Burst enqueue - Enqueue the maximum available objects if the specified count cannot be fulfilled

The advantages of this data structure over a linked list queue are as follows:

- Faster; only requires a single Compare-And-Swap instruction of sizeof(void *) instead of several double-Compare-And-Swap instructions.
- Simpler than a full lockless queue.
- Adapted to bulk enqueue/dequeue operations. As pointers are stored in a table, a dequeue of several objects will not produce as many cache misses as in a linked queue. Also, a bulk dequeue of many objects does not cost more than a dequeue of a simple object.

The disadvantages:

- Size is fixed
- Having many rings costs more in terms of memory than a linked list queue. An empty ring contains at least N pointers.

A simplified representation of a Ring is shown in with consumer and producer head and tail pointers to objects stored in the data structure.

Fig. 4.4: Ring Structure

4.4.1 References for Ring Implementation in FreeBSD*

The following code was added in FreeBSD 8.0, and is used in some network device drivers (at least in Intel drivers):

- bufring.h in FreeBSD
- bufring.c in FreeBSD

4.4.2 Lockless Ring Buffer in Linux*

The following is a link describing the Linux Lockless Ring Buffer Design.

4.4.3 Additional Features

Name

A ring is identified by a unique name. It is not possible to create two rings with the same name (rte_ring_create() returns NULL if this is attempted).

Water Marking

The ring can have a high water mark (threshold). Once an enqueue operation reaches the high water mark, the producer is notified, if the water mark is configured.

This mechanism can be used, for example, to exert a back pressure on I/O to inform the LAN to PAUSE.

Debug

When debug is enabled (CONFIG_RTE_LIBRTE_RING_DEBUG is set), the library stores some per-ring statistic counters about the number of enqueues/dequeues. These statistics are per-core to avoid concurrent accesses or atomic operations.

4.4.4 Use Cases

Use cases for the Ring library include:

- Communication between applications in the DPDK
- Used by memory pool allocator

4.4.5 Anatomy of a Ring Buffer

This section explains how a ring buffer operates. The ring structure is composed of two head and tail couples; one is used by producers and one is used by the consumers. The figures of the following sections refer to them as prod_head, prod_tail, cons_head and cons_tail.

Each figure represents a simplified state of the ring, which is a circular buffer. The content of the function local variables is represented on the top of the figure, and the content of ring structure is represented on the bottom of the figure.

Single Producer Enqueue

This section explains what occurs when a producer adds an object to the ring. In this example, only the producer head and tail (prod_head and prod_tail) are modified, and there is only one producer.

The initial state is to have a prod_head and prod_tail pointing at the same location.

Enqueue First Step

First, *ring->prod_head* and ring->cons_tail are copied in local variables. The prod_next local variable points to the next element of the table, or several elements after in case of bulk enqueue.

If there is not enough room in the ring (this is detected by checking cons_tail), it returns an error.

Fig. 4.5: Enqueue first step

Enqueue Second Step

The second step is to modify *ring->prod_head* in ring structure to point to the same location as prod_next.

A pointer to the added object is copied in the ring (obj4).

Fig. 4.6: Enqueue second step

Enqueue Last Step

Once the object is added in the ring, ring->prod_tail in the ring structure is modified to point to the same location as *ring->prod_head*. The enqueue operation is finished.

Fig. 4.7: Enqueue last step

Single Consumer Dequeue

This section explains what occurs when a consumer dequeues an object from the ring. In this example, only the consumer head and tail (cons_head and cons_tail) are modified and there is only one consumer.

The initial state is to have a cons_head and cons_tail pointing at the same location.

Dequeue First Step

First, ring->cons_head and ring->prod_tail are copied in local variables. The cons_next local variable points to the next element of the table, or several elements after in the case of bulk dequeue.

If there are not enough objects in the ring (this is detected by checking prod_tail), it returns an error.

Fig. 4.8: Dequeue last step

Dequeue Second Step

The second step is to modify ring->cons_head in the ring structure to point to the same location as cons_next.

The pointer to the dequeued object (obj1) is copied in the pointer given by the user.

Fig. 4.9: Dequeue second step

Dequeue Last Step

Finally, ring->cons_tail in the ring structure is modified to point to the same location as ring->cons_head. The dequeue operation is finished.

Fig. 4.10: Dequeue last step

Multiple Producers Enqueue

This section explains what occurs when two producers concurrently add an object to the ring. In this example, only the producer head and tail (prod_head and prod_tail) are modified.

The initial state is to have a prod_head and prod_tail pointing at the same location.

Multiple Consumer Enqueue First Step

On both cores, *ring->prod_head* and ring->cons_tail are copied in local variables. The prod_next local variable points to the next element of the table, or several elements after in the case of bulk enqueue.

If there is not enough room in the ring (this is detected by checking cons_tail), it returns an error.

Multiple Consumer Enqueue Second Step

The second step is to modify ring->prod_head in the ring structure to point to the same location as prod_next. This operation is done using a Compare And Swap (CAS) instruction, which does the following operations atomically:

- If ring->prod_head is different to local variable prod_head, the CAS operation fails, and the code restarts at first step.
- Otherwise, ring->prod_head is set to local prod_next, the CAS operation is successful, and processing continues.

In the figure, the operation succeeded on core 1, and step one restarted on core 2.

Fig. 4.11: Multiple consumer enqueue first step

Fig. 4.12: Multiple consumer enqueue second step

Multiple Consumer Enqueue Third Step

The CAS operation is retried on core 2 with success.

The core 1 updates one element of the ring(obj4), and the core 2 updates another one (obj5).

Fig. 4.13: Multiple consumer enqueue third step

Multiple Consumer Enqueue Fourth Step

Each core now wants to update ring->prod_tail. A core can only update it if ring->prod_tail is equal to the prod_head local variable. This is only true on core 1. The operation is finished on core 1.

Multiple Consumer Enqueue Last Step

Once ring->prod_tail is updated by core 1, core 2 is allowed to update it too. The operation is also finished on core 2.

Modulo 32-bit Indexes

In the preceding figures, the prod_head, prod_tail, cons_head and cons_tail indexes are represented by arrows. In the actual implementation, these values are not between 0 and size(ring)-1 as would be assumed. The indexes are between 0 and 2^32 -1, and we mask their value when we access the pointer table (the ring itself). 32-bit modulo also implies that operations on indexes (such as, add/subtract) will automatically do 2^32 modulo if the result overflows the 32-bit number range.

The following are two examples that help to explain how indexes are used in a ring.

Note: To simplify the explanation, operations with modulo 16-bit are used instead of modulo 32-bit. In addition, the four indexes are defined as unsigned 16-bit integers, as opposed to unsigned 32-bit integers in the more realistic case.

This ring contains 11000 entries.

This ring contains 12536 entries.

Note: For ease of understanding, we use modulo 65536 operations in the above examples. In real execution cases, this is redundant for low efficiency, but is done automatically when the result overflows.

Fig. 4.14: Multiple consumer enqueue fourth step

Fig. 4.15: Multiple consumer enqueue last step

The code always maintains a distance between producer and consumer between 0 and size(ring)-1. Thanks to this property, we can do subtractions between 2 index values in a modulo-32bit base: that's why the overflow of the indexes is not a problem.

At any time, entries and free_entries are between 0 and size(ring)-1, even if only the first term of subtraction has overflowed:

```
uint32_t entries = (prod_tail - cons_head);
uint32_t free_entries = (mask + cons_tail -prod_head);
```

4.4.6 References

- bufring.h in FreeBSD (version 8)
- bufring.c in FreeBSD (version 8)
- Linux Lockless Ring Buffer Design

4.5 Mempool Library

A memory pool is an allocator of a fixed-sized object. In the DPDK, it is identified by name and uses a ring to store free objects. It provides some other optional services such as a per-core object cache and an alignment helper to ensure that objects are padded to spread them equally on all DRAM or DDR3 channels.

This library is used by the *Mbuf Library* and the *Environment Abstraction Layer* (for logging history).

4.5.1 Cookies

In debug mode (CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG is enabled), cookies are added at the beginning and end of allocated blocks. The allocated objects then contain overwrite protection fields to help debugging buffer overflows.

4.5.2 Stats

In debug mode (CONFIG_RTE_LIBRTE_MEMPOOL_DEBUG is enabled), statistics about get from/put in the pool are stored in the mempool structure. Statistics are per-lcore to avoid concurrent access to statistics counters.

Fig. 4.16: Modulo 32-bit indexes - Example 1

Fig. 4.17: Modulo 32-bit indexes - Example 2

4.5.3 Memory Alignment Constraints

Depending on hardware memory configuration, performance can be greatly improved by adding a specific padding between objects. The objective is to ensure that the beginning of each object starts on a different channel and rank in memory so that all channels are equally loaded.

This is particularly true for packet buffers when doing L3 forwarding or flow classification. Only the first 64 bytes are accessed, so performance can be increased by spreading the start addresses of objects among the different channels.

The number of ranks on any DIMM is the number of independent sets of DRAMs that can be accessed for the full data bit-width of the DIMM. The ranks cannot be accessed simultaneously since they share the same data path. The physical layout of the DRAM chips on the DIMM itself does not necessarily relate to the number of ranks.

When running an application, the EAL command line options provide the ability to add the number of memory channels and ranks.

Note: The command line must always have the number of memory channels specified for the processor.

Examples of alignment for different DIMM architectures are shown in Fig. 4.18 and Fig. 4.19.

Fig. 4.18: Two Channels and Quad-ranked DIMM Example

In this case, the assumption is that a packet is 16 blocks of 64 bytes, which is not true.

The Intel® 5520 chipset has three channels, so in most cases, no padding is required between objects (except for objects whose size are n x 3 x 64 bytes blocks).

Fig. 4.19: Three Channels and Two Dual-ranked DIMM Example

When creating a new pool, the user can specify to use this feature or not.

4.5.4 Local Cache

In terms of CPU usage, the cost of multiple cores accessing a memory pool's ring of free buffers may be high since each access requires a compare-and-set (CAS) operation. To avoid having too many access requests to the memory pool's ring, the memory pool allocator can maintain a per-core cache and do bulk requests to the memory pool's ring, via the cache with many fewer locks on the actual memory pool structure. In this way, each core has full access to its own cache (with locks) of free objects and only when the cache fills does the core need to shuffle some of the free objects back to the pools ring or obtain more objects when the cache is empty.

While this may mean a number of buffers may sit idle on some core's cache, the speed at which a core can access its own cache for a specific memory pool without locks provides performance gains.

The cache is composed of a small, per-core table of pointers and its length (used as a stack). This cache can be enabled or disabled at creation of the pool.

The maximum size of the cache is static and is defined at compilation time (CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE).

Fig. 4.20 shows a cache in operation.

Fig. 4.20: A mempool in Memory with its Associated Ring

4.5.5 Use Cases

All allocations that require a high level of performance should use a pool-based memory allocator. Below are some examples:

- *Mbuf Library*
- *Environment Abstraction Layer* , for logging service
- Any application that needs to allocate fixed-sized objects in the data plane and that will be continuously utilized by the system.

4.6 Mbuf Library

The mbuf library provides the ability to allocate and free buffers (mbufs) that may be used by the DPDK application to store message buffers. The message buffers are stored in a mempool, using the *Mempool Library*.

A rte_mbuf struct can carry network packet buffers or generic control buffers (indicated by the CTRL_MBUF_FLAG). This can be extended to other types. The rte_mbuf header structure is kept as small as possible and currently uses just two cache lines, with the most frequently used fields being on the first of the two cache lines.

4.6.1 Design of Packet Buffers

For the storage of the packet data (including protocol headers), two approaches were considered:

1. Embed metadata within a single memory buffer the structure followed by a fixed size area for the packet data.
2. Use separate memory buffers for the metadata structure and for the packet data.

The advantage of the first method is that it only needs one operation to allocate/free the whole memory representation of a packet. On the other hand, the second method is more flexible and allows the complete separation of the allocation of metadata structures from the allocation of packet data buffers.

The first method was chosen for the DPDK. The metadata contains control information such as message type, length, offset to the start of the data and a pointer for additional mbuf structures allowing buffer chaining.

Message buffers that are used to carry network packets can handle buffer chaining where multiple buffers are required to hold the complete packet. This is the case for jumbo frames that are composed of many mbufs linked together through their next field.

For a newly allocated mbuf, the area at which the data begins in the message buffer is RTE_PKTMBUF_HEADROOM bytes after the beginning of the buffer, which is cache aligned. Message buffers may be used to carry control information, packets, events, and so on between different entities in the system. Message buffers may also use their buffer pointers to point to other message buffer data sections or other structures.

Fig. 4.21 and Fig. 4.22 show some of these scenarios.

Fig. 4.21: An mbuf with One Segment

Fig. 4.22: An mbuf with Three Segments

The Buffer Manager implements a fairly standard set of buffer access functions to manipulate network packets.

4.6.2 Buffers Stored in Memory Pools

The Buffer Manager uses the *Mempool Library* to allocate buffers. Therefore, it ensures that the packet header is interleaved optimally across the channels and ranks for L3 processing. An mbuf contains a field indicating the pool that it originated from. When calling rte_ctrlmbuf_free(m) or rte_pktmbuf_free(m), the mbuf returns to its original pool.

4.6.3 Constructors

Packet and control mbuf constructors are provided by the API. The rte_pktmbuf_init() and rte_ctrlmbuf_init() functions initialize some fields in the mbuf structure that are not modified by the user once created (mbuf type, origin pool, buffer start address, and so on). This function is given as a callback function to the rte_mempool_create() function at pool creation time.

4.6.4 Allocating and Freeing mbufs

Allocating a new mbuf requires the user to specify the mempool from which the mbuf should be taken. For any newly-allocated mbuf, it contains one segment, with a length of 0. The offset to data is initialized to have some bytes of headroom in the buffer (RTE_PKTMBUF_HEADROOM).

Freeing a mbuf means returning it into its original mempool. The content of an mbuf is not modified when it is stored in a pool (as a free mbuf). Fields initialized by the constructor do not need to be re-initialized at mbuf allocation.

When freeing a packet mbuf that contains several segments, all of them are freed and returned to their original mempool.

4.6.5 Manipulating mbufs

This library provides some functions for manipulating the data in a packet mbuf. For instance:

- Get data length
- Get a pointer to the start of data
- Prepend data before data
- Append data after data
- Remove data at the beginning of the buffer (rte_pktmbuf_adj())
- Remove data at the end of the buffer (rte_pktmbuf_trim()) Refer to the *DPDK API Reference* for details.

4.6.6 Meta Information

Some information is retrieved by the network driver and stored in an mbuf to make processing easier. For instance, the VLAN, the RSS hash result (see *Poll Mode Driver*) and a flag indicating that the checksum was computed by hardware.

An mbuf also contains the input port (where it comes from), and the number of segment mbufs in the chain.

For chained buffers, only the first mbuf of the chain stores this meta information.

For instance, this is the case on RX side for the IEEE1588 packet timestamp mechanism, the VLAN tagging and the IP checksum computation.

On TX side, it is also possible for an application to delegate some processing to the hardware if it supports it. For instance, the PKT_TX_IP_CKSUM flag allows to offload the computation of the IPv4 checksum.

The following examples explain how to configure different TX offloads on a vxlan-encapsulated tcp packet: `out_eth/out_ip/out_udp/vxlan/in_eth/in_ip/in_tcp/payload`

- calculate checksum of out_ip:

  ```
  mb->l2_len = len(out_eth)
  mb->l3_len = len(out_ip)
  mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
  set out_ip checksum to 0 in the packet
  ```

  This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM.

- calculate checksum of out_ip and out_udp:

  ```
   mb->l2_len = len(out_eth)
   mb->l3_len = len(out_ip)
   mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_UDP_CKSUM
   set out_ip checksum to 0 in the packet
   set out_udp checksum to pseudo header using rte_ipv4_phdr_cksum()

  This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM
  and DEV_TX_OFFLOAD_UDP_CKSUM.
  ```

- calculate checksum of in_ip:

  ```
  mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
  mb->l3_len = len(in_ip)
  ```

```
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM
set in_ip checksum to 0 in the packet
```

This is similar to case 1), but l2_len is different. It is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM. Note that it can only work if outer L4 checksum is 0.

- calculate checksum of in_ip and in_tcp:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CSUM | PKT_TX_TCP_CKSUM
set in_ip checksum to 0 in the packet
set in_tcp checksum to pseudo header using rte_ipv4_phdr_cksum()
```

This is similar to case 2), but l2_len is different. It is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM and DEV_TX_OFFLOAD_TCP_CKSUM. Note that it can only work if outer L4 checksum is 0.

- segment inner TCP:

```
mb->l2_len = len(out_eth + out_ip + out_udp + vxlan + in_eth)
mb->l3_len = len(in_ip)
mb->l4_len = len(in_tcp)
mb->ol_flags |= PKT_TX_IPV4 | PKT_TX_IP_CKSUM | PKT_TX_TCP_CKSUM |
  PKT_TX_TCP_SEG;
set in_ip checksum to 0 in the packet
set in_tcp checksum to pseudo header without including the IP
  payload length using rte_ipv4_phdr_cksum()
```

This is supported on hardware advertising DEV_TX_OFFLOAD_TCP_TSO. Note that it can only work if outer L4 checksum is 0.

- calculate checksum of out_ip, in_ip, in_tcp:

```
 mb->outer_l2_len = len(out_eth)
 mb->outer_l3_len = len(out_ip)
 mb->l2_len = len(out_udp + vxlan + in_eth)
 mb->l3_len = len(in_ip)
 mb->ol_flags |= PKT_TX_OUTER_IPV4 | PKT_TX_OUTER_IP_CKSUM  | \
   PKT_TX_IP_CKSUM |  PKT_TX_TCP_CKSUM;
 set out_ip checksum to 0 in the packet
 set in_ip checksum to 0 in the packet
 set in_tcp checksum to pseudo header using rte_ipv4_phdr_cksum()

This is supported on hardware advertising DEV_TX_OFFLOAD_IPV4_CKSUM,
DEV_TX_OFFLOAD_UDP_CKSUM and DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM.
```

The list of flags and their precise meaning is described in the mbuf API documentation (rte_mbuf.h). Also refer to the testpmd source code (specifically the csumonly.c file) for details.

4.6.7 Direct and Indirect Buffers

A direct buffer is a buffer that is completely separate and self-contained. An indirect buffer behaves like a direct buffer but for the fact that the buffer pointer and data offset in it refer to data in another direct buffer. This is useful in situations where packets need to be duplicated or fragmented, since indirect buffers provide the means to reuse the same packet data across multiple buffers.

A buffer becomes indirect when it is “attached” to a direct buffer using the rte_pktmbuf_attach() function. Each buffer has a reference counter field and whenever an indirect buffer is attached to the direct buffer, the reference counter on the direct buffer is incremented. Similarly, whenever the indirect buffer is detached, the reference counter on the direct buffer is decremented. If the resulting reference counter is equal to 0, the direct buffer is freed since it is no longer in use.

There are a few things to remember when dealing with indirect buffers. First of all, it is not possible to attach an indirect buffer to another indirect buffer. Secondly, for a buffer to become indirect, its reference counter must be equal to 1, that is, it must not be already referenced by another indirect buffer. Finally, it is not possible to reattach an indirect buffer to the direct buffer (unless it is detached first).

While the attach/detach operations can be invoked directly using the recommended rte_pktmbuf_attach() and rte_pktmbuf_detach() functions, it is suggested to use the higher-level rte_pktmbuf_clone() function, which takes care of the correct initialization of an indirect buffer and can clone buffers with multiple segments.

Since indirect buffers are not supposed to actually hold any data, the memory pool for indirect buffers should be configured to indicate the reduced memory consumption. Examples of the initialization of a memory pool for indirect buffers (as well as use case examples for indirect buffers) can be found in several of the sample applications, for example, the IPv4 Multicast sample application.

4.6.8 Debug

In debug mode (CONFIG_RTE_MBUF_DEBUG is enabled), the functions of the mbuf library perform sanity checks before any operation (such as, buffer corruption, bad type, and so on).

4.6.9 Use Cases

All networking application should use mbufs to transport network packets.

4.7 Poll Mode Driver

The DPDK includes 1 Gigabit, 10 Gigabit and 40 Gigabit and para virtualized virtio Poll Mode Drivers.

A Poll Mode Driver (PMD) consists of APIs, provided through the BSD driver running in user space, to configure the devices and their respective queues. In addition, a PMD accesses the RX and TX descriptors directly without any interrupts (with the exception of Link Status Change interrupts) to quickly receive, process and deliver packets in the user’s application. This section describes the requirements of the PMDs, their global design principles and proposes a high-level architecture and a generic external API for the Ethernet PMDs.

4.7.1 Requirements and Assumptions

The DPDK environment for packet processing applications allows for two models, run-to-completion and pipe-line:

- In the *run-to-completion* model, a specific port's RX descriptor ring is polled for packets through an API. Packets are then processed on the same core and placed on a port's TX descriptor ring through an API for transmission.
- In the *pipe-line* model, one core polls one or more port's RX descriptor ring through an API. Packets are received and passed to another core via a ring. The other core continues to process the packet which then may be placed on a port's TX descriptor ring through an API for transmission.

In a synchronous run-to-completion model, each logical core assigned to the DPDK executes a packet processing loop that includes the following steps:

- Retrieve input packets through the PMD receive API
- Process each received packet one at a time, up to its forwarding
- Send pending output packets through the PMD transmit API

Conversely, in an asynchronous pipe-line model, some logical cores may be dedicated to the retrieval of received packets and other logical cores to the processing of previously received packets. Received packets are exchanged between logical cores through rings. The loop for packet retrieval includes the following steps:

- Retrieve input packets through the PMD receive API
- Provide received packets to processing lcores through packet queues

The loop for packet processing includes the following steps:

- Retrieve the received packet from the packet queue
- Process the received packet, up to its retransmission if forwarded

To avoid any unnecessary interrupt processing overhead, the execution environment must not use any asynchronous notification mechanisms. Whenever needed and appropriate, asynchronous communication should be introduced as much as possible through the use of rings.

Avoiding lock contention is a key issue in a multi-core environment. To address this issue, PMDs are designed to work with per-core private resources as much as possible. For example, a PMD maintains a separate transmit queue per-core, per-port. In the same way, every receive queue of a port is assigned to and polled by a single logical core (lcore).

To comply with Non-Uniform Memory Access (NUMA), memory management is designed to assign to each logical core a private buffer pool in local memory to minimize remote memory access. The configuration of packet buffer pools should take into account the underlying physical memory architecture in terms of DIMMS, channels and ranks. The application must ensure that appropriate parameters are given at memory pool creation time. See *Mempool Library*.

4.7.2 Design Principles

The API and architecture of the Ethernet* PMDs are designed with the following guidelines in mind.

PMDs must help global policy-oriented decisions to be enforced at the upper application level. Conversely, NIC PMD functions should not impede the benefits expected by upper-level global policies, or worse prevent such policies from being applied.

For instance, both the receive and transmit functions of a PMD have a maximum number of packets/descriptors to poll. This allows a run-to-completion processing stack to statically fix or to dynamically adapt its overall behavior through different global loop policies, such as:

- Receive, process immediately and transmit packets one at a time in a piecemeal fashion.
- Receive as many packets as possible, then process all received packets, transmitting them immediately.
- Receive a given maximum number of packets, process the received packets, accumulate them and finally send all accumulated packets to transmit.

To achieve optimal performance, overall software design choices and pure software optimization techniques must be considered and balanced against available low-level hardware-based optimization features (CPU cache properties, bus speed, NIC PCI bandwidth, and so on). The case of packet transmission is an example of this software/hardware tradeoff issue when optimizing burst-oriented network packet processing engines. In the initial case, the PMD could export only an rte_eth_tx_one function to transmit one packet at a time on a given queue. On top of that, one can easily build an rte_eth_tx_burst function that loops invoking the rte_eth_tx_one function to transmit several packets at a time. However, an rte_eth_tx_burst function is effectively implemented by the PMD to minimize the driver-level transmit cost per packet through the following optimizations:

- Share among multiple packets the un-amortized cost of invoking the rte_eth_tx_one function.
- Enable the rte_eth_tx_burst function to take advantage of burst-oriented hardware features (prefetch data in cache, use of NIC head/tail registers) to minimize the number of CPU cycles per packet, for example by avoiding unnecessary read memory accesses to ring transmit descriptors, or by systematically using arrays of pointers that exactly fit cache line boundaries and sizes.
- Apply burst-oriented software optimization techniques to remove operations that would otherwise be unavoidable, such as ring index wrap back management.

Burst-oriented functions are also introduced via the API for services that are intensively used by the PMD. This applies in particular to buffer allocators used to populate NIC rings, which provide functions to allocate/free several buffers at a time. For example, an mbuf_multiple_alloc function returning an array of pointers to rte_mbuf buffers which speeds up the receive poll function of the PMD when replenishing multiple descriptors of the receive ring.

4.7.3 Logical Cores, Memory and NIC Queues Relationships

The DPDK supports NUMA allowing for better performance when a processor's logical cores and interfaces utilize its local memory. Therefore, mbuf allocation associated with local PCIe* interfaces should be allocated from memory pools created in the local memory. The buffers should, if possible, remain on the local processor to obtain the best performance results and RX and TX buffer descriptors should be populated with mbufs allocated from a mempool allocated from local memory.

The run-to-completion model also performs better if packet or data manipulation is in local memory instead of a remote processors memory. This is also true for the pipe-line model provided all logical cores used are located on the same processor.

Multiple logical cores should never share receive or transmit queues for interfaces since this would require global locks and hinder performance.

4.7.4 Device Identification and Configuration

Device Identification

Each NIC port is uniquely designated by its (bus/bridge, device, function) PCI identifiers assigned by the PCI probing/enumeration function executed at DPDK initialization. Based on their PCI identifier, NIC ports are assigned two other identifiers:

- A port index used to designate the NIC port in all functions exported by the PMD API.
- A port name used to designate the port in console messages, for administration or debugging purposes. For ease of use, the port name includes the port index.

Device Configuration

The configuration of each NIC port includes the following operations:

- Allocate PCI resources
- Reset the hardware (issue a Global Reset) to a well-known default state
- Set up the PHY and the link
- Initialize statistics counters

The PMD API must also export functions to start/stop the all-multicast feature of a port and functions to set/unset the port in promiscuous mode.

Some hardware offload features must be individually configured at port initialization through specific configuration parameters. This is the case for the Receive Side Scaling (RSS) and Data Center Bridging (DCB) features for example.

On-the-Fly Configuration

All device features that can be started or stopped “on the fly” (that is, without stopping the device) do not require the PMD API to export dedicated functions for this purpose.

All that is required is the mapping address of the device PCI registers to implement the configuration of these features in specific functions outside of the drivers.

For this purpose, the PMD API exports a function that provides all the information associated with a device that can be used to set up a given device feature outside of the driver. This includes the PCI vendor identifier, the PCI device identifier, the mapping address of the PCI device registers, and the name of the driver.

The main advantage of this approach is that it gives complete freedom on the choice of the API used to configure, to start, and to stop such features.

As an example, refer to the configuration of the IEEE1588 feature for the Intel® 82576 Gigabit Ethernet Controller and the Intel® 82599 10 Gigabit Ethernet Controller controllers in the testpmd application.

Other features such as the L3/L4 5-Tuple packet filtering feature of a port can be configured in the same way. Ethernet* flow control (pause frame) can be configured on the individual port. Refer to the testpmd source code for details. Also, L4 (UDP/TCP/ SCTP) checksum offload by the NIC can be enabled for an individual packet as long as the packet mbuf is set up correctly. See *Hardware Offload* for details.

Configuration of Transmit and Receive Queues

Each transmit queue is independently configured with the following information:

- The number of descriptors of the transmit ring
- The socket identifier used to identify the appropriate DMA memory zone from which to allocate the transmit ring in NUMA architectures
- The values of the Prefetch, Host and Write-Back threshold registers of the transmit queue
- The *minimum* transmit packets to free threshold (tx_free_thresh). When the number of descriptors used to transmit packets exceeds this threshold, the network adaptor should be checked to see if it has written back descriptors. A value of 0 can be passed during the TX queue configuration to indicate the default value should be used. The default value for tx_free_thresh is 32. This ensures that the PMD does not search for completed descriptors until at least 32 have been processed by the NIC for this queue.
- The *minimum* RS bit threshold. The minimum number of transmit descriptors to use before setting the Report Status (RS) bit in the transmit descriptor. Note that this parameter may only be valid for Intel 10 GbE network adapters. The RS bit is set on the last descriptor used to transmit a packet if the number of descriptors used since the last RS bit setting, up to the first descriptor used to transmit the packet, exceeds the transmit RS bit threshold (tx_rs_thresh). In short, this parameter controls which transmit descriptors are written back to host memory by the network adapter. A value of 0 can be passed during the TX queue configuration to indicate that the default value should be used. The default value for tx_rs_thresh is 32. This ensures that at least 32 descriptors are used before the network adapter writes back the most recently used descriptor. This saves upstream PCIe* bandwidth resulting from TX descriptor write-backs. It is important to note that the TX Write-back threshold (TX wthresh) should be set to 0 when tx_rs_thresh is greater than 1. Refer to the Intel® 82599 10 Gigabit Ethernet Controller Datasheet for more details.

The following constraints must be satisfied for tx_free_thresh and tx_rs_thresh:

- tx_rs_thresh must be greater than 0.
- tx_rs_thresh must be less than the size of the ring minus 2.
- tx_rs_thresh must be less than or equal to tx_free_thresh.
- tx_free_thresh must be greater than 0.
- tx_free_thresh must be less than the size of the ring minus 3.
- For optimal performance, TX wthresh should be set to 0 when tx_rs_thresh is greater than 1.

One descriptor in the TX ring is used as a sentinel to avoid a hardware race condition, hence the maximum threshold constraints.

Note: When configuring for DCB operation, at port initialization, both the number of transmit queues and the number of receive queues must be set to 128.

Hardware Offload

Depending on driver capabilities advertised by `rte_eth_dev_info_get()`, the PMD may support hardware offloading feature like checksumming, TCP segmentation or VLAN insertion.

The support of these offload features implies the addition of dedicated status bit(s) and value field(s) into the rte_mbuf data structure, along with their appropriate handling by the receive/transmit functions exported by each PMD. The list of flags and their precise meaning is described in the mbuf API documentation and in the in *Mbuf Library*, section "Meta Information".

4.7.5 Poll Mode Driver API

Generalities

By default, all functions exported by a PMD are lock-free functions that are assumed not to be invoked in parallel on different logical cores to work on the same target object. For instance, a PMD receive function cannot be invoked in parallel on two logical cores to poll the same RX queue of the same port. Of course, this function can be invoked in parallel by different logical cores on different RX queues. It is the responsibility of the upper-level application to enforce this rule.

If needed, parallel accesses by multiple logical cores to shared queues can be explicitly protected by dedicated inline lock-aware functions built on top of their corresponding lock-free functions of the PMD API.

Generic Packet Representation

A packet is represented by an rte_mbuf structure, which is a generic metadata structure containing all necessary housekeeping information. This includes fields and status bits corresponding to offload hardware features, such as checksum computation of IP headers or VLAN tags.

The rte_mbuf data structure includes specific fields to represent, in a generic way, the offload features provided by network controllers. For an input packet, most fields of the rte_mbuf structure are filled in by the PMD receive function with the information contained in the receive descriptor. Conversely, for output packets, most fields of rte_mbuf structures are used by the PMD transmit function to initialize transmit descriptors.

The mbuf structure is fully described in the *Mbuf Library* chapter.

Ethernet Device API

The Ethernet device API exported by the Ethernet PMDs is described in the *DPDK API Reference*.

Extended Statistics API

The extended statistics API allows each individual PMD to expose a unique set of statistics. The client of the API provides an array of `struct rte_eth_xstats` type. Each `struct` `rte_eth_xstats` contains a string and value pair. The amount of xstats exposed, and position of the statistic in the array must remain constant during runtime.

A naming scheme exists for the strings exposed to clients of the API. This is to allow scraping of the API for statistics of interest. The naming scheme uses strings split by a single underscore _. The scheme is as follows:

- direction
- detail 1
- detail 2
- detail n
- unit

Examples of common statistics xstats strings, formatted to comply to the scheme proposed above:

- `rx_bytes`
- `rx_crc_errors`
- `tx_multicast_packets`

The scheme, although quite simple, allows flexibility in presenting and reading information from the statistic strings. The following example illustrates the naming scheme:`rx_packets`. In this example, the string is split into two components. The first component `rx` indicates that the statistic is associated with the receive side of the NIC. The second component `packets` indicates that the unit of measure is packets.

A more complicated example: `tx_size_128_to_255_packets`. In this example, `tx` indicates transmission, `size` is the first detail, `128` etc are more details, and `packets` indicates that this is a packet counter.

Some additions in the metadata scheme are as follows:

- If the first part does not match `rx` or `tx`, the statistic does not have an affinity with either receive of transmit.
- If the first letter of the second part is `q` and this `q` is followed by a number, this statistic is part of a specific queue.

An example where queue numbers are used is as follows: `tx_q7_bytes` which indicates this statistic applies to queue number 7, and represents the number of transmitted bytes on that queue.

4.8 IVSHMEM Library

The DPDK IVSHMEM library facilitates fast zero-copy data sharing among virtual machines (host-to-guest or guest-to-guest) by means of QEMU's IVSHMEM mechanism.

The library works by providing a command line for QEMU to map several hugepages into a single IVSHMEM device. For the guest to know what is inside any given IVSHMEM device (and to distinguish between DPDK and non-DPDK IVSHMEM devices), a metadata file is also mapped into the IVSHMEM segment. No work needs to be done by the guest application to map IVSHMEM devices into memory; they are automatically recognized by the DPDK Environment Abstraction Layer (EAL).

A typical DPDK IVSHMEM use case looks like the following.

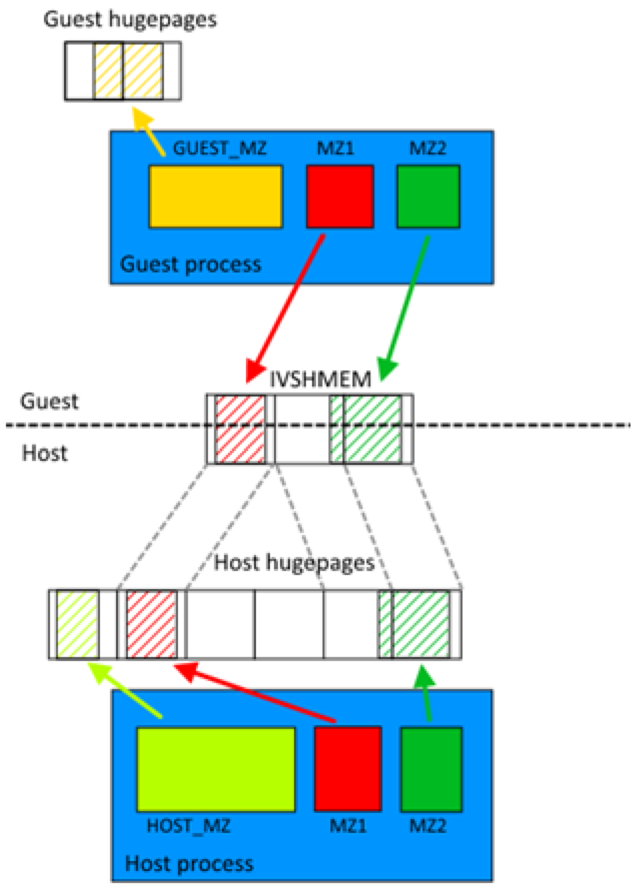


Fig. 4.23: Typical Ivshmem use case

The same could work with several virtual machines, providing host-to-VM or VM-to-VM communication. The maximum number of metadata files is 32 (by default) and each metadata file can contain different (or even the same) hugepages. The only constraint is that each VM has to have access to the memory it is sharing with other entities (be it host or another VM). For example, if the user wants to share the same memzone across two VMs, each VM must have that memzone in its metadata file.

4.8.1 IVHSHMEM Library API Overview

The following is a simple guide to using the IVSHMEM Library API:

- Call rte_ivshmem_metadata_create() to create a new metadata file. The metadata name is used to distinguish between multiple metadata files.
- Populate each metadata file with DPDK data structures. This can be done using the following API calls:
  - rte_ivhshmem_metadata_add_memzone() to add rte_memzone to metadata file
  - rte_ivshmem_metadata_add_ring() to add rte_ring to metadata file

  - rte_ivshmem_metadata_add_mempool() to add rte_mempool to metadata file
- Finally, call rte_ivshmem_metadata_cmdline_generate() to generate the command line for QEMU. Multiple metadata files (and thus multiple command lines) can be supplied to a single VM.

Note: Only data structures fully residing in DPDK hugepage memory work correctly. Supported data structures created by malloc(), mmap() or otherwise using non-DPDK memory cause undefined behavior and even a segmentation fault.

4.8.2 IVSHMEM Environment Configuration

The steps needed to successfully run IVSHMEM applications are the following:

- Compile a special version of QEMU from sources.

  The source code can be found on the QEMU website (currently, version 1.4.x is supported, but version 1.5.x is known to work also), however, the source code will need to be patched to support using regular files as the IVSHMEM memory backend. The patch is not included in the DPDK package, but is available on the Intel®DPDK-vswitch project webpage (either separately or in a DPDK vSwitch package).

- Enable IVSHMEM library in the DPDK build configuration.

  In the default configuration, IVSHMEM library is not compiled. To compile the IVSHMEM library, one has to either use one of the provided IVSHMEM targets (for example, x86_64-ivshmem-linuxapp-gcc), or set CONFIG_RTE_LIBRTE_IVSHMEM to "y" in the build configuration.

- Set up hugepage memory on the virtual machine.

  The guest applications run as regular DPDK (primary) processes and thus need their own hugepage memory set up inside the VM. The process is identical to the one described in the *DPDK Getting Started Guide*.

4.8.3 Best Practices for Writing IVSHMEM Applications

When considering the use of IVSHMEM for sharing memory, security implications need to be carefully evaluated. IVSHMEM is not suitable for untrusted guests, as IVSHMEM is essentially a window into the host process memory. This also has implications for the multiple VM scenarios. While the IVSHMEM library tries to share as little memory as possible, it is quite probable that data designated for one VM might also be present in an IVSMHMEM device designated for another VM. Consequently, any shared memory corruption will affect both host and all VMs sharing that particular memory.

IVSHMEM applications essentially behave like multi-process applications, so it is important to implement access serialization to data and thread safety. DPDK ring structures are already thread-safe, however, any custom data structures that the user might need would have to be thread-safe also.

Similar to regular DPDK multi-process applications, it is not recommended to use function pointers as functions might have different memory addresses in different processes.

It is best to avoid freeing the rte_mbuf structure on a different machine from where it was allocated, that is, if the mbuf was allocated on the host, the host should free it. Consequently, any packet transmission and reception should also happen on the same machine (whether virtual or physical). Failing to do so may lead to data corruption in the mempool cache.

Despite the IVSHMEM mechanism being zero-copy and having good performance, it is still desirable to do processing in batches and follow other procedures described in *Performance Optimization*.

4.8.4 Best Practices for Running IVSHMEM Applications

For performance reasons, it is best to pin host processes and QEMU processes to different cores so that they do not interfere with each other. If NUMA support is enabled, it is also desirable to keep host process' hugepage memory and QEMU process on the same NUMA node.

For the best performance across all NUMA nodes, each QEMU core should be pinned to host CPU core on the appropriate NUMA node. QEMU's virtual NUMA nodes should also be set up to correspond to physical NUMA nodes. More on how to set up DPDK and QEMU NUMA support can be found in *DPDK Getting Started Guide* and QEMU documentation respectively. A script called cpu_layout.py is provided with the DPDK package (in the tools directory) that can be used to identify which CPU cores correspond to which NUMA node.

The QEMU IVSHMEM command line creation should be considered the last step before starting the virtual machine. Currently, there is no hot plug support for QEMU IVSHMEM devices, so one cannot add additional memory to an IVSHMEM device once it has been created. Therefore, the correct sequence to run an IVSHMEM application is to run host application first, obtain the command lines for each IVSHMEM device and then run all QEMU instances with guest applications afterwards.

It is important to note that once QEMU is started, it holds on to the hugepages it uses for IVSHMEM devices. As a result, if the user wishes to shut down or restart the IVSHMEM host application, it is not enough to simply shut the application down. The virtual machine must also be shut down (if not, it will hold onto outdated host data).

4.9 Link Bonding Poll Mode Driver Library

In addition to Poll Mode Drivers (PMDs) for physical and virtual hardware, DPDK also includes a pure-software library that allows physical PMD's to be bonded together to create a single logical PMD.

Fig. 4.24: Bonded PMDs

The Link Bonding PMD library(librte_pmd_bond) supports bonding of groups of `rte_eth_dev` ports of the same speed and duplex to provide similar the capabilities to that found in Linux bonding driver to allow the aggregation of multiple (slave) NICs into a single logical interface between a server and a switch. The new bonded PMD will then process these interfaces based on the mode of operation specified to provide support for features such as redundant links, fault tolerance and/or load balancing.

The librte_pmd_bond library exports a C API which provides an API for the creation of bonded devices as well as the configuration and management of the bonded device and its slave devices.

Note: The Link Bonding PMD Library is enabled by default in the build configuration files, the library can be disabled by setting `CONFIG_RTE_LIBRTE_PMD_BOND=n` and recompiling the DPDK.

4.9.1 Link Bonding Modes Overview

Currently the Link Bonding PMD library supports 4 modes of operation:

- **Round-Robin (Mode 0):**

  Fig. 4.25: Round-Robin (Mode 0)

  This mode provides load balancing and fault tolerance by transmission of packets in sequential order from the first available slave device through the last. Packets are bulk dequeued from devices then serviced in a round-robin manner. This mode does not guarantee in order reception of packets and down stream should be able to handle out of order packets.

- **Active Backup (Mode 1):**

  Fig. 4.26: Active Backup (Mode 1)

  In this mode only one slave in the bond is active at any time, a different slave becomes active if, and only if, the primary active slave fails, thereby providing fault tolerance to slave failure. The single logical bonded interface's MAC address is externally visible on only one NIC (port) to avoid confusing the network switch.

- **Balance XOR (Mode 2):**

  Fig. 4.27: Balance XOR (Mode 2)

  This mode provides transmit load balancing (based on the selected transmission policy) and fault tolerance. The default policy (layer2) uses a simple calculation based on the packet flow source and destination MAC addresses as well as the number of active slaves available to the bonded device to classify the packet to a specific slave to transmit on. Alternate transmission policies supported are layer 2+3, this takes the IP source and destination addresses into the calculation of the transmit slave port and the final supported policy is layer 3+4, this uses IP source and destination addresses as well as the TCP/UDP source and destination port.

Note: The coloring differences of the packets are used to identify different flow classification calculated by the selected transmit policy

---

- **Broadcast (Mode 3):**
- **Link Aggregation 802.3AD (Mode 4):**

Fig. 4.28: Broadcast (Mode 3)

This mode provides fault tolerance by transmission of packets on all slave ports.

Fig. 4.29: Link Aggregation 802.3AD (Mode 4)

This mode provides dynamic link aggregation according to the 802.3ad specification. It negotiates and monitors aggregation groups that share the same speed and duplex settings using the selected balance transmit policy for balancing outgoing traffic.

DPDK implementation of this mode provide some additional requirements of the application.

1. It needs to call `rte_eth_tx_burst` and `rte_eth_rx_burst` with intervals period of less than 100ms.
2. Calls to `rte_eth_tx_burst` must have a buffer size of at least 2xN, where N is the number of slaves. This is a space required for LACP frames. Additionally LACP packets are included in the statistics, but they are not returned to the application.

- **Transmit Load Balancing (Mode 5):**

Fig. 4.30: Transmit Load Balancing (Mode 5)

This mode provides an adaptive transmit load balancing. It dynamically changes the transmitting slave, according to the computed load. Statistics are collected in 100ms intervals and scheduled every 10ms.

4.9.2 Implementation Details

The librte_pmd_bond bonded device are compatible with the Ethernet device API exported by the Ethernet PMDs described in the *DPDK API Reference*.

The Link Bonding Library supports the creation of bonded devices at application startup time during EAL initialization using the `--vdev` option as well as programmatically via the C API `rte_eth_bond_create` function.

Bonded devices support the dynamical addition and removal of slave devices using the `rte_eth_bond_slave_add` / `rte_eth_bond_slave_remove` APIs.

After a slave device is added to a bonded device slave is stopped using `rte_eth_dev_stop` and then reconfigured using `rte_eth_dev_configure` the RX and TX queues are also reconfigured using `rte_eth_tx_queue_setup` / `rte_eth_rx_queue_setup` with the parameters use to configure the bonding device. If RSS is enabled for bonding device, this mode is also enabled on new slave and configured as well.

Setting up multi-queue mode for bonding device to RSS, makes it fully RSS-capable, so all slaves are synchronized with its configuration. This mode is intended to provide RSS configuration on slaves transparent for client application implementation.

Bonding device stores its own version of RSS settings i.e. RETA, RSS hash function and RSS key, used to set up its slaves. That let to define the meaning of RSS configuration of bonding device as desired configuration of whole bonding (as one unit), without pointing any of slave inside. It is required to ensure consistency and made it more error-proof.

RSS hash function set for bonding device, is a maximal set of RSS hash functions supported by all bonded slaves. RETA size is a GCD of all its RETA's sizes, so it can be easily used as a pattern providing expected behavior, even if slave RETAs' sizes are different. If RSS Key is not set for bonded device, it's not changed on the slaves and default key for device is used.

All settings are managed through the bonding port API and always are propagated in one direction (from bonding to slaves).

Link Status Change Interrupts / Polling

Link bonding devices support the registration of a link status change callback, using the `rte_eth_dev_callback_register` API, this will be called when the status of the bonding device changes. For example in the case of a bonding device which has 3 slaves, the link status will change to up when one slave becomes active or change to down when all slaves become inactive. There is no callback notification when a single slave changes state and the previous conditions are not met. If a user wishes to monitor individual slaves then they must register callbacks with that slave directly.

The link bonding library also supports devices which do not implement link status change interrupts, this is achieved by polling the devices link status at a defined period which is set using the `rte_eth_bond_link_monitoring_set` API, the default polling interval is 10ms. When a device is added as a slave to a bonding device it is determined using the `RTE_PCI_DRV_INTR_LSC` flag whether the device supports interrupts or whether the link status should be monitored by polling it.

Requirements / Limitations

The current implementation only supports devices that support the same speed and duplex to be added as a slaves to the same bonded device. The bonded device inherits these attributes from the first active slave added to the bonded device and then all further slaves added to the bonded device must support these parameters.

A bonding device must have a minimum of one slave before the bonding device itself can be started.

To use a bonding device dynamic RSS configuration feature effectively, it is also required, that all slaves should be RSS-capable and support, at least one common hash function available for each of them. Changing RSS key is only possible, when all slave devices support the same key size.

To prevent inconsistency on how slaves process packets, once a device is added to a bonding device, RSS configuration should be managed through the bonding device API, and not directly on the slave.

Like all other PMD, all functions exported by a PMD are lock-free functions that are assumed not to be invoked in parallel on different logical cores to work on the same target object.

It should also be noted that the PMD receive function should not be invoked directly on a slave devices after they have been to a bonded device since packets read directly from the slave device will no longer be available to the bonded device to read.

Configuration

Link bonding devices are created using the `rte_eth_bond_create` API which requires a unique device name, the bonding mode, and the socket Id to allocate the bonding device's resources on. The other configurable parameters for a bonded device are its slave devices, its primary slave, a user defined MAC address and transmission policy to use if the device is in balance XOR mode.

Slave Devices

Bonding devices support up to a maximum of `RTE_MAX_ETHPORTS` slave devices of the same speed and duplex. Ethernet devices can be added as a slave to a maximum of one bonded device. Slave devices are reconfigured with the configuration of the bonded device on being added to a bonded device.

The bonded also guarantees to return the MAC address of the slave device to its original value of removal of a slave from it.

Primary Slave

The primary slave is used to define the default port to use when a bonded device is in active backup mode. A different port will only be used if, and only if, the current primary port goes down. If the user does not specify a primary port it will default to being the first port added to the bonded device.

MAC Address

The bonded device can be configured with a user specified MAC address, this address will be inherited by the some/all slave devices depending on the operating mode. If the device is in active backup mode then only the primary device will have the user specified MAC, all other slaves will retain their original MAC address. In mode 0, 2, 3, 4 all slaves devices are configure with the bonded devices MAC address.

If a user defined MAC address is not defined then the bonded device will default to using the primary slaves MAC address.

Balance XOR Transmit Policies

There are 3 supported transmission policies for bonded device running in Balance XOR mode. Layer 2, Layer 2+3, Layer 3+4.

- **Layer 2:** Ethernet MAC address based balancing is the default transmission policy for Balance XOR bonding mode. It uses a simple XOR calculation on the source MAC address and destination MAC address of the packet and then calculate the modulus of this value to calculate the slave device to transmit the packet on.
- **Layer 2 + 3:** Ethernet MAC address & IP Address based balancing uses a combination of source/destination MAC addresses and the source/destination IP addresses of the data packet to decide which slave port the packet will be transmitted on.

- **Layer 3 + 4:** IP Address & UDP Port based balancing uses a combination of source/destination IP Address and the source/destination UDP ports of the packet of the data packet to decide which slave port the packet will be transmitted on.

All these policies support 802.1Q VLAN Ethernet packets, as well as IPv4, IPv6 and UDP protocols for load balancing.

4.9.3 Using Link Bonding Devices

The librte_pmd_bond library supports two modes of device creation, the libraries export full C API or using the EAL command line to statically configure link bonding devices at application startup. Using the EAL option it is possible to use link bonding functionality transparently without specific knowledge of the libraries API, this can be used, for example, to add bonding functionality, such as active backup, to an existing application which has no knowledge of the link bonding C API.

Using the Poll Mode Driver from an Application

Using the librte_pmd_bond libraries API it is possible to dynamically create and manage link bonding device from within any application. Link bonding devices are created using the `rte_eth_bond_create` API which requires a unique device name, the link bonding mode to initial the device in and finally the socket Id which to allocate the devices resources onto. After successful creation of a bonding device it must be configured using the generic Ethernet device configure API `rte_eth_dev_configure` and then the RX and TX queues which will be used must be setup using `rte_eth_tx_queue_setup` / `rte_eth_rx_queue_setup`.

Slave devices can be dynamically added and removed from a link bonding device using the `rte_eth_bond_slave_add` / `rte_eth_bond_slave_remove` APIs but at least one slave device must be added to the link bonding device before it can be started using `rte_eth_dev_start`.

The link status of a bonded device is dictated by that of its slaves, if all slave device link status are down or if all slaves are removed from the link bonding device then the link status of the bonding device will go down.

It is also possible to configure / query the configuration of the control parameters of a bonded device using the provided APIs `rte_eth_bond_mode_set/get`, `rte_eth_bond_primary_set/get`, `rte_eth_bond_mac_set/reset` and `rte_eth_bond_xmit_policy_set/get`.

Using Link Bonding Devices from the EAL Command Line

Link bonding devices can be created at application startup time using the `--vdev` EAL command line option. The device name must start with the eth_bond prefix followed by numbers or letters. The name must be unique for each device. Each device can have multiple options arranged in a comma separated list. Multiple devices definitions can be arranged by calling the `--vdev` option multiple times.

Device names and bonding options must be separated by commas as shown below:

```
$RTE_TARGET/app/testpmd -c f -n 4 --vdev 'eth_bond0,bond_opt0=..,bond opt1=..'--vdev 'eth_bond1
```

Link Bonding EAL Options

There are multiple ways of definitions that can be assessed and combined as long as the following two rules are respected:

- A unique device name, in the format of eth_bondX is provided, where X can be any combination of numbers and/or letters, and the name is no greater than 32 characters long.
- A least one slave device is provided with for each bonded device definition.
- The operation mode of the bonded device being created is provided.

The different options are:

- mode: Integer value defining the bonding mode of the device. Currently supports modes 0,1,2,3,4,5 (round-robin, active backup, balance, broadcast, link aggregation, transmit load balancing).

```
mode=2
```

- slave: Defines the PMD device which will be added as slave to the bonded device. This option can be selected multiple times, for each device to be added as a slave. Physical devices should be specified using their PCI address, in the format domain:bus:devid.function

```
slave=0000:0a:00.0,slave=0000:0a:00.1
```

- primary: Optional parameter which defines the primary slave port, is used in active backup mode to select the primary slave for data TX/RX if it is available. The primary port also is used to select the MAC address to use when it is not defined by the user. This defaults to the first slave added to the device if it is specified. The primary device must be a slave of the bonded device.

```
primary=0000:0a:00.0
```

- socket_id: Optional parameter used to select which socket on a NUMA device the bonded devices resources will be allocated on.

```
socket_id=0
```

- mac: Optional parameter to select a MAC address for link bonding device, this overrides the value of the primary slave device.

```
mac=00:1e:67:1d:fd:1d
```

- xmit_policy: Optional parameter which defines the transmission policy when the bonded device is in balance mode. If not user specified this defaults to l2 (layer 2) forwarding, the other transmission policies available are l23 (layer 2+3) and l34 (layer 3+4)

```
xmit_policy=l23
```

- lsc_poll_period_ms: Optional parameter which defines the polling interval in milli-seconds at which devices which don't support lsc interrupts are checked for a change in the devices link status

```
lsc_poll_period_ms=100
```

- up_delay: Optional parameter which adds a delay in milli-seconds to the propagation of a devices link status changing to up, by default this parameter is zero.

```
up_delay=10
```

- down_delay: Optional parameter which adds a delay in milli-seconds to the propagation of a devices link status changing to down, by default this parameter is zero.

```
down_delay=50
```

Examples of Usage

Create a bonded device in round robin mode with two slaves specified by their PCI address:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_bond0,mode=0, slave=0000:00a:00.01,slave=0000
```

Create a bonded device in round robin mode with two slaves specified by their PCI address and an overriding MAC address:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_bond0,mode=0, slave=0000:00a:00.01,slave=0000
```

Create a bonded device in active backup mode with two slaves specified, and a primary slave specified by their PCI addresses:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_bond0,mode=1, slave=0000:00a:00.01,slave=0000
```

Create a bonded device in balance mode with two slaves specified by their PCI addresses, and a transmission policy of layer 3 + 4 forwarding:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_bond0,mode=2, slave=0000:00a:00.01,slave=0000
```

4.10 Timer Library

The Timer library provides a timer service to DPDK execution units to enable execution of callback functions asynchronously. Features of the library are:

- Timers can be periodic (multi-shot) or single (one-shot).
- Timers can be loaded from one core and executed on another. It has to be specified in the call to rte_timer_reset().
- Timers provide high precision (depends on the call frequency to rte_timer_manage() that checks timer expiration for the local core).
- If not required in the application, timers can be disabled at compilation time by not calling the rte_timer_manage() to increase performance.

The timer library uses the rte_get_timer_cycles() function that uses the High Precision Event Timer (HPET) or the CPUs Time Stamp Counter (TSC) to provide a reliable time reference.

This library provides an interface to add, delete and restart a timer. The API is based on BSD callout() with a few differences. Refer to the callout manual.

4.10.1 Implementation Details

Timers are tracked on a per-lcore basis, with all pending timers for a core being maintained in order of timer expiry in a skiplist data structure. The skiplist used has ten levels and each entry in the table appears in each level with probability ¼^level. This means that all entries are present in level 0, 1 in every 4 entries is present at level 1, one in every 16 at level 2 and so on up to level 9. This means that adding and removing entries from the timer list for a core can be done in log(n) time, up to 4^10 entries, that is, approximately 1,000,000 timers per lcore.

A timer structure contains a special field called status, which is a union of a timer state (stopped, pending, running, config) and an owner (lcore id). Depending on the timer state, we know if a timer is present in a list or not:

- STOPPED: no owner, not in a list
- CONFIG: owned by a core, must not be modified by another core, maybe in a list or not, depending on previous state
- PENDING: owned by a core, present in a list
- RUNNING: owned by a core, must not be modified by another core, present in a list

Resetting or stopping a timer while it is in a CONFIG or RUNNING state is not allowed. When modifying the state of a timer, a Compare And Swap instruction should be used to guarantee that the status (state+owner) is modified atomically.

Inside the rte_timer_manage() function, the skiplist is used as a regular list by iterating along the level 0 list, which contains all timer entries, until an entry which has not yet expired has been encountered. To improve performance in the case where there are entries in the timer list but none of those timers have yet expired, the expiry time of the first list entry is maintained within the per-core timer list structure itself. On 64-bit platforms, this value can be checked without the need to take a lock on the overall structure. (Since expiry times are maintained as 64-bit values, a check on the value cannot be done on 32-bit platforms without using either a compare-and-swap (CAS) instruction or using a lock, so this additional check is skipped in favor of checking as normal once the lock has been taken.) On both 64-bit and 32-bit platforms, a call to rte_timer_manage() returns without taking a lock in the case where the timer list for the calling core is empty.

4.10.2 Use Cases

The timer library is used for periodic calls, such as garbage collectors, or some state machines (ARP, bridging, and so on).

4.10.3 References

- callout manual - The callout facility that provides timers with a mechanism to execute a function at a given time.
- HPET - Information about the High Precision Event Timer (HPET).

4.11 Hash Library

The DPDK provides a Hash Library for creating hash table for fast lookup. The hash table is a data structure optimized for searching through a set of entries that are each identified by a unique key. For increased performance the DPDK Hash requires that all the keys have the same number of bytes which is set at the hash creation time.

4.11.1 Hash API Overview

The main configuration parameters for the hash are:

- Total number of hash entries
- Size of the key in bytes

The hash also allows the configuration of some low-level implementation related parameters such as:

- Hash function to translate the key into a bucket index

The main methods exported by the hash are:

- Add entry with key: The key is provided as input. If a new entry is successfully added to the hash for the specified key, or there is already an entry in the hash for the specified key, then the position of the entry is returned. If the operation was not successful, for example due to lack of free entries in the hash, then a negative value is returned;
- Delete entry with key: The key is provided as input. If an entry with the specified key is found in the hash, then the entry is removed from the hash and the position where the entry was found in the hash is returned. If no entry with the specified key exists in the hash, then a negative value is returned
- Lookup for entry with key: The key is provided as input. If an entry with the specified key is found in the hash (lookup hit), then the position of the entry is returned, otherwise (lookup miss) a negative value is returned.

Apart from these method explained above, the API allows the user three more options:

- Add / lookup / delete with key and precomputed hash: Both the key and its precomputed hash are provided as input. This allows the user to perform these operations faster, as hash is already computed.
- Add / lookup with key and data: A pair of key-value is provided as input. This allows the user to store not only the key, but also data which may be either a 8-byte integer or a pointer to external data (if data size is more than 8 bytes).
- Combination of the two options above: User can provide key, precomputed hash and data.

Also, the API contains a method to allow the user to look up entries in bursts, achieving higher performance than looking up individual entries, as the function prefetches next entries at the time it is operating with the first ones, which reduces significantly the impact of the necessary memory accesses. Notice that this method uses a pipeline of 8 entries (4 stages of 2 entries), so it is highly recommended to use at least 8 entries per burst.

The actual data associated with each key can be either managed by the user using a separate table that mirrors the hash in terms of number of entries and position of each entry, as shown in the Flow Classification use case describes in the following sections, or stored in the hash table itself.

The example hash tables in the L2/L3 Forwarding sample applications defines which port to forward a packet to based on a packet flow identified by the five-tuple lookup. However, this table could also be used for more sophisticated features and provide many other functions and actions that could be performed on the packets and flows.

4.11.2 Implementation Details

The hash table has two main tables:

- First table is an array of entries which is further divided into buckets, with the same number of consecutive array entries in each bucket. Each entry contains the computed primary and secondary hashes of a given key (explained below), and an index to the second table.
- The second table is an array of all the keys stored in the hash table and its data associated to each key.

The hash library uses the cuckoo hash method to resolve collisions. For any input key, there are two possible buckets (primary and secondary/alternative location) where that key can be stored in the hash, therefore only the entries within those bucket need to be examined when the key is looked up. The lookup speed is achieved by reducing the number of entries to be scanned from the total number of hash entries down to the number of entries in the two hash buckets, as opposed to the basic method of linearly scanning all the entries in the array. The hash uses a hash function (configurable) to translate the input key into a 4-byte key signature. The bucket index is the key signature modulo the number of hash buckets.

Once the buckets are identified, the scope of the hash add, delete and lookup operations is reduced to the entries in those buckets (it is very likely that entries are in the primary bucket).

To speed up the search logic within the bucket, each hash entry stores the 4-byte key signature together with the full key for each hash entry. For large key sizes, comparing the input key against a key from the bucket can take significantly more time than comparing the 4-byte signature of the input key against the signature of a key from the bucket. Therefore, the signature comparison is done first and the full key comparison done only when the signatures matches. The full key comparison is still necessary, as two input keys from the same bucket can still potentially have the same 4-byte hash signature, although this event is relatively rare for hash functions providing good uniform distributions for the set of input keys.

Example of lookup:

First of all, the primary bucket is identified and entry is likely to be stored there. If signature was stored there, we compare its key against the one provided and return the position where it was stored and/or the data associated to that key if there is a match. If signature is not in the primary bucket, the secondary bucket is looked up, where same procedure is carried out. If there is no match there either, key is considered not to be in the table.

Example of addition:

Like lookup, the primary and secondary buckets are identified. If there is an empty slot in the primary bucket, primary and secondary signatures are stored in that slot, key and data (if any) are added to the second table and an index to the position in the second table is stored in the slot of the first table. If there is no space in the primary bucket, one of the entries on that bucket is pushed to its alternative location, and the key to be added is inserted in its position. To know where the alternative bucket of the evicted entry is, the secondary signature is looked up and alternative bucket index is calculated from doing the modulo, as seen above. If there is room in the alternative bucket, the evicted entry is stored in it. If not, same process is repeated (one of the entries gets pushed) until a non full bucket is found. Notice that despite all the entry movement in the first table, the second table is not touched, which would impact greatly in performance.

In the very unlikely event that table enters in a loop where same entries are being evicted indefinitely, key is considered not able to be stored. With random keys, this method allows the user to get around 90% of the table utilization, without having to drop any stored entry (LRU) or allocate more memory (extended buckets).

4.11.3 Entry distribution in hash table

As mentioned above, Cuckoo hash implementation pushes elements out of their bucket, if there is a new entry to be added which primary location coincides with their current bucket, being pushed to their alternative location. Therefore, as user adds more entries to the hash table, distribution of the hash values in the buckets will change, being most of them in their primary location and a few in their secondary location, which the later will increase, as table gets busier. This information is quite useful, as performance may be lower as more entries are evicted to their secondary location.

See the tables below showing example entry distribution as table utilization increases.

Table 4.1: Entry distribution measured with an example table with 1024 random entries using jhash algorithm

% Table used	% In Primary location	% In Secondary location
25	100	0
50	96.1	3.9
75	88.2	11.8
80	86.3	13.7
85	83.1	16.9
90	77.3	22.7
95.8	64.5	35.5

Table 4.2: Entry distribution measured with an example table with 1 million random entries using jhash algorithm

% Table used	% In Primary location	% In Secondary location
50	96	4
75	86.9	13.1
80	83.9	16.1
85	80.1	19.9
90	74.8	25.2
94.5	67.4	32.6

Note: Last values on the tables above are the average maximum table utilization with random keys and using Jenkins hash function.

4.11.4 Use Case: Flow Classification

Flow classification is used to map each input packet to the connection/flow it belongs to. This operation is necessary as the processing of each input packet is usually done in the context of their connection, so the same set of operations is applied to all the packets from the same flow.

Applications using flow classification typically have a flow table to manage, with each separate flow having an entry associated with it in this table. The size of the flow table entry is application specific, with typical values of 4, 16, 32 or 64 bytes.

Each application using flow classification typically has a mechanism defined to uniquely identify a flow based on a number of fields read from the input packet that make up the flow key. One example is to use the DiffServ 5-tuple made up of the following fields of the IP and transport layer packet headers: Source IP Address, Destination IP Address, Protocol, Source Port, Destination Port.

The DPDK hash provides a generic method to implement an application specific flow classification mechanism. Given a flow table implemented as an array, the application should create a hash object with the same number of entries as the flow table and with the hash key size set to the number of bytes in the selected flow key.

The flow table operations on the application side are described below:

- Add flow: Add the flow key to hash. If the returned position is valid, use it to access the flow entry in the flow table for adding a new flow or updating the information associated with an existing flow. Otherwise, the flow addition failed, for example due to lack of free entries for storing new flows.
- Delete flow: Delete the flow key from the hash. If the returned position is valid, use it to access the flow entry in the flow table to invalidate the information associated with the flow.
- Lookup flow: Lookup for the flow key in the hash. If the returned position is valid (flow lookup hit), use the returned position to access the flow entry in the flow table. Otherwise (flow lookup miss) there is no flow registered for the current packet.

4.11.5 References

- Donald E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition), 1998, Addison-Wesley Professional

4.12 LPM Library

The DPDK LPM library component implements the Longest Prefix Match (LPM) table search method for 32-bit keys that is typically used to find the best route match in IP forwarding applications.

4.12.1 LPM API Overview

The main configuration parameter for LPM component instances is the maximum number of rules to support. An LPM prefix is represented by a pair of parameters (32- bit key, depth), with depth in the range of 1 to 32. An LPM rule is represented by an LPM prefix and some user data associated with the prefix. The prefix serves as the unique identifier of the LPM rule. In this implementation, the user data is 1-byte long and is called next hop, in correlation with its main use of storing the ID of the next hop in a routing table entry.

The main methods exported by the LPM component are:

- Add LPM rule: The LPM rule is provided as input. If there is no rule with the same prefix present in the table, then the new rule is added to the LPM table. If a rule with the same prefix is already present in the table, the next hop of the rule is updated. An error is returned when there is no available rule space left.

- Delete LPM rule: The prefix of the LPM rule is provided as input. If a rule with the specified prefix is present in the LPM table, then it is removed.
- Lookup LPM key: The 32-bit key is provided as input. The algorithm selects the rule that represents the best match for the given key and returns the next hop of that rule. In the case that there are multiple rules present in the LPM table that have the same 32-bit key, the algorithm picks the rule with the highest depth as the best match rule, which means that the rule has the highest number of most significant bits matching between the input key and the rule key.

4.12.2 Implementation Details

The current implementation uses a variation of the DIR-24-8 algorithm that trades memory usage for improved LPM lookup speed. The algorithm allows the lookup operation to be performed with typically a single memory read access. In the statistically rare case when the best match rule is having a depth bigger than 24, the lookup operation requires two memory read accesses. Therefore, the performance of the LPM lookup operation is greatly influenced by whether the specific memory location is present in the processor cache or not.

The main data structure is built using the following elements:

- A table with 2^24 entries.
- A number of tables (RTE_LPM_TBL8_NUM_GROUPS) with 2^8 entries.

The first table, called tbl24, is indexed using the first 24 bits of the IP address to be looked up, while the second table(s), called tbl8, is indexed using the last 8 bits of the IP address. This means that depending on the outcome of trying to match the IP address of an incoming packet to the rule stored in the tbl24 we might need to continue the lookup process in the second level.

Since every entry of the tbl24 can potentially point to a tbl8, ideally, we would have 2^24 tbl8s, which would be the same as having a single table with 2^32 entries. This is not feasible due to resource restrictions. Instead, this approach takes advantage of the fact that rules longer than 24 bits are very rare. By splitting the process in two different tables/levels and limiting the number of tbl8s, we can greatly reduce memory consumption while maintaining a very good lookup speed (one memory access, most of the times).

An entry in tbl24 contains the following fields:

- next hop / index to the tbl8
- valid flag
- external entry flag
- depth of the rule (length)

The first field can either contain a number indicating the tbl8 in which the lookup process should continue or the next hop itself if the longest prefix match has already been found. The two flags are used to determine whether the entry is valid or not and whether the search process have finished or not respectively. The depth or length of the rule is the number of bits of the rule that is stored in a specific entry.

An entry in a tbl8 contains the following fields:

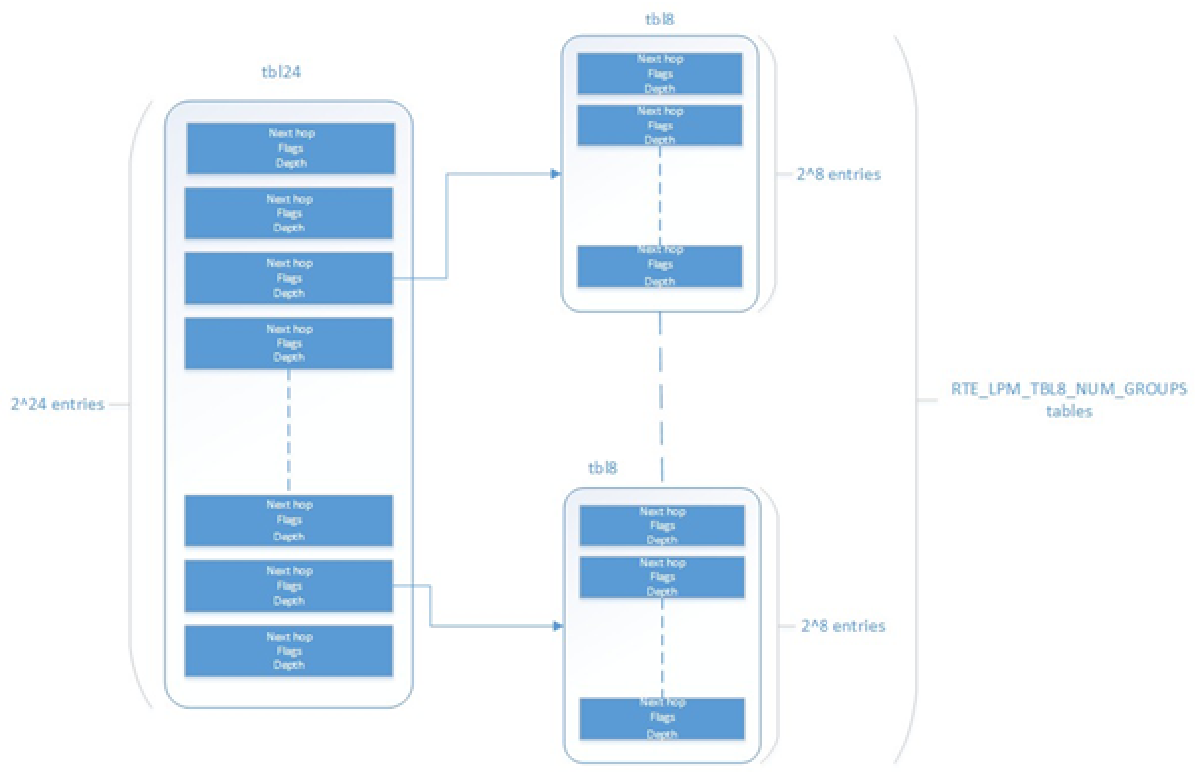


Fig. 4.31: Table split into different levels

- next hop
- valid
- valid group
- depth

Next hop and depth contain the same information as in the tbl24. The two flags show whether the entry and the table are valid respectively.

The other main data structure is a table containing the main information about the rules (IP and next hop). This is a higher level table, used for different things:

- Check whether a rule already exists or not, prior to addition or deletion, without having to actually perform a lookup.
- When deleting, to check whether there is a rule containing the one that is to be deleted. This is important, since the main data structure will have to be updated accordingly.

Addition

When adding a rule, there are different possibilities. If the rule's depth is exactly 24 bits, then:

- Use the rule (IP address) as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then set its next hop to its value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 0 (meaning the lookup process ends at this point, since this is the longest prefix that matches).

If the rule's depth is exactly 32 bits, then:

- Use the first 24 bits of the rule as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then look for a free tbl8, set the index to the tbl8 to this value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 1 (meaning the lookup process must continue since the rule hasn't been explored completely).

If the rule's depth is any other value, prefix expansion must be performed. This means the rule is copied to all the entries (as long as they are not in use) which would also cause a match.

As a simple example, let's assume the depth is 20 bits. This means that there are 2^(24 - 20) = 16 different combinations of the first 24 bits of an IP address that would cause a match. Hence, in this case, we copy the exact same entry to every position indexed by one of these combinations.

By doing this we ensure that during the lookup process, if a rule matching the IP address exists, it is found in either one or two memory accesses, depending on whether we need to move to the next table or not. Prefix expansion is one of the keys of this algorithm, since it improves the speed dramatically by adding redundancy.

Lookup

The lookup process is much simpler and quicker. In this case:

- Use the first 24 bits of the IP address as an index to the tbl24. If the entry is not in use, then it means we don't have a rule matching this IP. If it is valid and the external entry flag is set to 0, then the next hop is returned.
- If it is valid and the external entry flag is set to 1, then we use the tbl8 index to find out the tbl8 to be checked, and the last 8 bits of the IP address as an index to this table. Similarly, if the entry is not in use, then we don't have a rule matching this IP address. If it is valid then the next hop is returned.

Limitations in the Number of Rules

There are different things that limit the number of rules that can be added. The first one is the maximum number of rules, which is a parameter passed through the API. Once this number is reached, it is not possible to add any more rules to the routing table unless one or more are removed.

The second reason is an intrinsic limitation of the algorithm. As explained before, to avoid high memory consumption, the number of tbl8s is limited in compilation time (this value is by default 256). If we exhaust tbl8s, we won't be able to add any more rules. How many of them are necessary for a specific routing table is hard to determine in advance.

A tbl8 is consumed whenever we have a new rule with depth bigger than 24, and the first 24 bits of this rule are not the same as the first 24 bits of a rule previously added. If they are, then the new rule will share the same tbl8 than the previous one, since the only difference between the two rules is within the last byte.

With the default value of 256, we can have up to 256 rules longer than 24 bits that differ on their first three bytes. Since routes longer than 24 bits are unlikely, this shouldn't be a problem in most setups. Even if it is, however, the number of tbl8s can be modified.

Use Case: IPv4 Forwarding

The LPM algorithm is used to implement Classless Inter-Domain Routing (CIDR) strategy used by routers implementing IPv4 forwarding.

References

- RFC1519 Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy, http://www.ietf.org/rfc/rfc1519
- Pankaj Gupta, Algorithms for Routing Lookups and Packet Classification, PhD Thesis, Stanford University, 2000 (http://klamath.stanford.edu/~pankaj/thesis/ thesis_1sided.pdf)

4.13 LPM6 Library

The LPM6 (LPM for IPv6) library component implements the Longest Prefix Match (LPM) table search method for 128-bit keys that is typically used to find the best match route in IPv6 forwarding applications.

4.13.1 LPM6 API Overview

The main configuration parameters for the LPM6 library are:

- Maximum number of rules: This defines the size of the table that holds the rules, and therefore the maximum number of rules that can be added.
- Number of tbl8s: A tbl8 is a node of the trie that the LPM6 algorithm is based on.

This parameter is related to the number of rules you can have, but there is no way to accurately predict the number needed to hold a specific number of rules, since it strongly depends on the depth and IP address of every rule. One tbl8 consumes 1 kb of memory. As a recommendation, 65536 tbl8s should be sufficient to store several thousand IPv6 rules, but the number can vary depending on the case.

An LPM prefix is represented by a pair of parameters (128-bit key, depth), with depth in the range of 1 to 128. An LPM rule is represented by an LPM prefix and some user data associated with the prefix. The prefix serves as the unique identifier for the LPM rule. In this implementation, the user data is 1-byte long and is called “next hop”, which corresponds to its main use of storing the ID of the next hop in a routing table entry.

The main methods exported for the LPM component are:

- Add LPM rule: The LPM rule is provided as input. If there is no rule with the same prefix present in the table, then the new rule is added to the LPM table. If a rule with the same prefix is already present in the table, the next hop of the rule is updated. An error is returned when there is no available space left.
- Delete LPM rule: The prefix of the LPM rule is provided as input. If a rule with the specified prefix is present in the LPM table, then it is removed.
- Lookup LPM key: The 128-bit key is provided as input. The algorithm selects the rule that represents the best match for the given key and returns the next hop of that rule. In

the case that there are multiple rules present in the LPM table that have the same 128-bit value, the algorithm picks the rule with the highest depth as the best match rule, which means the rule has the highest number of most significant bits matching between the input key and the rule key.

Implementation Details

This is a modification of the algorithm used for IPv4 (see Section 19.2 "Implementation Details"). In this case, instead of using two levels, one with a tbl24 and a second with a tbl8, 14 levels are used.

The implementation can be seen as a multi-bit trie where the *stride* or number of bits inspected on each level varies from level to level. Specifically, 24 bits are inspected on the root node, and the remaining 104 bits are inspected in groups of 8 bits. This effectively means that the trie has 14 levels at the most, depending on the rules that are added to the table.

The algorithm allows the lookup operation to be performed with a number of memory accesses that directly depends on the length of the rule and whether there are other rules with bigger depths and the same key in the data structure. It can vary from 1 to 14 memory accesses, with 5 being the average value for the lengths that are most commonly used in IPv6.

The main data structure is built using the following elements:

- A table with 224 entries
- A number of tables, configurable by the user through the API, with 28 entries

The first table, called tbl24, is indexed using the first 24 bits of the IP address be looked up, while the rest of the tables, called tbl8s, are indexed using the rest of the bytes of the IP address, in chunks of 8 bits. This means that depending on the outcome of trying to match the IP address of an incoming packet to the rule stored in the tbl24 or the subsequent tbl8s we might need to continue the lookup process in deeper levels of the tree.

Similar to the limitation presented in the algorithm for IPv4, to store every possible IPv6 rule, we would need a table with 2^128 entries. This is not feasible due to resource restrictions.

By splitting the process in different tables/levels and limiting the number of tbl8s, we can greatly reduce memory consumption while maintaining a very good lookup speed (one memory access per level).

An entry in a table contains the following fields:

- next hop / index to the tbl8
- depth of the rule (length)
- valid flag
- valid group flag
- external entry flag

The first field can either contain a number indicating the tbl8 in which the lookup process should continue or the next hop itself if the longest prefix match has already been found. The depth or length of the rule is the number of bits of the rule that is stored in a specific entry. The flags are used to determine whether the entry/table is valid or not and whether the search process have finished or not respectively.

Both types of tables share the same structure.

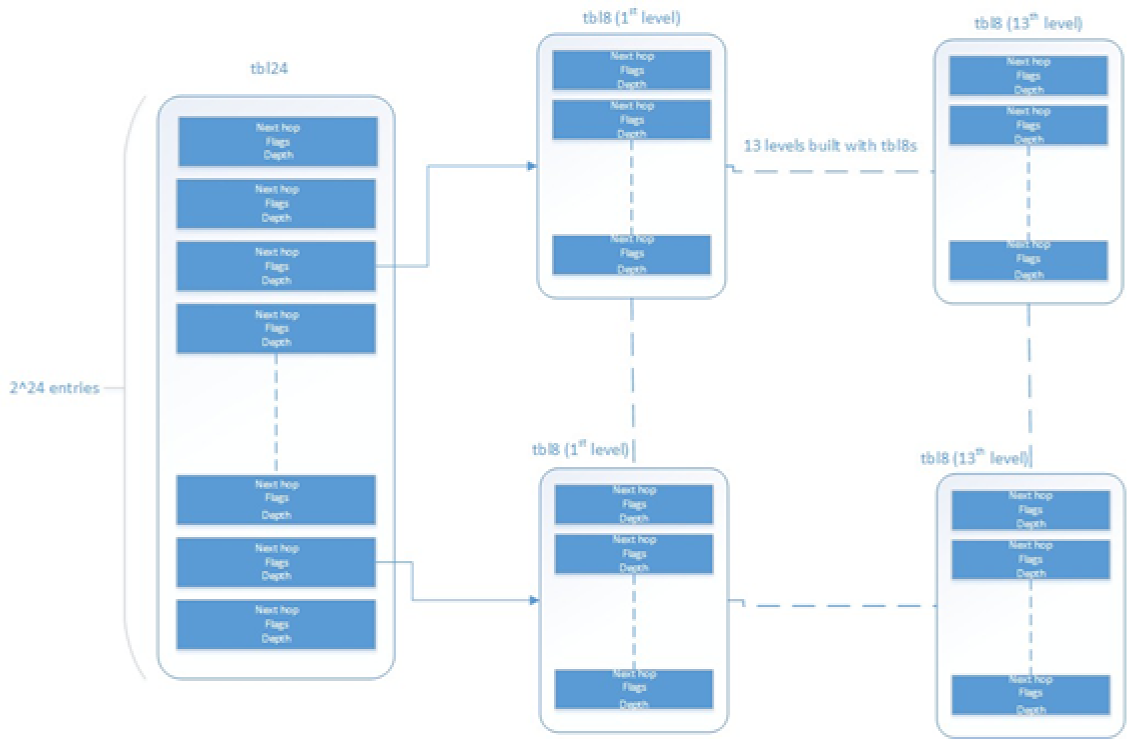


Fig. 4.32: Table split into different levels

The other main data structure is a table containing the main information about the rules (IP, next hop and depth). This is a higher level table, used for different things:

- Check whether a rule already exists or not, prior to addition or deletion, without having to actually perform a lookup.

When deleting, to check whether there is a rule containing the one that is to be deleted. This is important, since the main data structure will have to be updated accordingly.

Addition

When adding a rule, there are different possibilities. If the rule's depth is exactly 24 bits, then:

- Use the rule (IP address) as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then set its next hop to its value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 0 (meaning the lookup process ends at this point, since this is the longest prefix that matches).

If the rule's depth is bigger than 24 bits but a multiple of 8, then:

- Use the first 24 bits of the rule as an index to the tbl24.
- If the entry is invalid (i.e. it doesn't already contain a rule) then look for a free tbl8, set the index to the tbl8 to this value, the valid flag to 1 (meaning this entry is in use), and the external entry flag to 1 (meaning the lookup process must continue since the rule hasn't been explored completely).
- Use the following 8 bits of the rule as an index to the next tbl8.

- Repeat the process until the tbl8 at the right level (depending on the depth) has been reached and fill it with the next hop, setting the next entry flag to 0.

If the rule's depth is any other value, prefix expansion must be performed. This means the rule is copied to all the entries (as long as they are not in use) which would also cause a match.

As a simple example, let's assume the depth is 20 bits. This means that there are 2^(24-20) = 16 different combinations of the first 24 bits of an IP address that would cause a match. Hence, in this case, we copy the exact same entry to every position indexed by one of these combinations.

By doing this we ensure that during the lookup process, if a rule matching the IP address exists, it is found in, at the most, 14 memory accesses, depending on how many times we need to move to the next table. Prefix expansion is one of the keys of this algorithm, since it improves the speed dramatically by adding redundancy.

Prefix expansion can be performed at any level. So, for example, is the depth is 34 bits, it will be performed in the third level (second tbl8-based level).

Lookup

The lookup process is much simpler and quicker. In this case:

- Use the first 24 bits of the IP address as an index to the tbl24. If the entry is not in use, then it means we don't have a rule matching this IP. If it is valid and the external entry flag is set to 0, then the next hop is returned.
- If it is valid and the external entry flag is set to 1, then we use the tbl8 index to find out the tbl8 to be checked, and the next 8 bits of the IP address as an index to this table. Similarly, if the entry is not in use, then we don't have a rule matching this IP address. If it is valid then check the external entry flag for a new tbl8 to be inspected.
- Repeat the process until either we find an invalid entry (lookup miss) or a valid entry with the external entry flag set to 0. Return the next hop in the latter case.

Limitations in the Number of Rules

There are different things that limit the number of rules that can be added. The first one is the maximum number of rules, which is a parameter passed through the API. Once this number is reached, it is not possible to add any more rules to the routing table unless one or more are removed.

The second limitation is in the number of tbl8s available. If we exhaust tbl8s, we won't be able to add any more rules. How to know how many of them are necessary for a specific routing table is hard to determine in advance.

In this algorithm, the maximum number of tbl8s a single rule can consume is 13, which is the number of levels minus one, since the first three bytes are resolved in the tbl24. However:

- Typically, on IPv6, routes are not longer than 48 bits, which means rules usually take up to 3 tbl8s.

As explained in the LPM for IPv4 algorithm, it is possible and very likely that several rules will share one or more tbl8s, depending on what their first bytes are. If they share the same first 24 bits, for instance, the tbl8 at the second level will be shared. This might happen again in deeper levels, so, effectively, two 48 bit-long rules may use the same three tbl8s if the only difference is in their last byte.

The number of tbl8s is a parameter exposed to the user through the API in this version of the algorithm, due to its impact in memory consumption and the number or rules that can be added to the LPM table. One tbl8 consumes 1 kilobyte of memory.

4.13.2 Use Case: IPv6 Forwarding

The LPM algorithm is used to implement the Classless Inter-Domain Routing (CIDR) strategy used by routers implementing IP forwarding.

4.14 Packet Distributor Library

The DPDK Packet Distributor library is a library designed to be used for dynamic load balancing of traffic while supporting single packet at a time operation. When using this library, the logical cores in use are to be considered in two roles: firstly a distributor lcore, which is responsible for load balancing or distributing packets, and a set of worker lcores which are responsible for receiving the packets from the distributor and operating on them. The model of operation is shown in the diagram below.

4.14.1 Distributor Core Operation

The distributor core does the majority of the processing for ensuring that packets are fairly shared among workers. The operation of the distributor is as follows:

1. Packets are passed to the distributor component by having the distributor lcore thread call the “rte_distributor_process()” API
2. The worker lcores all share a single cache line with the distributor core in order to pass messages and packets to and from the worker. The process API call will poll all the worker cache lines to see what workers are requesting packets.
3. As workers request packets, the distributor takes packets from the set of packets passed in and distributes them to the workers. As it does so, it examines the “tag” – stored in the RSS hash field in the mbuf – for each packet and records what tags are being processed by each worker.
4. If the next packet in the input set has a tag which is already being processed by a worker, then that packet will be queued up for processing by that worker and given to it in preference to other packets when that work next makes a request for work. This ensures that no two packets with the same tag are processed in parallel, and that all packets with the same tag are processed in input order.
5. Once all input packets passed to the process API have either been distributed to workers or been queued up for a worker which is processing a given tag, then the process API returns to the caller.

Other functions which are available to the distributor lcore are:

- rte_distributor_returned_pkts()
- rte_distributor_flush()

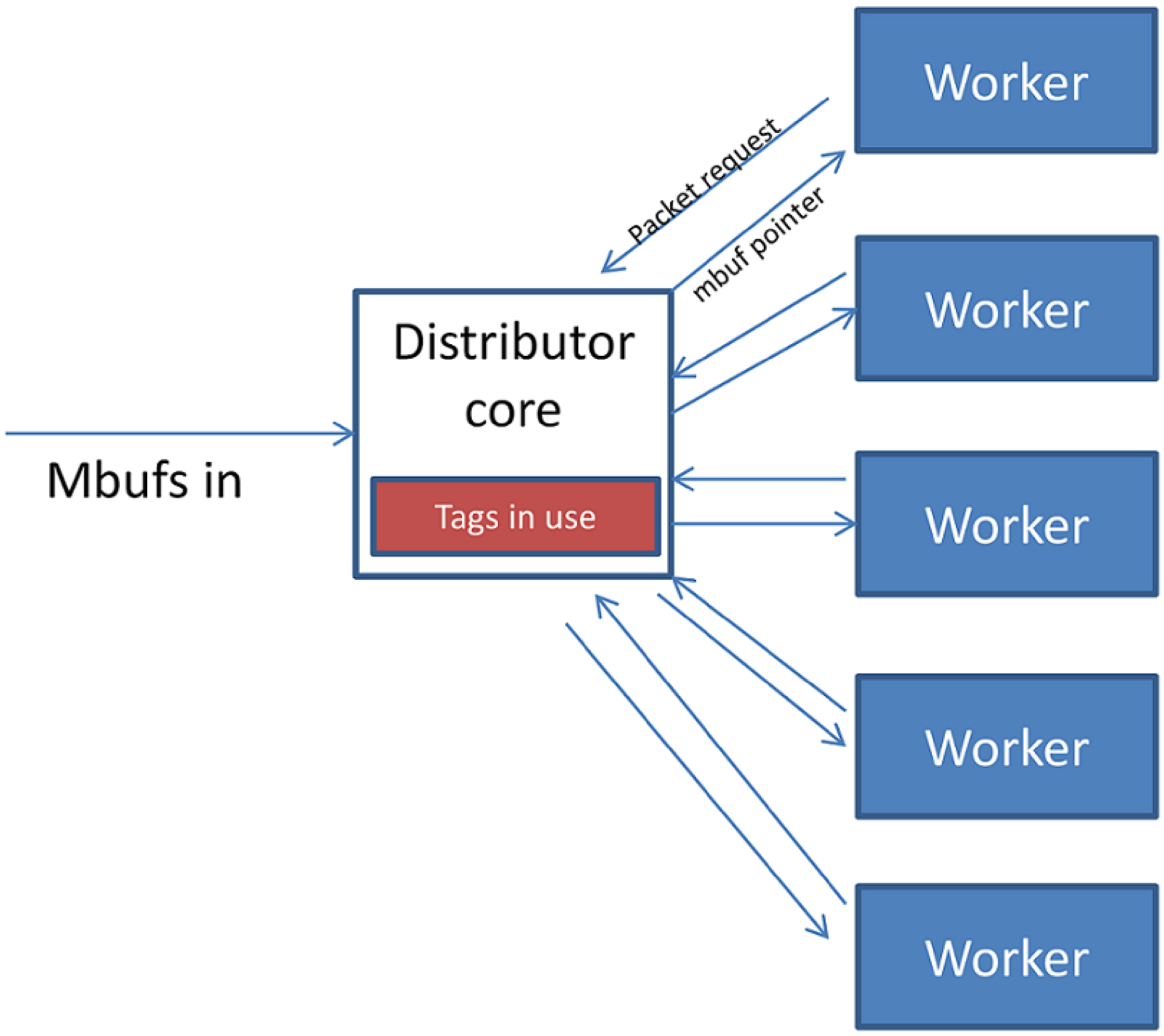


Fig. 4.33: Packet Distributor mode of operation

- rte_distributor_clear_returns()

Of these the most important API call is “rte_distributor_returned_pkts()” which should only be called on the lcore which also calls the process API. It returns to the caller all packets which have finished processing by all worker cores. Within this set of returned packets, all packets sharing the same tag will be returned in their original order.

NOTE: If worker lcores buffer up packets internally for transmission in bulk afterwards, the packets sharing a tag will likely get out of order. Once a worker lcore requests a new packet, the distributor assumes that it has completely finished with the previous packet and therefore that additional packets with the same tag can safely be distributed to other workers – who may then flush their buffered packets sooner and cause packets to get out of order.

NOTE: No packet ordering guarantees are made about packets which do not share a common packet tag.

Using the process and returned_pkts API, the following application workflow can be used, while allowing packet order within a packet flow – identified by a tag – to be maintained.

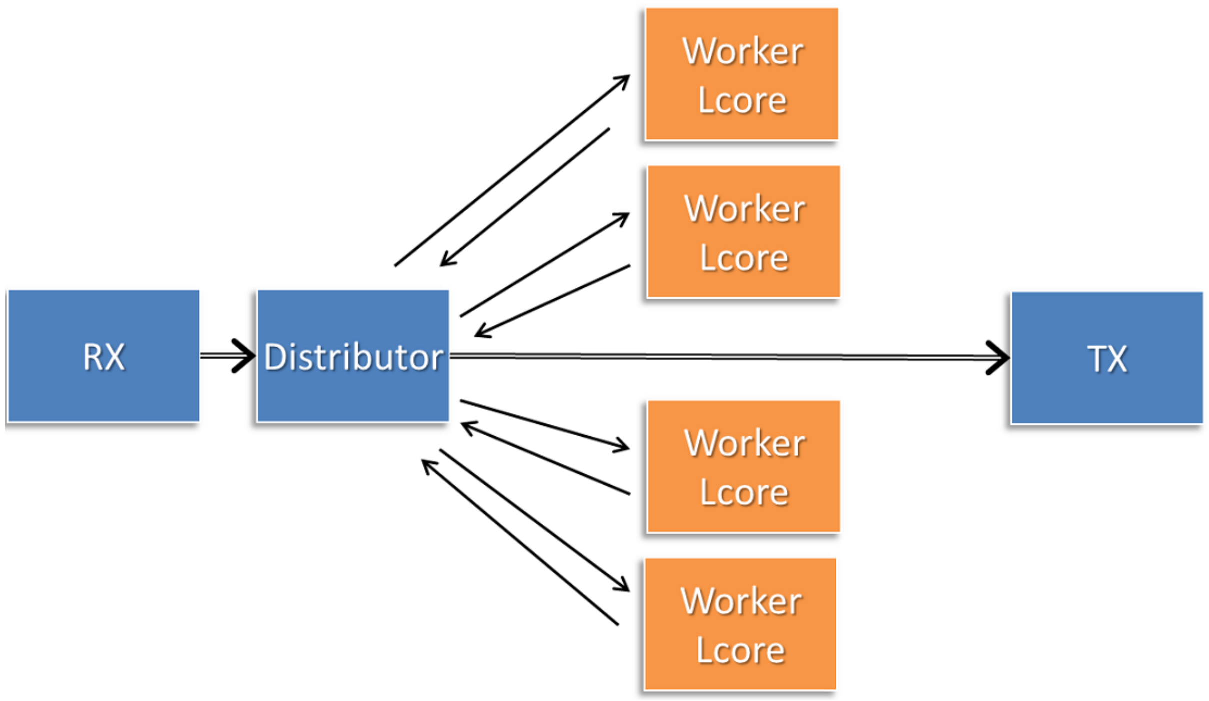


Fig. 4.34: Application workflow

The flush and clear_returns API calls, mentioned previously, are likely of less use that the process and returned_pkts APIS, and are principally provided to aid in unit testing of the library. Descriptions of these functions and their use can be found in the DPDK API Reference document.

4.14.2 Worker Operation

Worker cores are the cores which do the actual manipulation of the packets distributed by the packet distributor. Each worker calls “rte_distributor_get_pkt()” API to request a new packet when it has finished processing the previous one. [The previous packet should be returned to the distributor component by passing it as the final parameter to this API call.]

Since it may be desirable to vary the number of worker cores, depending on the traffic load i.e. to save power at times of lighter load, it is possible to have a worker stop processing packets by calling "rte_distributor_return_pkt()" to indicate that it has finished the current packet and does not want a new one.

4.15 Reorder Library

The Reorder Library provides a mechanism for reordering mbufs based on their sequence number.

4.15.1 Operation

The reorder library is essentially a buffer that reorders mbufs. The user inserts out of order mbufs into the reorder buffer and pulls in-order mbufs from it.

At a given time, the reorder buffer contains mbufs whose sequence number are inside the sequence window. The sequence window is determined by the minimum sequence number and the number of entries that the buffer was configured to hold. For example, given a reorder buffer with 200 entries and a minimum sequence number of 350, the sequence window has low and high limits of 350 and 550 respectively.

When inserting mbufs, the reorder library differentiates between valid, early and late mbufs depending on the sequence number of the inserted mbuf:

- valid: the sequence number is inside the window.
- late: the sequence number is outside the window and less than the low limit.
- early: the sequence number is outside the window and greater than the high limit.

The reorder buffer directly returns late mbufs and tries to accommodate early mbufs.

4.15.2 Implementation Details

The reorder library is implemented as a pair of buffers, which referred to as the *Order* buffer and the *Ready* buffer.

On an insert call, valid mbufs are inserted directly into the Order buffer and late mbufs are returned to the user with an error.

In the case of early mbufs, the reorder buffer will try to move the window (incrementing the minimum sequence number) so that the mbuf becomes a valid one. To that end, mbufs in the Order buffer are moved into the Ready buffer. Any mbufs that have not arrived yet are ignored and therefore will become late mbufs. This means that as long as there is room in the Ready buffer, the window will be moved to accommodate early mbufs that would otherwise be outside the reordering window.

For example, assuming that we have a buffer of 200 entries with a 350 minimum sequence number, and we need to insert an early mbuf with 565 sequence number. That means that we would need to move the windows at least 15 positions to accommodate the mbuf. The reorder buffer would try to move mbufs from at least the next 15 slots in the Order buffer to the Ready buffer, as long as there is room in the Ready buffer. Any gaps in the Order buffer at that point are skipped, and those packet will be reported as late packets when they arrive. The process of moving packets to the Ready buffer continues beyond the minimum required until a gap, i.e. missing mbuf, in the Order buffer is encountered.

When draining mbufs, the reorder buffer would return mbufs in the Ready buffer first and then from the Order buffer until a gap is found (mbufs that have not arrived yet).

4.15.3 Use Case: Packet Distributor

An application using the DPDK packet distributor could make use of the reorder library to transmit packets in the same order they were received.

A basic packet distributor use case would consist of a distributor with multiple workers cores. The processing of packets by the workers is not guaranteed to be in order, hence a reorder buffer can be used to order as many packets as possible.

In such a scenario, the distributor assigns a sequence number to mbufs before delivering them to the workers. As the workers finish processing the packets, the distributor inserts those mbufs into the reorder buffer and finally transmit drained mbufs.

NOTE: Currently the reorder buffer is not thread safe so the same thread is responsible for inserting and draining mbufs.

4.16 IP Fragmentation and Reassembly Library

The IP Fragmentation and Reassembly Library implements IPv4 and IPv6 packet fragmentation and reassembly.

4.16.1 Packet fragmentation

Packet fragmentation routines divide input packet into number of fragments. Both rte_ipv4_fragment_packet() and rte_ipv6_fragment_packet() functions assume that input mbuf data points to the start of the IP header of the packet (i.e. L2 header is already stripped out). To avoid copying of the actual packet's data zero-copy technique is used (rte_pktmbuf_attach). For each fragment two new mbufs are created:

- Direct mbuf – mbuf that will contain L3 header of the new fragment.
- Indirect mbuf – mbuf that is attached to the mbuf with the original packet. It's data field points to the start of the original packets data plus fragment offset.

Then L3 header is copied from the original mbuf into the 'direct' mbuf and updated to reflect new fragmented status. Note that for IPv4, header checksum is not recalculated and is set to zero.

Finally 'direct' and 'indirect' mbufs for each fragment are linked together via mbuf's next filed to compose a packet for the new fragment.

The caller has an ability to explicitly specify which mempools should be used to allocate 'direct' and 'indirect' mbufs from.

For more information about direct and indirect mbufs, refer to the *DPDK Programmers guide 7.7 Direct and Indirect Buffers.*

4.16.2 Packet reassembly

IP Fragment Table

Fragment table maintains information about already received fragments of the packet.

Each IP packet is uniquely identified by triple <Source IP address>, <Destination IP address>, <ID>.

Note that all update/lookup operations on Fragment Table are not thread safe. So if different execution contexts (threads/processes) will access the same table simultaneously, then some external syncing mechanism have to be provided.

Each table entry can hold information about packets consisting of up to RTE_LIBRTE_IP_FRAG_MAX (by default: 4) fragments.

Code example, that demonstrates creation of a new Fragment table:

```
frag_cycles = (rte_get_tsc_hz() + MS_PER_S - 1) / MS_PER_S * max_flow_ttl;
bucket_num = max_flow_num + max_flow_num / 4;
frag_tbl = rte_ip_frag_table_create(max_flow_num, bucket_entries, max_flow_num, frag_cycles, sc
```

Internally Fragment table is a simple hash table. The basic idea is to use two hash functions and <bucket_entries> * associativity. This provides 2 * <bucket_entries> possible locations in the hash table for each key. When the collision occurs and all 2 * <bucket_entries> are occupied, instead of reinserting existing keys into alternative locations, ip_frag_tbl_add() just returns a failure.

Also, entries that resides in the table longer then <max_cycles> are considered as invalid, and could be removed/replaced by the new ones.

Note that reassembly demands a lot of mbuf's to be allocated. At any given time up to (2 * bucket_entries * RTE_LIBRTE_IP_FRAG_MAX * <maximum number of mbufs per packet>) can be stored inside Fragment Table waiting for remaining fragments.

Packet Reassembly

Fragmented packets processing and reassembly is done by the rte_ipv4_frag_reassemble_packet()/rte_ipv6_frag_reassemble_packet. Functions. They either return a pointer to valid mbuf that contains reassembled packet, or NULL (if the packet can't be reassembled for some reason).

These functions are responsible for:

1. Search the Fragment Table for entry with packet's <IPv4 Source Address, IPv4 Destination Address, Packet ID>.
2. If the entry is found, then check if that entry already timed-out. If yes, then free all previously received fragments, and remove information about them from the entry.
3. If no entry with such key is found, then try to create a new one by one of two ways:
   (a) Use as empty entry.
   (b) Delete a timed-out entry, free mbufs associated with it mbufs and store a new entry with specified key in it.
4. Update the entry with new fragment information and check if a packet can be reassembled (the packet's entry contains all fragments).

(a) If yes, then, reassemble the packet, mark table's entry as empty and return the reassembled mbuf to the caller.

(b) If no, then return a NULL to the caller.

If at any stage of packet processing an error is encountered (e.g: can't insert new entry into the Fragment Table, or invalid/timed-out fragment), then the function will free all associated with the packet fragments, mark the table entry as invalid and return NULL to the caller.

Debug logging and Statistics Collection

The RTE_LIBRTE_IP_FRAG_TBL_STAT config macro controls statistics collection for the Fragment Table. This macro is not enabled by default.

The RTE_LIBRTE_IP_FRAG_DEBUG controls debug logging of IP fragments processing and reassembling. This macro is disabled by default. Note that while logging contains a lot of detailed information, it slows down packet processing and might cause the loss of a lot of packets.

4.17 Multi-process Support

In the DPDK, multi-process support is designed to allow a group of DPDK processes to work together in a simple transparent manner to perform packet processing, or other workloads, on Intel® architecture hardware. To support this functionality, a number of additions have been made to the core DPDK Environment Abstraction Layer (EAL).

The EAL has been modified to allow different types of DPDK processes to be spawned, each with different permissions on the hugepage memory used by the applications. For now, there are two types of process specified:

- primary processes, which can initialize and which have full permissions on shared memory
- secondary processes, which cannot initialize shared memory, but can attach to pre- initialized shared memory and create objects in it.

Standalone DPDK processes are primary processes, while secondary processes can only run alongside a primary process or after a primary process has already configured the hugepage shared memory for them.

To support these two process types, and other multi-process setups described later, two additional command-line parameters are available to the EAL:

- –proc-type: for specifying a given process instance as the primary or secondary DPDK instance
- –file-prefix: to allow processes that do not want to co-operate to have different memory regions

A number of example applications are provided that demonstrate how multiple DPDK processes can be used together. These are more fully documented in the “Multi- process Sample Application” chapter in the *DPDK Sample Application's User Guide*.

4.17.1 Memory Sharing

The key element in getting a multi-process application working using the DPDK is to ensure that memory resources are properly shared among the processes making up the multi-process application. Once there are blocks of shared memory available that can be accessed by multiple processes, then issues such as inter-process communication (IPC) becomes much simpler.

On application start-up in a primary or standalone process, the DPDK records to memory-mapped files the details of the memory configuration it is using - hugepages in use, the virtual addresses they are mapped at, the number of memory channels present, etc. When a secondary process is started, these files are read and the EAL recreates the same memory configuration in the secondary process so that all memory zones are shared between processes and all pointers to that memory are valid, and point to the same objects, in both processes.

Note: Refer to Section 23.3 "Multi-process Limitations" for details of how Linux kernel Address-Space Layout Randomization (ASLR) can affect memory sharing.

Fig. 4.35: Memory Sharing in the DPDK Multi-process Sample Application

The EAL also supports an auto-detection mode (set by EAL –proc-type=auto flag), whereby an DPDK process is started as a secondary instance if a primary instance is already running.

4.17.2 Deployment Models

Symmetric/Peer Processes

DPDK multi-process support can be used to create a set of peer processes where each process performs the same workload. This model is equivalent to having multiple threads each running the same main-loop function, as is done in most of the supplied DPDK sample applications. In this model, the first of the processes spawned should be spawned using the –proc-type=primary EAL flag, while all subsequent instances should be spawned using the –proc-type=secondary flag.

The simple_mp and symmetric_mp sample applications demonstrate this usage model. They are described in the "Multi-process Sample Application" chapter in the *DPDK Sample Application's User Guide*.

Asymmetric/Non-Peer Processes

An alternative deployment model that can be used for multi-process applications is to have a single primary process instance that acts as a load-balancer or server distributing received packets among worker or client threads, which are run as secondary processes. In this case, extensive use of rte_ring objects is made, which are located in shared hugepage memory.

The client_server_mp sample application shows this usage model. It is described in the "Multi-process Sample Application" chapter in the *DPDK Sample Application's User Guide*.

Running Multiple Independent DPDK Applications

In addition to the above scenarios involving multiple DPDK processes working together, it is possible to run multiple DPDK processes side-by-side, where those processes are all working independently. Support for this usage scenario is provided using the –file-prefix parameter to the EAL.

By default, the EAL creates hugepage files on each hugetlbfs filesystem using the rtemap_X filename, where X is in the range 0 to the maximum number of hugepages -1. Similarly, it creates shared configuration files, memory mapped in each process, using the /var/run/.rte_config filename, when run as root (or $HOME/.rte_config when run as a non-root user; if filesystem and device permissions are set up to allow this). The rte part of the filenames of each of the above is configurable using the file-prefix parameter.

In addition to specifying the file-prefix parameter, any DPDK applications that are to be run side-by-side must explicitly limit their memory use. This is done by passing the -m flag to each process to specify how much hugepage memory, in megabytes, each process can use (or passing –socket-mem to specify how much hugepage memory on each socket each process can use).

Note: Independent DPDK instances running side-by-side on a single machine cannot share any network ports. Any network ports being used by one process should be blacklisted in every other process.

Running Multiple Independent Groups of DPDK Applications

In the same way that it is possible to run independent DPDK applications side- by-side on a single system, this can be trivially extended to multi-process groups of DPDK applications running side-by-side. In this case, the secondary processes must use the same –file-prefix parameter as the primary process whose shared memory they are connecting to.

Note: All restrictions and issues with multiple independent DPDK processes running side-by-side apply in this usage scenario also.

4.17.3 Multi-process Limitations

There are a number of limitations to what can be done when running DPDK multi-process applications. Some of these are documented below:

- The multi-process feature requires that the exact same hugepage memory mappings be present in all applications. The Linux security feature - Address-Space Layout Randomization (ASLR) can interfere with this mapping, so it may be necessary to disable this feature in order to reliably run multi-process applications.

Warning: Disabling Address-Space Layout Randomization (ASLR) may have security implications, so it is recommended that it be disabled only when absolutely necessary, and only when the implications of this change have been understood.

- All DPDK processes running as a single application and using shared memory must have distinct coremask arguments. It is not possible to have a primary and secondary instance, or two secondary instances, using any of the same logical cores. Attempting to do so can cause corruption of memory pool caches, among other issues.
- The delivery of interrupts, such as Ethernet* device link status interrupts, do not work in secondary processes. All interrupts are triggered inside the primary process only. Any application needing interrupt notification in multiple processes should provide its own mechanism to transfer the interrupt information from the primary process to any secondary process that needs the information.
- The use of function pointers between multiple processes running based of different compiled binaries is not supported, since the location of a given function in one process may be different to its location in a second. This prevents the librte_hash library from behaving properly as in a multi-threaded instance, since it uses a pointer to the hash function internally.

To work around this issue, it is recommended that multi-process applications perform the hash calculations by directly calling the hashing function from the code and then using the rte_hash_add_with_hash()/rte_hash_lookup_with_hash() functions instead of the functions which do the hashing internally, such as rte_hash_add()/rte_hash_lookup().

- Depending upon the hardware in use, and the number of DPDK processes used, it may not be possible to have HPET timers available in each DPDK instance. The minimum number of HPET comparators available to Linux* userspace can be just a single comparator, which means that only the first, primary DPDK process instance can open and mmap /dev/hpet. If the number of required DPDK processes exceeds that of the number of available HPET comparators, the TSC (which is the default timer in this release) must be used as a time source across all processes instead of the HPET.

4.18 Kernel NIC Interface

The DPDK Kernel NIC Interface (KNI) allows userspace applications access to the Linux* control plane.

The benefits of using the DPDK KNI are:

- Faster than existing Linux TUN/TAP interfaces (by eliminating system calls and copy_to_user()/copy_from_user() operations.
- Allows management of DPDK ports using standard Linux net tools such as ethtool, ifconfig and tcpdump.
- Allows an interface with the kernel network stack.

The components of an application using the DPDK Kernel NIC Interface are shown in Fig. 4.36.

4.18.1 The DPDK KNI Kernel Module

The KNI kernel loadable module provides support for two types of devices:

- A Miscellaneous device (/dev/kni) that:
  - Creates net devices (via ioctl calls).

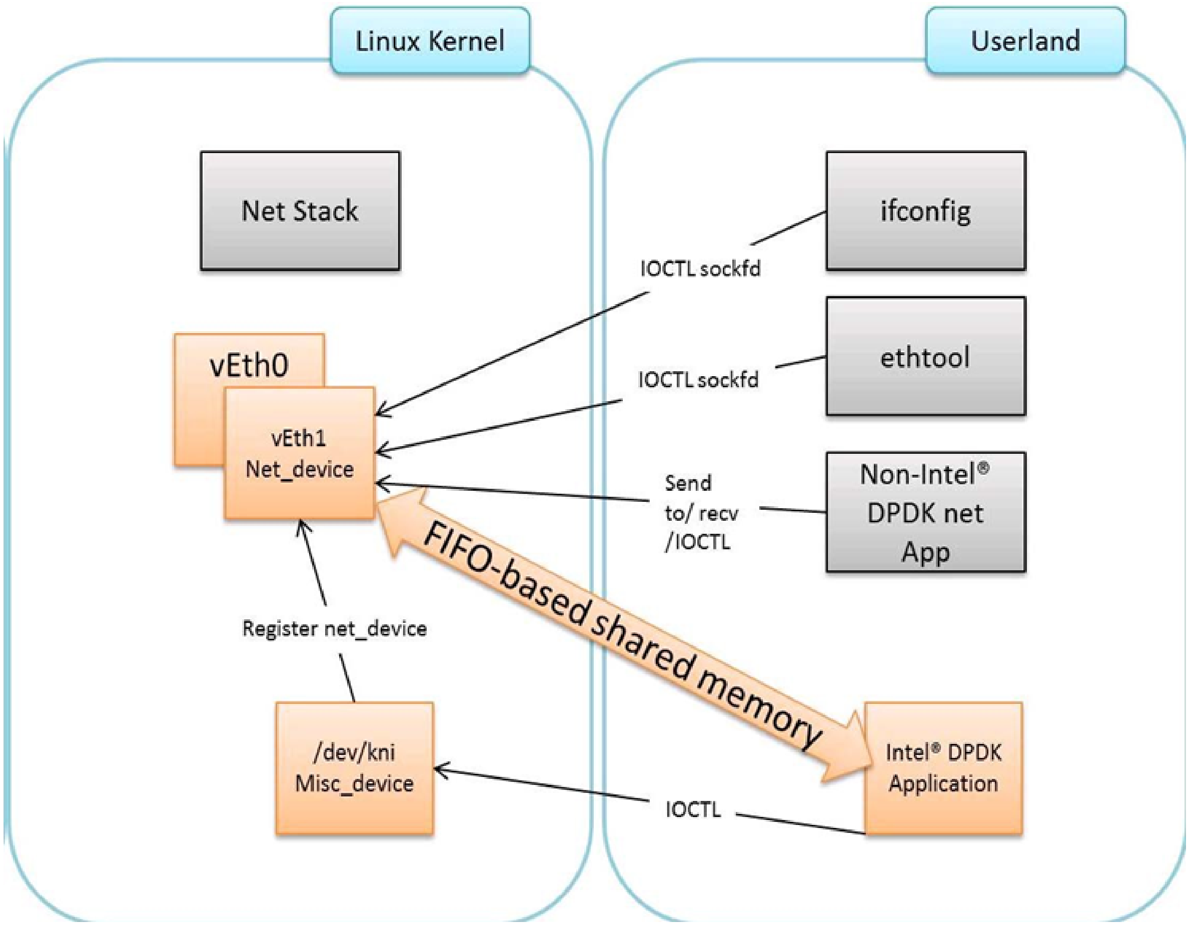


Fig. 4.36: Components of a DPDK KNI Application

  - Maintains a kernel thread context shared by all KNI instances (simulating the RX side of the net driver).
  - For single kernel thread mode, maintains a kernel thread context shared by all KNI instances (simulating the RX side of the net driver).
  - For multiple kernel thread mode, maintains a kernel thread context for each KNI instance (simulating the RX side of the new driver).
- Net device:
  - Net functionality provided by implementing several operations such as netdev_ops, header_ops, ethtool_ops that are defined by struct net_device, including support for DPDK mbufs and FIFOs.
  - The interface name is provided from userspace.
  - The MAC address can be the real NIC MAC address or random.

4.18.2 KNI Creation and Deletion

The KNI interfaces are created by a DPDK application dynamically. The interface name and FIFO details are provided by the application through an ioctl call using the rte_kni_device_info struct which contains:

- The interface name.
- Physical addresses of the corresponding memzones for the relevant FIFOs.
- Mbuf mempool details, both physical and virtual (to calculate the offset for mbuf pointers).
- PCI information.
- Core affinity.

Refer to rte_kni_common.h in the DPDK source code for more details.

The physical addresses will be re-mapped into the kernel address space and stored in separate KNI contexts.

The KNI interfaces can be deleted by a DPDK application dynamically after being created. Furthermore, all those KNI interfaces not deleted will be deleted on the release operation of the miscellaneous device (when the DPDK application is closed).

4.18.3 DPDK mbuf Flow

To minimize the amount of DPDK code running in kernel space, the mbuf mempool is managed in userspace only. The kernel module will be aware of mbufs, but all mbuf allocation and free operations will be handled by the DPDK application only.

Fig. 4.37 shows a typical scenario with packets sent in both directions.

4.18.4 Use Case: Ingress

On the DPDK RX side, the mbuf is allocated by the PMD in the RX thread context. This thread will enqueue the mbuf in the rx_q FIFO. The KNI thread will poll all KNI active devices for the rx_q. If an mbuf is dequeued, it will be converted to a sk_buff and sent to the net stack via

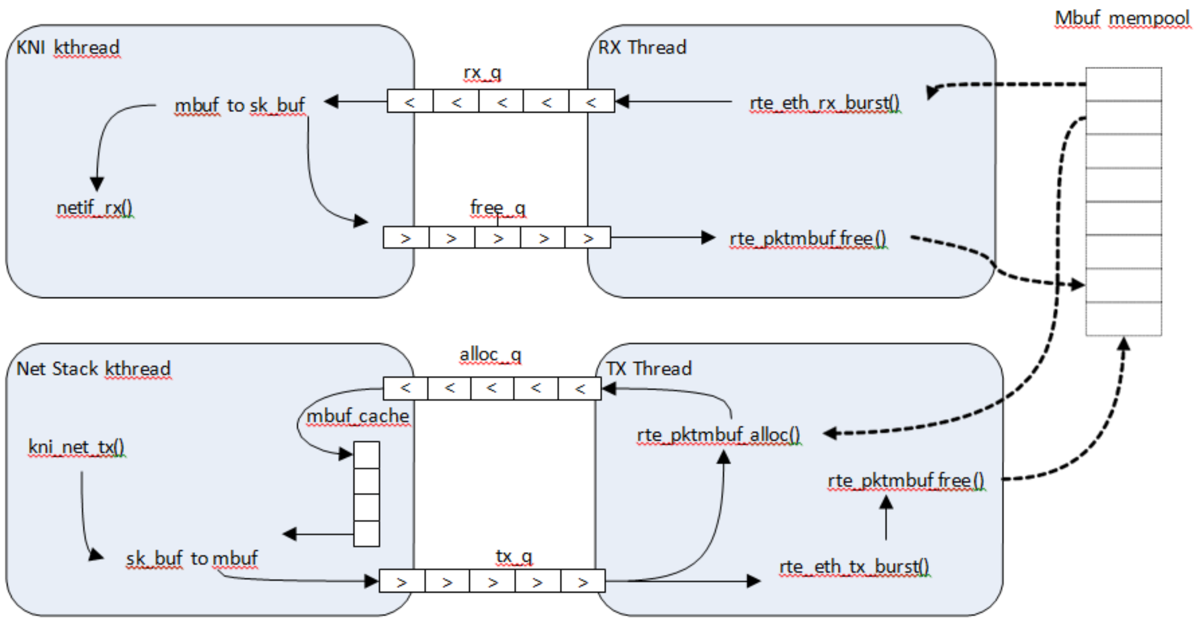


Fig. 4.37: Packet Flow via mbufs in the DPDK KNI

netif_rx(). The dequeued mbuf must be freed, so the same pointer is sent back in the free_q FIFO.

The RX thread, in the same main loop, polls this FIFO and frees the mbuf after dequeuing it.

4.18.5 Use Case: Egress

For packet egress the DPDK application must first enqueue several mbufs to create an mbuf cache on the kernel side.

The packet is received from the Linux net stack, by calling the kni_net_tx() callback. The mbuf is dequeued (without waiting due the cache) and filled with data from sk_buff. The sk_buff is then freed and the mbuf sent in the tx_q FIFO.

The DPDK TX thread dequeues the mbuf and sends it to the PMD (via rte_eth_tx_burst()). It then puts the mbuf back in the cache.

4.18.6 Ethtool

Ethtool is a Linux-specific tool with corresponding support in the kernel where each net device must register its own callbacks for the supported operations. The current implementation uses the igb/ixgbe modified Linux drivers for ethtool support. Ethtool is not supported in i40e and VMs (VF or EM devices).

4.18.7 Link state and MTU change

Link state and MTU change are network interface specific operations usually done via ifconfig. The request is initiated from the kernel side (in the context of the ifconfig process) and handled by the user space DPDK application. The application polls the request, calls the application handler and returns the response back into the kernel space.

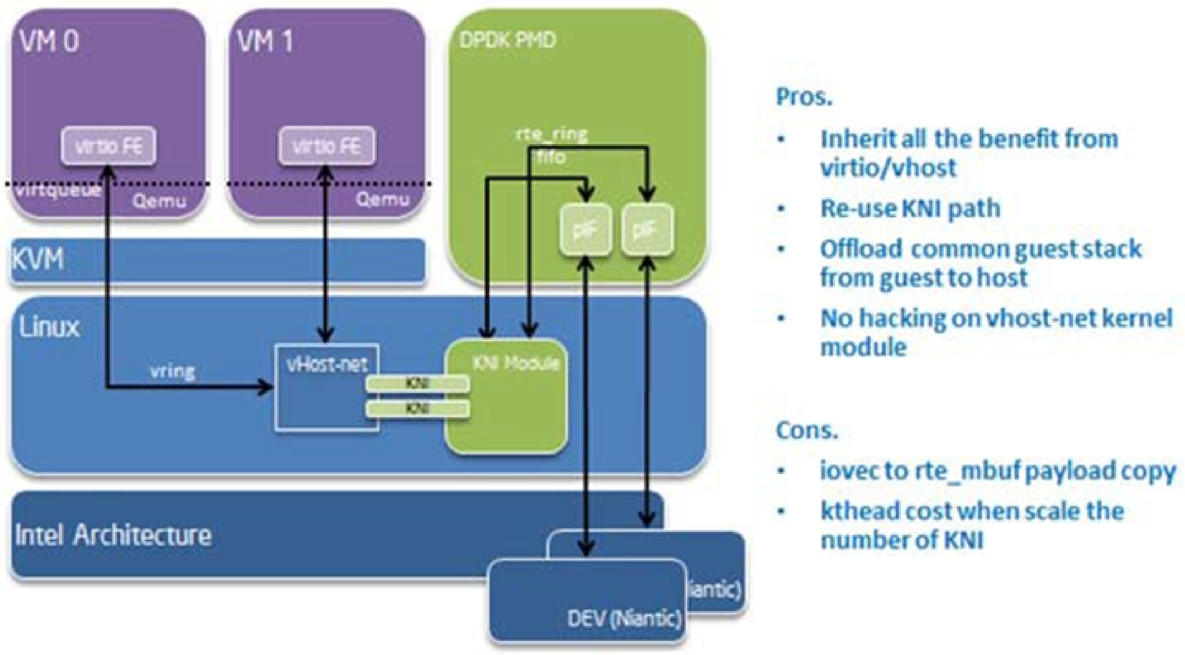
The application handlers can be registered upon interface creation or explicitly registered/unregistered in runtime. This provides flexibility in multiprocess scenarios (where the KNI is created in the primary process but the callbacks are handled in the secondary one). The constraint is that a single process can register and handle the requests.

4.18.8 KNI Working as a Kernel vHost Backend

vHost is a kernel module usually working as the backend of virtio (a para- virtualization driver framework) to accelerate the traffic from the guest to the host. The DPDK Kernel NIC interface provides the ability to hookup vHost traffic into userspace DPDK application. Together with the DPDK PMD virtio, it significantly improves the throughput between guest and host. In the scenario where DPDK is running as fast path in the host, kni-vhost is an efficient path for the traffic.

Overview

vHost-net has three kinds of real backend implementations. They are: 1) tap, 2) macvtap and 3) RAW socket. The main idea behind kni-vhost is making the KNI work as a RAW socket, attaching it as the backend instance of vHost-net. It is using the existing interface with vHost-net, so it does not require any kernel hacking, and is fully-compatible with the kernel vhost module. As vHost is still taking responsibility for communicating with the front-end virtio, it naturally supports both legacy virtio -net and the DPDK PMD virtio. There is a little penalty that comes from the non-polling mode of vhost. However, it scales throughput well when using KNI in multi-thread mode.



Pros.

- Inherit all the benefit from virtio/vhost
- Re-use KNI path
- Offload common guest stack from guest to host
- No hacking on vhost-net kernel module

Cons.

- iovec to rte_mbuf payload copy
- kthead cost when scale the number of KNI

Fig. 4.38: vHost-net Architecture Overview

Packet Flow

There is only a minor difference from the original KNI traffic flows. On transmit side, vhost kthread calls the RAW socket's ops sendmsg and it puts the packets into the KNI transmit FIFO. On the receive side, the kni kthread gets packets from the KNI receive FIFO, puts them into the queue of the raw socket, and wakes up the task in vhost kthread to begin receiving. All the packet copying, irrespective of whether it is on the transmit or receive side, happens in the context of vhost kthread. Every vhost-net device is exposed to a front end virtio device in the guest.

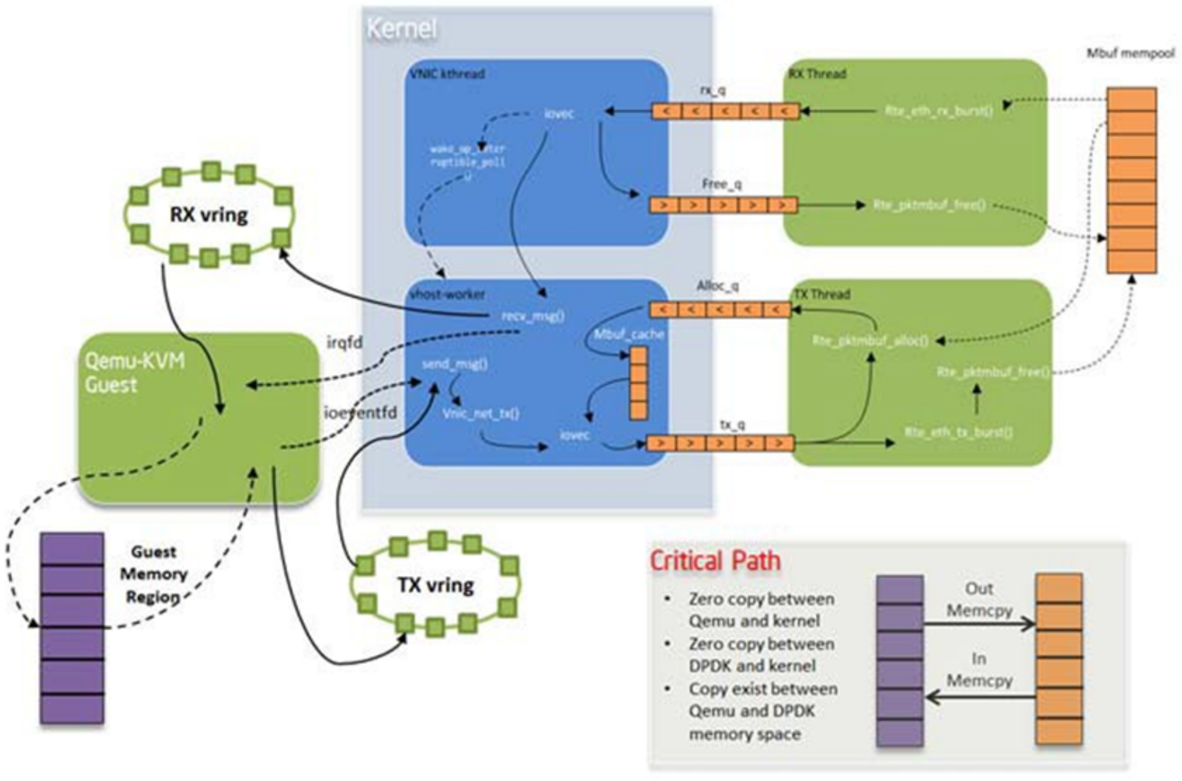


Fig. 4.39: KNI Traffic Flow

Sample Usage

Before starting to use KNI as the backend of vhost, the CONFIG_RTE_KNI_VHOST configuration option must be turned on. Otherwise, by default, KNI will not enable its backend support capability.

Of course, as a prerequisite, the vhost/vhost-net kernel CONFIG should be chosen before compiling the kernel.

1. Compile the DPDK and insert uio_pci_generic/igb_uio kernel modules as normal.
2. Insert the KNI kernel module:

   ```
   insmod ./rte_kni.ko
   ```

   If using KNI in multi-thread mode, use the following command line:

   ```
   insmod ./rte_kni.ko kthread_mode=multiple
   ```

3. Running the KNI sample application:

   ```
   examples/kni/build/app/kni -c -0xf0 -n 4 -- -p 0x3 -P --config="(0,4,6),(1,5,7)"
   ```

   This command runs the kni sample application with two physical ports. Each port pins two forwarding cores (ingress/egress) in user space.

4. Assign a raw socket to vhost-net during qemu-kvm startup. The DPDK does not provide a script to do this since it is easy for the user to customize. The following shows the key steps to launch qemu-kvm with kni-vhost:

   ```
   #!/bin/bash
   echo 1 > /sys/class/net/vEth0/sock_en
   fd=`cat /sys/class/net/vEth0/sock_fd`
   qemu-kvm \
   -name vm1 -cpu host -m 2048 -smp 1 -hda /opt/vm-fc16.img \
   -netdev tap,fd=$fd,id=hostnet1,vhost=on \
   -device virti-net-pci,netdev=hostnet1,id=net1,bus=pci.0,addr=0x4
   ```

It is simple to enable raw socket using sysfs sock_en and get raw socket fd using sock_fd under the KNI device node.

Then, using the qemu-kvm command with the -netdev option to assign such raw socket fd as vhost's backend.

Note: The key word tap must exist as qemu-kvm now only supports vhost with a tap backend, so here we cheat qemu-kvm by an existing fd.

Compatibility Configure Option

There is a CONFIG_RTE_KNI_VHOST_VNET_HDR_EN configuration option in DPDK configuration file. By default, it set to n, which means do not turn on the virtio net header, which is used to support additional features (such as, csum offload, vlan offload, generic-segmentation and so on), since the kni-vhost does not yet support those features.

Even if the option is turned on, kni-vhost will ignore the information that the header contains. When working with legacy virtio on the guest, it is better to turn off unsupported offload features using ethtool -K. Otherwise, there may be problems such as an incorrect L4 checksum error.

4.19 Thread Safety of DPDK Functions

The DPDK is comprised of several libraries. Some of the functions in these libraries can be safely called from multiple threads simultaneously, while others cannot. This section allows the developer to take these issues into account when building their own application.

The run-time environment of the DPDK is typically a single thread per logical core. In some cases, it is not only multi-threaded, but multi-process. Typically, it is best to avoid sharing data structures between threads and/or processes where possible. Where this is not possible, then the execution blocks must access the data in a thread- safe manner. Mechanisms such as atomics or locking can be used that will allow execution blocks to operate serially. However, this can have an effect on the performance of the application.

4.19.1 Fast-Path APIs

Applications operating in the data plane are performance sensitive but certain functions within those libraries may not be safe to call from multiple threads simultaneously. The hash, LPM and mempool libraries and RX/TX in the PMD are examples of this.

The hash and LPM libraries are, by design, thread unsafe in order to maintain performance. However, if required the developer can add layers on top of these libraries to provide thread safety. Locking is not needed in all situations, and in both the hash and LPM libraries, lookups of values can be performed in parallel in multiple threads. Adding, removing or modifying values, however, cannot be done in multiple threads without using locking when a single hash or LPM table is accessed. Another alternative to locking would be to create multiple instances of these tables allowing each thread its own copy.

The RX and TX of the PMD are the most critical aspects of a DPDK application and it is recommended that no locking be used as it will impact performance. Note, however, that these functions can safely be used from multiple threads when each thread is performing I/O on a different NIC queue. If multiple threads are to use the same hardware queue on the same NIC port, then locking, or some other form of mutual exclusion, is necessary.

The ring library is based on a lockless ring-buffer algorithm that maintains its original design for thread safety. Moreover, it provides high performance for either multi- or single-consumer/producer enqueue/dequeue operations. The mempool library is based on the DPDK lockless ring library and therefore is also multi-thread safe.

4.19.2 Performance Insensitive API

Outside of the performance sensitive areas described in Section 25.1, the DPDK provides a thread-safe API for most other libraries. For example, malloc and memzone functions are safe for use in multi-threaded and multi-process environments.

The setup and configuration of the PMD is not performance sensitive, but is not thread safe either. It is possible that the multiple read/writes during PMD setup and configuration could be corrupted in a multi-thread environment. Since this is not performance sensitive, the developer can choose to add their own layer to provide thread-safe setup and configuration. It is expected that, in most applications, the initial configuration of the network ports would be done by a single thread at startup.

4.19.3 Library Initialization

It is recommended that DPDK libraries are initialized in the main thread at application startup rather than subsequently in the forwarding threads. However, the DPDK performs checks to ensure that libraries are only initialized once. If initialization is attempted more than once, an error is returned.

In the multi-process case, the configuration information of shared memory will only be initialized by the master process. Thereafter, both master and secondary processes can allocate/release any objects of memory that finally rely on rte_malloc or memzones.

4.19.4 Interrupt Thread

The DPDK works almost entirely in Linux user space in polling mode. For certain infrequent operations, such as receiving a PMD link status change notification, callbacks may be called in an additional thread outside the main DPDK processing threads. These function callbacks should avoid manipulating DPDK objects that are also managed by the normal DPDK threads, and if they need to do so, it is up to the application to provide the appropriate locking or mutual exclusion restrictions around those objects.

4.20 Quality of Service (QoS) Framework

This chapter describes the DPDK Quality of Service (QoS) framework.

4.20.1 Packet Pipeline with QoS Support

An example of a complex packet processing pipeline with QoS support is shown in the following figure.

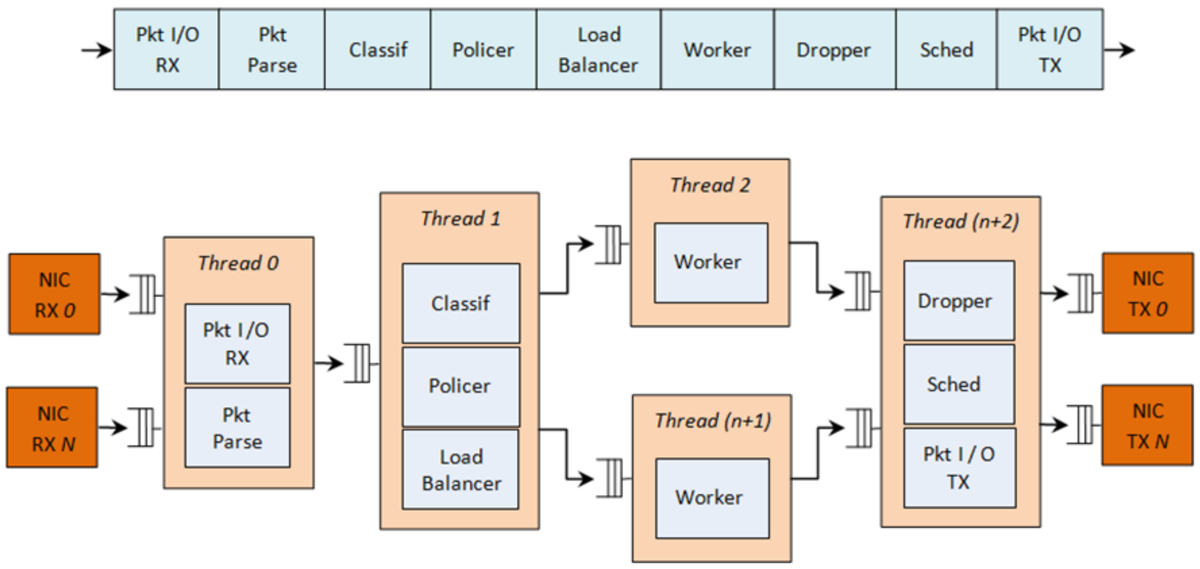


Fig. 4.40: Complex Packet Processing Pipeline with QoS Support

This pipeline can be built using reusable DPDK software libraries. The main blocks implementing QoS in this pipeline are: the policer, the dropper and the scheduler. A functional description of each block is provided in the following table.

Table 4.3: Packet Processing Pipeline Implementing QoS

#	Block	Functional Description
1	Packet I/O RX & TX	Packet reception/ transmission from/to multiple NIC ports. Poll mode drivers (PMDs) for Intel 1 GbE/10 GbE NICs.
2	Packet parser	Identify the protocol stack of the input packet. Check the integrity of the packet headers.
3	Flow classification	Map the input packet to one of the known traffic flows. Exact match table lookup using configurable hash function (jhash, CRC and so on) and bucket logic to handle collisions.
4	Policer	Packet metering using srTCM (RFC 2697) or trTCM (RFC2698) algorithms.
5	Load Balancer	Distribute the input packets to the application workers. Provide uniform load to each worker. Preserve the affinity of traffic flows to workers and the packet order within each flow.
6	Worker threads	Placeholders for the customer specific application workload (for example, IP stack and so on).
7	Dropper	Congestion management using the Random Early Detection (RED) algorithm (specified by the Sally Floyd - Van Jacobson paper) or Weighted RED (WRED). Drop packets based on the current scheduler queue load level and packet priority. When congestion is experienced, lower priority packets are dropped first.
8	Hierarchical Scheduler	5-level hierarchical scheduler (levels are: output port, subport, pipe, traffic class and queue) with thousands (typically 64K) leaf nodes (queues). Implements traffic shaping (for subport and pipe levels), strict priority (for traffic class level) and Weighted Round Robin (WRR) (for queues within each pipe traffic class).

The infrastructure blocks used throughout the packet processing pipeline are listed in the following table.

Table 4.4: Infrastructure Blocks Used by the Packet Processing Pipeline

#	Block	Functional Description
1	Buffer manager	Support for global buffer pools and private per-thread buffer caches.
2	Queue manager	Support for message passing between pipeline blocks.
3	Power saving	Support for power saving during low activity periods.

The mapping of pipeline blocks to CPU cores is configurable based on the performance level required by each specific application and the set of features enabled for each block. Some blocks might consume more than one CPU core (with each CPU core running a different instance of the same block on different input packets), while several other blocks could be mapped to the same CPU core.

4.20.2 Hierarchical Scheduler

The hierarchical scheduler block, when present, usually sits on the TX side just before the transmission stage. Its purpose is to prioritize the transmission of packets from different users and different traffic classes according to the policy specified by the Service Level Agreements (SLAs) of each network node.

Overview

The hierarchical scheduler block is similar to the traffic manager block used by network processors that typically implement per flow (or per group of flows) packet queuing and scheduling. It typically acts like a buffer that is able to temporarily store a large number of packets just before their transmission (enqueue operation); as the NIC TX is requesting more packets for transmission, these packets are later on removed and handed over to the NIC TX with the packet selection logic observing the predefined SLAs (dequeue operation).

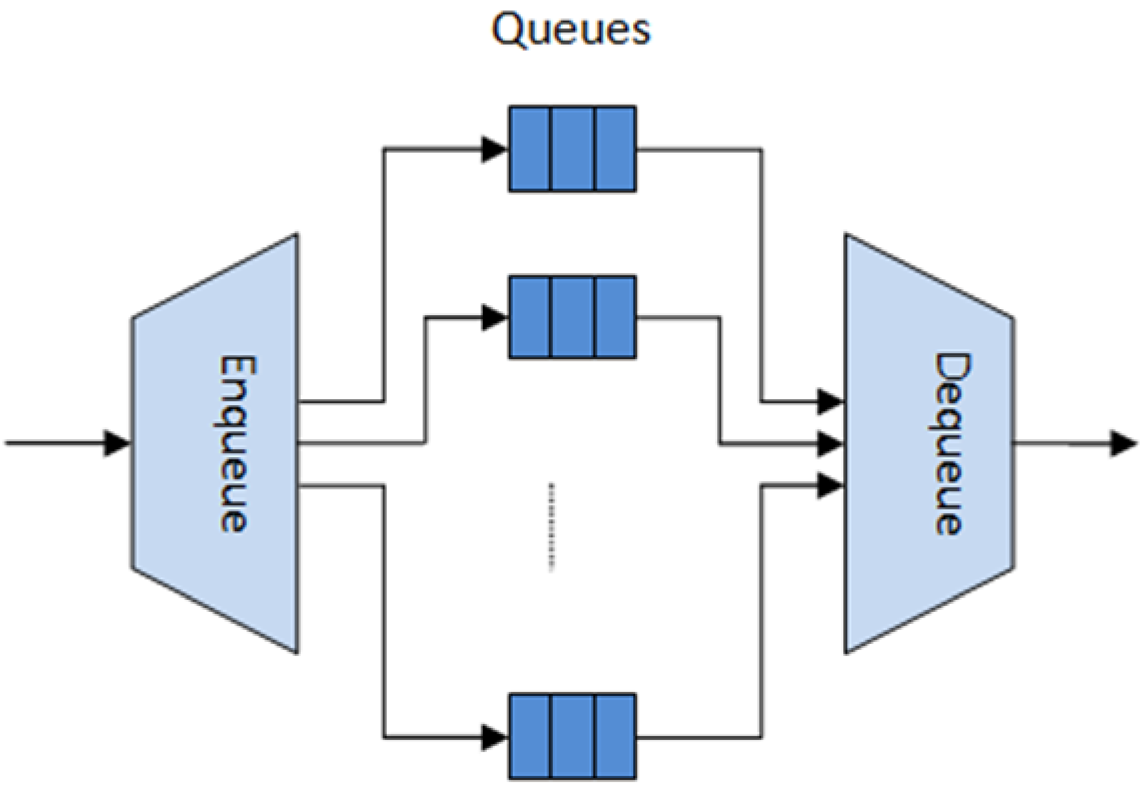


Fig. 4.41: Hierarchical Scheduler Block Internal Diagram

The hierarchical scheduler is optimized for a large number of packet queues. When only a small number of queues are needed, message passing queues should be used instead of this block. See Section 26.2.5 “Worst Case Scenarios for Performance” for a more detailed discussion.

Scheduling Hierarchy

The scheduling hierarchy is shown in Fig. 4.42. The first level of the hierarchy is the Ethernet TX port 1/10/40 GbE, with subsequent hierarchy levels defined as subport, pipe, traffic class and queue.

Typically, each subport represents a predefined group of users, while each pipe represents an individual user/subscriber. Each traffic class is the representation of a different traffic type with specific loss rate, delay and jitter requirements, such as voice, video or data transfers. Each queue hosts packets from one or multiple connections of the same type belonging to the same user.

The functionality of each hierarchical level is detailed in the following table.

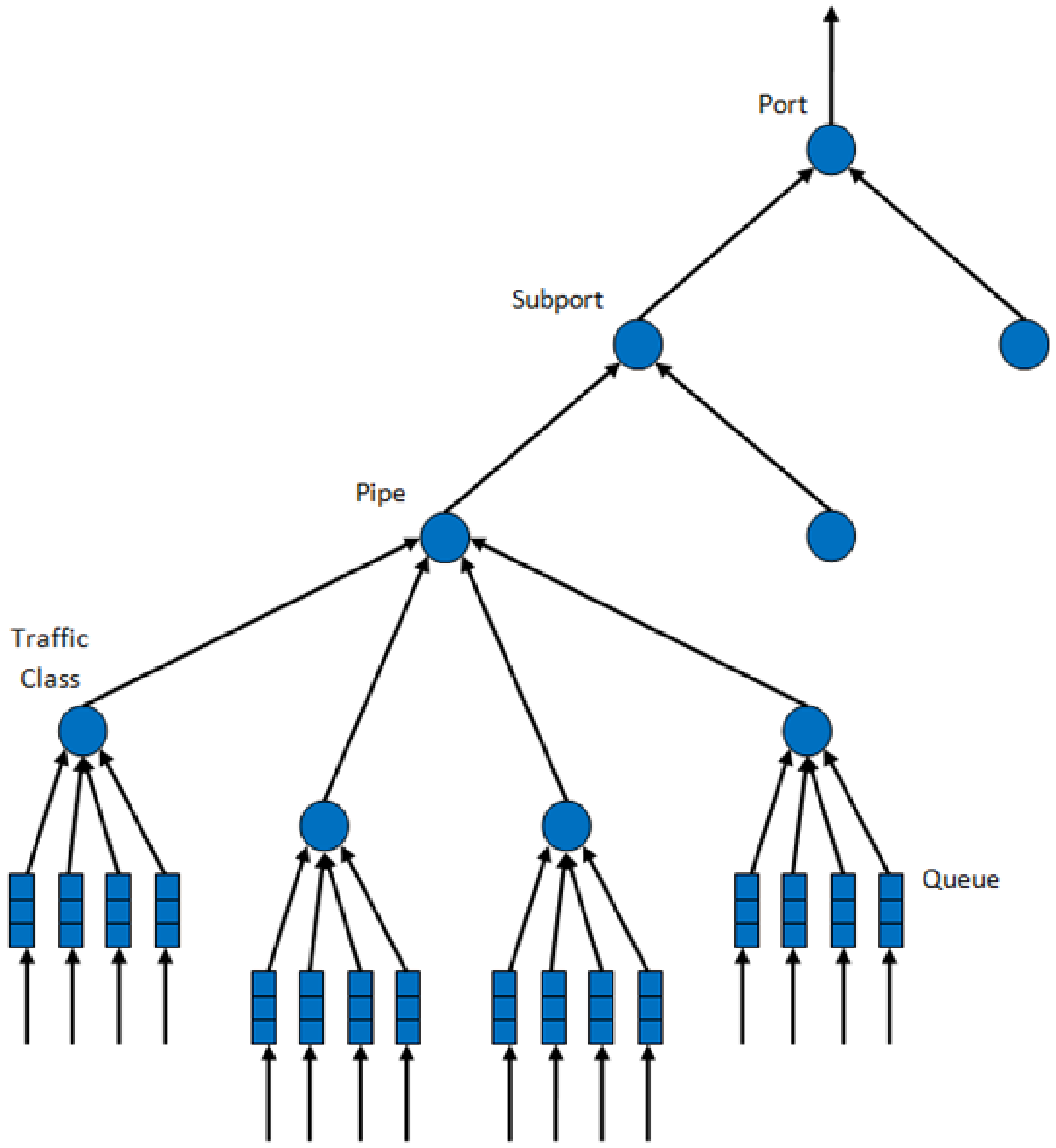


Fig. 4.42: Scheduling Hierarchy per Port

Table 4.5: Port Scheduling Hierarchy

#	Level	Siblings per Parent	Functional Description
1	Port	•	1. Output Ethernet port 1/10/40 GbE. 2. Multiple ports are scheduled in round robin order with all ports having equal priority.
2	Subport	Configurable (default: 8)	1. Traffic shaping using token bucket algorithm (one token bucket per subport). 2. Upper limit enforced per Traffic Class (TC) at the subport level. 3. Lower priority TCs able to reuse subport bandwidth currently unused by higher priority TCs.
3	Pipe	Configurable (default: 4K)	1. Traffic shaping using the token bucket algorithm (one token bucket per pipe.
4	Traffic Class (TC)	4	1. TCs of the same pipe handled in strict priority order. 2. Upper limit enforced per TC at the pipe level. 3. Lower priority TCs able to reuse pipe bandwidth currently unused by higher priority TCs. 4. When subport

Application Programming Interface (API)

Port Scheduler Configuration API

The rte_sched.h file contains configuration functions for port, subport and pipe.

Port Scheduler Enqueue API

The port scheduler enqueue API is very similar to the API of the DPDK PMD TX function.

```
int rte_sched_port_enqueue(struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_pkts
```

Port Scheduler Dequeue API

The port scheduler dequeue API is very similar to the API of the DPDK PMD RX function.

```
int rte_sched_port_dequeue(struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_pkts
```

Usage Example

```
/* File "application.c" */

#define N_PKTS_RX   64
#define N_PKTS_TX   48
#define NIC_RX_PORT 0
#define NIC_RX_QUEUE 0
#define NIC_TX_PORT 1
#define NIC_TX_QUEUE 0

struct rte_sched_port *port = NULL;
struct rte_mbuf *pkts_rx[N_PKTS_RX], *pkts_tx[N_PKTS_TX];
uint32_t n_pkts_rx, n_pkts_tx;

/* Initialization */

<initialization code>

/* Runtime */
while (1) {
    /* Read packets from NIC RX queue */

    n_pkts_rx = rte_eth_rx_burst(NIC_RX_PORT, NIC_RX_QUEUE, pkts_rx, N_PKTS_RX);

    /* Hierarchical scheduler enqueue */

    rte_sched_port_enqueue(port, pkts_rx, n_pkts_rx);

    /* Hierarchical scheduler dequeue */

    n_pkts_tx = rte_sched_port_dequeue(port, pkts_tx, N_PKTS_TX);

    /* Write packets to NIC TX queue */

    rte_eth_tx_burst(NIC_TX_PORT, NIC_TX_QUEUE, pkts_tx, n_pkts_tx);
}
```

Implementation

Internal Data Structures per Port

A schematic of the internal data structures in shown in with details in.

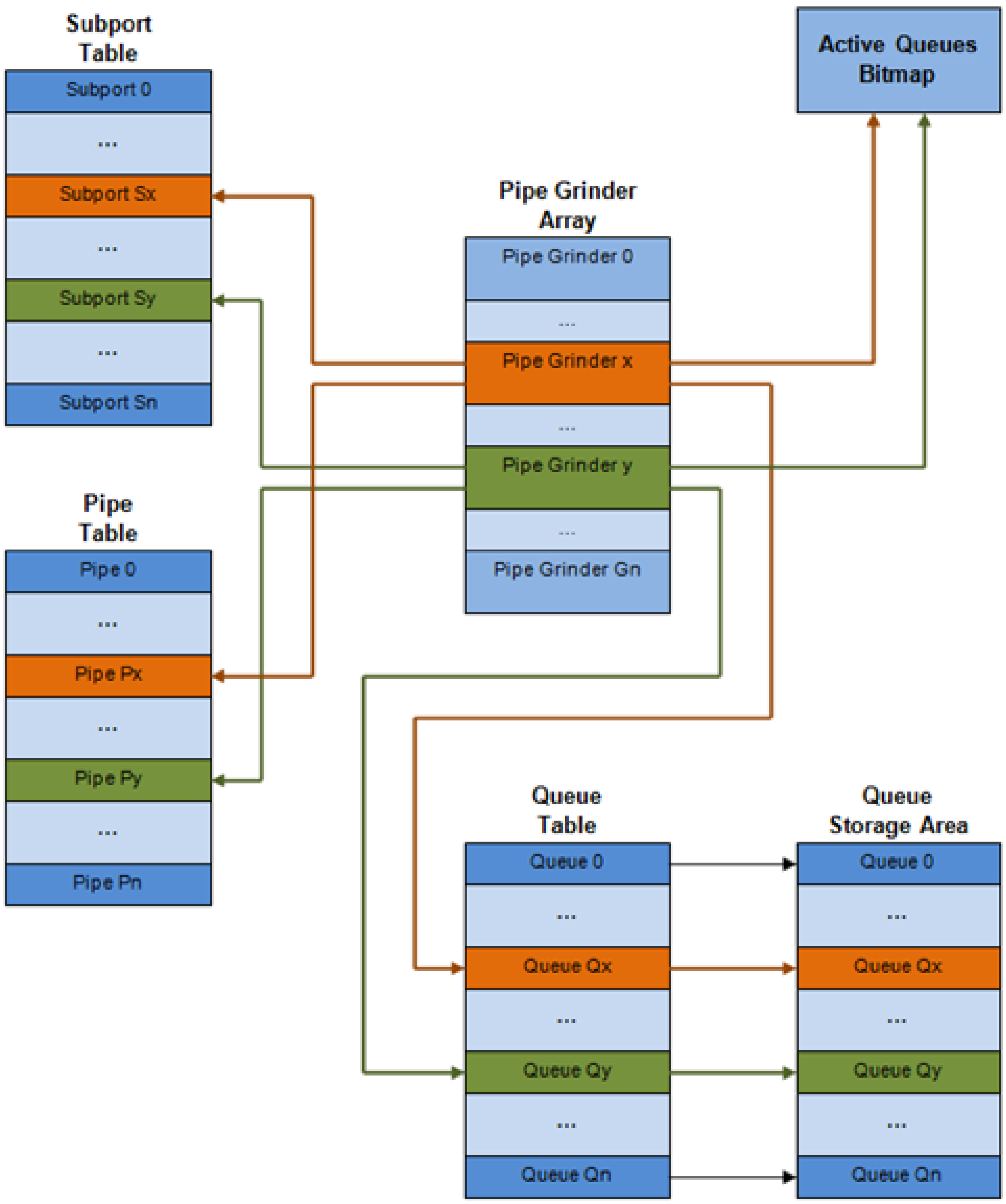


Fig. 4.43: Internal Data Structures per Port

Table 4.6: Scheduler Internal Data Structures per Port

#	Data structure	Size (bytes)	# per port	Access type		Description
				Enq	Deq	
1	Subport table entry	64	# subports per port	•	Rd, Wr	Persistent subport data (credits, etc).
2	Pipe table entry	64	# pipes per port	•	Rd, Wr	Persistent data for pipe, its TCs and its queues (credits, etc) that is updated during run-time. The pipe configuration parameters do not change during run-time. The same pipe configuration parameters are shared by multiple pipes, therefore they are not part of pipe table entry.
3	Queue table entry	4	#queues per port	Rd, Wr	Rd, Wr	Persistent queue data (read and write pointers). The queue size is the same per TC for all queues, allowing the queue base address to be computed using a fast formula, so these two parameters are not part of queue table entry. The queue table entries

Multicore Scaling Strategy

The multicore scaling strategy is:

1. Running different physical ports on different threads. The enqueue and dequeue of the same port are run by the same thread.
2. Splitting the same physical port to different threads by running different sets of subports of the same physical port (virtual ports) on different threads. Similarly, a subport can be split into multiple subports that are each run by a different thread. The enqueue and dequeue of the same port are run by the same thread. This is only required if, for performance reasons, it is not possible to handle a full port with a single core.

Enqueue and Dequeue for the Same Output Port Running enqueue and dequeue operations for the same output port from different cores is likely to cause significant impact on scheduler's performance and it is therefore not recommended.

The port enqueue and dequeue operations share access to the following data structures:

1. Packet descriptors
2. Queue table
3. Queue storage area
4. Bitmap of active queues

The expected drop in performance is due to:

1. Need to make the queue and bitmap operations thread safe, which requires either using locking primitives for access serialization (for example, spinlocks/ semaphores) or using atomic primitives for lockless access (for example, Test and Set, Compare And Swap, an so on). The impact is much higher in the former case.
2. Ping-pong of cache lines storing the shared data structures between the cache hierarchies of the two cores (done transparently by the MESI protocol cache coherency CPU hardware).

Therefore, the scheduler enqueue and dequeue operations have to be run from the same thread, which allows the queues and the bitmap operations to be non-thread safe and keeps the scheduler data structures internal to the same core.

Performance Scaling Scaling up the number of NIC ports simply requires a proportional increase in the number of CPU cores to be used for traffic scheduling.

Enqueue Pipeline

The sequence of steps per packet:

1. *Access* the mbuf to read the data fields required to identify the destination queue for the packet. These fields are: port, subport, traffic class and queue within traffic class, and are typically set by the classification stage.
2. *Access* the queue structure to identify the write location in the queue array. If the queue is full, then the packet is discarded.

3. *Access* the queue array location to store the packet (i.e. write the mbuf pointer).

It should be noted the strong data dependency between these steps, as steps 2 and 3 cannot start before the result from steps 1 and 2 becomes available, which prevents the processor out of order execution engine to provide any significant performance optimizations.

Given the high rate of input packets and the large amount of queues, it is expected that the data structures accessed to enqueue the current packet are not present in the L1 or L2 data cache of the current core, thus the above 3 memory accesses would result (on average) in L1 and L2 data cache misses. A number of 3 L1/L2 cache misses per packet is not acceptable for performance reasons.

The workaround is to prefetch the required data structures in advance. The prefetch operation has an execution latency during which the processor should not attempt to access the data structure currently under prefetch, so the processor should execute other work. The only other work available is to execute different stages of the enqueue sequence of operations on other input packets, thus resulting in a pipelined implementation for the enqueue operation.

Fig. 4.44 illustrates a pipelined implementation for the enqueue operation with 4 pipeline stages and each stage executing 2 different input packets. No input packet can be part of more than one pipeline stage at a given time.

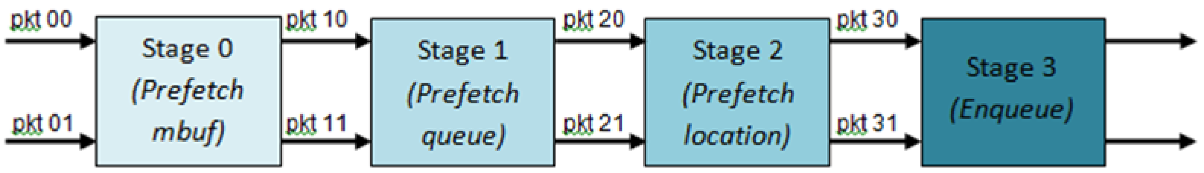


Fig. 4.44: Prefetch Pipeline for the Hierarchical Scheduler Enqueue Operation

The congestion management scheme implemented by the enqueue pipeline described above is very basic: packets are enqueued until a specific queue becomes full, then all the packets destined to the same queue are dropped until packets are consumed (by the dequeue operation). This can be improved by enabling RED/WRED as part of the enqueue pipeline which looks at the queue occupancy and packet priority in order to yield the enqueue/drop decision for a specific packet (as opposed to enqueuing all packets / dropping all packets indiscriminately).

Dequeue State Machine

The sequence of steps to schedule the next packet from the current pipe is:

1. Identify the next active pipe using the bitmap scan operation, *prefetch* pipe.
2. *Read* pipe data structure. Update the credits for the current pipe and its subport. Identify the first active traffic class within the current pipe, select the next queue using WRR, *prefetch* queue pointers for all the 16 queues of the current pipe.
3. *Read* next element from the current WRR queue and *prefetch* its packet descriptor.
4. *Read* the packet length from the packet descriptor (mbuf structure). Based on the packet length and the available credits (of current pipe, pipe traffic class, subport and subport traffic class), take the go/no go scheduling decision for the current packet.

To avoid the cache misses, the above data structures (pipe, queue, queue array, mbufs) are prefetched in advance of being accessed. The strategy of hiding the latency of the prefetch operations is to switch from the current pipe (in grinder A) to another pipe (in grinder B) immediately after a prefetch is issued for the current pipe. This gives enough time to the prefetch operation to complete before the execution switches back to this pipe (in grinder A).

The dequeue pipe state machine exploits the data presence into the processor cache, therefore it tries to send as many packets from the same pipe TC and pipe as possible (up to the available packets and credits) before moving to the next active TC from the same pipe (if any) or to another active pipe.

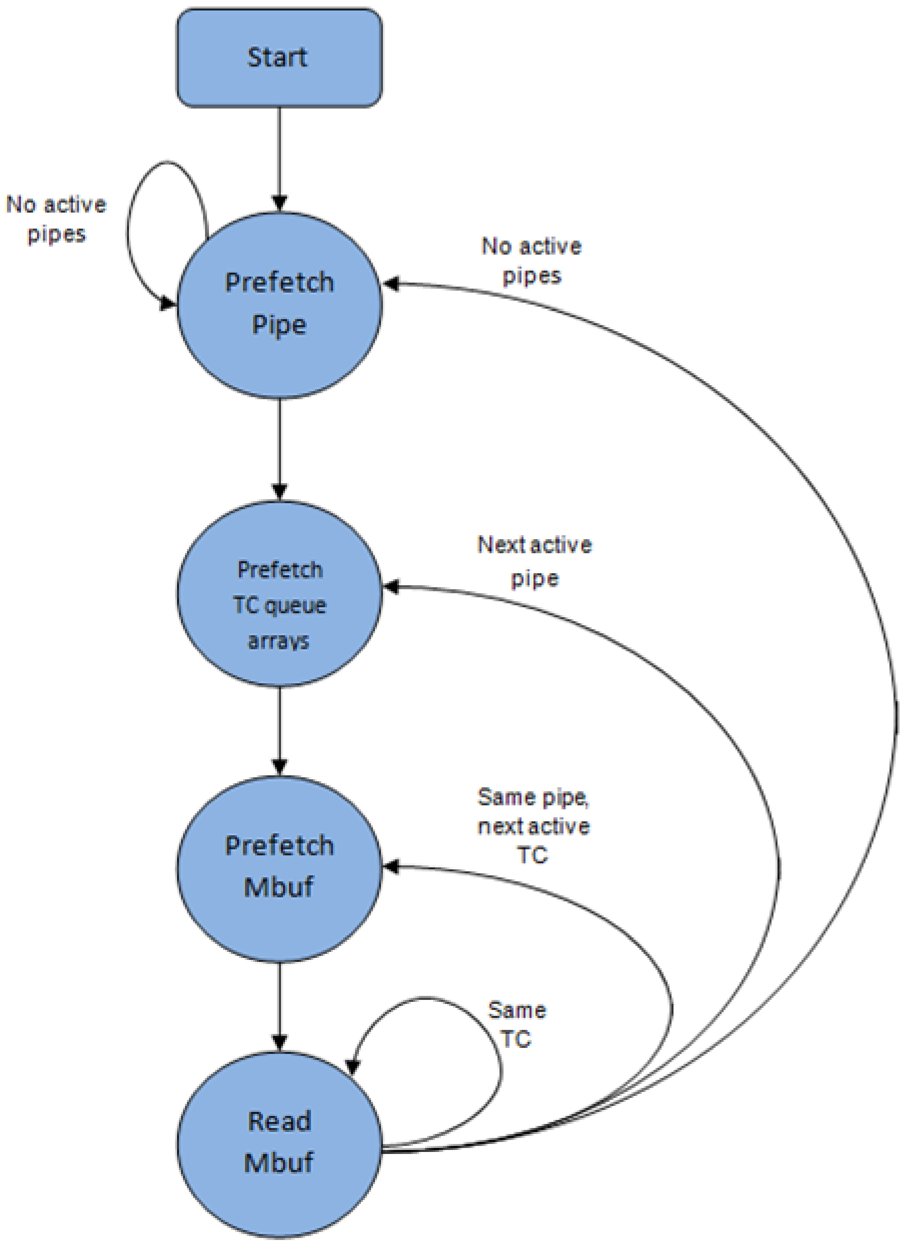


Fig. 4.45: Pipe Prefetch State Machine for the Hierarchical Scheduler Dequeue Operation

Timing and Synchronization

The output port is modeled as a conveyor belt of byte slots that need to be filled by the scheduler with data for transmission. For 10 GbE, there are 1.25 billion byte slots that need to be filled by the port scheduler every second. If the scheduler is not fast enough to fill the slots, provided that enough packets and credits exist, then some slots will be left unused and bandwidth will be wasted.

In principle, the hierarchical scheduler dequeue operation should be triggered by NIC TX. Usually, once the occupancy of the NIC TX input queue drops below a predefined threshold, the port scheduler is woken up (interrupt based or polling based, by continuously monitoring the queue occupancy) to push more packets into the queue.

Internal Time Reference The scheduler needs to keep track of time advancement for the credit logic, which requires credit updates based on time (for example, subport and pipe traffic shaping, traffic class upper limit enforcement, and so on).

Every time the scheduler decides to send a packet out to the NIC TX for transmission, the scheduler will increment its internal time reference accordingly. Therefore, it is convenient to keep the internal time reference in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium. This way, as a packet is scheduled for transmission, the time is incremented with (n + h), where n is the packet length in bytes and h is the number of framing overhead bytes per packet.

Internal Time Reference Re-synchronization The scheduler needs to align its internal time reference to the pace of the port conveyor belt. The reason is to make sure that the scheduler does not feed the NIC TX with more bytes than the line rate of the physical medium in order to prevent packet drop (by the scheduler, due to the NIC TX input queue being full, or later on, internally by the NIC TX).

The scheduler reads the current time on every dequeue invocation. The CPU time stamp can be obtained by reading either the Time Stamp Counter (TSC) register or the High Precision Event Timer (HPET) register. The current CPU time stamp is converted from number of CPU clocks to number of bytes: *time_bytes = time_cycles / cycles_per_byte, where cycles_per_byte* is the amount of CPU cycles that is equivalent to the transmission time for one byte on the wire (e.g. for a CPU frequency of 2 GHz and a 10GbE port,*cycles_per_byte = 1.6*).

The scheduler maintains an internal time reference of the NIC time. Whenever a packet is scheduled, the NIC time is incremented with the packet length (including framing overhead). On every dequeue invocation, the scheduler checks its internal reference of the NIC time against the current time:

1. If NIC time is in the future (NIC time >= current time), no adjustment of NIC time is needed. This means that scheduler is able to schedule NIC packets before the NIC actually needs those packets, so the NIC TX is well supplied with packets;
2. If NIC time is in the past (NIC time < current time), then NIC time should be adjusted by setting it to the current time. This means that the scheduler is not able to keep up with the speed of the NIC byte conveyor belt, so NIC bandwidth is wasted due to poor packet supply to the NIC TX.

Scheduler Accuracy and Granularity The scheduler round trip delay (SRTD) is the time (number of CPU cycles) between two consecutive examinations of the same pipe by the scheduler.

To keep up with the output port (that is, avoid bandwidth loss), the scheduler should be able to schedule n packets faster than the same n packets are transmitted by NIC TX.

The scheduler needs to keep up with the rate of each individual pipe, as configured for the pipe token bucket, assuming that no port oversubscription is taking place. This means that the size of the pipe token bucket should be set high enough to prevent it from overflowing due to big SRTD, as this would result in credit loss (and therefore bandwidth loss) for the pipe.

Credit Logic

Scheduling Decision The scheduling decision to send next packet from (subport S, pipe P, traffic class TC, queue Q) is favorable (packet is sent) when all the conditions below are met:

- Pipe P of subport S is currently selected by one of the port grinders;
- Traffic class TC is the highest priority active traffic class of pipe P;
- Queue Q is the next queue selected by WRR within traffic class TC of pipe P;
- Subport S has enough credits to send the packet;
- Subport S has enough credits for traffic class TC to send the packet;
- Pipe P has enough credits to send the packet;
- Pipe P has enough credits for traffic class TC to send the packet.

If all the above conditions are met, then the packet is selected for transmission and the necessary credits are subtracted from subport S, subport S traffic class TC, pipe P, pipe P traffic class TC.

Framing Overhead As the greatest common divisor for all packet lengths is one byte, the unit of credit is selected as one byte. The number of credits required for the transmission of a packet of n bytes is equal to (n+h), where h is equal to the number of framing overhead bytes per packet.

Table 4.7: Ethernet Frame Overhead Fields

#	Packet field	Length (bytes)	Comments
1	Preamble	7	
2	Start of Frame Delimiter (SFD)	1	
3	Frame Check Sequence (FCS)	4	Considered overhead only if not included in the mbuf packet length field.
4	Inter Frame Gap (IFG)	12	
5	Total	24	

Traffic Shaping The traffic shaping for subport and pipe is implemented using a token bucket per subport/per pipe. Each token bucket is implemented using one saturated counter that keeps track of the number of available credits.

The token bucket generic parameters and operations are presented in Table 4.8 and Table 4.9.

Table 4.8: Token Bucket Generic Operations

#	Token Bucket Parameter	Unit	Description
1	bucket_rate	Credits per second	Rate of adding credits to the bucket.
2	bucket_size	Credits	Max number of credits that can be stored in the bucket.

Table 4.9: Token Bucket Generic Parameters

#	Token Bucket Operation	Description
1	Initialization	Bucket set to a predefined value, e.g. zero or half of the bucket size.
2	Credit update	Credits are added to the bucket on top of existing ones, either periodically or on demand, based on the bucket_rate. Credits cannot exceed the upper limit defined by the bucket_size, so any credits to be added to the bucket while the bucket is full are dropped.
3	Credit consumption	As result of packet scheduling, the necessary number of credits is removed from the bucket. The packet can only be sent if enough credits are in the bucket to send the full packet (packet bytes and framing overhead for the packet).

To implement the token bucket generic operations described above, the current design uses the persistent data structure presented in Table 4.10, while the implementation of the token bucket operations is described in Table 4.11.

Table 4.10: Token Bucket Persistent Data Structure

#	Token bucket field	Unit	Description
1	tb_time	Bytes	Time of the last credit update. Measured in bytes instead of seconds or CPU cycles for ease of credit consumption operation (as the current time is also maintained in bytes). See Section 26.2.4.5.1 “Internal Time Reference” for an explanation of why the time is maintained in byte units.
2	tb_period	Bytes	Time period that should elapse since the last credit update in order for the bucket to be awarded tb_credits_per_period worth or credits.
3	tb_credits_per_period	Bytes	Credit allowance per tb_period.
4	tb_size	Bytes	Bucket size, i.e. upper limit for the tb_credits.
5	tb_credits	Bytes	Number of credits currently in the bucket.

The bucket rate (in bytes per second) can be computed with the following formula:

*bucket_rate = (tb_credits_per_period / tb_period) * r*

where, r = port line rate (in bytes per second).

Table 4.11: Token Bucket Operations

#	Token bucket operation	Description
1	Initialization	*tb_credits = 0; or tb_credits = tb_size / 2;*
2	Credit update	Credit update options: • Every time a packet is sent for a port, update the credits of all the the subports and pipes of that port. Not feasible. • Every time a packet is sent, update the credits for the pipe and subport. Very accurate, but not needed (a lot of calculations). • Every time a pipe is selected (that is, picked by one of the grinders), update the credits for the pipe and its subport. The current implementation is using option 3. According to Section 26.2.4.4 “Dequeue State Machine”, the pipe and subport credits are updated every time a pipe is selected by the dequeue process before the pipe and subport credits are actually used. The implementation uses a tradeoff between accuracy and speed by updating the bucket credits only when at least a full *tb_period* has elapsed since the last update. • Full accuracy can be achieved by selecting the value for *tb_period* for which *tb_credits_per_period = 1*. • When full accuracy is not required, better performance is achieved by setting *tb_credits* to a larger value. Update operations: • n_periods = (time - tb_time) / tb_period; • tb_credits += n_periods * tb_credits_per_period; • tb_credits = min(tb_credits, tb_size); • tb_time += n_periods * tb_period;

Traffic Classes

Implementation of Strict Priority Scheduling Strict priority scheduling of traffic classes within the same pipe is implemented by the pipe dequeue state machine, which selects the queues in ascending order. Therefore, queues 0..3 (associated with TC 0, highest priority TC) are handled before queues 4..7 (TC 1, lower priority than TC 0), which are handled before queues 8..11 (TC 2), which are handled before queues 12..15 (TC 3, lowest priority TC).

Upper Limit Enforcement The traffic classes at the pipe and subport levels are not traffic shaped, so there is no token bucket maintained in this context. The upper limit for the traffic classes at the subport and pipe levels is enforced by periodically refilling the subport / pipe traffic class credit counter, out of which credits are consumed every time a packet is scheduled for that subport / pipe, as described in Table 4.12 and Table 4.13.

Table 4.12: Subport/Pipe Traffic Class Upper Limit Enforcement Persistent Data Structure

#	Subport or pipe field	Unit	Description
1	tc_time	Bytes	Time of the next update (upper limit refill) for the 4 TCs of the current subport / pipe. See Section 26.2.4.5.1, "Internal Time Reference" for the explanation of why the time is maintained in byte units.
2	tc_period	Bytes	Time between two consecutive updates for the 4 TCs of the current subport / pipe. This is expected to be many times bigger than the typical value of the token bucket tb_period.
3	tc_credits_per_period	Bytes	Upper limit for the number of credits allowed to be consumed by the current TC during each enforcement period tc_period.
4	tc_credits	Bytes	Current upper limit for the number of credits that can be consumed by the current traffic class for the remainder of the current enforcement period.

Table 4.13: Subport/Pipe Traffic Class Upper Limit Enforcement Operations

#	Traffic Class Operation	Description

1	Initialization	tc_credits = tc_credits_per_period;
		tc_time = tc_period;

2	Credit update	Update operations:
		if (time >= tc_time) {
		tc_credits = tc_credits_per_period;
		tc_time = time + tc_period;
		}

3	Credit consumption (on packet scheduling)	As result of packet scheduling, the TC limit is decreased with the necessary number of credits. The packet can only be sent if enough credits are currently available in the TC limit to send the full packet (packet bytes and framing overhead for the packet).
		Scheduling operations:
		pkt_credits = pk_len + frame_overhead;
		if (tc_credits >= pkt_credits) {tc_credits -= pkt_credits;}

Weighted Round Robin (WRR) The evolution of the WRR design solution from simple to complex is shown in Table 4.14.

Table 4.14: Weighted Round Robin (WRR)

#	All Queues Active?	Equal Weights for All Queues?	All Packets Equal?	Strategy
1	Yes	Yes	Yes	**Byte level round robin** *Next queue* queue #i, i = *(i + 1) % n*
2	Yes	Yes	No	**Packet level round robin** Consuming one byte from queue #i requires consuming exactly one token for queue #i. T(i) = Accumulated number of tokens previously consumed from queue #i. Every time a packet is consumed from queue #i, T(i) is updated as: T(i) += *pkt_len*. *Next queue* : queue with the smallest T.
3	Yes	No	No	**Packet level weighted round robin** This case can be reduced to the previous case by introducing a cost per byte that is different for each queue. Queues with lower weights have a higher cost per byte. This way, it is still meaningful to compare the consumption amongst different queues in order to select the next queue. w(i) = Weight of queue #i t(i) = Tokens per byte for queue #i, defined as the

Subport Traffic Class Oversubscription

Problem Statement Oversubscription for subport traffic class X is a configuration-time event that occurs when more bandwidth is allocated for traffic class X at the level of subport member pipes than allocated for the same traffic class at the parent subport level.

The existence of the oversubscription for a specific subport and traffic class is solely the result of pipe and subport-level configuration as opposed to being created due to dynamic evolution of the traffic load at run-time (as congestion is).

When the overall demand for traffic class X for the current subport is low, the existence of the oversubscription condition does not represent a problem, as demand for traffic class X is completely satisfied for all member pipes. However, this can no longer be achieved when the aggregated demand for traffic class X for all subport member pipes exceeds the limit configured at the subport level.

Solution Space summarizes some of the possible approaches for handling this problem, with the third approach selected for implementation.

Table 4.15: Subport Traffic Class Oversubscription

No.	Approach	Description
1	Don't care	First come, first served. This approach is not fair amongst subport member pipes, as pipes that are served first will use up as much bandwidth for TC X as they need, while pipes that are served later will receive poor service due to bandwidth for TC X at the subport level being scarce.
2	Scale down all pipes	All pipes within the subport have their bandwidth limit for TC X scaled down by the same factor. This approach is not fair among subport member pipes, as the low end pipes (that is, pipes configured with low bandwidth) can potentially experience severe service degradation that might render their service unusable (if available bandwidth for these pipes drops below the minimum requirements for a workable service), while the service degradation for high end pipes might not be noticeable at all.
3	Cap the high demand pipes	Each subport member pipe receives an equal share of the bandwidth available at run-time for TC X at the subport level. Any bandwidth left unused by the low-demand pipes is redistributed in equal portions to the high-demand pipes. This way, the high-demand pipes are truncated while the low-demand pipes are not impacted.

Typically, the subport TC oversubscription feature is enabled only for the lowest priority traffic class (TC 3), which is typically used for best effort traffic, with the management plane preventing this condition from occurring for the other (higher priority) traffic classes.

To ease implementation, it is also assumed that the upper limit for subport TC 3 is set to 100% of the subport rate, and that the upper limit for pipe TC 3 is set to 100% of pipe rate for all subport member pipes.

Implementation Overview The algorithm computes a watermark, which is periodically updated based on the current demand experienced by the subport member pipes, whose purpose is to limit the amount of traffic that each pipe is allowed to send for TC 3. The watermark is computed at the subport level at the beginning of each traffic class upper limit enforcement period and the same value is used by all the subport member pipes throughout the current enforcement period. illustrates how the watermark computed as subport level at the beginning of each period is propagated to all subport member pipes.

At the beginning of the current enforcement period (which coincides with the end of the previous enforcement period), the value of the watermark is adjusted based on the amount of bandwidth allocated to TC 3 at the beginning of the previous period that was not left unused by the subport member pipes at the end of the previous period.

If there was subport TC 3 bandwidth left unused, the value of the watermark for the current period is increased to encourage the subport member pipes to consume more bandwidth. Otherwise, the value of the watermark is decreased to enforce equality of bandwidth consumption among subport member pipes for TC 3.

The increase or decrease in the watermark value is done in small increments, so several enforcement periods might be required to reach the equilibrium state. This state can change at any moment due to variations in the demand experienced by the subport member pipes for TC 3, for example, as a result of demand increase (when the watermark needs to be lowered) or demand decrease (when the watermark needs to be increased).

When demand is low, the watermark is set high to prevent it from impeding the subport member pipes from consuming more bandwidth. The highest value for the watermark is picked as the highest rate configured for a subport member pipe. Table 4.16 and Table 4.17 illustrates the watermark operation.

Table 4.16: Watermark Propagation from Subport Level to Member Pipes at the Beginning of Each Traffic Class Upper Limit Enforcement Period

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level**: subport_period_id= 0 **Pipe level**: pipe_period_id = 0

2	Credit update	**Subport Level**:
		if (time>=subport_tc_time)
		{ subport_wm = water_mark_update();
		subport_tc_time = time + subport_tc_period;
		subport_period_id++;
		}
		Pipelevel:
		if(pipe_period_id != subport_period_id)
		{
		pipe_ov_credits = subport_wm * pipe_weight;
		pipe_period_id = subport_period_id;
		}

3	Credit consumption (on packet scheduling)	**Pipe level:**
		pkt_credits = pk_len + frame_overhead;
		if(pipe_ov_credits >= pkt_credits{
		pipe_ov_credits -= pkt_credits;
		}

Table 4.17: Watermark Calculation

No.	Subport Traffic Class Operation	Description
1	Initialization	**Subport level:** wm = WM_MAX

2	Credit update	**Subport level (water_mark_update):**
		tc0_cons = subport_tc0_credits_per_period - subport_tc0_credits;
		tc1_cons = subport_tc1_credits_per_period - subport_tc1_credits;
		tc2_cons = subport_tc2_credits_per_period - subport_tc2_credits;
		tc3_cons = subport_tc3_credits_per_period - subport_tc3_credits;
		tc3_cons_max = subport_tc3_credits_per_period - (tc0_cons + tc1_cons + tc2_cons);
		if(tc3_consumption > (tc3_consumption_max - MTU)){
		    wm -= wm >> 7;
		    if(wm < WM_MIN)
		    wm = WM_MIN;
		} else {
		    wm += (wm >> 7) + 1;
		    if(wm > WM_MAX) wm = WM_MAX;
		}

Worst Case Scenarios for Performance

Lots of Active Queues with Not Enough Credits

The more queues the scheduler has to examine for packets and credits in order to select one packet, the lower the performance of the scheduler is.

The scheduler maintains the bitmap of active queues, which skips the non-active queues, but in order to detect whether a specific pipe has enough credits, the pipe has to be drilled down using the pipe dequeue state machine, which consumes cycles regardless of the scheduling result (no packets are produced or at least one packet is produced).

This scenario stresses the importance of the policer for the scheduler performance: if the pipe does not have enough credits, its packets should be dropped as soon as possible (before they reach the hierarchical scheduler), thus rendering the pipe queues as not active, which allows the dequeue side to skip that pipe with no cycles being spent on investigating the pipe credits that would result in a “not enough credits” status.

Single Queue with 100% Line Rate

The port scheduler performance is optimized for a large number of queues. If the number of queues is small, then the performance of the port scheduler for the same level of active traffic is expected to be worse than the performance of a small set of message passing queues.

4.20.3 Dropper

The purpose of the DPDK dropper is to drop packets arriving at a packet scheduler to avoid congestion. The dropper supports the Random Early Detection (RED), Weighted Random Early Detection (WRED) and tail drop algorithms. Fig. 4.46 illustrates how the dropper integrates with the scheduler. The DPDK currently does not support congestion management so the dropper provides the only method for congestion avoidance.

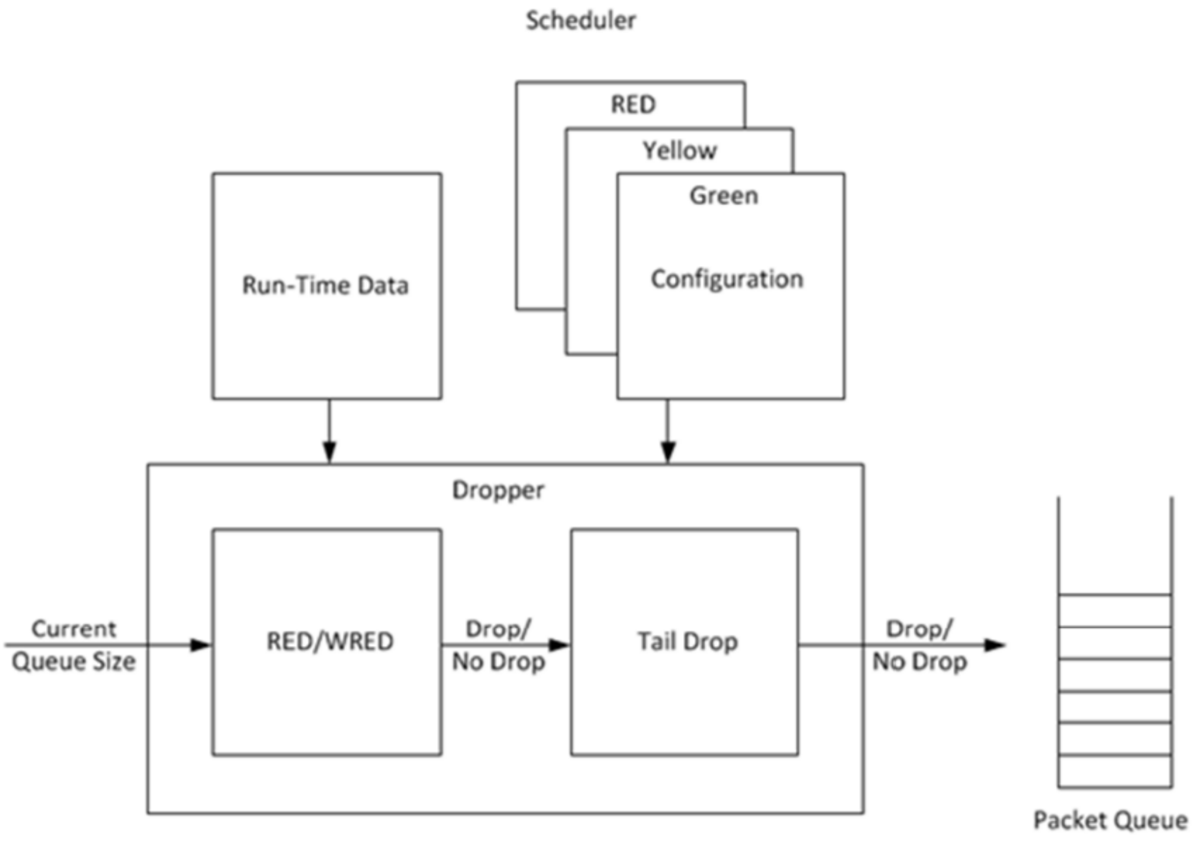


Fig. 4.46: High-level Block Diagram of the DPDK Dropper

The dropper uses the Random Early Detection (RED) congestion avoidance algorithm as documented in the reference publication. The purpose of the RED algorithm is to monitor a packet queue, determine the current congestion level in the queue and decide whether an arriving packet should be enqueued or dropped. The RED algorithm uses an Exponential Weighted

Moving Average (EWMA) filter to compute average queue size which gives an indication of the current congestion level in the queue.

For each enqueue operation, the RED algorithm compares the average queue size to minimum and maximum thresholds. Depending on whether the average queue size is below, above or in between these thresholds, the RED algorithm calculates the probability that an arriving packet should be dropped and makes a random decision based on this probability.

The dropper also supports Weighted Random Early Detection (WRED) by allowing the scheduler to select different RED configurations for the same packet queue at run-time. In the case of severe congestion, the dropper resorts to tail drop. This occurs when a packet queue has reached maximum capacity and cannot store any more packets. In this situation, all arriving packets are dropped.

The flow through the dropper is illustrated in Fig. 4.47. The RED/WRED algorithm is exercised first and tail drop second.

The use cases supported by the dropper are:

- – Initialize configuration data
- – Initialize run-time data
- – Enqueue (make a decision to enqueue or drop an arriving packet)
- – Mark empty (record the time at which a packet queue becomes empty)

The configuration use case is explained in *Section2.23.3.1*, the enqueue operation is explained in *Section 2.23.3.2* and the mark empty operation is explained in *Section 2.23.3.3*.

Configuration

A RED configuration contains the parameters given in Table 4.18.

Table 4.18: RED Configuration Parameters

Parameter	Minimum	Maximum	Typical
Minimum Threshold	0	1022	1/4 x queue size
Maximum Threshold	1	1023	1/2 x queue size
Inverse Mark Probability	1	255	10
EWMA Filter Weight	1	12	9

The meaning of these parameters is explained in more detail in the following sections. The format of these parameters as specified to the dropper module API corresponds to the format used by Cisco* in their RED implementation. The minimum and maximum threshold parameters are specified to the dropper module in terms of number of packets. The mark probability parameter is specified as an inverse value, for example, an inverse mark probability parameter value of 10 corresponds to a mark probability of 1/10 (that is, 1 in 10 packets will be dropped). The EWMA filter weight parameter is specified as an inverse log value, for example, a filter weight parameter value of 9 corresponds to a filter weight of 1/29.

Enqueue Operation

In the example shown in Fig. 4.48, q (actual queue size) is the input value, avg (average queue size) and count (number of packets since the last drop) are run-time values, decision is

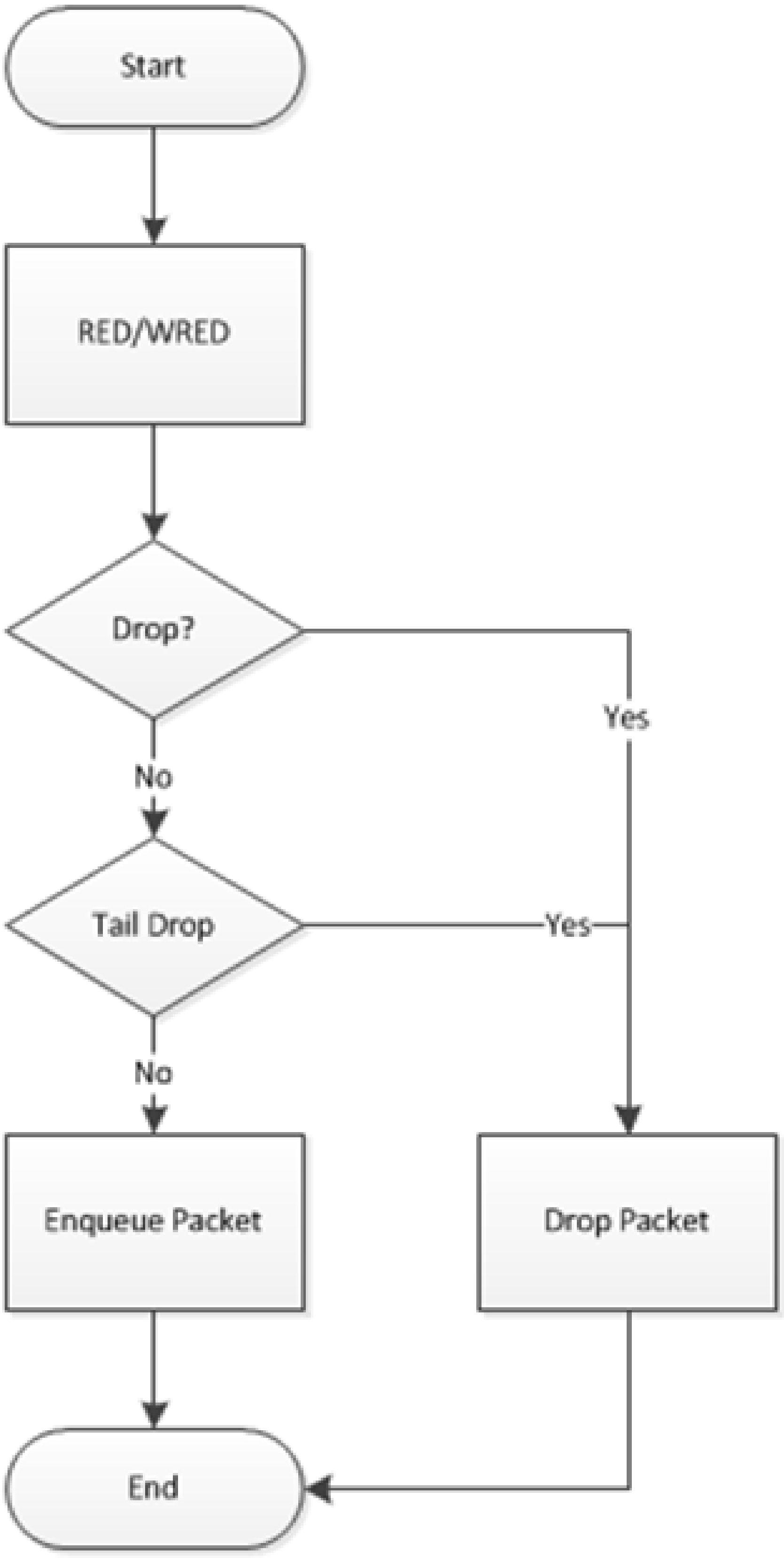


Fig. 4.47: Flow Through the Dropper

the output value and the remaining values are configuration parameters.

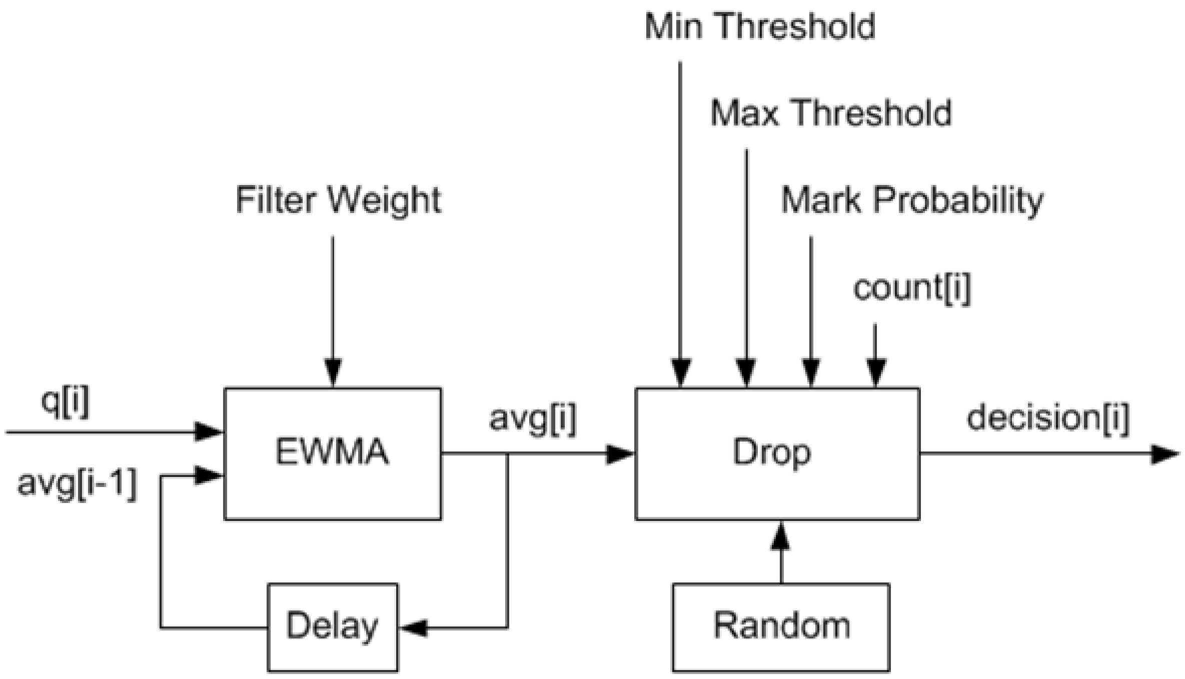


Fig. 4.48: Example Data Flow Through Dropper

EWMA Filter Microblock

The purpose of the EWMA Filter microblock is to filter queue size values to smooth out transient changes that result from "bursty" traffic. The output value is the average queue size which gives a more stable view of the current congestion level in the queue.

The EWMA filter has one configuration parameter, filter weight, which determines how quickly or slowly the average queue size output responds to changes in the actual queue size input. Higher values of filter weight mean that the average queue size responds more quickly to changes in actual queue size.

Average Queue Size Calculation when the Queue is not Empty The definition of the EWMA filter is given in the following equation.

$$avg[i] = (1 - w_q) \times avg[i-1] + w_q \times q[i]$$

Where:

- *avg* = average queue size
- *wq* = filter weight
- *q* = actual queue size

Note:

The filter weight, wq = 1/2^n, where n is the filter weight parameter value passed to the dropper modu
on configuration (see *Section2.23.3.1*).

Average Queue Size Calculation when the Queue is Empty

The EWMA filter does not read time stamps and instead assumes that enqueue operations will happen quite regularly. Special handling is required when the queue becomes empty as the queue could be empty for a short time or a long time. When the queue becomes empty, average queue size should decay gradually to zero instead of dropping suddenly to zero or remaining stagnant at the last computed value. When a packet is enqueued on an empty queue, the average queue size is computed using the following formula:

$$avg[i] = avg[i-1] \times (1 - w_q)^m$$

Where:

- *m* = the number of enqueue operations that could have occurred on this queue while the queue was empty

In the dropper module, *m* is defined as:

$$m = \left(\frac{time - qtime}{s}\right)$$

Where:

- *time* = current time
- *qtime* = time the queue became empty
- *s* = typical time between successive enqueue operations on this queue

The time reference is in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium (see Section 26.2.4.5.1 "Internal Time Reference"). The parameter s is defined in the dropper module as a constant with the value: s=2^22. This corresponds to the time required by every leaf node in a hierarchy with 64K leaf nodes to transmit one 64-byte packet onto the wire and represents the worst case scenario. For much smaller scheduler hierarchies, it may be necessary to reduce the parameter s, which is defined in the red header source file (rte_red.h) as:

```
#define RTE_RED_S
```

Since the time reference is in bytes, the port speed is implied in the expression: *time-qtime*. The dropper does not have to be configured with the actual port speed. It adjusts automatically to low speed and high speed links.

Implementation A numerical method is used to compute the factor (1-wq)^m that appears in Equation 2.

This method is based on the following identity:

$$a \equiv 2^{(b \times \log_2(a))}$$

This allows us to express the following:

$$(1 - w_q)^m = 2^{(m \times \log_2(1 - w_q))}$$

In the dropper module, a look-up table is used to compute log2(1-wq) for each value of wq supported by the dropper module. The factor (1-wq)^m can then be obtained by multiplying the table value by *m* and applying shift operations. To avoid overflow in the multiplication, the value, *m*, and the look-up table values are limited to 16 bits. The total size of the look-up table is 56 bytes. Once the factor (1-wq)^m is obtained using this method, the average queue size can be calculated from Equation 2.

Alternative Approaches Other methods for calculating the factor (1-wq)^m in the expression for computing average queue size when the queue is empty (Equation 2) were considered. These approaches include:

- Floating-point evaluation
- Fixed-point evaluation using a small look-up table (512B) and up to 16 multiplications (this is the approach used in the FreeBSD* ALTQ RED implementation)
- Fixed-point evaluation using a small look-up table (512B) and 16 SSE multiplications (SSE optimized version of the approach used in the FreeBSD* ALTQ RED implementation)
- Large look-up table (76 KB)

The method that was finally selected (described above in Section 26.3.2.2.1) out performs all of these approaches in terms of run-time performance and memory requirements and also achieves accuracy comparable to floating-point evaluation. Table 4.19 lists the performance of each of these alternative approaches relative to the method that is used in the dropper. As can be seen, the floating-point implementation achieved the worst performance.

Table 4.19: Relative Performance of Al

Method	Relative Performance
Current dropper method (see *Section 23.3.2.1.3*)	100%
Fixed-point method with small (512B) look-up table	148%
SSE method with small (512B) look-up table	114%
Large (76KB) look-up table	118%
Floating-point	595%
Note: In this case, since performance is expressed as time spent executing the operation in a specific condi	

Drop Decision Block

The Drop Decision block:

- Compares the average queue size with the minimum and maximum thresholds
- Calculates a packet drop probability
- Makes a random decision to enqueue or drop an arriving packet

The calculation of the drop probability occurs in two stages. An initial drop probability is calculated based on the average queue size, the minimum and maximum thresholds and the mark probability. An actual drop probability is then computed from the initial drop probability. The actual drop probability takes the count run-time value into consideration so that the actual drop probability increases as more packets arrive to the packet queue since the last packet was dropped.

Initial Packet Drop Probability The initial drop probability is calculated using the following equation.

$$p_b = \begin{cases} 0, & avg < min_{th} \\ max_p \left(\dfrac{avg - min_{th}}{max_{th} - min_{th}} \right), & min_{th} \leq avg < max_{th} \\ 1, & avg \geq max_{th} \end{cases}$$

Where:

- *maxp* = mark probability
- *avg* = average queue size
- *minth* = minimum threshold
- *maxth* = maximum threshold

The calculation of the packet drop probability using Equation 3 is illustrated in Fig. 4.49. If the average queue size is below the minimum threshold, an arriving packet is enqueued. If the average queue size is at or above the maximum threshold, an arriving packet is dropped. If the average queue size is between the minimum and maximum thresholds, a drop probability is calculated to determine if the packet should be enqueued or dropped.

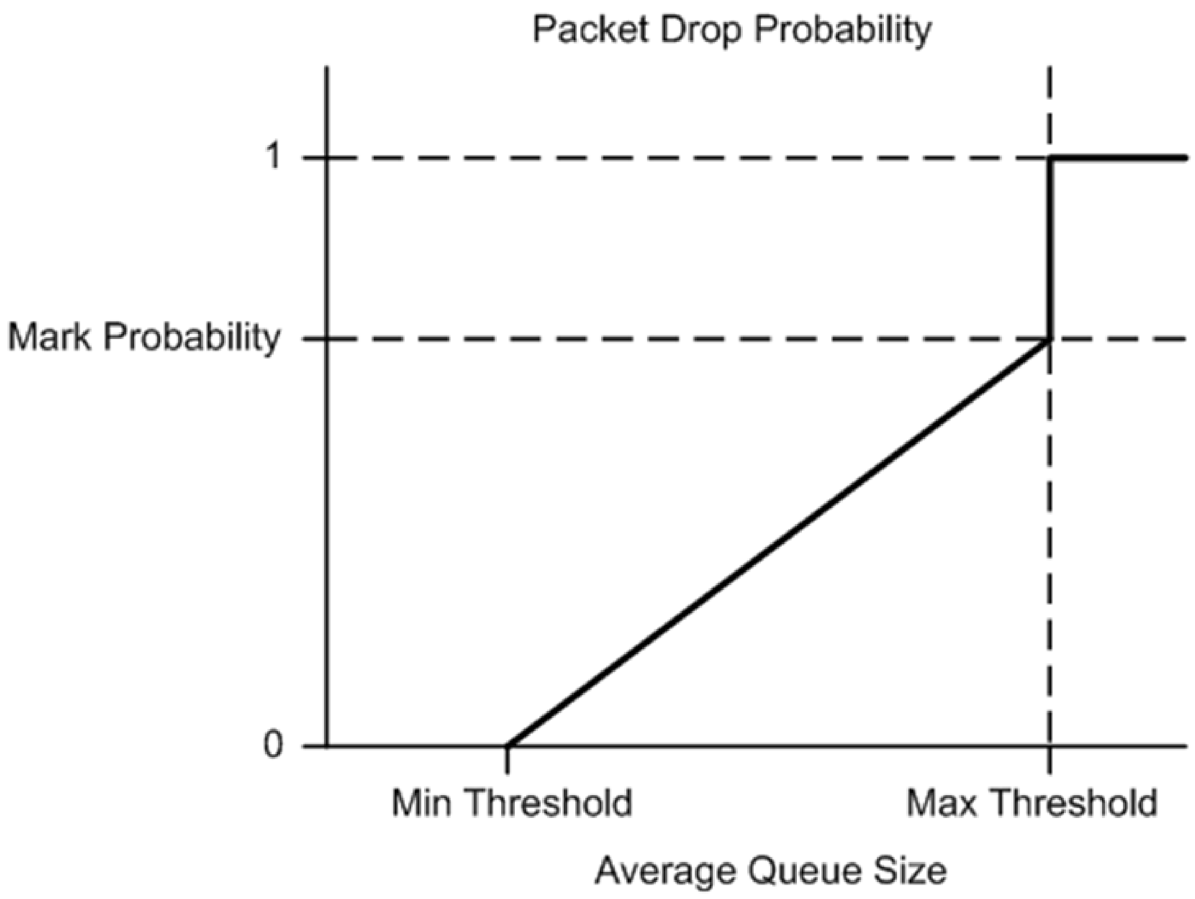


Fig. 4.49: Packet Drop Probability for a Given RED Configuration

Actual Drop Probability If the average queue size is between the minimum and maximum thresholds, then the actual drop probability is calculated from the following equation.

$$p_a = \frac{p_b}{(2 - count \times p_b)}$$

Where:

- *Pb* = initial drop probability (from Equation 3)
- *count* = number of packets that have arrived since the last drop

The constant 2, in Equation 4 is the only deviation from the drop probability formulae given in the reference document where a value of 1 is used instead. It should be noted that the value pa computed from can be negative or greater than 1. If this is the case, then a value of 1 should be used instead.

The initial and actual drop probabilities are shown in Fig. 4.50. The actual drop probability is shown for the case where the formula given in the reference document1 is used (blue curve) and also for the case where the formula implemented in the dropper module, is used (red curve). The formula in the reference document results in a significantly higher drop rate compared to the mark probability configuration parameter specified by the user. The choice to deviate from the reference document is simply a design decision and one that has been taken by other RED implementations, for example, FreeBSD* ALTQ RED.

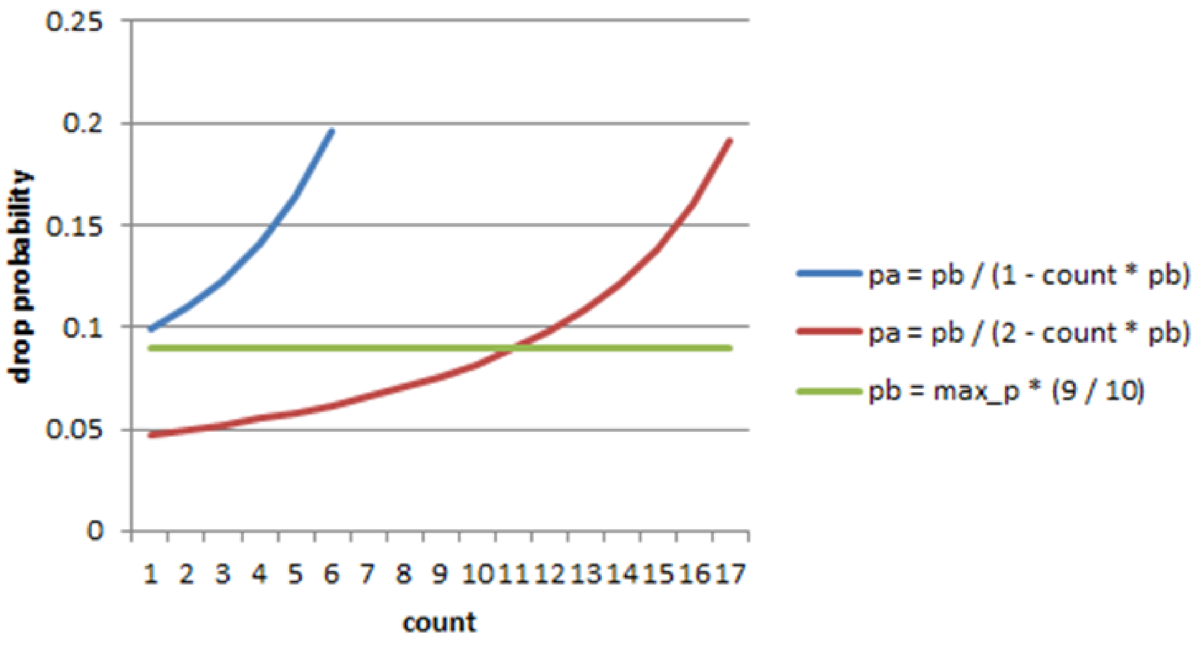


Fig. 4.50: Initial Drop Probability (pb), Actual Drop probability (pa) Computed Using a Factor 1 (Blue Curve) and a Factor 2 (Red Curve)

Queue Empty Operation

The time at which a packet queue becomes empty must be recorded and saved with the RED run-time data so that the EWMA filter block can calculate the average queue size on the next enqueue operation. It is the responsibility of the calling application to inform the dropper module through the API that a queue has become empty.

Source Files Location

The source files for the DPDK dropper are located at:

- DPDK/lib/librte_sched/rte_red.h
- DPDK/lib/librte_sched/rte_red.c

Integration with the DPDK QoS Scheduler

RED functionality in the DPDK QoS scheduler is disabled by default. To enable it, use the DPDK configuration parameter:

```
CONFIG_RTE_SCHED_RED=y
```

This parameter must be set to y. The parameter is found in the build configuration files in the DPDK/config directory, for example, DPDK/config/common_linuxapp. RED configuration parameters are specified in the rte_red_params structure within the rte_sched_port_params structure that is passed to the scheduler on initialization. RED parameters are specified separately for four traffic classes and three packet colors (green, yellow and red) allowing the scheduler to implement Weighted Random Early Detection (WRED).

Integration with the DPDK QoS Scheduler Sample Application

The DPDK QoS Scheduler Application reads a configuration file on start-up. The configuration file includes a section containing RED parameters. The format of these parameters is described in *Section2.23.3.1*. A sample RED configuration is shown below. In this example, the queue size is 64 packets.

Note: For correct operation, the same EWMA filter weight parameter (wred weight) should be used for each packet color (green, yellow, red) in the same traffic class (tc).

```
 ; RED params per traffic class and color (Green / Yellow / Red)

[red]
tc 0 wred min = 28 22 16
tc 0 wred max = 32 32 32
tc 0 wred inv prob = 10 10 10
tc 0 wred weight = 9 9 9

tc 1 wred min = 28 22 16
tc 1 wred max = 32 32 32
tc 1 wred inv prob = 10 10 10
tc 1 wred weight = 9 9 9

tc 2 wred min = 28 22 16
tc 2 wred max = 32 32 32
tc 2 wred inv prob = 10 10 10
tc 2 wred weight = 9 9 9

tc 3 wred min = 28 22 16
tc 3 wred max = 32 32 32
tc 3 wred inv prob = 10 10 10
tc 3 wred weight = 9 9 9
```

With this configuration file, the RED configuration that applies to green, yellow and red packets in traffic class 0 is shown in Table 4.20.

Table 4.20: RED Configuration Corresponding to RED Configuration File

RED Parameter	Configuration Name	Green	Yellow	Red
Minimum Threshold	tc 0 wred min	28	22	16
Maximum Threshold	tc 0 wred max	32	32	32
Mark Probability	tc 0 wred inv prob	10	10	10
EWMA Filter Weight	tc 0 wred weight	9	9	9

Application Programming Interface (API)

Enqueue API

The syntax of the enqueue API is as follows:

```
int rte_red_enqueue(const struct rte_red_config *red_cfg, struct rte_red *red, const unsigned
```

The arguments passed to the enqueue API are configuration data, run-time data, the current size of the packet queue (in packets) and a value representing the current time. The time reference is in units of bytes, where a byte signifies the time duration required by the physical interface to send out a byte on the transmission medium (see Section 26.2.4.5.1 "Internal Time Reference"). The dropper reuses the scheduler time stamps for performance reasons.

Empty API

The syntax of the empty API is as follows:

```
void rte_red_mark_queue_empty(struct rte_red *red, const uint64_t time)
```

The arguments passed to the empty API are run-time data and the current time in bytes.

4.20.4 Traffic Metering

The traffic metering component implements the Single Rate Three Color Marker (srTCM) and Two Rate Three Color Marker (trTCM) algorithms, as defined by IETF RFC 2697 and 2698 respectively. These algorithms meter the stream of incoming packets based on the allowance defined in advance for each traffic flow. As result, each incoming packet is tagged as green, yellow or red based on the monitored consumption of the flow the packet belongs to.

Functional Overview

The srTCM algorithm defines two token buckets for each traffic flow, with the two buckets sharing the same token update rate:

- Committed (C) bucket: fed with tokens at the rate defined by the Committed Information Rate (CIR) parameter (measured in IP packet bytes per second). The size of the C bucket is defined by the Committed Burst Size (CBS) parameter (measured in bytes);
- Excess (E) bucket: fed with tokens at the same rate as the C bucket. The size of the E bucket is defined by the Excess Burst Size (EBS) parameter (measured in bytes).

The trTCM algorithm defines two token buckets for each traffic flow, with the two buckets being updated with tokens at independent rates:

- Committed (C) bucket: fed with tokens at the rate defined by the Committed Information Rate (CIR) parameter (measured in bytes of IP packet per second). The size of the C bucket is defined by the Committed Burst Size (CBS) parameter (measured in bytes);
- Peak (P) bucket: fed with tokens at the rate defined by the Peak Information Rate (PIR) parameter (measured in IP packet bytes per second). The size of the P bucket is defined by the Peak Burst Size (PBS) parameter (measured in bytes).

Please refer to RFC 2697 (for srTCM) and RFC 2698 (for trTCM) for details on how tokens are consumed from the buckets and how the packet color is determined.

Color Blind and Color Aware Modes

For both algorithms, the color blind mode is functionally equivalent to the color aware mode with input color set as green. For color aware mode, a packet with red input color can only get the red output color, while a packet with yellow input color can only get the yellow or red output colors.

The reason why the color blind mode is still implemented distinctly than the color aware mode is that color blind mode can be implemented with fewer operations than the color aware mode.

Implementation Overview

For each input packet, the steps for the srTCM / trTCM algorithms are:

- Update the C and E / P token buckets. This is done by reading the current time (from the CPU timestamp counter), identifying the amount of time since the last bucket update and computing the associated number of tokens (according to the pre-configured bucket rate). The number of tokens in the bucket is limited by the pre-configured bucket size;
- Identify the output color for the current packet based on the size of the IP packet and the amount of tokens currently available in the C and E / P buckets; for color aware mode only, the input color of the packet is also considered. When the output color is not red, a number of tokens equal to the length of the IP packet are subtracted from the C or E /P or both buckets, depending on the algorithm and the output color of the packet.

4.21 Power Management

The DPDK Power Management feature allows users space applications to save power by dynamically adjusting CPU frequency or entering into different C-States.

- Adjusting the CPU frequency dynamically according to the utilization of RX queue.
- Entering into different deeper C-States according to the adaptive algorithms to speculate brief periods of time suspending the application if no packets are received.

The interfaces for adjusting the operating CPU frequency are in the power management library. C-State control is implemented in applications according to the different use cases.

4.21.1 CPU Frequency Scaling

The Linux kernel provides a cpufreq module for CPU frequency scaling for each lcore. For example, for cpuX, /sys/devices/system/cpu/cpuX/cpufreq/ has the following sys files for frequency scaling:

- affected_cpus
- bios_limit
- cpuinfo_cur_freq
- cpuinfo_max_freq
- cpuinfo_min_freq
- cpuinfo_transition_latency
- related_cpus
- scaling_available_frequencies
- scaling_available_governors
- scaling_cur_freq
- scaling_driver
- scaling_governor
- scaling_max_freq
- scaling_min_freq
- scaling_setspeed

In the DPDK, scaling_governor is configured in user space. Then, a user space application can prompt the kernel by writing scaling_setspeed to adjust the CPU frequency according to the strategies defined by the user space application.

4.21.2 Core-load Throttling through C-States

Core state can be altered by speculative sleeps whenever the specified lcore has nothing to do. In the DPDK, if no packet is received after polling, speculative sleeps can be triggered according the strategies defined by the user space application.

4.21.3 API Overview of the Power Library

The main methods exported by power library are for CPU frequency scaling and include the following:

- **Freq up**: Prompt the kernel to scale up the frequency of the specific lcore.
- **Freq down**: Prompt the kernel to scale down the frequency of the specific lcore.
- **Freq max**: Prompt the kernel to scale up the frequency of the specific lcore to the maximum.
- **Freq min**: Prompt the kernel to scale down the frequency of the specific lcore to the minimum.

- **Get available freqs**: Read the available frequencies of the specific lcore from the sys file.
- **Freq get**: Get the current frequency of the specific lcore.
- **Freq set**: Prompt the kernel to set the frequency for the specific lcore.

4.21.4 User Cases

The power management mechanism is used to save power when performing L3 forwarding.

4.21.5 References

- l3fwd-power: The sample application in DPDK that performs L3 forwarding with power management.
- The "L3 Forwarding with Power Management Sample Application" chapter in the *DPDK Sample Application's User Guide*.

4.22 Packet Classification and Access Control

The DPDK provides an Access Control library that gives the ability to classify an input packet based on a set of classification rules.

The ACL library is used to perform an N-tuple search over a set of rules with multiple categories and find the best match (highest priority) for each category. The library API provides the following basic operations:

- Create a new Access Control (AC) context.
- Add rules into the context.
- For all rules in the context, build the runtime structures necessary to perform packet classification.
- Perform input packet classifications.
- Destroy an AC context and its runtime structures and free the associated memory.

4.22.1 Overview

Rule definition

The current implementation allows the user for each AC context to specify its own rule (set of fields) over which packet classification will be performed. Though there are few restrictions on the rule fields layout:

- First field in the rule definition has to be one byte long.
- All subsequent fields has to be grouped into sets of 4 consecutive bytes.

This is done mainly for performance reasons - search function processes the first input byte as part of the flow setup and then the inner loop of the search function is unrolled to process four input bytes at a time.

To define each field inside an AC rule, the following structure is used:

```
struct rte_acl_field_def {
    uint8_t type;         /*< type - ACL_FIELD_TYPE. */
    uint8_t size;         /*< size of field 1,2,4, or 8. */
    uint8_t field_index;  /*< index of field inside the rule. */
    uint8_t input_index;  /*< 0-N input index. */
    uint32_t offset;      /*< offset to start of field. */
};
```

- type The field type is one of three choices:
  - _MASK - for fields such as IP addresses that have a value and a mask defining the number of relevant bits.
  - _RANGE - for fields such as ports that have a lower and upper value for the field.
  - _BITMASK - for fields such as protocol identifiers that have a value and a bit mask.
- size The size parameter defines the length of the field in bytes. Allowable values are 1, 2, 4, or 8 bytes. Note that due to the grouping of input bytes, 1 or 2 byte fields must be defined as consecutive fields that make up 4 consecutive input bytes. Also, it is best to define fields of 8 or more bytes as 4 byte fields so that the build processes can eliminate fields that are all wild.
- field_index A zero-based value that represents the position of the field inside the rule; 0 to N-1 for N fields.
- input_index As mentioned above, all input fields, except the very first one, must be in groups of 4 consecutive bytes. The input index specifies to which input group that field belongs to.
- offset The offset field defines the offset for the field. This is the offset from the beginning of the buffer parameter for the search.

For example, to define classification for the following IPv4 5-tuple structure:

```
struct ipv4_5tuple {
    uint8_t proto;
    uint32_t ip_src;
    uint32_t ip_dst;
    uint16_t port_src;
    uint16_t port_dst;
};
```

The following array of field definitions can be used:

```
struct rte_acl_field_def ipv4_defs[5] = {
    /* first input field - always one byte long. */
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct ipv4_5tuple, proto),
    },

    /* next input field (IPv4 source address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
       .offset = offsetof (struct ipv4_5tuple, ip_src),
    },

    /* next input field (IPv4 destination address) - 4 consecutive bytes. */
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
       .offset = offsetof (struct ipv4_5tuple, ip_dst),
    },

    /*
     * Next 2 fields (src & dst ports) form 4 consecutive bytes.
     * They share the same input index.
     */
    {
        .type = RTE_ACL_FIELD_TYPE_RANGE,
        .size = sizeof (uint16_t),
        .field_index = 3,
        .input_index = 3,
        .offset = offsetof (struct ipv4_5tuple, port_src),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_RANGE,
        .size = sizeof (uint16_t),
        .field_index = 4,
        .input_index = 3,
        .offset = offsetof (struct ipv4_5tuple, port_dst),
    },
};
```

A typical example of such an IPv4 5-tuple rule is a follows:

```
source addr/mask  destination addr/mask  source ports dest ports protocol/mask
192.168.1.0/24    192.168.2.31/32        0:65535      1234:1234  17/0xff
```

Any IPv4 packets with protocol ID 17 (UDP), source address 192.168.1.[0-255], destination address 192.168.2.31, source port [0-65535] and destination port 1234 matches the above rule.

To define classification for the IPv6 2-tuple: <protocol, IPv6 source address> over the following IPv6 header structure:

```
struct struct ipv6_hdr {
    uint32_t vtc_flow;     /* IP version, traffic class & flow label. */
    uint16_t payload_len;  /* IP packet length - includes sizeof(ip_header). */
    uint8_t proto;         /* Protocol, next header. */
    uint8_t hop_limits;    /* Hop limits. */
    uint8_t src_addr[16];  /* IP address of source host. */
    uint8_t dst_addr[16];  /* IP address of destination host(s). */
} __attribute__((__packed__));
```

The following array of field definitions can be used:

```
struct struct rte_acl_field_def ipv6_2tuple_defs[5] = {
    {
        .type = RTE_ACL_FIELD_TYPE_BITMASK,
        .size = sizeof (uint8_t),
        .field_index = 0,
        .input_index = 0,
        .offset = offsetof (struct ipv6_hdr, proto),
    },
    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 1,
        .input_index = 1,
        .offset = offsetof (struct ipv6_hdr, src_addr[0]),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 2,
        .input_index = 2,
        .offset = offsetof (struct ipv6_hdr, src_addr[4]),
    },

    {
        .type = RTE_ACL_FIELD_TYPE_MASK,
        .size = sizeof (uint32_t),
        .field_index = 3,
        .input_index = 3,
       .offset = offsetof (struct ipv6_hdr, src_addr[8]),
    },

    {
       .type = RTE_ACL_FIELD_TYPE_MASK,
       .size = sizeof (uint32_t),
       .field_index = 4,
       .input_index = 4,
       .offset = offsetof (struct ipv6_hdr, src_addr[12]),
    },
};
```

A typical example of such an IPv6 2-tuple rule is a follows:

```
source addr/mask                              protocol/mask
2001:db8:1234:0000:0000:0000:0000:0000/48     6/0xff
```

Any IPv6 packets with protocol ID 6 (TCP), and source address inside the range [2001:db8:1234:0000:0000:0000:0000:0000 - 2001:db8:1234:ffff:ffff:ffff:ffff:ffff] matches the above rule.

When creating a set of rules, for each rule, additional information must be supplied also:

- **priority**: A weight to measure the priority of the rules (higher is better). If the input tuple matches more than one rule, then the rule with the higher priority is returned. Note that if the input tuple matches more than one rule and these rules have equal priority, it is undefined which rule is returned as a match. It is recommended to assign a unique priority for each rule.
- **category_mask**: Each rule uses a bit mask value to select the relevant category(s) for the rule. When a lookup is performed, the result for each category is returned. This effectively provides a “parallel lookup” by enabling a single search to return multiple results if, for example, there were four different sets of ACL rules, one for access control, one for routing, and so on. Each set could be assigned its own category and by combining them into a single database, one lookup returns a result for each of the four sets.
- **userdata**: A user-defined field that could be any value except zero. For each category, a successful match returns the userdata field of the highest priority matched rule.

Note: When adding new rules into an ACL context, all fields must be in host byte order (LSB). When the search is performed for an input tuple, all fields in that tuple must be in network byte order (MSB).

RT memory size limit

Build phase (rte_acl_build()) creates for a given set of rules internal structure for further run-time traversal. With current implementation it is a set of multi-bit tries (with stride == 8). Depending on the rules set, that could consume significant amount of memory. In attempt to conserve some space ACL build process tries to split the given rule-set into several non-intersecting subsets and construct a separate trie for each of them. Depending on the rule-set, it might reduce RT memory requirements but might increase classification time. There is a possibility at build-time to specify maximum memory limit for internal RT structures for given AC context. It could be done via **max_size** field of the **rte_acl_config** structure. Setting it to the value greater than zero, instructs rte_acl_build() to:

- attempt to minimize number of tries in the RT table, but
- make sure that size of RT table wouldn't exceed given value.

Setting it to zero makes rte_acl_build() to use the default behavior: try to minimize size of the RT structures, but doesn't expose any hard limit on it.

That gives the user the ability to decisions about performance/space trade-off. For example:

```
struct rte_acl_ctx * acx;
struct rte_acl_config cfg;
int ret;

/*
 * assuming that acx points to already created and
 * populated with rules AC context and cfg filled properly.
 */

 /* try to build AC context, with RT structures less then 8MB. */
 cfg.max_size = 0x800000;
 ret = rte_acl_build(acx, &cfg);

 /*
  * RT structures can't fit into 8MB for given context.
  * Try to build without exposing any hard limit.
  */
 if (ret == -ERANGE) {
    cfg.max_size = 0;
    ret = rte_acl_build(acx, &cfg);
 }
```

Classification methods

After rte_acl_build() over given AC context has finished successfully, it can be used to perform classification - search for a rule with highest priority over the input data. There are several implementations of classify algorithm:

- **RTE_ACL_CLASSIFY_SCALAR**: generic implementation, doesn't require any specific HW support.

- **RTE_ACL_CLASSIFY_SSE**: vector implementation, can process up to 8 flows in parallel. Requires SSE 4.1 support.
- **RTE_ACL_CLASSIFY_AVX2**: vector implementation, can process up to 16 flows in parallel. Requires AVX2 support.

It is purely a runtime decision which method to choose, there is no build-time difference. All implementations operates over the same internal RT structures and use similar principles. The main difference is that vector implementations can manually exploit IA SIMD instructions and process several input data flows in parallel. At startup ACL library determines the highest available classify method for the given platform and sets it as default one. Though the user has an ability to override the default classifier function for a given ACL context or perform particular search using non-default classify method. In that case it is user responsibility to make sure that given platform supports selected classify implementation.

4.22.2 Application Programming Interface (API) Usage

Note: For more details about the Access Control API, please refer to the *DPDK API Reference*.

The following example demonstrates IPv4, 5-tuple classification for rules defined above with multiple categories in more detail.

Classify with Multiple Categories

```
struct rte_acl_ctx * acx;
struct rte_acl_config cfg;
int ret;

/* define a structure for the rule with up to 5 fields. */

RTE_ACL_RULE_DEF(acl_ipv4_rule, RTE_DIM(ipv4_defs));

/* AC context creation parameters. */

struct rte_acl_param prm = {
    .name = "ACL_example",
    .socket_id = SOCKET_ID_ANY,
    .rule_size = RTE_ACL_RULE_SZ(RTE_DIM(ipv4_defs)),

    /* number of fields per rule. */

    .max_rule_num = 8, /* maximum number of rules in the AC context. */
};

struct acl_ipv4_rule acl_rules[] = {

    /* matches all packets traveling to 192.168.0.0/16, applies for categories: 0,1 */
    {
        .data = {.userdata = 1, .category_mask = 3, .priority = 1},

        /* destination IPv4 */
        .field[2] = {.value.u32 = IPv4(192,168,0,0),. mask_range.u32 = 16,},

        /* source port */
```

```
            .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

            /* destination port */
           .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
        },

        /* matches all packets traveling to 192.168.1.0/24, applies for categories: 0 */
        {
            .data = {.userdata = 2, .category_mask = 1, .priority = 2},

            /* destination IPv4 */
            .field[2] = {.value.u32 = IPv4(192,168,1,0),. mask_range.u32 = 24,},

            /* source port */
            .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

            /* destination port */
            .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
        },

        /* matches all packets traveling from 10.1.1.1, applies for categories: 1 */
        {
            .data = {.userdata = 3, .category_mask = 2, .priority = 3},

            /* source IPv4 */
            .field[1] = {.value.u32 = IPv4(10,1,1,1),. mask_range.u32 = 32,},

            /* source port */
            .field[3] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},

            /* destination port */
            .field[4] = {.value.u16 = 0, .mask_range.u16 = 0xffff,},
        },

    };


    /* create an empty AC context  */

    if ((acx = rte_acl_create(&prm)) == NULL) {

        /* handle context create failure. */

    }

    /* add rules to the context */

    ret = rte_acl_add_rules(acx, acl_rules, RTE_DIM(acl_rules));
    if (ret != 0) {
       /* handle error at adding ACL rules. */
    }

    /* prepare AC build config. */

    cfg.num_categories = 2;
    cfg.num_fields = RTE_DIM(ipv4_defs);

    memcpy(cfg.defs, ipv4_defs, sizeof (ipv4_defs));

    /* build the runtime structures for added rules, with 2 categories. */

    ret = rte_acl_build(acx, &cfg);
    if (ret != 0) {
        /* handle error at build runtime structures for ACL context. */
    }
```

For a tuple with source IP address: 10.1.1.1 and destination IP address: 192.168.1.15, once the following lines are executed:

```
uint32_t results[4]; /* make classify for 4 categories. */

rte_acl_classify(acx, data, results, 1, 4);
```

then the results[] array contains:

```
results[4] = {2, 3, 0, 0};
```

- For category 0, both rules 1 and 2 match, but rule 2 has higher priority, therefore results[0] contains the userdata for rule 2.
- For category 1, both rules 1 and 3 match, but rule 3 has higher priority, therefore results[1] contains the userdata for rule 3.
- For categories 2 and 3, there are no matches, so results[2] and results[3] contain zero, which indicates that no matches were found for those categories.

For a tuple with source IP address: 192.168.1.1 and destination IP address: 192.168.2.11, once the following lines are executed:

```
uint32_t results[4]; /* make classify by 4 categories. */

rte_acl_classify(acx, data, results, 1, 4);
```

the results[] array contains:

```
results[4] = {1, 1, 0, 0};
```

- For categories 0 and 1, only rule 1 matches.
- For categories 2 and 3, there are no matches.

For a tuple with source IP address: 10.1.1.1 and destination IP address: 201.212.111.12, once the following lines are executed:

```
uint32_t results[4]; /* make classify by 4 categories. */
rte_acl_classify(acx, data, results, 1, 4);
```

the results[] array contains:

```
results[4] = {0, 3, 0, 0};
```

- For category 1, only rule 3 matches.
- For categories 0, 2 and 3, there are no matches.

4.23 Packet Framework

4.23.1 Design Objectives

The main design objectives for the DPDK Packet Framework are:

- Provide standard methodology to build complex packet processing pipelines. Provide reusable and extensible templates for the commonly used pipeline functional blocks;

- Provide capability to switch between pure software and hardware-accelerated implementations for the same pipeline functional block;
- Provide the best trade-off between flexibility and performance. Hardcoded pipelines usually provide the best performance, but are not flexible, while developing flexible frameworks is never a problem, but performance is usually low;
- Provide a framework that is logically similar to Open Flow.

4.23.2 Overview

Packet processing applications are frequently structured as pipelines of multiple stages, with the logic of each stage glued around a lookup table. For each incoming packet, the table defines the set of actions to be applied to the packet, as well as the next stage to send the packet to.

The DPDK Packet Framework minimizes the development effort required to build packet processing pipelines by defining a standard methodology for pipeline development, as well as providing libraries of reusable templates for the commonly used pipeline blocks.

The pipeline is constructed by connecting the set of input ports with the set of output ports through the set of tables in a tree-like topology. As result of lookup operation for the current packet in the current table, one of the table entries (on lookup hit) or the default table entry (on lookup miss) provides the set of actions to be applied on the current packet, as well as the next hop for the packet, which can be either another table, an output port or packet drop.

An example of packet processing pipeline is presented in Fig. 4.51:

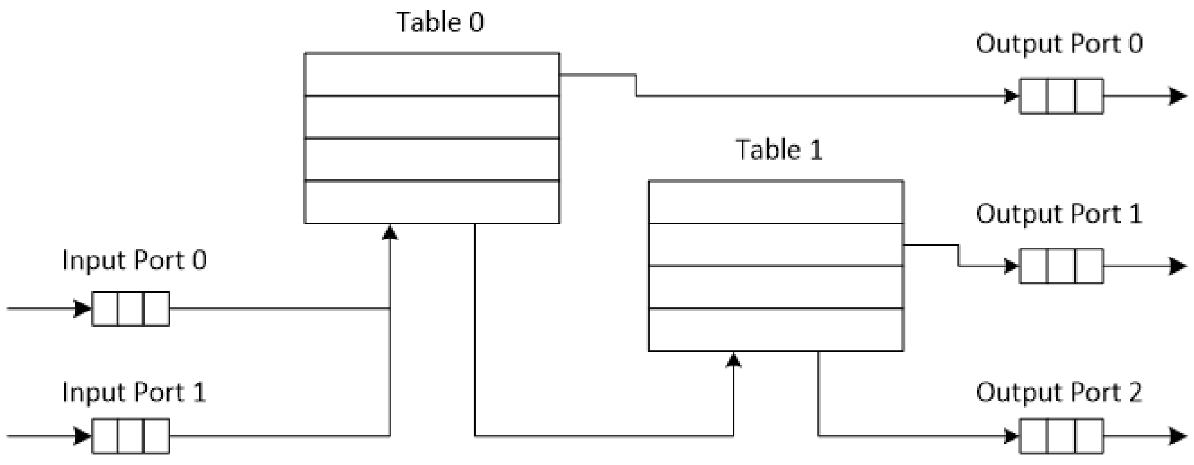


Fig. 4.51: Example of Packet Processing Pipeline where Input Ports 0 and 1 are Connected with Output Ports 0, 1 and 2 through Tables 0 and 1

4.23.3 Port Library Design

Port Types

Table 4.21 is a non-exhaustive list of ports that can be implemented with the Packet Framework.

Table 4.21: Port Types

#	Port type	Description
1	SW ring	SW circular buffer used for message passing between the application threads. Uses the DPDK rte_ring primitive. Expected to be the most commonly used type of port.
2	HW ring	Queue of buffer descriptors used to interact with NIC, switch or accelerator ports. For NIC ports, it uses the DPDK rte_eth_rx_queue or rte_eth_tx_queue primitives.
3	IP re-assembly	Input packets are either IP fragments or complete IP datagrams. Output packets are complete IP datagrams.
4	IP fragmentation	Input packets are jumbo (IP datagrams with length bigger than MTU) or non-jumbo packets. Output packets are non-jumbo packets.
5	Traffic manager	Traffic manager attached to a specific NIC output port, performing congestion management and hierarchical scheduling according to pre-defined SLAs.
6	KNI	Send/receive packets to/from Linux kernel space.
7	Source	Input port used as packet generator. Similar to Linux kernel /dev/zero character device.
8	Sink	Output port used to drop all input packets. Similar to Linux kernel /dev/null character device.

Port Interface

Each port is unidirectional, i.e. either input port or output port. Each input/output port is required to implement an abstract interface that defines the initialization and run-time operation of the port. The port abstract interface is described in.

Table 4.22: 20 Port Abstract Interface

#	Port Operation	Description
1	Create	Create the low-level port object (e.g. queue). Can internally allocate memory.
2	Free	Free the resources (e.g. memory) used by the low-level port object.
3	RX	Read a burst of input packets. Non-blocking operation. Only defined for input ports.
4	TX	Write a burst of input packets. Non-blocking operation. Only defined for output ports.
5	Flush	Flush the output buffer. Only defined for output ports.

4.23.4 Table Library Design

Table Types

Table 4.23 is a non-exhaustive list of types of tables that can be implemented with the Packet Framework.

Table 4.23: Table Types

#	Table Type	Description
1	Hash table	Lookup key is n-tuple based. Typically, the lookup key is hashed to produce a signature that is used to identify a bucket of entries where the lookup key is searched next. The signature associated with the lookup key of each input packet is either read from the packet descriptor (pre-computed signature) or computed at table lookup time. The table lookup, add entry and delete entry operations, as well as any other pipeline block that pre-computes the signature all have to use the same hashing algorithm to generate the signature. Typically used to implement flow classification tables, ARP caches, routing table for tunnelling protocols, etc.
2	Longest Prefix Match (LPM)	Lookup key is the IP address. Each table entries has an associated IP prefix (IP and depth). The table lookup operation selects the IP prefix that is matched by the lookup key; in case of multiple matches, the entry with the longest prefix depth wins. Typically used to implement IP routing tables.
3	Access Control List (ACLs)	Lookup key is 7-tuple of two VLAN/MPLS labels, IP destination address, IP source addresses, L4 protocol, L4 destination port, L4 source port. Each table entry has an associated ACL and priority. The ACL contains bit masks for the VLAN/MPLS labels, IP prefix for IP destination address, IP prefix for IP source addresses, L4 protocol and bitmask, L4 destination port and bit mask, L4 source port and bit mask. The table lookup operation selects the ACL that is matched by the lookup key; in case of multiple matches, the

Table Interface

Each table is required to implement an abstract interface that defines the initialization and run-time operation of the table. The table abstract interface is described in Table 4.24.

Table 4.24: Table Abstract Interface

#	Table operation	Description
1	Create	Create the low-level data structures of the lookup table. Can internally allocate memory.
2	Free	Free up all the resources used by the lookup table.
3	Add entry	Add new entry to the lookup table.
4	Delete entry	Delete specific entry from the lookup table.
5	Lookup	Look up a burst of input packets and return a bit mask specifying the result of the lookup operation for each packet: a set bit signifies lookup hit for the corresponding packet, while a cleared bit a lookup miss. For each lookup hit packet, the lookup operation also returns a pointer to the table entry that was hit, which contains the actions to be applied on the packet and any associated metadata. For each lookup miss packet, the actions to be applied on the packet and any associated metadata are specified by the default table entry preconfigured for lookup miss.

Hash Table Design

Hash Table Overview

Hash tables are important because the key lookup operation is optimized for speed: instead of having to linearly search the lookup key through all the keys in the table, the search is limited to only the keys stored in a single table bucket.

Associative Arrays

An associative array is a function that can be specified as a set of (key, value) pairs, with each key from the possible set of input keys present at most once. For a given associative array, the possible operations are:

1. *add (key, value)*: When no value is currently associated with *key*, then the (key, *value*) association is created. When *key* is already associated value *value0*, then the association (*key*, *value0*) is removed and association *(key, value)* is created;
2. *delete key*: When no value is currently associated with *key*, this operation has no effect. When *key* is already associated *value*, then association *(key, value)* is removed;
3. *lookup key*: When no value is currently associated with *key*, then this operation returns void value (lookup miss). When *key* is associated with *value*, then this operation returns *value*. The *(key, value)* association is not changed.

The matching criterion used to compare the input key against the keys in the associative array is *exact match*, as the key size (number of bytes) and the key value (array of bytes) have to match exactly for the two keys under comparison.

Hash Function

A hash function deterministically maps data of variable length (key) to data of fixed size (hash value or key signature). Typically, the size of the key is bigger than the size of the key signature. The hash function basically compresses a long key into a short signature. Several keys can share the same signature (collisions).

High quality hash functions have uniform distribution. For large number of keys, when dividing the space of signature values into a fixed number of equal intervals (buckets), it is desirable to have the key signatures evenly distributed across these intervals (uniform distribution), as opposed to most of the signatures going into only a few of the intervals and the rest of the intervals being largely unused (non-uniform distribution).

Hash Table

A hash table is an associative array that uses a hash function for its operation. The reason for using a hash function is to optimize the performance of the lookup operation by minimizing the number of table keys that have to be compared against the input key.

Instead of storing the (key, value) pairs in a single list, the hash table maintains multiple lists (buckets). For any given key, there is a single bucket where that key might exist, and this bucket is uniquely identified based on the key signature. Once the key signature is computed and the hash table bucket identified, the key is either located in this bucket or it is not present in the hash table at all, so the key search can be narrowed down from the full set of keys currently in the table to just the set of keys currently in the identified table bucket.

The performance of the hash table lookup operation is greatly improved, provided that the table keys are evenly distributed among the hash table buckets, which can be achieved by using a hash function with uniform distribution. The rule to map a key to its bucket can simply be to use the key signature (modulo the number of table buckets) as the table bucket ID:

> *bucket_id = f_hash(key) % n_buckets;*

By selecting the number of buckets to be a power of two, the modulo operator can be replaced by a bitwise AND logical operation:

> *bucket_id = f_hash(key) & (n_buckets - 1);*

considering *n_bits* as the number of bits set in *bucket_mask = n_buckets - 1*, this means that all the keys that end up in the same hash table bucket have the lower *n_bits* of their signature identical. In order to reduce the number of keys in the same bucket (collisions), the number of hash table buckets needs to be increased.

In packet processing context, the sequence of operations involved in hash table operations is described in Fig. 4.52:

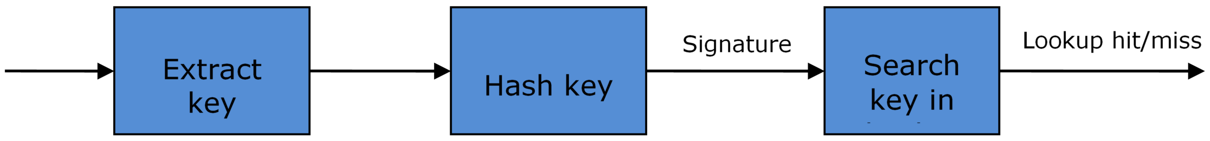


Fig. 4.52: Sequence of Steps for Hash Table Operations in a Packet Processing Context

Hash Table Use Cases

Flow Classification

Description: The flow classification is executed at least once for each input packet. This operation maps each incoming packet against one of the known traffic flows in the flow database that typically contains millions of flows.

Hash table name: Flow classification table

Number of keys: Millions

Key format: n-tuple of packet fields that uniquely identify a traffic flow/connection. Example: DiffServ 5-tuple of (Source IP address, Destination IP address, L4 protocol, L4 protocol source port, L4 protocol destination port). For IPv4 protocol and L4 protocols like TCP, UDP or SCTP, the size of the DiffServ 5-tuple is 13 bytes, while for IPv6 it is 37 bytes.

Key value (key data): actions and action meta-data describing what processing to be applied for the packets of the current flow. The size of the data associated with each traffic flow can vary from 8 bytes to kilobytes.

Address Resolution Protocol (ARP)

Description: Once a route has been identified for an IP packet (so the output interface and the IP address of the next hop station are known), the MAC address of the next hop station is needed in order to send this packet onto the next leg of the journey towards its destination (as identified by its destination IP address). The MAC address of the next hop station becomes the destination MAC address of the outgoing Ethernet frame.

Hash table name: ARP table

Number of keys: Thousands

Key format: The pair of (Output interface, Next Hop IP address), which is typically 5 bytes for IPv4 and 17 bytes for IPv6.

Key value (key data): MAC address of the next hop station (6 bytes).

Hash Table Types

Table 4.25 lists the hash table configuration parameters shared by all different hash table types.

Table 4.25: Configuration Parameters Common for All Hash Table Types

#	Parameter	Details
1	Key size	Measured as number of bytes. All keys have the same size.
2	Key value (key data) size	Measured as number of bytes.
3	Number of buckets	Needs to be a power of two.
4	Maximum number of keys	Needs to be a power of two.
5	Hash function	Examples: jhash, CRC hash, etc.
6	Hash function seed	Parameter to be passed to the hash function.
7	Key offset	Offset of the lookup key byte array within the packet meta-data stored in the packet buffer.

Bucket Full Problem On initialization, each hash table bucket is allocated space for exactly 4 keys. As keys are added to the table, it can happen that a given bucket already has 4 keys when a new key has to be added to this bucket. The possible options are:

1. **Least Recently Used (LRU) Hash Table.** One of the existing keys in the bucket is deleted and the new key is added in its place. The number of keys in each bucket never grows bigger than 4. The logic to pick the key to be dropped from the bucket is LRU. The hash table lookup operation maintains the order in which the keys in the same bucket are hit, so every time a key is hit, it becomes the new Most Recently Used (MRU) key, i.e. the last candidate for drop. When a key is added to the bucket, it also becomes the new MRU key. When a key needs to be picked and dropped, the first candidate for drop, i.e. the current LRU key, is always picked. The LRU logic requires maintaining specific data structures per each bucket.
2. **Extendable Bucket Hash Table.** The bucket is extended with space for 4 more keys. This is done by allocating additional memory at table initialization time, which is used to create a pool of free keys (the size of this pool is configurable and always a multiple of 4). On key add operation, the allocation of a group of 4 keys only happens successfully within the limit of free keys, otherwise the key add operation fails. On key delete operation, a group of 4 keys is freed back to the pool of free keys when the key to be deleted is the only key that was used within its group of 4 keys at that time. On key lookup operation, if the current bucket is in extended state and a match is not found in the first group of 4 keys, the search continues beyond the first group of 4 keys, potentially until all keys in this bucket are examined. The extendable bucket logic requires maintaining specific data structures per table and per each bucket.

Table 4.26: Configuration Parameters Specific to Extendable Bucket Hash Table

#	Parameter	Details
1	Number of additional keys	Needs to be a power of two, at least equal to 4.

Signature Computation The possible options for key signature computation are:

1. **Pre-computed key signature.** The key lookup operation is split between two CPU cores. The first CPU core (typically the CPU core that performs packet RX) extracts the key from the input packet, computes the key signature and saves both the key and the key

signature in the packet buffer as packet meta-data. The second CPU core reads both the key and the key signature from the packet meta-data and performs the bucket search step of the key lookup operation.

2. **Key signature computed on lookup ("do-sig" version).** The same CPU core reads the key from the packet meta-data, uses it to compute the key signature and also performs the bucket search step of the key lookup operation.

Table 4.27: Configuration Parameters Specific to Pre-computed Key Signature Hash Table

#	Parameter	Details
1	Signature offset	Offset of the pre-computed key signature within the packet meta-data.

Key Size Optimized Hash Tables For specific key sizes, the data structures and algorithm of key lookup operation can be specially handcrafted for further performance improvements, so following options are possible:

1. **Implementation supporting configurable key size.**
2. **Implementation supporting a single key size.** Typical key sizes are 8 bytes and 16 bytes.

Bucket Search Logic for Configurable Key Size Hash Tables

The performance of the bucket search logic is one of the main factors influencing the performance of the key lookup operation. The data structures and algorithm are designed to make the best use of Intel CPU architecture resources like: cache memory space, cache memory bandwidth, external memory bandwidth, multiple execution units working in parallel, out of order instruction execution, special CPU instructions, etc.

The bucket search logic handles multiple input packets in parallel. It is built as a pipeline of several stages (3 or 4), with each pipeline stage handling two different packets from the burst of input packets. On each pipeline iteration, the packets are pushed to the next pipeline stage: for the 4-stage pipeline, two packets (that just completed stage 3) exit the pipeline, two packets (that just completed stage 2) are now executing stage 3, two packets (that just completed stage 1) are now executing stage 2, two packets (that just completed stage 0) are now executing stage 1 and two packets (next two packets to read from the burst of input packets) are entering the pipeline to execute stage 0. The pipeline iterations continue until all packets from the burst of input packets execute the last stage of the pipeline.

The bucket search logic is broken into pipeline stages at the boundary of the next memory access. Each pipeline stage uses data structures that are stored (with high probability) into the L1 or L2 cache memory of the current CPU core and breaks just before the next memory access required by the algorithm. The current pipeline stage finalizes by prefetching the data structures required by the next pipeline stage, so given enough time for the prefetch to complete, when the next pipeline stage eventually gets executed for the same packets, it will read the data structures it needs from L1 or L2 cache memory and thus avoid the significant penalty incurred by L2 or L3 cache memory miss.

By prefetching the data structures required by the next pipeline stage in advance (before they are used) and switching to executing another pipeline stage for different packets, the number of L2 or L3 cache memory misses is greatly reduced, hence one of the main reasons for improved performance. This is because the cost of L2/L3 cache memory miss on memory read accesses is high, as usually due to data dependency between instructions, the CPU execution units have to stall until the read operation is completed from L3 cache memory or external DRAM memory. By using prefetch instructions, the latency of memory read accesses is hidden, provided that it is preformed early enough before the respective data structure is actually used.

By splitting the processing into several stages that are executed on different packets (the packets from the input burst are interlaced), enough work is created to allow the prefetch instructions to complete successfully (before the prefetched data structures are actually accessed) and also the data dependency between instructions is loosened. For example, for the 4-stage pipeline, stage 0 is executed on packets 0 and 1 and then, before same packets 0 and 1 are used (i.e. before stage 1 is executed on packets 0 and 1), different packets are used: packets 2 and 3 (executing stage 1), packets 4 and 5 (executing stage 2) and packets 6 and 7 (executing stage 3). By executing useful work while the data structures are brought into the L1 or L2 cache memory, the latency of the read memory accesses is hidden. By increasing the gap between two consecutive accesses to the same data structure, the data dependency between instructions is loosened; this allows making the best use of the super-scalar and out-of-order execution CPU architecture, as the number of CPU core execution units that are active (rather than idle or stalled due to data dependency constraints between instructions) is maximized.

The bucket search logic is also implemented without using any branch instructions. This avoids the important cost associated with flushing the CPU core execution pipeline on every instance of branch misprediction.

Configurable Key Size Hash Table Fig. 4.53, Table 4.28 and Table 4.29 detail the main data structures used to implement configurable key size hash tables (either LRU or extendable bucket, either with pre-computed signature or “do-sig”).

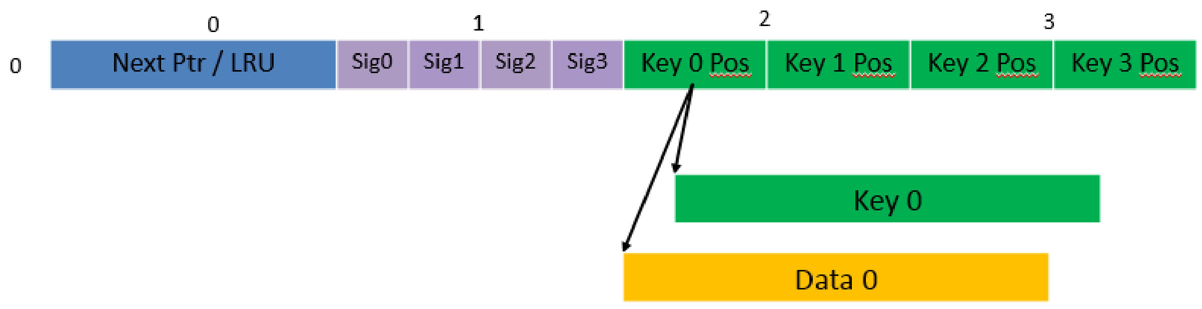


Fig. 4.53: Data Structures for Configurable Key Size Hash Tables

Table 4.28: Main Large Data Structures (Arrays) used for Configurable Key Size Hash Tables

#	Array name	Number of entries	Entry size (bytes)	Description
1	Bucket array	n_buckets (configurable)	32	Buckets of the hash table.
2	Bucket extensions array	n_buckets_ext (configurable)	32	This array is only created for extendable bucket tables.
3	Key array	n_keys	key_size (configurable)	Keys added to the hash table.
4	Data array	n_keys	entry_size (configurable)	Key values (key data) associated with the hash table keys.

Table 4.29: Field Description for Bucket Array Entry (Configurable Key Size Hash Tables)

#	Field name	Field size (bytes)	Description
1	Next Ptr/LRU	8	For LRU tables, this fields represents the LRU list for the current bucket stored as array of 4 entries of 2 bytes each. Entry 0 stores the index (0 .. 3) of the MRU key, while entry 3 stores the index of the LRU key. For extendable bucket tables, this field represents the next pointer (i.e. the pointer to the next group of 4 keys linked to the current bucket). The next pointer is not NULL if the bucket is currently extended or NULL otherwise. To help the branchless implementation, bit 0 (least significant bit) of this field is set to 1 if the next pointer is not NULL and to 0 otherwise.
2	Sig[0 .. 3]	4 x 2	If key X (X = 0 .. 3) is valid, then sig X bits 15 .. 1 store the most significant 15 bits of key X signature and sig X bit 0 is set to 1. If key X is not valid, then sig X is set to zero.
3	Key Pos [0 .. 3]	4 x 4	If key X is valid (X = 0 .. 3), then Key Pos X represents the index into the key array where key X is stored, as well as the index into the data array where the value associated with key X is stored. If key X is not valid, then the value of Key Pos X is undefined.

Fig. 4.54 and Table 4.30 detail the bucket search pipeline stages (either LRU or extendable bucket, either with pre-computed signature or "do-sig"). For each pipeline stage, the described operations are applied to each of the two packets handled by that stage.

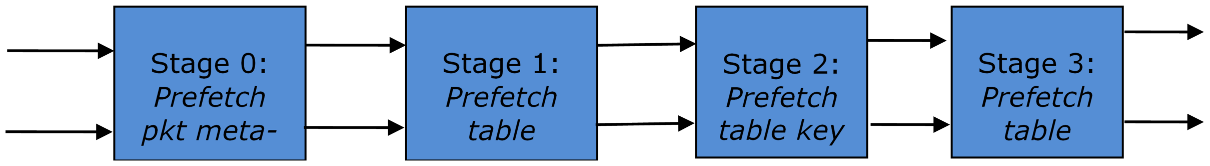


Fig. 4.54: Bucket Search Pipeline for Key Lookup Operation (Configurable Key Size Hash Tables)

Table 4.30: Description of the Bucket Search Pipeline Stages (Configurable Key Size Hash Tables)

#	Stage name	Description
0	Prefetch packet meta-data	Select next two packets from the burst of input packets. Prefetch packet meta-data containing the key and key signature.
1	Prefetch table bucket	Read the key signature from the packet meta-data (for extendable bucket hash tables) or read the key from the packet meta-data and compute key signature (for LRU tables). Identify the bucket ID using the key signature. Set bit 0 of the signature to 1 (to match only signatures of valid keys from the table). Prefetch the bucket.
2	Prefetch table key	Read the key signatures from the bucket. Compare the signature of the input key against the 4 key signatures from the packet. As result, the following is obtained: *match* = equal to TRUE if there was at least one signature match and to FALSE in the case of no signature match; *match_many* = equal to TRUE is there were more than one signature matches (can be up to 4 signature matches in the worst case scenario) and to FALSE otherwise; *match_pos* = the index of the first key that produced signature match (only valid if match is true). For extendable bucket hash tables only, set *match_many* to TRUE if next pointer is valid. Prefetch the bucket key indicated by *match_pos* (even if *match_pos* does not point to valid key valid).
3	Prefetch table data	Read the bucket key indicated by *match_pos*. Compare the bucket key against the input key. As re-

Additional notes:

1. The pipelined version of the bucket search algorithm is executed only if there are at least 7 packets in the burst of input packets. If there are less than 7 packets in the burst of input packets, a non-optimized implementation of the bucket search algorithm is executed.
2. Once the pipelined version of the bucket search algorithm has been executed for all the packets in the burst of input packets, the non-optimized implementation of the bucket search algorithm is also executed for any packets that did not produce a lookup hit, but have the *match_many* flag set. As result of executing the non-optimized version, some of these packets may produce a lookup hit or lookup miss. This does not impact the performance of the key lookup operation, as the probability of matching more than one signature in the same group of 4 keys or of having the bucket in extended state (for extendable bucket hash tables only) is relatively small.

Key Signature Comparison Logic

The key signature comparison logic is described in Table 4.31.

Table 4.31: Lookup Tables for Match, Match_Many and Match_Pos

#	mask	match (1 bit)	match_many (1 bit)	match_pos (2 bits)
0	0000	0	0	00
1	0001	1	0	00
2	0010	1	0	01
3	0011	1	1	00
4	0100	1	0	10
5	0101	1	1	00
6	0110	1	1	01
7	0111	1	1	00
8	1000	1	0	11
9	1001	1	1	00
10	1010	1	1	01
11	1011	1	1	00
12	1100	1	1	10
13	1101	1	1	00
14	1110	1	1	01
15	1111	1	1	00

The input *mask* hash bit X (X = 0 .. 3) set to 1 if input signature is equal to bucket signature X and set to 0 otherwise. The outputs *match*, *match_many* and *match_pos* are 1 bit, 1 bit and 2 bits in size respectively and their meaning has been explained above.

As displayed in Table 4.32, the lookup tables for *match* and *match_many* can be collapsed into a single 32-bit value and the lookup table for *match_pos* can be collapsed into a 64-bit value. Given the input *mask*, the values for *match*, *match_many* and *match_pos* can be obtained by indexing their respective bit array to extract 1 bit, 1 bit and 2 bits respectively with branchless logic.

Table 4.32: Collapsed Lookup Tables for Match, Match_Many and Match_Pos

	Bit array	Hexadecimal value
match	1111_1111_1111_1110	0xFFFELLU
match_many	1111_1110_1110_1000	0xFEE8LLU
match_pos	0001_0010_0001_0011__0001_0010_0001_0000	0x12131210LLU

The pseudo-code for match, match_many and match_pos is:

```
match = (0xFFFELLU >> mask) & 1;

match_many = (0xFEE8LLU >> mask) & 1;

match_pos = (0x12131210LLU >> (mask << 1)) & 3;
```

Single Key Size Hash Tables Fig. 4.55, Fig. 4.56, Table 4.33 and Table 4.34 detail the main data structures used to implement 8-byte and 16-byte key hash tables (either LRU or extendable bucket, either with pre-computed signature or "do-sig").

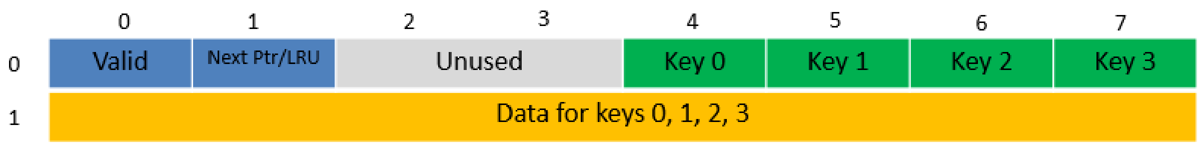


Fig. 4.55: Data Structures for 8-byte Key Hash Tables

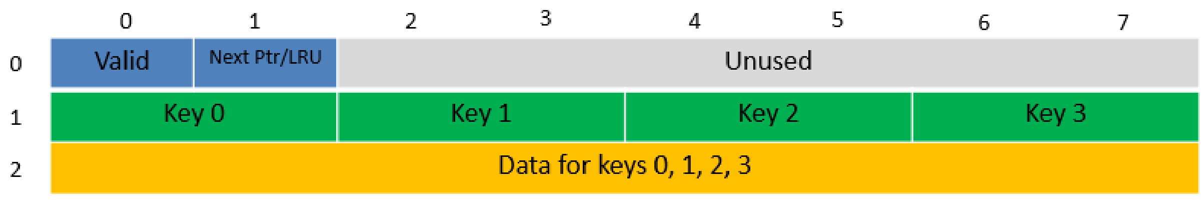


Fig. 4.56: Data Structures for 16-byte Key Hash Tables

Table 4.33: Main Large Data Structures (Arrays) used for 8-byte and 16-byte Key Size Hash Tables

#	Array name	Number of entries	Entry size (bytes)	Description
1	Bucket array	n_buckets (configurable)	*8-byte key size:* 64 + 4 x entry_size *16-byte key size:* 128 + 4 x entry_size	Buckets of the hash table.
2	Bucket extensions array	n_buckets_ext (configurable)	*8-byte key size:* 64 + 4 x entry_size *16-byte key size:* 128 + 4 x entry_size	This array is only created for extendable bucket tables.

Table 4.34: Field Description for Bucket Array Entry (8-byte and 16-byte Key Hash Tables)

#	Field name	Field size (bytes)	Description
1	Valid	8	Bit X (X = 0 .. 3) is set to 1 if key X is valid or to 0 otherwise. Bit 4 is only used for extendable bucket tables to help with the implementation of the branchless logic. In this case, bit 4 is set to 1 if next pointer is valid (not NULL) or to 0 otherwise.
2	Next Ptr/LRU	8	For LRU tables, this fields represents the LRU list for the current bucket stored as array of 4 entries of 2 bytes each. Entry 0 stores the index (0 .. 3) of the MRU key, while entry 3 stores the index of the LRU key. For extendable bucket tables, this field represents the next pointer (i.e. the pointer to the next group of 4 keys linked to the current bucket). The next pointer is not NULL if the bucket is currently extended or NULL otherwise.
3	Key [0 .. 3]	4 x key_size	Full keys.
4	Data [0 .. 3]	4 x entry_size	Full key values (key data) associated with keys 0 .. 3.

and detail the bucket search pipeline used to implement 8-byte and 16-byte key hash tables (either LRU or extendable bucket, either with pre-computed signature or “do-sig”). For each pipeline stage, the described operations are applied to each of the two packets handled by that stage.

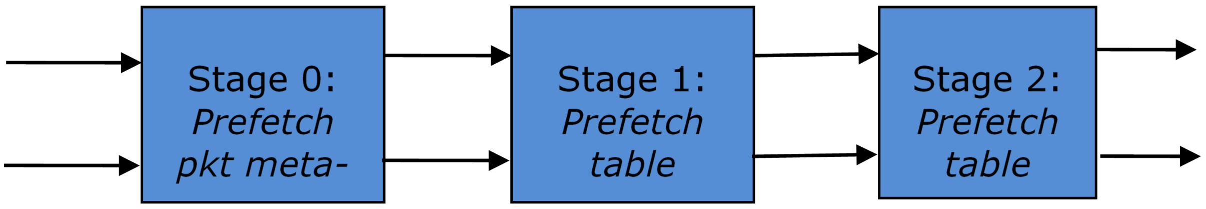


Fig. 4.57: Bucket Search Pipeline for Key Lookup Operation (Single Key Size Hash Tables)

Table 4.35: Description of the Bucket Search Pipeline Stages (8-byte and 16-byte Key Hash Tables)

#	Stage name	Description
0	Prefetch packet meta-data	1. Select next two packets from the burst of input packets. 2. Prefetch packet meta-data containing the key and key signature.
1	Prefetch table bucket	1. Read the key signature from the packet meta-data (for extendable bucket hash tables) or read the key from the packet meta-data and compute key signature (for LRU tables). 2. Identify the bucket ID using the key signature. 3. Prefetch the bucket.
2	Prefetch table data	1. Read the bucket. 2. Compare all 4 bucket keys against the input key. 3. Report input key as lookup hit only when a match is identified (more than one key match is not possible) 4. For LRU tables only, use branchless logic to update the bucket LRU list (the current key becomes the new MRU) only on lookup hit. 5. Prefetch the key value (key data) associated with the matched key (to avoid branches, this is done on both lookup hit and miss).

Additional notes:

1. The pipelined version of the bucket search algorithm is executed only if there are at least 5 packets in the burst of input packets. If there are less than 5 packets in the burst of input packets, a non-optimized implementation of the bucket search algorithm is executed.
2. For extendable bucket hash tables only, once the pipelined version of the bucket search algorithm has been executed for all the packets in the burst of input packets, the non-optimized implementation of the bucket search algorithm is also executed for any packets that did not produce a lookup hit, but have the bucket in extended state. As result of executing the non-optimized version, some of these packets may produce a lookup hit or lookup miss. This does not impact the performance of the key lookup operation, as the probability of having the bucket in extended state is relatively small.

4.23.5 Pipeline Library Design

A pipeline is defined by:

1. The set of input ports;
2. The set of output ports;
3. The set of tables;
4. The set of actions.

The input ports are connected with the output ports through tree-like topologies of interconnected tables. The table entries contain the actions defining the operations to be executed on the input packets and the packet flow within the pipeline.

Connectivity of Ports and Tables

To avoid any dependencies on the order in which pipeline elements are created, the connectivity of pipeline elements is defined after all the pipeline input ports, output ports and tables have been created.

General connectivity rules:

1. Each input port is connected to a single table. No input port should be left unconnected;
2. The table connectivity to other tables or to output ports is regulated by the next hop actions of each table entry and the default table entry. The table connectivity is fluid, as the table entries and the default table entry can be updated during run-time.
    - A table can have multiple entries (including the default entry) connected to the same output port. A table can have different entries connected to different output ports. Different tables can have entries (including default table entry) connected to the same output port.
    - A table can have multiple entries (including the default entry) connected to another table, in which case all these entries have to point to the same table. This constraint is enforced by the API and prevents tree-like topologies from being created (allowing table chaining only), with the purpose of simplifying the implementation of the pipeline run-time execution engine.

Port Actions

Port Action Handler

An action handler can be assigned to each input/output port to define actions to be executed on each input packet that is received by the port. Defining the action handler for a specific input/output port is optional (i.e. the action handler can be disabled).

For input ports, the action handler is executed after RX function. For output ports, the action handler is executed before the TX function.

The action handler can decide to drop packets.

Table Actions

Table Action Handler

An action handler to be executed on each input packet can be assigned to each table. Defining the action handler for a specific table is optional (i.e. the action handler can be disabled).

The action handler is executed after the table lookup operation is performed and the table entry associated with each input packet is identified. The action handler can only handle the user-defined actions, while the reserved actions (e.g. the next hop actions) are handled by the Packet Framework. The action handler can decide to drop the input packet.

Reserved Actions

The reserved actions are handled directly by the Packet Framework without the user being able to change their meaning through the table action handler configuration. A special category of the reserved actions is represented by the next hop actions, which regulate the packet flow between input ports, tables and output ports through the pipeline. Table 4.36 lists the next hop actions.

Table 4.36: Next Hop Actions (Reserved)

#	Next hop action	Description
1	Drop	Drop the current packet.
2	Send to output port	Send the current packet to specified output port. The output port ID is metadata stored in the same table entry.
3	Send to table	Send the current packet to specified table. The table ID is metadata stored in the same table entry.

User Actions

For each table, the meaning of user actions is defined through the configuration of the table action handler. Different tables can be configured with different action handlers, therefore the meaning of the user actions and their associated meta-data is private to each table. Within the same table, all the table entries (including the table default entry) share the same definition for the user actions and their associated meta-data, with each table entry having its own set of enabled user actions and its own copy of the action meta-data. Table 4.37 contains a non-exhaustive list of user action examples.

Table 4.37: User Action Examples

#	User action	Description
1	Metering	Per flow traffic metering using the srTCM and trTCM algorithms.
2	Statistics	Update the statistics counters maintained per flow.
3	App ID	Per flow state machine fed by variable length sequence of packets at the flow initialization with the purpose of identifying the traffic type and application.
4	Push/pop labels	Push/pop VLAN/MPLS labels to/from the current packet.
5	Network Address Translation (NAT)	Translate between the internal (LAN) and external (WAN) IP destination/source address and/or L4 protocol destination/source port.
6	TTL update	Decrement IP TTL and, in case of IPv4 packets, update the IP checksum.

4.23.6 Multicore Scaling

A complex application is typically split across multiple cores, with cores communicating through SW queues. There is usually a performance limit on the number of table lookups and actions that can be fitted on the same CPU core due to HW constraints like: available CPU cycles, cache memory size, cache transfer BW, memory transfer BW, etc.

As the application is split across multiple CPU cores, the Packet Framework facilitates the creation of several pipelines, the assignment of each such pipeline to a different CPU core and the interconnection of all CPU core-level pipelines into a single application-level complex pipeline. For example, if CPU core A is assigned to run pipeline P1 and CPU core B pipeline P2, then the interconnection of P1 with P2 could be achieved by having the same set of SW queues act like output ports for P1 and input ports for P2.

This approach enables the application development using the pipeline, run-to-completion (clustered) or hybrid (mixed) models.

It is allowed for the same core to run several pipelines, but it is not allowed for several cores to run the same pipeline.

Shared Data Structures

The threads performing table lookup are actually table writers rather than just readers. Even if the specific table lookup algorithm is thread-safe for multiple readers (e. g. read-only access of the search algorithm data structures is enough to conduct the lookup operation), once the table entry for the current packet is identified, the thread is typically expected to update the action meta-data stored in the table entry (e.g. increment the counter tracking the number of packets that hit this table entry), and thus modify the table entry. During the time this thread is accessing this table entry (either writing or reading; duration is application specific), for data consistency reasons, no other threads (threads performing table lookup or entry add/delete operations) are allowed to modify this table entry.

Mechanisms to share the same table between multiple threads:

1. **Multiple writer threads.** Threads need to use synchronization primitives like semaphores (distinct semaphore per table entry) or atomic instructions. The cost of semaphores is usually high, even when the semaphore is free. The cost of atomic instructions is normally higher than the cost of regular instructions.
2. **Multiple writer threads, with single thread performing table lookup operations and multiple threads performing table entry add/delete operations.** The threads performing table entry add/delete operations send table update requests to the reader (typically through message passing queues), which does the actual table updates and then sends the response back to the request initiator.
3. **Single writer thread performing table entry add/delete operations and multiple reader threads that perform table lookup operations with read-only access to the table entries.** The reader threads use the main table copy while the writer is updating the mirror copy. Once the writer update is done, the writer can signal to the readers and busy wait until all readers swaps between the mirror copy (which now becomes the main copy) and the mirror copy (which now becomes the main copy).

4.23.7 Interfacing with Accelerators

The presence of accelerators is usually detected during the initialization phase by inspecting the HW devices that are part of the system (e.g. by PCI bus enumeration). Typical devices with acceleration capabilities are:

- Inline accelerators: NICs, switches, FPGAs, etc;
- Look-aside accelerators: chipsets, FPGAs, etc.

Usually, to support a specific functional block, specific implementation of Packet Framework tables and/or ports and/or actions has to be provided for each accelerator, with all the implementations sharing the same API: pure SW implementation (no acceleration), implementation using accelerator A, implementation using accelerator B, etc. The selection between these implementations could be done at build time or at run-time (recommended), based on which accelerators are present in the system, with no application changes required.

4.24 Vhost Library

The vhost library implements a user space vhost driver. It supports both vhost-cuse (cuse: user space character device) and vhost-user(user space socket server). It also creates, manages and destroys vhost devices for corresponding virtio devices in the guest. Vhost supported vSwitch could register callbacks to this library, which will be called when a vhost device is activated or deactivated by guest virtual machine.

4.24.1 Vhost API Overview

- Vhost driver registration

  rte_vhost_driver_register registers the vhost driver into the system. For vhost-cuse, character device file will be created under the /dev directory. Character device name is specified as the parameter. For vhost-user, a Unix domain socket server will be created with the parameter as the local socket path.

- Vhost session start

  rte_vhost_driver_session_start starts the vhost session loop. Vhost session is an infinite blocking loop. Put the session in a dedicate DPDK thread.

- Callback register

  Vhost supported vSwitch could call rte_vhost_driver_callback_register to register two callbacks, new_destory and destroy_device. When virtio device is activated or deactivated by guest virtual machine, the callback will be called, then vSwitch could put the device onto data core or remove the device from data core by setting or unsetting VIRTIO_DEV_RUNNING on the device flags.

- Read/write packets from/to guest virtual machine

  rte_vhost_enqueue_burst transmit host packets to guest. rte_vhost_dequeue_burst receives packets from guest.

- Feature enable/disable

  Now one negotiate-able feature in vhost is merge-able. vSwitch could enable/disable this feature for performance consideration.

4.24.2 Vhost Implementation

Vhost cuse implementation

When vSwitch registers the vhost driver, it will register a cuse device driver into the system and creates a character device file. This cuse driver will receive vhost open/release/IOCTL message from QEMU simulator.

When the open call is received, vhost driver will create a vhost device for the virtio device in the guest.

When VHOST_SET_MEM_TABLE IOCTL is received, vhost searches the memory region to find the starting user space virtual address that maps the memory of guest virtual machine. Through this virtual address and the QEMU pid, vhost could find the file QEMU uses to map the guest memory. Vhost maps this file into its address space, in this way vhost could fully access the guest physical memory, which means vhost could access the shared virtio ring and the guest physical address specified in the entry of the ring.

The guest virtual machine tells the vhost whether the virtio device is ready for processing or is de-activated through VHOST_NET_SET_BACKEND message. The registered callback from vSwitch will be called.

When the release call is released, vhost will destroy the device.

Vhost user implementation

When vSwitch registers a vhost driver, it will create a Unix domain socket server into the system. This server will listen for a connection and process the vhost message from QEMU simulator.

When there is a new socket connection, it means a new virtio device has been created in the guest virtual machine, and the vhost driver will create a vhost device for this virtio device.

For messages with a file descriptor, the file descriptor could be directly used in the vhost process as it is already installed by Unix domain socket.

- VHOST_SET_MEM_TABLE
- VHOST_SET_VRING_KICK
- VHOST_SET_VRING_CALL
- VHOST_SET_LOG_FD
- VHOST_SET_VRING_ERR

For VHOST_SET_MEM_TABLE message, QEMU will send us information for each memory region and its file descriptor in the ancillary data of the message. The fd is used to map that region.

There is no VHOST_NET_SET_BACKEND message as in vhost cuse to signal us whether virtio device is ready or should be stopped. VHOST_SET_VRING_KICK is used as the signal to put the vhost device onto data plane. VHOST_GET_VRING_BASE is used as the signal to remove vhost device from data plane.

When the socket connection is closed, vhost will destroy the device.

4.24.3 Vhost supported vSwitch reference

For more vhost details and how to support vhost in vSwitch, please refer to vhost example in the DPDK Sample Applications Guide.

4.25 Port Hotplug Framework

The Port Hotplug Framework provides DPDK applications with the ability to attach and detach ports at runtime. Because the framework depends on PMD implementation, the ports that PMDs cannot handle are out of scope of this framework. Furthermore, after detaching a port from a DPDK application, the framework doesn't provide a way for removing the devices from the system. For the ports backed by a physical NIC, the kernel will need to support PCI Hotplug feature.

4.25.1 Overview

The basic requirements of the Port Hotplug Framework are:

- DPDK applications that use the Port Hotplug Framework must manage their own ports.

  The Port Hotplug Framework is implemented to allow DPDK applications to manage ports. For example, when DPDK applications call the port attach function, the attached port number is returned. DPDK applications can also detach the port by port number.

- Kernel support is needed for attaching or detaching physical device ports.

  To attach new physical device ports, the device will be recognized by userspace driver I/O framework in kernel at first. Then DPDK applications can call the Port Hotplug functions to attach the ports. For detaching, steps are vice versa.

- Before detaching, they must be stopped and closed.

  DPDK applications must call “rte_eth_dev_stop()” and “rte_eth_dev_close()” APIs before detaching ports. These functions will start finalization sequence of the PMDs.

- The framework doesn’t affect legacy DPDK applications behavior.

  If the Port Hotplug functions aren’t called, all legacy DPDK apps can still work without modifications.

4.25.2 Port Hotplug API overview

- Attaching a port

  “rte_eth_dev_attach()” API attaches a port to DPDK application, and returns the attached port number. Before calling the API, the device should be recognized by an userspace driver I/O framework. The API receives a pci address like “0000:01:00.0” or a virtual device name like “eth_pcap0,iface=eth0”. In the case of virtual device name, the format is the same as the general “–vdev” option of DPDK.

- Detaching a port

  “rte_eth_dev_detach()” API detaches a port from DPDK application, and returns a pci address of the detached device or a virtual device name of the device.

4.25.3 Reference

“testpmd” supports the Port Hotplug Framework.

4.25.4 Limitations

- The Port Hotplug APIs are not thread safe.
- The framework can only be enabled with Linux. BSD is not supported.
- To detach a port, the port should be backed by a device that igb_uio manages. VFIO is not supported.
- Not all PMDs support detaching feature. To know whether a PMD can support detaching, search for the “RTE_PCI_DRV_DETACHABLE” flag in PMD implementation. If the flag is defined in the PMD, detaching is supported.

Part 2: Development Environment

4.26 Source Organization

This section describes the organization of sources in the DPDK framework.

4.26.1 Makefiles and Config

Note: In the following descriptions, `RTE_SDK` is the environment variable that points to the base directory into which the tarball was extracted. See *Useful Variables Provided by the Build System* for descriptions of other variables.

Makefiles that are provided by the DPDK libraries and applications are located in `$(RTE_SDK)/mk`.

Config templates are located in `$(RTE_SDK)/config`. The templates describe the options that are enabled for each target. The config file also contains items that can be enabled and disabled for many of the DPDK libraries, including debug options. The user should look at the config file and become familiar with these options. The config file is also used to create a header file, which will be located in the new build directory.

4.26.2 Libraries

Libraries are located in subdirectories of `$(RTE_SDK)/lib`. By convention a library refers to any code that provides an API to an application. Typically, it generates an archive file (`.a`), but a kernel module would also go in the same directory.

The lib directory contains:

```
lib
+-- librte_cmdline      # Command line interface helper
+-- librte_distributor  # Packet distributor
+-- librte_eal          # Environment abstraction layer
+-- librte_ether        # Generic interface to poll mode driver
+-- librte_hash         # Hash library
+-- librte_ip_frag      # IP fragmentation library
+-- librte_ivshmem      # QEMU IVSHMEM library
+-- librte_kni          # Kernel NIC interface
+-- librte_kvargs       # Argument parsing library
+-- librte_lpm          # Longest prefix match library
+-- librte_mbuf         # Packet and control mbuf manipulation
+-- librte_mempool      # Memory pool manager (fixed sized objects)
+-- librte_meter        # QoS metering library
+-- librte_net          # Various IP-related headers
+-- librte_power        # Power management library
+-- librte_ring         # Software rings (act as lockless FIFOs)
+-- librte_sched        # QoS scheduler and dropper library
+-- librte_timer        # Timer library
```

4.26.3 Drivers

Drivers are special libraries which provide poll-mode driver implementations for devices: either hardware devices or pseudo/virtual devices. They are contained in the *drivers* subdirectory, classified by type, and each compiles to a library with the format `librte_pmd_X.a` where `X` is the driver name.

The drivers directory has a *net* subdirectory which contains:

```
drivers/net
+-- af_packet          # Poll mode driver based on Linux af_packet
+-- bonding            # Bonding poll mode driver
+-- cxgbe               # Chelsio Terminator 10GbE/40GbE poll mode driver
+-- e1000               # 1GbE poll mode drivers (igb and em)
+-- enic                # Cisco VIC Ethernet NIC Poll-mode Driver
+-- fm10k               # Host interface PMD driver for FM10000 Series
+-- i40e                # 40GbE poll mode driver
+-- ixgbe               # 10GbE poll mode driver
+-- mlx4                # Mellanox ConnectX-3 poll mode driver
+-- null                # NULL poll mode driver for testing
+-- pcap                # PCAP poll mode driver
+-- ring                # Ring poll mode driver
+-- szedata2            # SZEDATA2 poll mode driver
+-- virtio              # Virtio poll mode driver
+-- vmxnet3             # VMXNET3 poll mode driver
+-- xenvirt             # Xen virtio poll mode driver
```

Note: Several of the `driver/net` directories contain a `base` sub-directory. The `base` directory generally contains code the shouldn't be modified directly by the user. Any enhancements should be done via the `X_osdep.c` and/or `X_osdep.h` files in that directory. Refer to the local README in the base directories for driver specific instructions.

4.26.4 Applications

Applications are source files that contain a `main()` function. They are located in the `$(RTE_SDK)/app` and `$(RTE_SDK)/examples` directories.

The app directory contains sample applications that are used to test DPDK (such as autotests) or the Poll Mode Drivers (test-pmd):

```
app
+-- chkincs             # Test program to check include dependencies
+-- cmdline_test        # Test the commandline library
+-- test                # Autotests to validate DPDK features
+-- test-acl            # Test the ACL library
+-- test-pipeline       # Test the IP Pipeline framework
+-- test-pmd            # Test and benchmark poll mode drivers
```

The examples directory contains sample applications that show how libraries can be used:

```
examples
+-- cmdline             # Example of using the cmdline library
+-- dpdk_qat            # Sample integration with Intel QuickAssist
+-- exception_path      # Sending packets to and from Linux TAP device
+-- helloworld          # Basic Hello World example
+-- ip_reassembly       # Example showing IP reassembly
+-- ip_fragmentation    # Example showing IPv4 fragmentation
+-- ipv4_multicast      # Example showing IPv4 multicast
+-- kni                 # Kernel NIC Interface (KNI) example
+-- l2fwd               # L2 forwarding with and without SR-IOV
+-- l3fwd               # L3 forwarding example
+-- l3fwd-power         # L3 forwarding example with power management
+-- l3fwd-vf            # L3 forwarding example with SR-IOV
+-- link_status_interrupt # Link status change interrupt example
+-- load_balancer       # Load balancing across multiple cores/sockets
+-- multi_process       # Example apps using multiple DPDK processes
+-- qos_meter           # QoS metering example
+-- qos_sched           # QoS scheduler and dropper example
+-- timer               # Example of using librte_timer library
+-- vmdq_dcb            # Example of VMDQ and DCB receiving
+-- vmdq                # Example of VMDQ receiving
+-- vhost               # Example of userspace vhost and switch
```

Note: The actual examples directory may contain additional sample applications to those shown above. Check the latest DPDK source files for details.

4.27 Development Kit Build System

The DPDK requires a build system for compilation activities and so on. This section describes the constraints and the mechanisms used in the DPDK framework.

There are two use-cases for the framework:

- Compilation of the DPDK libraries and sample applications; the framework generates specific binary libraries, include files and sample applications
- Compilation of an external application or library, using an installed binary DPDK

4.27.1 Building the Development Kit Binary

The following provides details on how to build the DPDK binary.

Build Directory Concept

After installation, a build directory structure is created. Each build directory contains include files, libraries, and applications:

```
~/DPDK$ ls
app                                 MAINTAINERS
config                              Makefile
COPYRIGHT                           mk
doc                                 scripts
examples                            lib
tools                               x86_64-native-linuxapp-gcc
x86_64-native-linuxapp-icc          i686-native-linuxapp-gcc
i686-native-linuxapp-icc

...
~/DEV/DPDK$ ls i686-native-linuxapp-gcc

app build hostapp include kmod lib Makefile


~/DEV/DPDK$ ls i686-native-linuxapp-gcc/app/
cmdline_test   dump_cfg     test     testpmd
cmdline_test.map       dump_cfg.map   test.map
        testpmd.map


~/DEV/DPDK$ ls i686-native-linuxapp-gcc/lib/

libethdev.a  librte_hash.a  librte_mbuf.a librte_pmd_ixgbe.a

librte_cmdline.a librte_lpm.a librte_mempool.a librte_ring.a
```

```
librte_eal.a librte_pmd_e1000.a librte_timer.a


~/DEV/DPDK$ ls i686-native-linuxapp-gcc/include/
arch                         rte_cpuflags.h       rte_memcpy.h
cmdline_cirbuf.h             rte_cycles.h         rte_memory.h
cmdline.h                    rte_debug.h          rte_mempool.h
cmdline_parse_etheraddr.h    rte_eal.h            rte_memzone.h
cmdline_parse.h              rte_errno.h          rte_pci_dev_ids.h
cmdline_parse_ipaddr.h       rte_ethdev.h         rte_pci.h
cmdline_parse_num.h          rte_ether.h          rte_per_lcore.h
cmdline_parse_portlist.h     rte_fbk_hash.h       rte_prefetch.h
cmdline_parse_string.h       rte_hash_crc.h       rte_random.h
cmdline_rdline.h             rte_hash.h           rte_ring.h
cmdline_socket.h             rte_interrupts.h     rte_rwlock.h
cmdline_vt100.h              rte_ip.h             rte_sctp.h
exec-env                     rte_jhash.h          rte_spinlock.h
rte_alarm.h                  rte_launch.h         rte_string_fns.h
rte_atomic.h                 rte_lcore.h          rte_tailq.h
rte_branch_prediction.h      rte_log.h            rte_tcp.h
rte_byteorder.h              rte_lpm.h            rte_timer.h
rte_common.h                 rte_malloc.h         rte_udp.h
rte_config.h                 rte_mbuf.h
```

A build directory is specific to a configuration that includes architecture + execution environment + toolchain. It is possible to have several build directories sharing the same sources with different configurations.

For instance, to create a new build directory called my_sdk_build_dir using the default configuration template config/defconfig_x86_64-linuxapp, we use:

```
cd ${RTE_SDK}
make config T=x86_64-native-linuxapp-gcc O=my_sdk_build_dir
```

This creates a new my_sdk_build_dir directory. After that, we can compile by doing:

```
cd my_sdk_build_dir
make
```

which is equivalent to:

```
make O=my_sdk_build_dir
```

The content of the my_sdk_build_dir is then:

```
-- .config                          # used configuration

-- Makefile                         # wrapper that calls head Makefile
                                    # with $PWD as build directory


    -- build                               #All temporary files used during build
    +--app                                 # process, including . o, .d, and .cmd files.
        |  +-- test                        # For libraries, we have the .a file.
        |  +-- test.o                      # For applications, we have the elf file.
        |  `-- ...
        +-- lib
            +-- librte_eal
            |   `-- ...
            +-- librte_mempool
            |  +--  mempool-file1.o
            |  +--  .mempool-file1.o.cmd
            |  +--  .mempool-file1.o.d
            |  +--   mempool-file2.o
            |  +--  .mempool-file2.o.cmd
            |  +--  .mempool-file2.o.d
            |  `--  mempool.a
            `-- ...

-- include                 # All include files installed by libraries
    +-- librte_mempool.h   # and applications are located in this
    +-- rte_eal.h          # directory. The installed files can depend
    +-- rte_spinlock.h     # on configuration if needed (environment,
    +-- rte_atomic.h       # architecture, ..)
    `-- \*.h ...

-- lib                     # all compiled libraries are copied in this
    +-- librte_eal.a       # directory
    +-- librte_mempool.a
    `-- \*.a ...

-- app                     # All compiled applications are installed
+ --test                   # here. It includes the binary in elf format
```

Refer to *Development Kit Root Makefile Help* for details about make commands that can be used from the root of DPDK.

4.27.2 Building External Applications

Since DPDK is in essence a development kit, the first objective of end users will be to create an application using this SDK. To compile an application, the user must set the RTE_SDK and RTE_TARGET environment variables.

```
export RTE_SDK=/opt/DPDK
export RTE_TARGET=x86_64-native-linuxapp-gcc
cd /path/to/my_app
```

For a new application, the user must create their own Makefile that includes some .mk files, such as ${RTE_SDK}/mk/rte.vars.mk, and ${RTE_SDK}/mk/ rte.app.mk. This is described in *Building Your Own Application*.

Depending on the chosen target (architecture, machine, executive environment, toolchain) defined in the Makefile or as an environment variable, the applications and libraries will compile using the appropriate .h files and will link with the appropriate .a files. These files are located in ${RTE_SDK}/arch-machine-execenv-toolchain, which is referenced internally by ${RTE_BIN_SDK}.

To compile their application, the user just has to call make. The compilation result will be located in /path/to/my_app/build directory.

Sample applications are provided in the examples directory.

4.27.3 Makefile Description

General Rules For DPDK Makefiles

In the DPDK, Makefiles always follow the same scheme:

1. Include $(RTE_SDK)/mk/rte.vars.mk at the beginning.
2. Define specific variables for RTE build system.

3. Include a specific $(RTE_SDK)/mk/rte.XYZ.mk, where XYZ can be app, lib, extapp, extlib, obj, gnuconfigure, and so on, depending on what kind of object you want to build. *See Makefile Types* below.
4. Include user-defined rules and variables.

   The following is a very simple example of an external application Makefile:

   ```
   include $(RTE_SDK)/mk/rte.vars.mk

   # binary name
   APP = helloworld

   # all source are stored in SRCS-y
   SRCS-y := main.c

   CFLAGS += -O3
   CFLAGS += $(WERROR_FLAGS)

   include $(RTE_SDK)/mk/rte.extapp.mk
   ```

Makefile Types

Depending on the .mk file which is included at the end of the user Makefile, the Makefile will have a different role. Note that it is not possible to build a library and an application in the same Makefile. For that, the user must create two separate Makefiles, possibly in two different directories.

In any case, the rte.vars.mk file must be included in the user Makefile as soon as possible.

Application

These Makefiles generate a binary application.

- rte.app.mk: Application in the development kit framework
- rte.extapp.mk: External application
- rte.hostapp.mk: Host application in the development kit framework

Library

Generate a .a library.

- rte.lib.mk: Library in the development kit framework
- rte.extlib.mk: external library
- rte.hostlib.mk: host library in the development kit framework

Install

- rte.install.mk: Does not build anything, it is only used to create links or copy files to the installation directory. This is useful for including files in the development kit framework.

Kernel Module

- rte.module.mk: Build a kernel module in the development kit framework.

Objects

- rte.obj.mk: Object aggregation (merge several .o in one) in the development kit framework.
- rte.extobj.mk: Object aggregation (merge several .o in one) outside the development kit framework.

Misc

- rte.doc.mk: Documentation in the development kit framework
- rte.gnuconfigure.mk: Build an application that is configure-based.
- rte.subdir.mk: Build several directories in the development kit framework.

Useful Variables Provided by the Build System

- RTE_SDK: The absolute path to the DPDK sources. When compiling the development kit, this variable is automatically set by the framework. It has to be defined by the user as an environment variable if compiling an external application.
- RTE_SRCDIR: The path to the root of the sources. When compiling the development kit, RTE_SRCDIR = RTE_SDK. When compiling an external application, the variable points to the root of external application sources.
- RTE_OUTPUT: The path to which output files are written. Typically, it is $(RTE_SRCDIR)/build, but it can be overridden by the O= option in the make command line.
- RTE_TARGET: A string identifying the target for which we are building. The format is arch-machine-execenv-toolchain. When compiling the SDK, the target is deduced by the build system from the configuration (.config). When building an external application, it must be specified by the user in the Makefile or as an environment variable.
- RTE_SDK_BIN: References $(RTE_SDK)/$(RTE_TARGET).
- RTE_ARCH: Defines the architecture (i686, x86_64). It is the same value as CONFIG_RTE_ARCH but without the double-quotes around the string.
- RTE_MACHINE: Defines the machine. It is the same value as CONFIG_RTE_MACHINE but without the double-quotes around the string.
- RTE_TOOLCHAIN: Defines the toolchain (gcc , icc). It is the same value as CONFIG_RTE_TOOLCHAIN but without the double-quotes around the string.
- RTE_EXEC_ENV: Defines the executive environment (linuxapp). It is the same value as CONFIG_RTE_EXEC_ENV but without the double-quotes around the string.

- RTE_KERNELDIR: This variable contains the absolute path to the kernel sources that will be used to compile the kernel modules. The kernel headers must be the same as the ones that will be used on the target machine (the machine that will run the application). By default, the variable is set to /lib/modules/$(shell uname -r)/build, which is correct when the target machine is also the build machine.

Variables that Can be Set/Overridden in a Makefile Only

- VPATH: The path list that the build system will search for sources. By default, RTE_SRCDIR will be included in VPATH.
- CFLAGS: Flags to use for C compilation. The user should use += to append data in this variable.
- LDFLAGS: Flags to use for linking. The user should use += to append data in this variable.
- ASFLAGS: Flags to use for assembly. The user should use += to append data in this variable.
- CPPFLAGS: Flags to use to give flags to C preprocessor (only useful when assembling .S files). The user should use += to append data in this variable.
- LDLIBS: In an application, the list of libraries to link with (for example, -L /path/to/libfoo -lfoo). The user should use += to append data in this variable.
- SRC-y: A list of source files (.c, .S, or .o if the source is a binary) in case of application, library or object Makefiles. The sources must be available from VPATH.
- INSTALL-y-$(INSTPATH): A list of files to be installed in $(INSTPATH). The files must be available from VPATH and will be copied in $(RTE_OUTPUT)/$(INSTPATH). Can be used in almost any RTE Makefile.
- SYMLINK-y-$(INSTPATH): A list of files to be installed in $(INSTPATH). The files must be available from VPATH and will be linked (symbolically) in $(RTE_OUTPUT)/$(INSTPATH). This variable can be used in almost any DPDK Makefile.
- PREBUILD: A list of prerequisite actions to be taken before building. The user should use += to append data in this variable.
- POSTBUILD: A list of actions to be taken after the main build. The user should use += to append data in this variable.
- PREINSTALL: A list of prerequisite actions to be taken before installing. The user should use += to append data in this variable.
- POSTINSTALL: A list of actions to be taken after installing. The user should use += to append data in this variable.
- PRECLEAN: A list of prerequisite actions to be taken before cleaning. The user should use += to append data in this variable.
- POSTCLEAN: A list of actions to be taken after cleaning. The user should use += to append data in this variable.
- DEPDIR-y: Only used in the development kit framework to specify if the build of the current directory depends on build of another one. This is needed to support parallel builds correctly.

Variables that can be Set/Overridden by the User on the Command Line Only

Some variables can be used to configure the build system behavior. They are documented in *Development Kit Root Makefile Help* and *External Application/Library Makefile Help*

- WERROR_CFLAGS: By default, this is set to a specific value that depends on the compiler. Users are encouraged to use this variable as follows:

  CFLAGS += $(WERROR_CFLAGS)

This avoids the use of different cases depending on the compiler (icc or gcc). Also, this variable can be overridden from the command line, which allows bypassing of the flags for testing purposes.

Variables that Can be Set/Overridden by the User in a Makefile or Command Line

- CFLAGS_my_file.o: Specific flags to add for C compilation of my_file.c.
- LDFLAGS_my_app: Specific flags to add when linking my_app.
- EXTRA_CFLAGS: The content of this variable is appended after CFLAGS when compiling.
- EXTRA_LDFLAGS: The content of this variable is appended after LDFLAGS when linking.
- EXTRA_LDLIBS: The content of this variable is appended after LDLIBS when linking.
- EXTRA_ASFLAGS: The content of this variable is appended after ASFLAGS when assembling.
- EXTRA_CPPFLAGS: The content of this variable is appended after CPPFLAGS when using a C preprocessor on assembly files.

4.28 Development Kit Root Makefile Help

The DPDK provides a root level Makefile with targets for configuration, building, cleaning, testing, installation and others. These targets are explained in the following sections.

4.28.1 Configuration Targets

The configuration target requires the name of the target, which is specified using T=mytarget and it is mandatory. The list of available targets are in $(RTE_SDK)/config (remove the defconfig _ prefix).

Configuration targets also support the specification of the name of the output directory, using O=mybuilddir. This is an optional parameter, the default output directory is build.

- Config

  This will create a build directory, and generates a configuration from a template. A Makefile is also created in the new build directory.

  Example:

  ```
  make config O=mybuild T=x86_64-native-linuxapp-gcc
  ```

4.28.2 Build Targets

Build targets support the optional specification of the name of the output directory, using O=mybuilddir. The default output directory is build.

- all, build or just make

  Build the DPDK in the output directory previously created by a make config.

  Example:

  ```
  make O=mybuild
  ```

- clean

  Clean all objects created using make build.

  Example:

  ```
  make clean O=mybuild
  ```

- %_sub

  Build a subdirectory only, without managing dependencies on other directories.

  Example:

  ```
  make lib/librte_eal_sub O=mybuild
  ```

- %_clean

  Clean a subdirectory only.

  Example:

  ```
  make lib/librte_eal_clean O=mybuild
  ```

4.28.3 Install Targets

- Install

  The list of available targets are in $(RTE_SDK)/config (remove the defconfig_ prefix).

  The GNU standards variables may be used: http://gnu.org/prep/standards/html_node/Directory-Variables.html and http://gnu.org/prep/standards/html_node/DESTDIR.html

  Example:

  ```
  make install DESTDIR=myinstall prefix=/usr
  ```

4.28.4 Test Targets

- test

  Launch automatic tests for a build directory specified using O=mybuilddir. It is optional, the default output directory is build.

  Example:

  ```
  make test O=mybuild
  ```

4.28.5 Documentation Targets

- doc

  Generate the documentation (API and guides).

- doc-api-html

  Generate the Doxygen API documentation in html.

- doc-guides-html

  Generate the guides documentation in html.

- doc-guides-pdf

  Generate the guides documentation in pdf.

4.28.6 Deps Targets

- depdirs

  This target is implicitly called by make config. Typically, there is no need for a user to call it, except if DEPDIRS-y variables have been updated in Makefiles. It will generate the file $(RTE_OUTPUT)/.depdirs.

  Example:

  ```
  make depdirs O=mybuild
  ```

- depgraph

  This command generates a dot graph of dependencies. It can be displayed to debug circular dependency issues, or just to understand the dependencies.

  Example:

  ```
  make depgraph O=mybuild > /tmp/graph.dot && dotty /tmp/ graph.dot
  ```

4.28.7 Misc Targets

- help

  Show a quick help.

4.28.8 Other Useful Command-line Variables

The following variables can be specified on the command line:

- V=

  Enable verbose build (show full compilation command line, and some intermediate commands).

- D=

  Enable dependency debugging. This provides some useful information about why a target is built or not.

- EXTRA_CFLAGS=, EXTRA_LDFLAGS=, EXTRA_LDLIBS=, EXTRA_ASFLAGS=, EXTRA_CPPFLAGS=

  Append specific compilation, link or asm flags.

- CROSS=

  Specify a cross toolchain header that will prefix all gcc/binutils applications. This only works when using gcc.

4.28.9 Make in a Build Directory

All targets described above are called from the SDK root $(RTE_SDK). It is possible to run the same Makefile targets inside the build directory. For instance, the following command:

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-native-linuxapp-gcc
make O=mybuild
```

is equivalent to:

```
cd $(RTE_SDK)
make config O=mybuild T=x86_64-native-linuxapp-gcc
cd mybuild

# no need to specify O= now
make
```

4.28.10 Compiling for Debug

To compile the DPDK and sample applications with debugging information included and the optimization level set to 0, the EXTRA_CFLAGS environment variable should be set before compiling as follows:

```
export EXTRA_CFLAGS='-O0 -g'
```

4.29 Extending the DPDK

This chapter describes how a developer can extend the DPDK to provide a new library, a new target, or support a new target.

4.29.1 Example: Adding a New Library libfoo

To add a new library to the DPDK, proceed as follows:

1. Add a new configuration option:

   ```
   for f in config/\*; do \
       echo CONFIG_RTE_LIBFOO=y >> $f; done
   ```

1. Create a new directory with sources:

   ```
   mkdir ${RTE_SDK}/lib/libfoo
   touch ${RTE_SDK}/lib/libfoo/foo.c
   touch ${RTE_SDK}/lib/libfoo/foo.h
   ```

1. Add a foo() function in libfoo.

   Definition is in foo.c:

   ```
   void foo(void)
   {
   }
   ```

   Declaration is in foo.h:

   ```
   extern void foo(void);
   ```

2. Update lib/Makefile:

   ```
   vi ${RTE_SDK}/lib/Makefile
   # add:
   # DIRS-$(CONFIG_RTE_LIBFOO) += libfoo
   ```

3. Create a new Makefile for this library, for example, derived from mempool Makefile:

   ```
   cp ${RTE_SDK}/lib/librte_mempool/Makefile ${RTE_SDK}/lib/libfoo/

   vi ${RTE_SDK}/lib/libfoo/Makefile
   # replace:
   # librte_mempool -> libfoo
   # rte_mempool -> foo
   ```

4. Update mk/DPDK.app.mk, and add -lfoo in LDLIBS variable when the option is enabled. This will automatically add this flag when linking a DPDK application.

5. Build the DPDK with the new library (we only show a specific target here):

   ```
   cd ${RTE_SDK}
   make config T=x86_64-native-linuxapp-gcc
   make
   ```

6. Check that the library is installed:

   ```
   ls build/lib
   ls build/include
   ```

Example: Using libfoo in the Test Application

The test application is used to validate all functionality of the DPDK. Once you have added a library, a new test case should be added in the test application.

- A new test_foo.c file should be added, that includes foo.h and calls the foo() function from test_foo(). When the test passes, the test_foo() function should return 0.
- Makefile, test.h and commands.c must be updated also, to handle the new test case.
- Test report generation: autotest.py is a script that is used to generate the test report that is available in the ${RTE_SDK}/doc/rst/test_report/autotests directory. This script must be updated also. If libfoo is in a new test family, the links in ${RTE_SDK}/doc/rst/test_report/test_report.rst must be updated.
- Build the DPDK with the updated test application (we only show a specific target here):

  ```
  cd ${RTE_SDK}
  make config T=x86_64-native-linuxapp-gcc
  make
  ```

4.30 Building Your Own Application

4.30.1 Compiling a Sample Application in the Development Kit Directory

When compiling a sample application (for example, hello world), the following variables must be exported: RTE_SDK and RTE_TARGET.

```
~/DPDK$ cd examples/helloworld/
~/DPDK/examples/helloworld$ export RTE_SDK=/home/user/DPDK
~/DPDK/examples/helloworld$ export RTE_TARGET=x86_64-native-linuxapp-gcc
~/DPDK/examples/helloworld$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

The binary is generated in the build directory by default:

```
~/DPDK/examples/helloworld$ ls build/app
helloworld helloworld.map
```

4.30.2 Build Your Own Application Outside the Development Kit

The sample application (Hello World) can be duplicated in a new directory as a starting point for your development:

```
~$ cp -r DPDK/examples/helloworld my_rte_app
~$ cd my_rte_app/
~/my_rte_app$ export RTE_SDK=/home/user/DPDK
~/my_rte_app$ export RTE_TARGET=x86_64-native-linuxapp-gcc
~/my_rte_app$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

4.30.3 Customizing Makefiles

Application Makefile

The default makefile provided with the Hello World sample application is a good starting point. It includes:

- $(RTE_SDK)/mk/rte.vars.mk at the beginning
- $(RTE_SDK)/mk/rte.extapp.mk at the end

The user must define several variables:

- APP: Contains the name of the application.
- SRCS-y: List of source files (*.c, *.S).

Library Makefile

It is also possible to build a library in the same way:

- Include $(RTE_SDK)/mk/rte.vars.mk at the beginning.
- Include $(RTE_SDK)/mk/rte.extlib.mk at the end.

The only difference is that APP should be replaced by LIB, which contains the name of the library. For example, libfoo.a.

Customize Makefile Actions

Some variables can be defined to customize Makefile actions. The most common are listed below. Refer to *Makefile Description* section in *Development Kit Build System*

chapter for details.

- VPATH: The path list where the build system will search for sources. By default, RTE_SRCDIR will be included in VPATH.
- CFLAGS_my_file.o: The specific flags to add for C compilation of my_file.c.
- CFLAGS: The flags to use for C compilation.
- LDFLAGS: The flags to use for linking.
- CPPFLAGS: The flags to use to provide flags to the C preprocessor (only useful when assembling .S files)
- LDLIBS: A list of libraries to link with (for example, -L /path/to/libfoo - lfoo)

4.31 External Application/Library Makefile help

External applications or libraries should include specific Makefiles from RTE_SDK, located in mk directory. These Makefiles are:

- ${RTE_SDK}/mk/rte.extapp.mk: Build an application
- ${RTE_SDK}/mk/rte.extlib.mk: Build a static library
- ${RTE_SDK}/mk/rte.extobj.mk: Build objects (.o)

4.31.1 Prerequisites

The following variables must be defined:

- ${RTE_SDK}: Points to the root directory of the DPDK.
- ${RTE_TARGET}: Reference the target to be used for compilation (for example, x86_64-native-linuxapp-gcc).

4.31.2 Build Targets

Build targets support the specification of the name of the output directory, using O=mybuilddir. This is optional; the default output directory is build.

- all, “nothing” (meaning just make)

  Build the application or the library in the specified output directory.

  Example:

  ```
  make O=mybuild
  ```

- clean

  Clean all objects created using make build.

  Example:

  ```
  make clean O=mybuild
  ```

4.31.3 Help Targets

- help

  Show this help.

4.31.4 Other Useful Command-line Variables

The following variables can be specified at the command line:

- S=

  Specify the directory in which the sources are located. By default, it is the current directory.

- M=

  Specify the Makefile to call once the output directory is created. By default, it uses $(S)/Makefile.

- V=

  Enable verbose build (show full compilation command line and some intermediate commands).

- D=

  Enable dependency debugging. This provides some useful information about why a target must be rebuilt or not.

- EXTRA_CFLAGS=, EXTRA_LDFLAGS=, EXTRA_ASFLAGS=, EXTRA_CPPFLAGS=

  Append specific compilation, link or asm flags.

- CROSS=

  Specify a cross-toolchain header that will prefix all gcc/binutils applications. This only works when using gcc.

4.31.5 Make from Another Directory

It is possible to run the Makefile from another directory, by specifying the output and the source dir. For example:

```
export RTE_SDK=/path/to/DPDK
export RTE_TARGET=x86_64-native-linuxapp-icc
make -f /path/to/my_app/Makefile S=/path/to/my_app O=/path/to/build_dir
```

Part 3: Performance Optimization

4.32 Performance Optimization Guidelines

4.32.1 Introduction

The following sections describe optimizations used in the DPDK and optimizations that should be considered for a new applications.

They also highlight the performance-impacting coding techniques that should, and should not be, used when developing an application using the DPDK.

And finally, they give an introduction to application profiling using a Performance Analyzer from Intel to optimize the software.

4.33 Writing Efficient Code

This chapter provides some tips for developing efficient code using the DPDK. For additional and more general information, please refer to the *Intel® 64 and IA-32 Architectures Optimization Reference Manual* which is a valuable reference to writing efficient code.

4.33.1 Memory

This section describes some key memory considerations when developing applications in the DPDK environment.

Memory Copy: Do not Use libc in the Data Plane

Many libc functions are available in the DPDK, via the Linux* application environment. This can ease the porting of applications and the development of the configuration plane. However, many of these functions are not designed for performance. Functions such as memcpy() or strcpy() should not be used in the data plane. To copy small structures, the preference is for a simpler technique that can be optimized by the compiler. Refer to the *VTune™ Performance Analyzer Essentials* publication from Intel Press for recommendations.

For specific functions that are called often, it is also a good idea to provide a self-made optimized function, which should be declared as static inline.

The DPDK API provides an optimized rte_memcpy() function.

Memory Allocation

Other functions of libc, such as malloc(), provide a flexible way to allocate and free memory. In some cases, using dynamic allocation is necessary, but it is really not advised to use malloc-like functions in the data plane because managing a fragmented heap can be costly and the allocator may not be optimized for parallel allocation.

If you really need dynamic allocation in the data plane, it is better to use a memory pool of fixed-size objects. This API is provided by librte_mempool. This data structure provides several services that increase performance, such as memory alignment of objects, lockless access to objects, NUMA awareness, bulk get/put and per-lcore cache. The rte_malloc () function uses a similar concept to mempools.

Concurrent Access to the Same Memory Area

Read-Write (RW) access operations by several lcores to the same memory area can generate a lot of data cache misses, which are very costly. It is often possible to use per-lcore variables, for example, in the case of statistics. There are at least two solutions for this:

- Use RTE_PER_LCORE variables. Note that in this case, data on lcore X is not available to lcore Y.
- Use a table of structures (one per lcore). In this case, each structure must be cache-aligned.

Read-mostly variables can be shared among lcores without performance losses if there are no RW variables in the same cache line.

NUMA

On a NUMA system, it is preferable to access local memory since remote memory access is slower. In the DPDK, the memzone, ring, rte_malloc and mempool APIs provide a way to create a pool on a specific socket.

Sometimes, it can be a good idea to duplicate data to optimize speed. For read-mostly variables that are often accessed, it should not be a problem to keep them in one socket only, since data will be present in cache.

Distribution Across Memory Channels

Modern memory controllers have several memory channels that can load or store data in parallel. Depending on the memory controller and its configuration, the number of channels and the way the memory is distributed across the channels varies. Each channel has a bandwidth limit, meaning that if all memory access operations are done on the first channel only, there is a potential bottleneck.

By default, the *Mempool Library* spreads the addresses of objects among memory channels.

4.33.2 Communication Between lcores

To provide a message-based communication between lcores, it is advised to use the DPDK ring API, which provides a lockless ring implementation.

The ring supports bulk and burst access, meaning that it is possible to read several elements from the ring with only one costly atomic operation (see Chapter 5 “Ring Library”). Performance is greatly improved when using bulk access operations.

The code algorithm that dequeues messages may be something similar to the following:

```
    #define MAX_BULK 32

    while (1) {
        /* Process as many elements as can be dequeued. */
        count = rte_ring_dequeue_burst(ring, obj_table, MAX_BULK);
        if (unlikely(count == 0))
            continue;

        my_process_bulk(obj_table, count);
   }
```

4.33.3 PMD Driver

The DPDK Poll Mode Driver (PMD) is also able to work in bulk/burst mode, allowing the factorization of some code for each call in the send or receive function.

Avoid partial writes. When PCI devices write to system memory through DMA, it costs less if the write operation is on a full cache line as opposed to part of it. In the PMD code, actions have been taken to avoid partial writes as much as possible.

Lower Packet Latency

Traditionally, there is a trade-off between throughput and latency. An application can be tuned to achieve a high throughput, but the end-to-end latency of an average packet will typically increase as a result. Similarly, the application can be tuned to have, on average, a low end-to-end latency, at the cost of lower throughput.

In order to achieve higher throughput, the DPDK attempts to aggregate the cost of processing each packet individually by processing packets in bursts.

Using the testpmd application as an example, the burst size can be set on the command line to a value of 16 (also the default value). This allows the application to request 16 packets at a time from the PMD. The testpmd application then immediately attempts to transmit all the packets that were received, in this case, all 16 packets.

The packets are not transmitted until the tail pointer is updated on the corresponding TX queue of the network port. This behavior is desirable when tuning for high throughput because the cost of tail pointer updates to both the RX and TX queues can be spread across 16 packets, effectively hiding the relatively slow MMIO cost of writing to the PCIe* device. However, this is not very desirable when tuning for low latency because the first packet that was received must also wait for another 15 packets to be received. It cannot be transmitted until the other 15 packets have also been processed because the NIC will not know to transmit the packets until the TX tail pointer has been updated, which is not done until all 16 packets have been processed for transmission.

To consistently achieve low latency, even under heavy system load, the application developer should avoid processing packets in bunches. The testpmd application can be configured from the command line to use a burst value of 1. This will allow a single packet to be processed at a time, providing lower latency, but with the added cost of lower throughput.

4.33.4 Locks and Atomic Operations

Atomic operations imply a lock prefix before the instruction, causing the processor's LOCK# signal to be asserted during execution of the following instruction. This has a big impact on performance in a multicore environment.

Performance can be improved by avoiding lock mechanisms in the data plane. It can often be replaced by other solutions like per-lcore variables. Also, some locking techniques are more efficient than others. For instance, the Read-Copy-Update (RCU) algorithm can frequently replace simple rwlocks.

4.33.5 Coding Considerations

Inline Functions

Small functions can be declared as static inline in the header file. This avoids the cost of a call instruction (and the associated context saving). However, this technique is not always efficient; it depends on many factors including the compiler.

Branch Prediction

The Intel® C/C++ Compiler (icc)/gcc built-in helper functions likely() and unlikely() allow the developer to indicate if a code branch is likely to be taken or not. For instance:

```
if (likely(x > 1))
    do_stuff();
```

4.33.6 Setting the Target CPU Type

The DPDK supports CPU microarchitecture-specific optimizations by means of CONFIG_RTE_MACHINE option in the DPDK configuration file. The degree of optimization depends on the compiler's ability to optimize for a specific microarchitecture, therefore it is preferable to use the latest compiler versions whenever possible.

If the compiler version does not support the specific feature set (for example, the Intel® AVX instruction set), the build process gracefully degrades to whatever latest feature set is supported by the compiler.

Since the build and runtime targets may not be the same, the resulting binary also contains a platform check that runs before the main() function and checks if the current machine is suitable for running the binary.

Along with compiler optimizations, a set of preprocessor defines are automatically added to the build process (regardless of the compiler version). These defines correspond to the instruction sets that the target CPU should be able to support. For example, a binary compiled for any SSE4.2-capable processor will have RTE_MACHINE_CPUFLAG_SSE4_2 defined, thus enabling compile-time code path selection for different platforms.

4.34 Profile Your Application

Intel processors provide performance counters to monitor events. Some tools provided by Intel can be used to profile and benchmark an application. See the *VTune Performance Analyzer Essentials* publication from Intel Press for more information.

For a DPDK application, this can be done in a Linux* application environment only.

The main situations that should be monitored through event counters are:

- Cache misses
- Branch mis-predicts
- DTLB misses
- Long latency instructions and exceptions

Refer to the Intel Performance Analysis Guide for details about application profiling.

4.35 Glossary

ACL Access Control List

API Application Programming Interface

ASLR Linux* kernel Address-Space Layout Randomization

BSD Berkeley Software Distribution

Clr Clear

CIDR Classless Inter-Domain Routing

Control Plane The control plane is concerned with the routing of packets and with providing a start or end point.

Core A core may include several lcores or threads if the processor supports hyperthreading.

Core Components A set of libraries provided by the DPDK, including eal, ring, mempool, mbuf, timers, and so on.

CPU Central Processing Unit

CRC Cyclic Redundancy Check

ctrlmbuf An *mbuf* carrying control data.

Data Plane In contrast to the control plane, the data plane in a network architecture are the layers involved when forwarding packets. These layers must be highly optimized to achieve good performance.

DIMM Dual In-line Memory Module

Doxygen A documentation generator used in the DPDK to generate the API reference.

DPDK Data Plane Development Kit

DRAM Dynamic Random Access Memory

EAL The Environment Abstraction Layer (EAL) provides a generic interface that hides the environment specifics from the applications and libraries. The services expected from the EAL are: development kit loading and launching, core affinity/ assignment procedures, system memory allocation/description, PCI bus access, inter-partition communication.

FIFO First In First Out

FPGA Field Programmable Gate Array

GbE Gigabit Ethernet

HW Hardware

HPET High Precision Event Timer; a hardware timer that provides a precise time reference on x86 platforms.

ID Identifier

IOCTL Input/Output Control

I/O Input/Output

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

lcore A logical execution unit of the processor, sometimes called a *hardware thread*.

KNI Kernel Network Interface

L1 Layer 1

L2 Layer 2

L3 Layer 3

L4 Layer 4

LAN Local Area Network

LPM Longest Prefix Match

master lcore The execution unit that executes the main() function and that launches other lcores.

mbuf An mbuf is a data structure used internally to carry messages (mainly network packets). The name is derived from BSD stacks. To understand the concepts of packet buffers or mbuf, refer to *TCP/IP Illustrated, Volume 2: The Implementation*.

MESI Modified Exclusive Shared Invalid (CPU cache coherency protocol)

MTU Maximum Transfer Unit

NIC Network Interface Card

OOO Out Of Order (execution of instructions within the CPU pipeline)

NUMA Non-uniform Memory Access

PCI Peripheral Connect Interface

PHY An abbreviation for the physical layer of the OSI model.

pktmbuf An *mbuf* carrying a network packet.

PMD Poll Mode Driver

QoS Quality of Service

RCU Read-Copy-Update algorithm, an alternative to simple rwlocks.

Rd Read

RED Random Early Detection

RSS Receive Side Scaling

RTE Run Time Environment. Provides a fast and simple framework for fast packet processing, in a lightweight environment as a Linux* application and using Poll Mode Drivers (PMDs) to increase speed.

Rx Reception

Slave lcore Any *lcore* that is not the *master lcore*.

Socket A physical CPU, that includes several *cores*.

SLA Service Level Agreement

srTCM Single Rate Three Color Marking

SRTD Scheduler Round Trip Delay

SW Software

Target In the DPDK, the target is a combination of architecture, machine, executive environment and toolchain. For example: i686-native-linuxapp-gcc.

TCP Transmission Control Protocol

TC Traffic Class

TLB Translation Lookaside Buffer

TLS Thread Local Storage

trTCM Two Rate Three Color Marking

TSC Time Stamp Counter

Tx Transmission

TUN/TAP TUN and TAP are virtual network kernel devices.

VLAN Virtual Local Area Network

Wr Write

WRED Weighted Random Early Detection

WRR Weighted Round Robin

Figures

CHAPTER 5

Network Interface Controller Drivers

5.1 BNX2X Poll Mode Driver

The BNX2X poll mode driver library (**librte_pmd_bnx2x**) implements support for **QLogic 578xx** 10/20 Gbps family of adapters as well as their virtual functions (VF) in SR-IOV context. It is supported on several standard Linux distros like Red Hat 7.x and SLES12 OS. It is compile-tested under FreeBSD OS.

More information can be found at QLogic Corporation's Official Website.

5.1.1 Supported Features

BNX2X PMD has support for:

- Base L2 features
- Unicast/multicast filtering
- Promiscuous mode
- Port hardware statistics
- SR-IOV VF

5.1.2 Non-supported Features

The features not yet supported include:

- TSS (Transmit Side Scaling)
- RSS (Receive Side Scaling)
- LRO/TSO offload
- Checksum offload
- SR-IOV PF

5.1.3 Co-existence considerations

- BCM578xx being a CNA can have both NIC and Storage personalities. However, coexistence with storage protocol drivers (cnic, bnx2fc and bnx2fi) is not supported on the same adapter. So storage personality has to be disabled on that adapter when used in DPDK applications.

- For SR-IOV case, bnx2x PMD will be used to bind to SR-IOV VF device and Linux native kernel driver (bnx2x) will be attached to SR-IOV PF.

5.1.4 Supported QLogic NICs

- 578xx

5.1.5 Prerequisites

- Requires firmware version **7.2.51.0**. It is included in most of the standard Linux distros. If it is not available visit QLogic Driver Download Center to get the required firmware.

5.1.6 Pre-Installation Configuration

Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_BNX2X_PMD` (default **y**)

  Toggle compilation of bnx2x driver.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG` (default **n**)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_INIT` (default **n**)

  Toggle display of initialization related messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_TX` (default **n**)

  Toggle display of transmit fast path run-time messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_RX` (default **n**)

  Toggle display of receive fast path run-time messages.

- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_PERIODIC` (default **n**)

  Toggle display of register reads and writes.

Driver Compilation

BNX2X PMD for Linux x86_64 gcc target, run the following “make” command:

```
cd <DPDK-source-directory>
make config T=x86_64-native-linuxapp-gcc install
```

To compile BNX2X PMD for Linux x86_64 clang target, run the following “make” command:

```
cd <DPDK-source-directory>
make config T=x86_64-native-linuxapp-clang install
```

To compile BNX2X PMD for Linux i686 gcc target, run the following "make" command:

```
cd <DPDK-source-directory>
make config T=i686-native-linuxapp-gcc install
```

To compile BNX2X PMD for Linux i686 gcc target, run the following "make" command:

```
cd <DPDK-source-directory>
make config T=i686-native-linuxapp-gcc install
```

To compile BNX2X PMD for FreeBSD x86_64 clang target, run the following "gmake" command:

```
cd <DPDK-source-directory>
gmake config T=x86_64-native-bsdapp-clang install
```

To compile BNX2X PMD for FreeBSD x86_64 gcc target, run the following "gmake" command:

```
cd <DPDK-source-directory>
gmake config T=x86_64-native-bsdapp-gcc install -Wl,-rpath=/usr/local/lib/gcc48 CC=gcc48
```

To compile BNX2X PMD for FreeBSD x86_64 gcc target, run the following "gmake" command:

```
cd <DPDK-source-directory>
gmake config T=x86_64-native-bsdapp-gcc install -Wl,-rpath=/usr/local/lib/gcc48 CC=gcc48
```

5.1.7 Linux

Linux Installation

Sample Application Notes

This section demonstrates how to launch `testpmd` with QLogic 578xx devices managed by `librte_pmd_bnx2x` in Linux operating system.

1. Request huge pages:

   ```
   echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages/nr_hugepages
   ```

2. Load `igb_uio` or `vfio-pci` driver:

   ```
   insmod ./x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
   ```

   or

   ```
   modprobe vfio-pci
   ```

3. Bind the QLogic adapters to `igb_uio` or `vfio-pci` loaded in the previous step:

   ```
   ./tools/dpdk_nic_bind.py --bind igb_uio 0000:84:00.0 0000:84:00.1
   ```

   or

   Setup VFIO permissions for regular users and then bind to `vfio-pci`:

   ```
   sudo chmod a+x /dev/vfio
   ```

   ```
   sudo chmod 0666 /dev/vfio/*
   ```

   ```
   ./tools/dpdk_nic_bind.py --bind vfio-pci 0000:84:00.0 0000:84:00.1
   ```

4. Start `testpmd` with basic parameters:

   ```
   ./x86_64-native-linuxapp-gcc/app/testpmd -c 0xf -n 4 -- -i
   ```

   Example output:

```
[...]
EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL:   probe driver: 14e4:168e rte_bnx2x_pmd
EAL:   PCI memory mapped at 0x7f14f6fe5000
EAL:   PCI memory mapped at 0x7f14f67e5000
EAL:   PCI memory mapped at 0x7f15fbd9b000
EAL: PCI device 0000:84:00.1 on NUMA socket 1
EAL:   probe driver: 14e4:168e rte_bnx2x_pmd
EAL:   PCI memory mapped at 0x7f14f5fe5000
EAL:   PCI memory mapped at 0x7f14f57e5000
EAL:   PCI memory mapped at 0x7f15fbd4f000
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: bnx2x_dev_tx_queue_setup(): fp[00] req_bd=512, thresh=512,
             usable_bd=1020, total_bd=1024,
                          tx_pages=4
PMD: bnx2x_dev_rx_queue_setup(): fp[00] req_bd=128, thresh=0,
             usable_bd=510, total_bd=512,
                          rx_pages=1, cq_pages=8
PMD: bnx2x_print_adapter_info():
[...]
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

SR-IOV: Prerequisites and sample Application Notes

This section provides instructions to configure SR-IOV with Linux OS.

1. Verify SR-IOV and ARI capabilities are enabled on the adapter using `lspci`:

   ```
   lspci -s <slot> -vvv
   ```

   Example output:

   ```
   [...]
   Capabilities: [1b8 v1] Alternative Routing-ID Interpretation (ARI)
   [...]
   Capabilities: [1c0 v1] Single Root I/O Virtualization (SR-IOV)
   [...]
   Kernel driver in use: igb_uio
   ```

2. Load the kernel module:

   ```
   modprobe bnx2x
   ```

   Example output:

   ```
   systemd-udevd[4848]: renamed network interface eth0 to ens5f0
   systemd-udevd[4848]: renamed network interface eth1 to ens5f1
   ```

3. Bring up the PF ports:

   ```
   ifconfig ens5f0 up
   ifconfig ens5f1 up
   ```

4. Create VF device(s):

   Echo the number of VFs to be created into “sriov_numvfs” sysfs entry of the parent PF.

   Example output:

   ```
   echo 2 > /sys/devices/pci0000:00/0000:00:03.0/0000:81:00.0/sriov_numvfs
   ```

5. Assign VF MAC address:

   Assign MAC address to the VF using iproute2 utility. The syntax is: ip link set <PF iface> vf <VF id> mac <macaddr>

   Example output:

   ```
   ip link set ens5f0 vf 0 mac 52:54:00:2f:9d:e8
   ```

6. PCI Passthrough:

   The VF devices may be passed through to the guest VM using virt-manager or virsh etc. bnx2x PMD should be used to bind the VF devices in the guest VM using the instructions outlined in the Application notes below.

5.2 CXGBE Poll Mode Driver

The CXGBE PMD (**librte_pmd_cxgbe**) provides poll mode driver support for **Chelsio T5** 10/40 Gbps family of adapters. CXGBE PMD has support for the latest Linux and FreeBSD operating systems.

More information can be found at Chelsio Communications Official Website.

5.2.1 Features

CXGBE PMD has support for:

- Multiple queues for TX and RX
- Receiver Side Steering (RSS)
- VLAN filtering
- Checksum offload
- Promiscuous mode
- All multicast mode
- Port hardware statistics
- Jumbo frames

5.2.2 Limitations

The Chelsio T5 devices provide two/four ports but expose a single PCI bus address, thus, librte_pmd_cxgbe registers itself as a PCI driver that allocates one Ethernet device per detected port.

For this reason, one cannot whitelist/blacklist a single port without whitelisting/blacklisting the other ports on the same device.

5.2.3 Supported Chelsio T5 NICs

- 1G NICs: T502-BT
- 10G NICs: T520-BT, T520-CR, T520-LL-CR, T520-SO-CR, T540-CR
- 40G NICs: T580-CR, T580-LP-CR, T580-SO-CR
- Other T5 NICs: T522-CR

5.2.4 Prerequisites

- Requires firmware version **1.13.32.0** and higher. Visit Chelsio Download Center to get latest firmware bundled with the latest Chelsio Unified Wire package.

  For Linux, installing and loading the latest cxgb4 kernel driver from the Chelsio Unified Wire package should get you the latest firmware. More information can be obtained from the User Guide that is bundled with the Chelsio Unified Wire package.

  For FreeBSD, the latest firmware obtained from the Chelsio Unified Wire package must be manually flashed via cxgbetool available in FreeBSD source repository.

  Instructions on how to manually flash the firmware are given in section *Linux Installation* for Linux and section *FreeBSD Installation* for FreeBSD.

5.2.5 Pre-Installation Configuration

Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_CXGBE_PMD` (default **y**)

  Toggle compilation of librte_pmd_cxgbe driver.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG` (default **n**)

  Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG_REG` (default **n**)

  Toggle display of registers related run-time check messages.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG_MBOX` (default **n**)

  Toggle display of firmware mailbox related run-time check messages.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG_TX` (default **n**)

  Toggle display of transmission data path run-time check messages.

- `CONFIG_RTE_LIBRTE_CXGBE_DEBUG_RX` (default **n**)

  Toggle display of receiving data path run-time check messages.

Driver Compilation

To compile CXGBE PMD for Linux x86_64 gcc target, run the following "make" command:

```
cd <DPDK-source-directory>
make config T=x86_64-native-linuxapp-gcc install
```

To compile CXGBE PMD for FreeBSD x86_64 clang target, run the following "gmake" command:

```
cd <DPDK-source-directory>
gmake config T=x86_64-native-bsdapp-clang install
```

5.2.6 Linux

Linux Installation

Steps to manually install the latest firmware from the downloaded Chelsio Unified Wire package for Linux operating system are as follows:

1. Load the kernel module:

   ```
   modprobe cxgb4
   ```

2. Use ifconfig to get the interface name assigned to Chelsio card:

   ```
   ifconfig -a | grep "00:07:43"
   ```

   Example output:

   ```
   p1p1      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C0
   p1p2      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C8
   ```

3. Install cxgbtool:

   ```
   cd <path_to_uwire>/tools/cxgbtool
   make install
   ```

4. Use cxgbtool to load the firmware config file onto the card:

   ```
   cxgbtool p1p1 loadcfg <path_to_uwire>/src/network/firmware/t5-config.txt
   ```

5. Use cxgbtool to load the firmware image onto the card:

   ```
   cxgbtool p1p1 loadfw <path_to_uwire>/src/network/firmware/t5fw-*.bin
   ```

6. Unload and reload the kernel module:

   ```
   modprobe -r cxgb4
   modprobe cxgb4
   ```

7. Verify with ethtool:

   ```
   ethtool -i p1p1 | grep "firmware"
   ```

   Example output:

   ```
   firmware-version: 1.13.32.0, TP 0.1.4.8
   ```

Running testpmd

This section demonstrates how to launch **testpmd** with Chelsio T5 devices managed by librte_pmd_cxgbe in Linux operating system.

1. Change to DPDK source directory where the target has been compiled in section *Driver Compilation*:

   ```
   cd <DPDK-source-directory>
   ```

2. Load the kernel module:

   ```
   modprobe cxgb4
   ```

3. Get the PCI bus addresses of the interfaces bound to cxgb4 driver:

   ```
   dmesg | tail -2
   ```

   Example output:

   ```
   cxgb4 0000:02:00.4 p1p1: renamed from eth0
   cxgb4 0000:02:00.4 p1p2: renamed from eth1
   ```

   Note: Both the interfaces of a Chelsio T5 2-port adapter are bound to the same PCI bus address.

4. Unload the kernel module:

   ```
   modprobe -ar cxgb4 csiostor
   ```

5. Request huge pages:

   ```
   echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages/nr_hugepages
   ```

6. Mount huge pages:

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

7. Load igb_uio or vfio-pci driver:

   ```
   insmod ./x86_64-native-linuxapp-gcc/kmod/igb_uio.ko
   ```

   or

   ```
   modprobe vfio-pci
   ```

8. Bind the Chelsio T5 adapters to igb_uio or vfio-pci loaded in the previous step:

   ```
   ./tools/dpdk_nic_bind.py --bind igb_uio 0000:02:00.4
   ```

   or

   Setup VFIO permissions for regular users and then bind to vfio-pci:

   ```
   sudo chmod a+x /dev/vfio

   sudo chmod 0666 /dev/vfio/*

   ./tools/dpdk_nic_bind.py --bind vfio-pci 0000:02:00.4
   ```

   Note: Currently, CXGBE PMD only supports the binding of PF4 for Chelsio T5 NICs.

9. Start testpmd with basic parameters:

   ```
   ./x86_64-native-linuxapp-gcc/app/testpmd -c 0xf -n 4 -w 0000:02:00.4 -- -i
   ```

   Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket -1
EAL:   probe driver: 1425:5401 rte_cxgbe_pmd
EAL:   PCI memory mapped at 0x7fd7c0200000
EAL:   PCI memory mapped at 0x7fd77cdfd000
EAL:   PCI memory mapped at 0x7fd7c10b7000
PMD: rte_cxgbe_pmd: fw: 1.13.32.0, TP: 0.1.4.8
PMD: rte_cxgbe_pmd: Coming up as MASTER: Initializing adapter
Interactive-mode selected
Configuring Port 0 (socket 0)
Port 0: 00:07:43:2D:EA:C0
Configuring Port 1 (socket 0)
Port 1: 00:07:43:2D:EA:C8
Checking link statuses...
PMD: rte_cxgbe_pmd: Port0: passive DA port module inserted
PMD: rte_cxgbe_pmd: Port1: passive DA port module inserted
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

Note: Flow control pause TX/RX is disabled by default and can be enabled via testpmd. Refer section *Enable/Disable Flow Control* for more details.

5.2.7 FreeBSD

FreeBSD Installation

Steps to manually install the latest firmware from the downloaded Chelsio Unified Wire package for FreeBSD operating system are as follows:

1. Load the kernel module:

   ```
   kldload if_cxgbe
   ```

2. Use dmesg to get the t5nex instance assigned to the Chelsio card:

   ```
   dmesg | grep "t5nex"
   ```

   Example output:

   ```
   t5nex0: <Chelsio T520-CR> irq 16 at device 0.4 on pci2
   cxl0: <port 0> on t5nex0
   cxl1: <port 1> on t5nex0
   t5nex0: PCIe x8, 2 ports, 14 MSI-X interrupts, 31 eq, 13 iq
   ```

   In the example above, a Chelsio T520-CR card is bound to a t5nex0 instance.

3. Install cxgbetool from FreeBSD source repository:

   ```
   cd <path_to_FreeBSD_source>/tools/tools/cxgbetool/
   make && make install
   ```

4. Use cxgbetool to load the firmware image onto the card:

   ```
   cxgbetool t5nex0 loadfw <path_to_uwire>/src/network/firmware/t5fw-*.bin
   ```

5. Unload and reload the kernel module:

   ```
   kldunload if_cxgbe
   kldload if_cxgbe
   ```

6. Verify with sysctl:

   ```
   sysctl -a | grep "t5nex" | grep "firmware"
   ```

   Example output:

   ```
   dev.t5nex.0.firmware_version: 1.13.32.0
   ```

Running testpmd

This section demonstrates how to launch **testpmd** with Chelsio T5 devices managed by librte_pmd_cxgbe in FreeBSD operating system.

1. Change to DPDK source directory where the target has been compiled in section *Driver Compilation*:

   ```
   cd <DPDK-source-directory>
   ```

2. Copy the contigmem kernel module to /boot/kernel directory:

   ```
   cp x86_64-native-bsdapp-clang/kmod/contigmem.ko /boot/kernel/
   ```

3. Add the following lines to /boot/loader.conf:

   ```
   # reserve 2 x 1G blocks of contiguous memory using contigmem driver
   hw.contigmem.num_buffers=2
   hw.contigmem.buffer_size=1073741824
   # load contigmem module during boot process
   contigmem_load="YES"
   ```

   The above lines load the contigmem kernel module during boot process and allocate 2 x 1G blocks of contiguous memory to be used for DPDK later on. This is to avoid issues with potential memory fragmentation during later system up time, which may result in failure of allocating the contiguous memory required for the contigmem kernel module.

4. Restart the system and ensure the contigmem module is loaded successfully:

   ```
   reboot
   kldstat | grep "contigmem"
   ```

   Example output:

   ```
   2    1 0xffffffff817f1000 3118     contigmem.ko
   ```

5. Repeat step 1 to ensure that you are in the DPDK source directory.

6. Load the cxgbe kernel module:

   ```
   kldload if_cxgbe
   ```

7. Get the PCI bus addresses of the interfaces bound to t5nex driver:

   ```
   pciconf -l | grep "t5nex"
   ```

   Example output:

   ```
   t5nex0@pci0:2:0:4: class=0x020000 card=0x00001425 chip=0x54011425 rev=0x00
   ```

   In the above example, the t5nex0 is bound to 2:0:4 bus address.

   Note: Both the interfaces of a Chelsio T5 2-port adapter are bound to the same PCI bus address.

8. Unload the kernel module:

   ```
   kldunload if_cxgbe
   ```

9. Set the PCI bus addresses to hw.nic_uio.bdfs kernel environment parameter:

   ```
   kenv hw.nic_uio.bdfs="2:0:4"
   ```

   This automatically binds 2:0:4 to nic_uio kernel driver when it is loaded in the next step.

   Note: Currently, CXGBE PMD only supports the binding of PF4 for Chelsio T5 NICs.

10. Load nic_uio kernel driver:

    ```
    kldload ./x86_64-native-bsdapp-clang/kmod/nic_uio.ko
    ```

11. Start testpmd with basic parameters:

    ```
    ./x86_64-native-bsdapp-clang/app/testpmd -c 0xf -n 4 -w 0000:02:00.4 -- -i
    ```

    Example output:

    ```
    [...]
    EAL: PCI device 0000:02:00.4 on NUMA socket 0
    EAL:   probe driver: 1425:5401 rte_cxgbe_pmd
    EAL:   PCI memory mapped at 0x8007ec000
    EAL:   PCI memory mapped at 0x842800000
    EAL:   PCI memory mapped at 0x80086c000
    PMD: rte_cxgbe_pmd: fw: 1.13.32.0, TP: 0.1.4.8
    PMD: rte_cxgbe_pmd: Coming up as MASTER: Initializing adapter
    Interactive-mode selected
    Configuring Port 0 (socket 0)
    Port 0: 00:07:43:2D:EA:C0
    Configuring Port 1 (socket 0)
    Port 1: 00:07:43:2D:EA:C8
    Checking link statuses...
    PMD: rte_cxgbe_pmd: Port0: passive DA port module inserted
    PMD: rte_cxgbe_pmd: Port1: passive DA port module inserted
    Port 0 Link Up - speed 10000 Mbps - full-duplex
    Port 1 Link Up - speed 10000 Mbps - full-duplex
    Done
    testpmd>
    ```

Note: Flow control pause TX/RX is disabled by default and can be enabled via testpmd. Refer section *Enable/Disable Flow Control* for more details.

5.2.8 Sample Application Notes

Enable/Disable Flow Control

Flow control pause TX/RX is disabled by default and can be enabled via testpmd as follows:

```
testpmd> set flow_ctrl rx on tx on 0 0 0 0 mac_ctrl_frame_fwd off autoneg on 0
testpmd> set flow_ctrl rx on tx on 0 0 0 0 mac_ctrl_frame_fwd off autoneg on 1
```

To disable again, run:

```
testpmd> set flow_ctrl rx off tx off 0 0 0 0 mac_ctrl_frame_fwd off autoneg off 0
testpmd> set flow_ctrl rx off tx off 0 0 0 0 mac_ctrl_frame_fwd off autoneg off 1
```

Jumbo Mode

There are two ways to enable sending and receiving of jumbo frames via testpmd. One method involves using the **mtu** command, which changes the mtu of an individual port without having to stop the selected port. Another method involves stopping all the ports first and then running **max-pkt-len** command to configure the mtu of all the ports with a single command.

- To configure each port individually, run the mtu command as follows:

  ```
  testpmd> port config mtu 0 9000
  testpmd> port config mtu 1 9000
  ```

- To configure all the ports at once, stop all the ports first and run the max-pkt-len command as follows:

  ```
  testpmd> port stop all
  testpmd> port config all max-pkt-len 9000
  ```

5.3 Driver for VM Emulated Devices

The DPDK EM poll mode driver supports the following emulated devices:

- qemu-kvm emulated Intel® 82540EM Gigabit Ethernet Controller (qemu e1000 device)
- VMware* emulated Intel® 82545EM Gigabit Ethernet Controller
- VMware emulated Intel® 8274L Gigabit Ethernet Controller.

5.3.1 Validated Hypervisors

The validated hypervisors are:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0
- KVM (Kernel Virtual Machine) with Qemu, version 0.15.1
- VMware ESXi 5.0, Update 1

5.3.2 Recommended Guest Operating System in Virtual Machine

The recommended guest operating system in a virtualized environment is:

- Fedora* 18 (64-bit)

For supported kernel versions, refer to the *DPDK Release Notes*.

5.3.3 Setting Up a KVM Virtual Machine

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with Qemu version, 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the DPDK Getting Started Guide

- Target Applications: testpmd

The setup procedure is as follows:

1. Download qemu-kvm-0.14.0 from http://sourceforge.net/projects/kvm/files/qemu-kvm/ and install it in the Host OS using the following steps:

   When using a recent kernel (2.6.25+) with kvm modules included:

   ```
   tar xzf qemu-kvm-release.tar.gz cd qemu-kvm-release
   ./configure --prefix=/usr/local/kvm
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   When using an older kernel or a kernel from a distribution without the kvm modules, you must download (from the same link), compile and install the modules yourself:

   ```
   tar xjf kvm-kmod-release.tar.bz2
   cd kvm-kmod-release
   ./configure
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   Note that qemu-kvm installs in the /usr/local/bin directory.

   For more details about KVM configuration and usage, please refer to: http://www.linux-kvm.org/page/HOWTO1.

2. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).

3. Start the Virtual Machine with at least one emulated e1000 device.

   Note: The Qemu provides several choices for the emulated network device backend. Most commonly used is a TAP networking backend that uses a TAP networking device in the host. For more information about Qemu supported networking backends and different options for configuring networking at Qemu, please refer to:

   — http://www.linux-kvm.org/page/Networking

   — http://wiki.qemu.org/Documentation/Networking

   — http://qemu.weilnetz.de/qemu-doc.html

   For example, to start a VM with two emulated e1000 devices, issue the following command:

   ```
   /usr/local/kvm/bin/qemu-system-x86_64 -cpu host -smp 4 -hda qemu1.raw -m 1024
   -net nic,model=e1000,vlan=1,macaddr=DE:AD:1E:00:00:01
   -net tap,vlan=1,ifname=tapvm01,script=no,downscript=no
   -net nic,model=e1000,vlan=2,macaddr=DE:AD:1E:00:00:02
   -net tap,vlan=2,ifname=tapvm02,script=no,downscript=no
   ```

   where:

   — -m = memory to assign

   — -smp = number of smp cores

   — -hda = virtual disk image

This command starts a new virtual machine with two emulated 82540EM devices, backed up with two TAP networking host interfaces, tapvm01 and tapvm02.

```
# ip tuntap show
tapvm01: tap
tapvm02: tap
```

4. Configure your TAP networking interfaces using ip/ifconfig tools.
5. Log in to the guest OS and check that the expected emulated devices exist:

```
# lspci -d 8086:100e
00:04.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03
00:05.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03
```

6. Install the DPDK and run testpmd.

5.3.4 Known Limitations of Emulated Devices

The following are known limitations:

1. The Qemu e1000 RX path does not support multiple descriptors/buffers per packet. Therefore, rte_mbuf should be big enough to hold the whole packet. For example, to allow testpmd to receive jumbo frames, use the following:

   testpmd [options] – –mbuf-size=<your-max-packet-size>

2. Qemu e1000 does not validate the checksum of incoming packets.
3. Qemu e1000 only supports one interrupt source, so link and Rx interrupt should be exclusive.
4. Qemu e1000 does not support interrupt auto-clear, application should disable interrupt immediately when woken up.

5.4 ENIC Poll Mode Driver

ENIC PMD is the DPDK poll-mode driver for the Cisco System Inc. VIC Ethernet NICs. These adapters are also referred to as vNICs below. If you are running or would like to run DPDK software applications on Cisco UCS servers using Cisco VIC adapters the following documentation is relevant.

5.4.1 Version Information

The version of the ENIC PMD driver is 1.0.0.6 and will be printed by ENIC PMD during the initialization.

5.4.2 How to obtain ENIC PMD integrated DPDK

ENIC PMD support is integrated into the DPDK suite. dpdk-<version>.tar.gz should be downloaded from http://dpdk.org

5.4.3 Configuration information

- **DPDK Configuration Parameters**

  The following configuration options are available for the ENIC PMD:

  - **CONFIG_RTE_LIBRTE_ENIC_PMD** (default y): Enables or disables inclusion of the ENIC PMD driver in the DPDK compilation.
  - **CONFIG_RTE_LIBRTE_ENIC_DEBUG** (default n): Enables or disables debug logging within the ENIC PMD driver.

- **vNIC Configuration Parameters**

  - **Number of Queues**

    The maximum number of receive and transmit queues are configurable on a per vNIC basis through the Cisco UCS Manager (CIMC or UCSM). These values should be configured to be greater than or equal to the nb_rx_q and nb_tx_q parameters expected to used in the call to the rte_eth_dev_configure() function.

  - **Size of Queues**

    Likewise, the number of receive and transmit descriptors are configurable on a per vNIC bases via the UCS Manager and should be greater than or equal to the nb_rx_desc and nb_tx_desc parameters expected to be used in the calls to rte_eth_rx_queue_setup() and rte_eth_tx_queue_setup() respectively.

  - **Interrupts**

    Only one interrupt per vNIC interface should be configured in the UCS manager regardless of the number receive/transmit queues. The ENIC PMD uses this interrupt to get information about errors in the fast path.

5.4.4 Limitations

- **VLAN 0 Priority Tagging**

  If a vNIC is configured in TRUNK mode by the UCS manager, the adapter will priority tag egress packets according to 802.1Q if they were not already VLAN tagged by software. If the adapter is connected to a properly configured switch, there will be no unexpected behavior.

  In test setups where an Ethernet port of a Cisco adapter in TRUNK mode is connected point-to-point to another adapter port or connected though a router instead of a switch, all ingress packets will be VLAN tagged. Programs such as l3fwd which do not account for VLAN tags in packets will misbehave. The solution is to enable VLAN stripping on ingress. The follow code fragment is example of how to accomplish this:

```
vlan_offload = rte_eth_dev_get_vlan_offload(port);
vlan_offload |= ETH_VLAN_STRIP_OFFLOAD;
rte_eth_dev_set_vlan_offload(port, vlan_offload);
```

5.4.5 How to build the suite?

The build instructions for the DPDK suite should be followed. By default the ENIC PMD library will be built into the DPDK library.

For configuring and using UIO and VFIO frameworks, please refer the documentation that comes with DPDK suite.

5.4.6 Supported Cisco VIC adapters

ENIC PMD supports all recent generations of Cisco VIC adapters including:

- VIC 1280
- VIC 1240
- VIC 1225
- VIC 1285
- VIC 1225T
- VIC 1227
- VIC 1227T
- VIC 1380
- VIC 1340
- VIC 1385
- VIC 1387
- **Flow director features are not supported on generation 1 Cisco VIC adapters**
  (M81KR and P81E)

5.4.7 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in Dependencies section of DPDK documentation.

5.4.8 Supported features

- Unicast, multicast and broadcast transmission and reception
- Receive queue polling
- Port Hardware Statistics
- Hardware VLAN acceleration
- IP checksum offload
- Receive side VLAN stripping
- Multiple receive and transmit queues
- Flow Director ADD, UPDATE, DELETE, STATS operation support for IPV4 5-TUPLE flows
- Promiscuous mode
- Setting RX VLAN (supported via UCSM/CIMC only)
- VLAN filtering (supported via UCSM/CIMC only)

- Execution of application by unprivileged system users
- IPV4, IPV6 and TCP RSS hashing

5.4.9 Known bugs and Unsupported features in this release

- Signature or flex byte based flow direction
- Drop feature of flow direction
- VLAN based flow direction
- non-IPV4 flow direction
- Setting of extended VLAN
- UDP RSS hashing

5.4.10 Prerequisites

- Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration
- Insert vfio-pci kernel module using the command 'modprobe vfio-pci' if the user wants to use VFIO framework
- Insert uio kernel module using the command 'modprobe uio' if the user wants to use UIO framework
- DPDK suite should be configured based on the user's decision to use VFIO or UIO framework
- If the vNIC device(s) to be used is bound to the kernel mode Ethernet driver (enic), use 'ifconfig' to bring the interface down. The dpdk_nic_bind.py tool can then be used to unbind the device's bus id from the enic kernel mode driver.
- Bind the intended vNIC to vfio-pci in case the user wants ENIC PMD to use VFIO framework using dpdk_nic_bind.py.
- Bind the intended vNIC to igb_uio in case the user wants ENIC PMD to use UIO framework using dpdk_nic_bind.py.

At this point the system should be ready to run DPDK applications. Once the application runs to completion, the vNIC can be detached from vfio-pci or igb_uio if necessary.

Root privilege is required to bind and unbind vNICs to/from VFIO/UIO. VFIO framework helps an unprivileged user to run the applications. For an unprivileged user to run the applications on DPDK and ENIC PMD, it may be necessary to increase the maximum locked memory of the user. The following command could be used to do this.

```
sudo sh -c "ulimit -l <value in Kilo Bytes>"
```

The value depends on the memory configuration of the application, DPDK and PMD. Typically, the limit has to be raised to higher than 2GB. e.g., 2621440

The compilation of any unused drivers can be disabled using the configuration file in config/ directory (e.g., config/common_linuxapp). This would help in bringing down the time taken for building the libraries and the initialization time of the application.

5.4.11 Additional Reference

- http://www.cisco.com/c/en/us/products/servers-unified-computing

5.4.12 Contact Information

Any questions or bugs should be reported to DPDK community and to the ENIC PMD maintainers:

- John Daley <johndale@cisco.com>
- Sujith Sankar <ssujith@cisco.com>

5.5 FM10K Poll Mode Driver

The FM10K poll mode driver library provides support for the Intel FM10000 (FM10K) family of 40GbE/100GbE adapters.

5.5.1 Limitations

Switch manager

The Intel FM10000 family of NICs integrate a hardware switch and multiple host interfaces. The FM10000 PMD driver only manages host interfaces. For the switch component another switch driver has to be loaded prior to to the FM10000 PMD driver. The switch driver can be acquired for Intel support or from the Match Interface project. Only Testpoint is validated with DPDK, the latest version that has been validated with DPDK2.2 is 4.1.6.

CRC striping

The FM10000 family of NICs strip the CRC for every packets coming into the host interface. So, CRC will be stripped even when the `rxmode.hw_strip_crc` member is set to 0 in `struct rte_eth_conf`.

Maximum packet length

The FM10000 family of NICS support a maximum of a 15K jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 15364, frames up to 15364 bytes can still reach the host interface.

5.6 IXGBE Driver

5.6.1 Vector PMD for IXGBE

Vector PMD uses Intel® SIMD instructions to optimize packet I/O. It improves load/store bandwidth efficiency of L1 data cache by using a wider SSE/AVX register 1 (1). The wider register gives space to hold multiple packet buffers so as to save instruction number when processing bulk of packets.

There is no change to PMD API. The RX/TX handler are the only two entries for vPMD packet I/O. They are transparently registered at runtime RX/TX execution if all condition checks pass.

1. To date, only an SSE version of IX GBE vPMD is available. To ensure that vPMD is in the binary code, ensure that the option CONFIG_RTE_IXGBE_INC_VECTOR=y is in the configure file.

Some constraints apply as pre-conditions for specific optimizations on bulk packet transfers. The following sections explain RX and TX constraints in the vPMD.

RX Constraints

Prerequisites and Pre-conditions

The following prerequisites apply:

- To enable vPMD to work for RX, bulk allocation for Rx must be allowed.

Ensure that the following pre-conditions are satisfied:

- rxq->rx_free_thresh >= RTE_PMD_IXGBE_RX_MAX_BURST
- rxq->rx_free_thresh < rxq->nb_rx_desc
- (rxq->nb_rx_desc % rxq->rx_free_thresh) == 0
- rxq->nb_rx_desc < (IXGBE_MAX_RING_DESC - RTE_PMD_IXGBE_RX_MAX_BURST)

These conditions are checked in the code.

Scattered packets are not supported in this mode. If an incoming packet is greater than the maximum acceptable length of one “mbuf” data size (by default, the size is 2 KB), vPMD for RX would be disabled.

By default, IXGBE_MAX_RING_DESC is set to 4096 and RTE_PMD_IXGBE_RX_MAX_BURST is set to 32.

Feature not Supported by RX Vector PMD

Some features are not supported when trying to increase the throughput in vPMD. They are:

- IEEE1588
- FDIR
- Header split
- RX checksum off load

Other features are supported using optional MACRO configuration. They include:

- HW VLAN strip
- HW extend dual VLAN
- Enabled by RX_OLFLAGS (RTE_IXGBE_RX_OLFLAGS_ENABLE=y)

To guarantee the constraint, configuration flags in dev_conf.rxmode will be checked:

- hw_vlan_strip
- hw_vlan_extend
- hw_ip_checksum
- header_split
- dev_conf

fdir_conf->mode will also be checked.

RX Burst Size

As vPMD is focused on high throughput, it assumes that the RX burst size is equal to or greater than 32 per burst. It returns zero if using nb_pkt < 32 as the expected packet number in the receive handler.

TX Constraint

Prerequisite

The only prerequisite is related to tx_rs_thresh. The tx_rs_thresh value must be greater than or equal to RTE_PMD_IXGBE_TX_MAX_BURST, but less or equal to RTE_IXGBE_TX_MAX_FREE_BUF_SZ. Consequently, by default the tx_rs_thresh value is in the range 32 to 64.

Feature not Supported by RX Vector PMD

TX vPMD only works when txq_flags is set to IXGBE_SIMPLE_FLAGS.

This means that it does not support TX multi-segment, VLAN offload and TX csum offload. The following MACROs are used for these three features:

- ETH_TXQ_FLAGS_NOMULTSEGS
- ETH_TXQ_FLAGS_NOVLANOFFL
- ETH_TXQ_FLAGS_NOXSUMSCTP
- ETH_TXQ_FLAGS_NOXSUMUDP
- ETH_TXQ_FLAGS_NOXSUMTCP

Sample Application Notes

testpmd

By default, using CONFIG_RTE_IXGBE_RX_OLFLAGS_ENABLE=y:

```
./x86_64-native-linuxapp-gcc/app/testpmd -c 300 -n 4 -- -i --burst=32 --rxfreet=32 --mbcache=25
```

When CONFIG_RTE_IXGBE_RX_OLFLAGS_ENABLE=n, better performance can be achieved:

```
./x86_64-native-linuxapp-gcc/app/testpmd -c 300 -n 4 -- -i --burst=32 --rxfree=32 --mbcache=25
```

l3fwd

When running l3fwd with vPMD, there is one thing to note. In the configuration, ensure that port_conf.rxmode.hw_ip_checksum=0. Otherwise, by default, RX vPMD is disabled.

load_balancer

As in the case of l3fwd, set configure port_conf.rxmode.hw_ip_checksum=0 to enable vPMD. In addition, for improved performance, use -bsz "(32,32),(64,64),(32,32)" in load_balancer to avoid using the default burst size of 144.

5.7 I40E/IXGBE/IGB Virtual Function Driver

Supported Intel® Ethernet Controllers (see the *DPDK Release Notes* for details) support the following modes of operation in a virtualized environment:

- **SR-IOV mode**: Involves direct assignment of part of the port resources to different guest operating systems using the PCI-SIG Single Root I/O Virtualization (SR IOV) standard, also known as "native mode" or "pass-through" mode. In this chapter, this mode is referred to as IOV mode.
- **VMDq mode**: Involves central management of the networking resources by an IO Virtual Machine (IOVM) or a Virtual Machine Monitor (VMM), also known as software switch acceleration mode. In this chapter, this mode is referred to as the Next Generation VMDq mode.

5.7.1 SR-IOV Mode Utilization in a DPDK Environment

The DPDK uses the SR-IOV feature for hardware-based I/O sharing in IOV mode. Therefore, it is possible to partition SR-IOV capability on Ethernet controller NIC resources logically and expose them to a virtual machine as a separate PCI function called a "Virtual Function". Refer to Fig. 5.1.

Therefore, a NIC is logically distributed among multiple virtual machines (as shown in Fig. 5.1), while still having global data in common to share with the Physical Function and other Virtual Functions. The DPDK fm10kvf, i40evf, igbvf or ixgbevf as a Poll Mode Driver (PMD) serves for the Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, Intel® 82599 10 Gigabit Ethernet Controller NIC, Intel® Fortville 10/40 Gigabit Ethernet Controller NIC's virtual PCI function, or PCIe host-interface of the Intel Ethernet Switch FM10000 Series. Meanwhile the DPDK Poll Mode Driver (PMD) also supports "Physical Function" of such NIC's on the host.

The DPDK PF/VF Poll Mode Driver (PMD) supports the Layer 2 switch on Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, Intel® 82599 10 Gigabit Ethernet

Controller, and Intel® Fortville 10/40 Gigabit Ethernet Controller NICs so that guest can choose it for inter virtual machine traffic in SR-IOV mode.

For more detail on SR-IOV, please refer to the following documents:

- SR-IOV provides hardware based I/O sharing
- PCI-SIG-Single Root I/O Virtualization Support on IA
- Scalable I/O Virtualized Servers

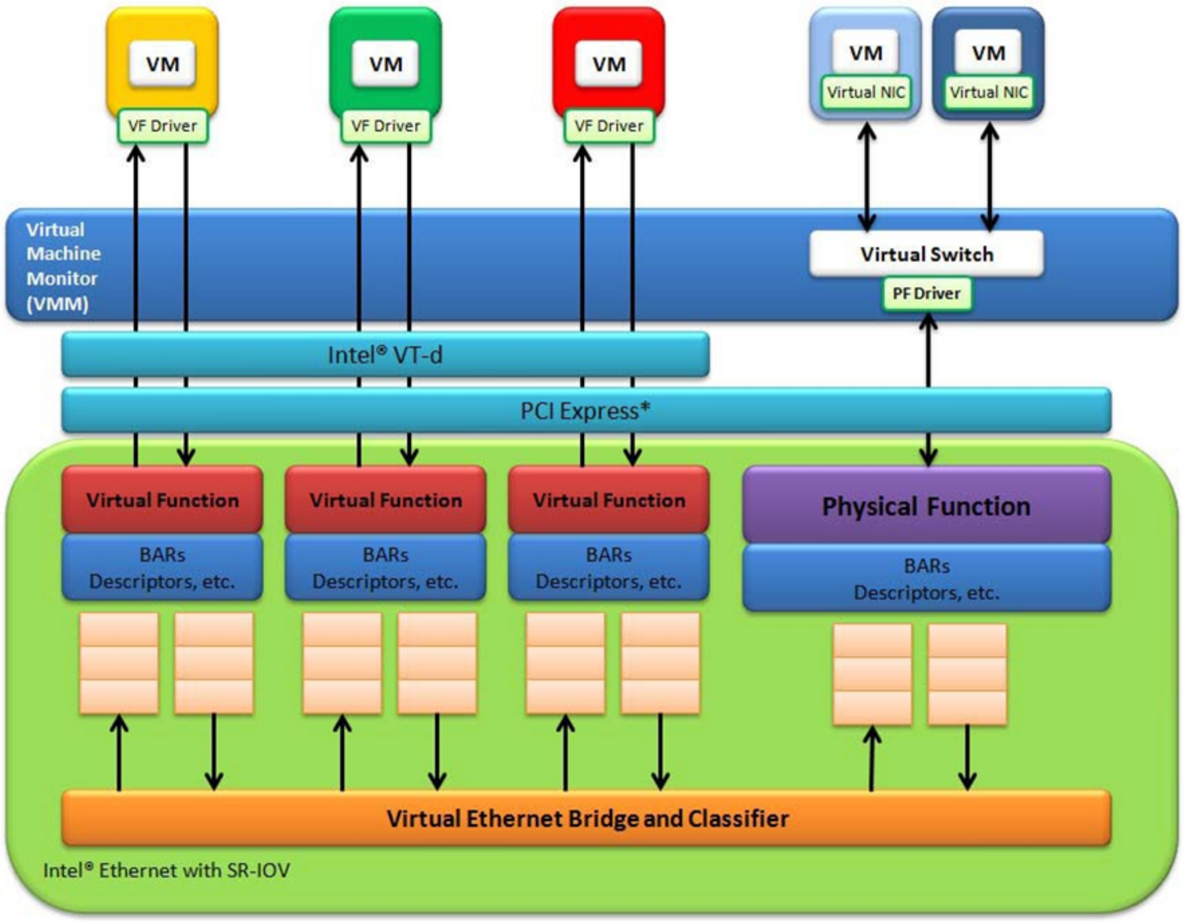


Fig. 5.1: Virtualization for a Single Port NIC in SR-IOV Mode

Physical and Virtual Function Infrastructure

The following describes the Physical Function and Virtual Functions infrastructure for the supported Ethernet Controller NICs.

Virtual Functions operate under the respective Physical Function on the same NIC Port and therefore have no access to the global NIC resources that are shared between other functions for the same NIC port.

A Virtual Function has basic access to the queue resources and control structures of the queues assigned to it. For global resource access, a Virtual Function has to send a request to the Physical Function for that port, and the Physical Function operates on the global resources on behalf of the Virtual Function. For this out-of-band communication, an SR-IOV enabled NIC provides a memory buffer for each Virtual Function, which is called a “Mailbox”.

The PCIE host-interface of Intel Ethernet Switch FM10000 Series VF infrastructure

In a virtualized environment, the programmer can enable a maximum of *64 Virtual Functions (VF)* globally per PCIE host-interface of the Intel Ethernet Switch FM10000 Series device. Each VF can have a maximum of 16 queue pairs. The Physical Function in host could be only configured by the Linux* fm10k driver (in the case of the Linux Kernel-based Virtual Machine [KVM]), DPDK PMD PF driver doesn't support it yet.

For example,

- Using Linux* fm10k driver:

  ```
  rmmod fm10k (To remove the fm10k module)
  insmod fm0k.ko max_vfs=2,2 (To enable two Virtual Functions per port)
  ```

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® Fortville 10/40 Gigabit Ethernet Controller VF Infrastructure

In a virtualized environment, the programmer can enable a maximum of *128 Virtual Functions (VF)* globally per Intel® Fortville 10/40 Gigabit Ethernet Controller NIC device. Each VF can have a maximum of 16 queue pairs. The Physical Function in host could be either configured by the Linux* i40e driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* i40e driver:

  ```
  rmmod i40e (To remove the i40e module)
  insmod i40e.ko max_vfs=2,2 (To enable two Virtual Functions per port)
  ```

- Using the DPDK PMD PF i40e driver:

  Kernel Params: iommu=pt, intel_iommu=on

  ```
  modprobe uio
  insmod igb_uio
  ./dpdk_nic_bind.py -b igb_uio bb:ss.f
  echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific PCI
  ```

  Launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

- Using the DPDK PMD PF ixgbe driver to enable VF RSS:

Same steps as above to install the modules of uio, igb_uio, specify max_vfs for PCI device, and launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

The available queue number(at most 4) per VF depends on the total number of pool, which is determined by the max number of VF at PF initialization stage and the number of queue specified in config:

– If the max number of VF is set in the range of 1 to 32:

  If the number of rxq is specified as 4(e.g. '–rxq 4' in testpmd), then there are totally 32 pools(ETH_32_POOLS), and each VF could have 4 or less(e.g. 2) queues;

  If the number of rxq is specified as 2(e.g. '–rxq 2' in testpmd), then there are totally 32 pools(ETH_32_POOLS), and each VF could have 2 queues;

– If the max number of VF is in the range of 33 to 64:

  If the number of rxq is 4 ('–rxq 4' in testpmd), then error message is expected as rxq is not correct at this case;

  If the number of rxq is 2 ('–rxq 2' in testpmd), then there is totally 64 pools(ETH_64_POOLS), and each VF have 2 queues;

On host, to enable VF RSS functionality, rx mq mode should be set as ETH_MQ_RX_VMDQ_RSS or ETH_MQ_RX_RSS mode, and SRIOV mode should be activated(max_vfs >= 1). It also needs config VF RSS information like hash function, RSS key, RSS key length.

```
testpmd -c 0xffff -n 4 -- --coremask=<core-mask> --rxq=4 --txq=4 -i
```

The limitation for VF RSS on Intel® 82599 10 Gigabit Ethernet Controller is: The hash and key are shared among PF and all VF, the RETA table with 128 entries is also shared among PF and all VF; So it could not to provide a method to query the hash and reta content per VF on guest, while, if possible, please query them on host(PF) for the shared RETA information.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® 82599 10 Gigabit Ethernet Controller VF Infrastructure

The programmer can enable a maximum of *63 Virtual Functions* and there must be *one Physical Function* per Intel® 82599 10 Gigabit Ethernet Controller NIC port. The reason for this is that the device allows for a maximum of 128 queues per port and a virtual/physical function has to have at least one queue pair (RX/TX). The current implementation of the DPDK ixgbevf driver supports a single queue pair (RX/TX) per Virtual Function. The Physical Function in host could be either configured by the Linux* ixgbe driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* ixgbe driver:

  ```
  rmmod ixgbe (To remove the ixgbe module)
  insmod ixgbe max_vfs=2,2 (To enable two Virtual Functions per port)
  ```

- Using the DPDK PMD PF ixgbe driver:

  Kernel Params: iommu=pt, intel_iommu=on

  ```
  modprobe uio
  insmod igb_uio
  ./dpdk_nic_bind.py -b igb_uio bb:ss.f
  echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific PCI
  ```

  Launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® 82576 Gigabit Ethernet Controller and Intel® Ethernet Controller I350 Family VF Infrastructure

In a virtualized environment, an Intel® 82576 Gigabit Ethernet Controller serves up to eight virtual machines (VMs). The controller has 16 TX and 16 RX queues. They are generally referred to (or thought of) as queue pairs (one TX and one RX queue). This gives the controller 16 queue pairs.

A pool is a group of queue pairs for assignment to the same VF, used for transmit and receive operations. The controller has eight pools, with each pool containing two queue pairs, that is, two TX and two RX queues assigned to each VF.

In a virtualized environment, an Intel® Ethernet Controller I350 family device serves up to eight virtual machines (VMs) per port. The eight queues can be accessed by eight different VMs if configured correctly (the i350 has 4x1GbE ports each with 8T X and 8 RX queues), that means, one Transmit and one Receive queue assigned to each VF.

For example,

- Using Linux* igb driver:

  ```
  rmmod igb (To remove the igb module)
  insmod igb max_vfs=2,2 (To enable two Virtual Functions per port)
  ```

- Using Intel® DPDK PMD PF igb driver:

  Kernel Params: iommu=pt, intel_iommu=on modprobe uio

  ```
  insmod igb_uio
  ./dpdk_nic_bind.py -b igb_uio bb:ss.f
  echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific pci
  ```

  Launch DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a four-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence, starting from 0 to 7. However:

- Virtual Functions 0 and 4 belong to Physical Function 0
- Virtual Functions 1 and 5 belong to Physical Function 1
- Virtual Functions 2 and 6 belong to Physical Function 2
- Virtual Functions 3 and 7 belong to Physical Function 3

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Validated Hypervisors

The validated hypervisor is:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0

However, the hypervisor is bypassed to configure the Virtual Function devices using the Mailbox interface, the solution is hypervisor-agnostic. Xen* and VMware* (when SR- IOV is supported) will also be able to support the DPDK with Virtual Function driver support.

Expected Guest Operating System in Virtual Machine

The expected guest operating systems in a virtualized environment are:

- Fedora* 14 (64-bit)
- Ubuntu* 10.04 (64-bit)

For supported kernel versions, refer to the *DPDK Release Notes*.

5.7.2 Setting Up a KVM Virtual Machine Monitor

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with Qemu version 0.14.0
- Guest Operating System: Fedora 14

- Linux Kernel Version: Refer to the *DPDK Getting Started Guide*
- Target Applications: l2fwd, l3fwd-vf

The setup procedure is as follows:

1. Before booting the Host OS, open **BIOS setup** and enable **Intel® VT features**.
2. While booting the Host OS kernel, pass the intel_iommu=on kernel command line argument using GRUB. When using DPDK PF driver on host, pass the iommu=pt kernel command line argument in GRUB.
3. Download qemu-kvm-0.14.0 from http://sourceforge.net/projects/kvm/files/qemu-kvm/ and install it in the Host OS using the following steps:

   When using a recent kernel (2.6.25+) with kvm modules included:

   ```
   tar xzf qemu-kvm-release.tar.gz
   cd qemu-kvm-release
   ./configure --prefix=/usr/local/kvm
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   When using an older kernel, or a kernel from a distribution without the kvm modules, you must download (from the same link), compile and install the modules yourself:

   ```
   tar xjf kvm-kmod-release.tar.bz2
   cd kvm-kmod-release
   ./configure
   make
   sudo make install
   sudo /sbin/modprobe kvm-intel
   ```

   qemu-kvm installs in the /usr/local/bin directory.

   For more details about KVM configuration and usage, please refer to:

   http://www.linux-kvm.org/page/HOWTO1.
4. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).
5. Download and install the latest ixgbe driver from:

   http://downloadcenter.intel.com/Detail_Desc.aspx?agr=Y&amp;DwnldID=14687
6. In the Host OS

   When using Linux kernel ixgbe driver, unload the Linux ixgbe driver and reload it with the max_vfs=2,2 argument:

   ```
   rmmod ixgbe
   modprobe ixgbe max_vfs=2,2
   ```

   When using DPDK PMD PF driver, insert DPDK kernel module igb_uio and set the number of VF by sysfs max_vfs:

   ```
   modprobe uio
   insmod igb_uio
   ./dpdk_nic_bind.py -b igb_uio 02:00.0 02:00.1 0e:00.0 0e:00.1
   echo 2 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:02\:00.1/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:0e\:00.0/max_vfs
   echo 2 > /sys/bus/pci/devices/0000\:0e\:00.1/max_vfs
   ```

Note: You need to explicitly specify number of vfs for each port, for example, in the command above, it creates two vfs for the first two ixgbe ports.

Let say we have a machine with four physical ixgbe ports:

0000:02:00.0

0000:02:00.1

0000:0e:00.0

0000:0e:00.1

The command above creates two vfs for device 0000:02:00.0:

```
ls -alrt /sys/bus/pci/devices/0000\:02\:00.0/virt*
lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn1 -> ../000
lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn0 -> ../000
```

It also creates two vfs for device 0000:02:00.1:

```
ls -alrt /sys/bus/pci/devices/0000\:02\:00.1/virt*
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn1 -> ../000
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn0 -> ../000
```

7. List the PCI devices connected and notice that the Host OS shows two Physical Functions (traditional ports) and four Virtual Functions (two for each port). This is the result of the previous step.

8. Insert the pci_stub module to hold the PCI devices that are freed from the default driver using the following command (see http://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM Section 4 for more information):

```
sudo /sbin/modprobe pci-stub
```

Unbind the default driver from the PCI devices representing the Virtual Functions. A script to perform this action is as follows:

```
echo "8086 10ed" > /sys/bus/pci/drivers/pci-stub/new_id
echo 0000:08:10.0 > /sys/bus/pci/devices/0000:08:10.0/driver/unbind
echo 0000:08:10.0 > /sys/bus/pci/drivers/pci-stub/bind
```

where, 0000:08:10.0 belongs to the Virtual Function visible in the Host OS.

9. Now, start the Virtual Machine by running the following command:

```
/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -smp 4 -boot c -hda lucid.qcow2 -device pci-
```

where:

— -m = memory to assign

—-smp = number of smp cores

— -boot = boot option

—-hda = virtual disk image

— -device = device to attach

> **Note:** — The pci-assign,host=08:10.0 alue indicates that you want to attach a PCI device to a Virtual Machine and the respective (Bus:Device.Function) numbers should be passed for the Virtual Function to be attached.
>
> — qemu-kvm-0.14.0 allows a maximum of four PCI devices assigned to a VM, but this is qemu-kvm version dependent since qemu-kvm-0.14.1 allows a maximum of five PCI devices.
>
> — qemu-system-x86_64 also has a -cpu command line option that is used to select the cpu_model to emulate in a Virtual Machine. Therefore, it can be used as:
>
> ```
> /usr/local/kvm/bin/qemu-system-x86_64 -cpu ?
>
> (to list all available cpu_models)
>
> /usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -cpu host -smp 4 -boot c -hda lucid.qcow2 -c
>
> (to use the same cpu_model equivalent to the host cpu)
> ```
>
> For more information, please refer to: http://wiki.qemu.org/Features/CPUModels.

10. Install and run DPDK host app to take over the Physical Function. Eg.

    ```
    make install T=x86_64-native-linuxapp-gcc
    ./x86_64-native-linuxapp-gcc/app/testpmd -c f -n 4 -- -i
    ```

11. Finally, access the Guest OS using vncviewer with the localhost:5900 port and check the lspci command output in the Guest OS. The virtual functions will be listed as available for use.

12. Configure and install the DPDK with an x86_64-native-linuxapp-gcc configuration on the Guest OS as normal, that is, there is no change to the normal installation procedure.

    ```
    make config T=x86_64-native-linuxapp-gcc O=x86_64-native-linuxapp-gcc
    cd x86_64-native-linuxapp-gcc
    make
    ```

Note: If you are unable to compile the DPDK and you are getting “error: CPU you selected does not support x86-64 instruction set”, power off the Guest OS and start the virtual machine with the correct -cpu option in the qemu- system-x86_64 command as shown in step 9. You must select the best x86_64 cpu_model to emulate or you can select host option if available.

Note: Run the DPDK l2fwd sample application in the Guest OS with Hugepages enabled. For the expected benchmark performance, you must pin the cores from the Guest OS to the Host OS (taskset can be used to do this) and you must also look at the PCI Bus layout on the board to ensure you are not running the traffic over the QPI Interface.

Note:

- The Virtual Machine Manager (the Fedora package name is virt-manager) is a utility for virtual machine management that can also be used to create, start, stop and delete virtual machines. If this option is used, step 2 and 6 in the instructions provided will be different.

- virsh, a command line utility for virtual machine management, can also be used to bind and unbind devices to a virtual machine in Ubuntu. If this option is used, step 6 in the instructions provided will be different.
- The Virtual Machine Monitor (see Fig. 5.2) is equivalent to a Host OS with KVM installed as described in the instructions.

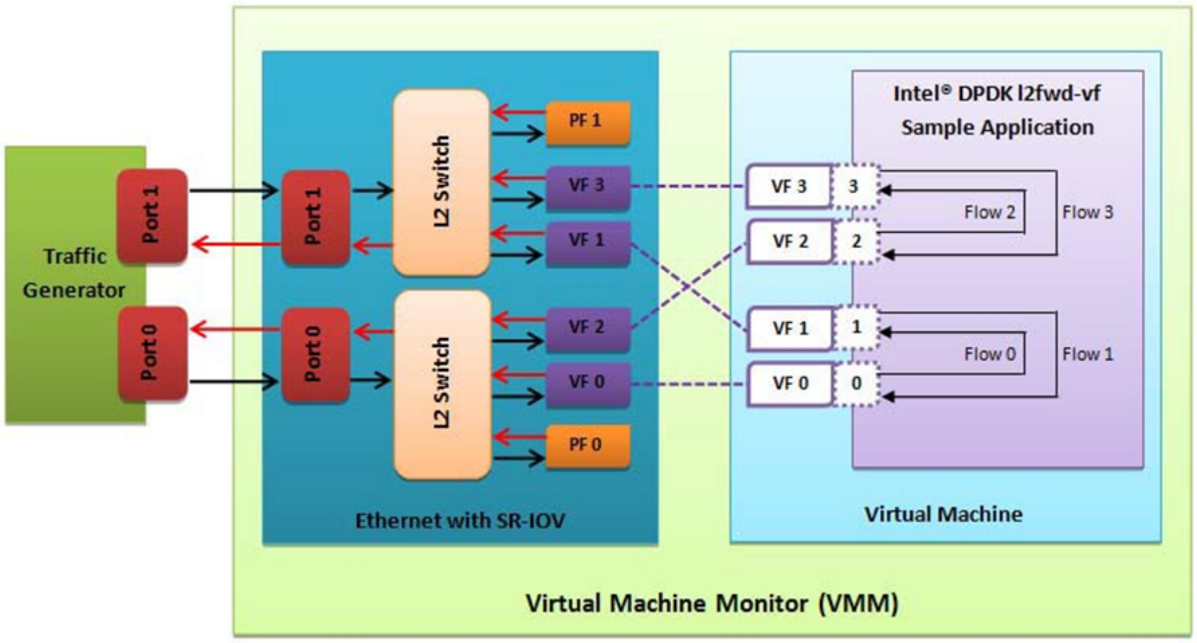


Fig. 5.2: Performance Benchmark Setup

5.7.3 DPDK SR-IOV PMD PF/VF Driver Usage Model

Fast Host-based Packet Processing

Software Defined Network (SDN) trends are demanding fast host-based packet handling. In a virtualization environment, the DPDK VF PMD driver performs the same throughput result as a non-VT native environment.

With such host instance fast packet processing, lots of services such as filtering, QoS, DPI can be offloaded on the host fast path.

Fig. 5.3 shows the scenario where some VMs directly communicate externally via a VFs, while others connect to a virtual switch and share the same uplink bandwidth.

5.7.4 SR-IOV (PF/VF) Approach for Inter-VM Communication

Inter-VM data communication is one of the traffic bottle necks in virtualization platforms. SR-IOV device assignment helps a VM to attach the real device, taking advantage of the bridge in the NIC. So VF-to-VF traffic within the same physical port (VM0<->VM1) have hardware acceleration. However, when VF crosses physical ports (VM0<->VM2), there is no such hardware bridge. In this case, the DPDK PMD PF driver provides host forwarding between such VMs.

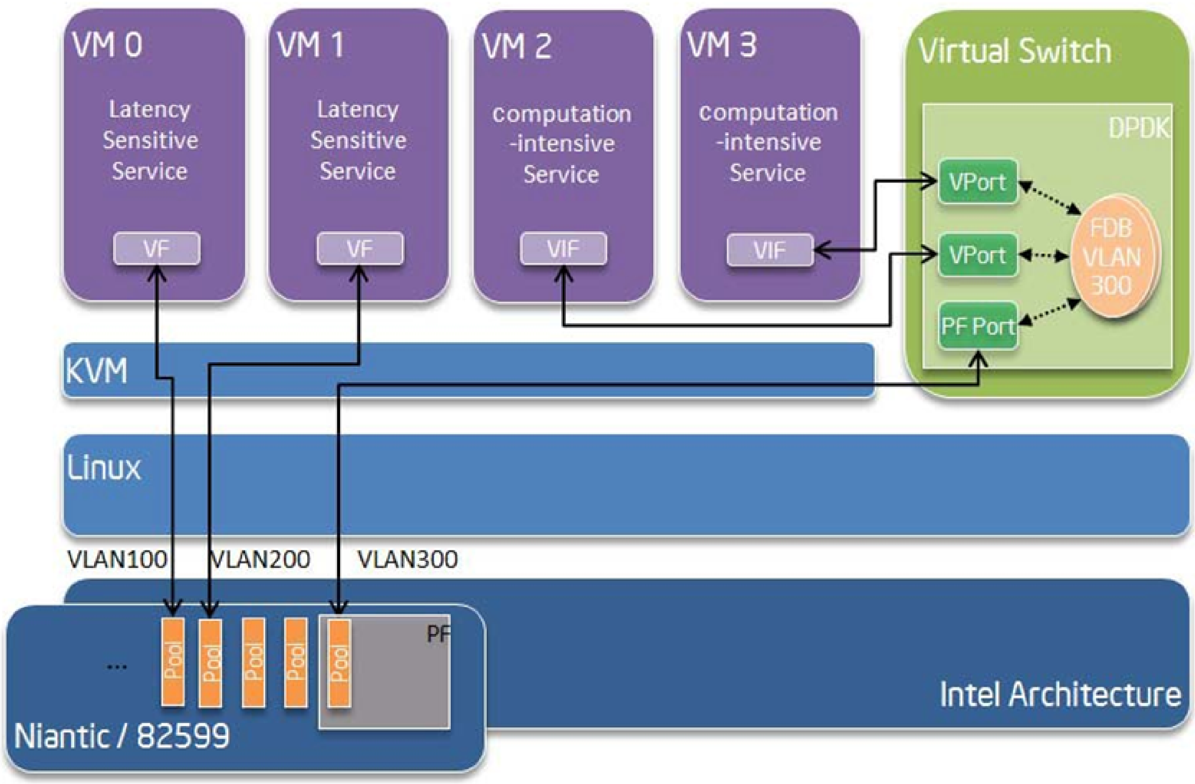


Fig. 5.3: Fast Host-based Packet Processing

Fig. 5.4 shows an example. In this case an update of the MAC address lookup tables in both the NIC and host DPDK application is required.

In the NIC, writing the destination of a MAC address belongs to another cross device VM to the PF specific pool. So when a packet comes in, its destination MAC address will match and forward to the host DPDK PMD application.

In the host DPDK application, the behavior is similar to L2 forwarding, that is, the packet is forwarded to the correct PF pool. The SR-IOV NIC switch forwards the packet to a specific VM according to the MAC destination address which belongs to the destination VF on the VM.

5.8 MLX4 poll mode driver library

The MLX4 poll mode driver library (**librte_pmd_mlx4**) implements support for **Mellanox ConnectX-3** and **Mellanox ConnectX-3 Pro** 10/40 Gbps adapters as well as their virtual functions (VF) in SR-IOV context.

Information and documentation about this family of adapters can be found on the Mellanox website. Help is also provided by the Mellanox community.

There is also a section dedicated to this poll mode driver.

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting `CONFIG_RTE_LIBRTE_MLX4_PMD=y` and recompiling DPDK.

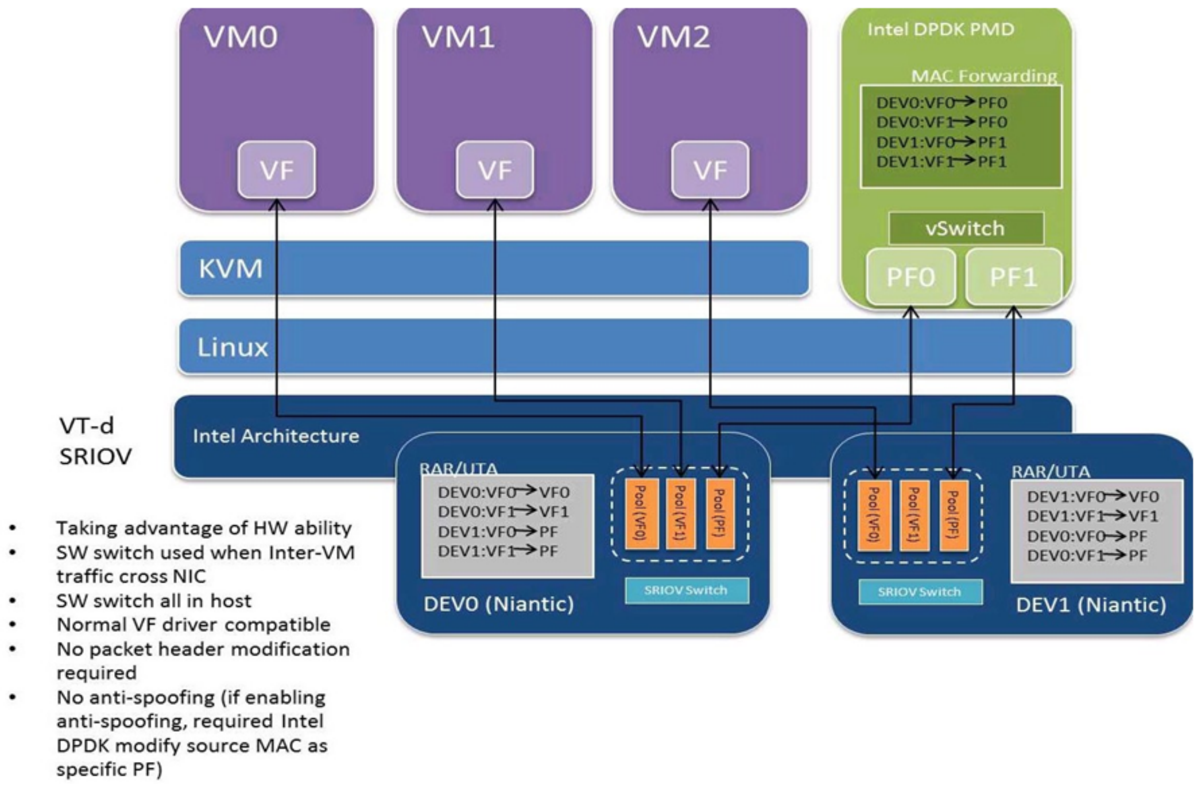


Fig. 5.4: Inter-VM Communication

Warning: `CONFIG_RTE_BUILD_COMBINE_LIBS` with `CONFIG_RTE_BUILD_SHARED_LIB` is not supported and thus the compilation will fail with this configuration.

5.8.1 Implementation details

Most Mellanox ConnectX-3 devices provide two ports but expose a single PCI bus address, thus unlike most drivers, librte_pmd_mlx4 registers itself as a PCI driver that allocates one Ethernet device per detected port.

For this reason, one cannot white/blacklist a single port without also white/blacklisting the others on the same device.

Besides its dependency on libibverbs (that implies libmlx4 and associated kernel support), librte_pmd_mlx4 relies heavily on system calls for control operations such as querying/updating the MTU and flow control parameters.

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel combined with hardware specifications that allow it to handle virtual memory addresses directly ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

This capability allows the PMD to coexist with kernel network interfaces which remain functional, although they stop receiving unicast packets as long as they share the same MAC address.

Compiling librte_pmd_mlx4 causes DPDK to be linked against libibverbs.

5.8.2 Features

- RSS, also known as RCA, is supported. In this mode the number of configured RX queues must be a power of two.
- VLAN filtering is supported.
- Link state information is provided.
- Promiscuous mode is supported.
- All multicast mode is supported.
- Multiple MAC addresses (unicast, multicast) can be configured.
- Scattered packets are supported for TX and RX.
- Inner L3/L4 (IP, TCP and UDP) TX/RX checksum offloading and validation.
- Outer L3 (IP) TX/RX checksum offloading and validation for VXLAN frames.
- Secondary process TX is supported.

5.8.3 Limitations

- RSS hash key cannot be modified.
- RSS RETA cannot be configured
- RSS always includes L3 (IPv4/IPv6) and L4 (UDP/TCP). They cannot be dissociated.
- Hardware counters are not implemented (they are software counters).
- Secondary process RX is not supported.

5.8.4 Configuration

Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX4_PMD` (default **n**)

  Toggle compilation of librte_pmd_mlx4 itself.
- `CONFIG_RTE_LIBRTE_MLX4_DEBUG` (default **n**)

  Toggle debugging code and stricter compilation flags. Enabling this option adds additional run-time checks and debugging messages at the cost of lower performance.
- `CONFIG_RTE_LIBRTE_MLX4_SGE_WR_N` (default **4**)

  Number of scatter/gather elements (SGEs) per work request (WR). Lowering this number improves performance but also limits the ability to receive scattered packets (packets that do not fit a single mbuf). The default value is a safe tradeoff.
- `CONFIG_RTE_LIBRTE_MLX4_MAX_INLINE` (default **0**)

  Amount of data to be inlined during TX operations. Improves latency but lowers throughput.

- `CONFIG_RTE_LIBRTE_MLX4_TX_MP_CACHE` (default **8**)

  Maximum number of cached memory pools (MPs) per TX queue. Each MP from which buffers are to be transmitted must be associated to memory regions (MRs). This is a slow operation that must be cached.

  This value is always 1 for RX queues since they use a single MP.

- `CONFIG_RTE_LIBRTE_MLX4_SOFT_COUNTERS` (default **1**)

  Toggle software counters. No counters are available if this option is disabled since hardware counters are not supported.

Environment variables

- `MLX4_INLINE_RECV_SIZE`

  A nonzero value enables inline receive for packets up to that size. May significantly improve performance in some cases but lower it in others. Requires careful testing.

Run-time configuration

- The only constraint when RSS mode is requested is to make sure the number of RX queues is a power of two. This is a hardware requirement.
- librte_pmd_mlx4 brings kernel network interfaces up during initialization because it is affected by their state. Forcing them down prevents packets reception.
- **ethtool** operations on related kernel interfaces also affect the PMD.

Kernel module parameters

The **mlx4_core** kernel module has several parameters that affect the behavior and/or the performance of librte_pmd_mlx4. Some of them are described below.

- **num_vfs** (integer or triplet, optionally prefixed by device address strings)

  Create the given number of VFs on the specified devices.

- **log_num_mgm_entry_size** (integer)

  Device-managed flow steering (DMFS) is required by DPDK applications. It is enabled by using a negative value, the last four bits of which have a special meaning.

  - **-1**: force device-managed flow steering (DMFS).
  - **-7**: configure optimized steering mode to improve performance with the following limitation: VLAN filtering is not supported with this mode. This is the recommended mode in case VLAN filter is not needed.

5.8.5 Prerequisites

This driver relies on external libraries and kernel drivers for resources allocations and initialization. The following dependencies are not part of DPDK and must be installed separately:

- **libibverbs**

  User space verbs framework used by librte_pmd_mlx4. This library provides a generic interface between the kernel and low-level user space drivers such as libmlx4.

  It allows slow and privileged operations (context initialization, hardware resources allocations) to be managed by the kernel and fast operations to never leave user space.

- **libmlx4**

  Low-level user space driver library for Mellanox ConnectX-3 devices, it is automatically loaded by libibverbs.

  This library basically implements send/receive calls to the hardware queues.

- **Kernel modules** (mlnx-ofed-kernel)

  They provide the kernel-side verbs API and low level device drivers that manage actual hardware initialization and resources sharing with user space processes.

  Unlike most other PMDs, these modules must remain loaded and bound to their devices:

  - mlx4_core: hardware driver managing Mellanox ConnectX-3 devices.
  - mlx4_en: Ethernet device driver that provides kernel network interfaces.
  - mlx4_ib: InifiniBand device driver.
  - ib_uverbs: user space driver for verbs (entry point for libibverbs).

- **Firmware update**

  Mellanox OFED releases include firmware updates for ConnectX-3 adapters.

  Because each release provides new features, these updates must be applied to match the kernel modules and libraries they come with.

Note: Both libraries are BSD and GPL licensed. Linux kernel modules are GPL licensed.

Currently supported by DPDK:

- Mellanox OFED **3.1**.
- Firmware version **2.35.5100** and higher.
- Supported architectures: **x86_64** and **POWER8**.

Getting Mellanox OFED

While these libraries and kernel modules are available on OpenFabrics Alliance's website and provided by package managers on most distributions, this PMD requires Ethernet extensions that may not be supported at the moment (this is a work in progress).

Mellanox OFED includes the necessary support and should be used in the meantime. For DPDK, only libibverbs, libmlx4, mlnx-ofed-kernel packages and firmware updates are required from that distribution.

Note: Several versions of Mellanox OFED are available. Installing the version this DPDK release was developed and tested against is strongly recommended. Please check the *prerequisites*.

5.8.6 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-3 devices managed by librte_pmd_mlx4.

1. Load the kernel modules:

   ```
   modprobe -a ib_uverbs mlx4_en mlx4_core mlx4_ib
   ```

   Alternatively if MLNX_OFED is fully installed, the following script can be run:

   ```
   /etc/init.d/openibd restart
   ```

   Note: User space I/O kernel modules (uio and igb_uio) are not used and do not have to be loaded.

2. Make sure Ethernet interfaces are in working order and linked to kernel verbs. Related sysfs entries should be present:

   ```
   ls -d /sys/class/net/*/device/infiniband_verbs/uverbs* | cut -d / -f 5
   ```

   Example output:

   ```
   eth2
   eth3
   eth4
   eth5
   ```

3. Optionally, retrieve their PCI bus addresses for whitelisting:

   ```
   {
       for intf in eth2 eth3 eth4 eth5;
       do
           (cd "/sys/class/net/${intf}/device/" && pwd -P);
       done;
   } |
   sed -n 's,.*/\(.*\),-w \1,p'
   ```

   Example output:

   ```
   -w 0000:83:00.0
   -w 0000:83:00.0
   -w 0000:84:00.0
   -w 0000:84:00.0
   ```

   Note: There are only two distinct PCI bus addresses because the Mellanox ConnectX-3 adapters installed on this system are dual port.

4. Request huge pages:

   ```
   echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages/nr_hugepages
   ```

5. Start testpmd with basic parameters:

```
testpmd -c 0xff00 -n 4 -w 0000:83:00.0 -w 0000:84:00.0 -- --rxq=2 --txq=2 -i
```

Example output:

```
[...]
EAL: PCI device 0000:83:00.0 on NUMA socket 1
EAL:   probe driver: 15b3:1007 librte_pmd_mlx4
PMD: librte_pmd_mlx4: PCI information matches, using device "mlx4_0" (VF: false)
PMD: librte_pmd_mlx4: 2 port(s) detected
PMD: librte_pmd_mlx4: port 1 MAC address is 00:02:c9:b5:b7:50
PMD: librte_pmd_mlx4: port 2 MAC address is 00:02:c9:b5:b7:51
EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL:   probe driver: 15b3:1007 librte_pmd_mlx4
PMD: librte_pmd_mlx4: PCI information matches, using device "mlx4_1" (VF: false)
PMD: librte_pmd_mlx4: 2 port(s) detected
PMD: librte_pmd_mlx4: port 1 MAC address is 00:02:c9:b5:ba:b0
PMD: librte_pmd_mlx4: port 2 MAC address is 00:02:c9:b5:ba:b1
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: librte_pmd_mlx4: 0x867d60: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867d60: RX queues number update: 0 -> 2
Port 0: 00:02:C9:B5:B7:50
Configuring Port 1 (socket 0)
PMD: librte_pmd_mlx4: 0x867da0: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867da0: RX queues number update: 0 -> 2
Port 1: 00:02:C9:B5:B7:51
Configuring Port 2 (socket 0)
PMD: librte_pmd_mlx4: 0x867de0: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867de0: RX queues number update: 0 -> 2
Port 2: 00:02:C9:B5:BA:B0
Configuring Port 3 (socket 0)
PMD: librte_pmd_mlx4: 0x867e20: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867e20: RX queues number update: 0 -> 2
Port 3: 00:02:C9:B5:BA:B1
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex
Port 3 Link Up - speed 40000 Mbps - full-duplex
Done
testpmd>
```

5.9 MLX5 poll mode driver

The MLX5 poll mode driver library (**librte_pmd_mlx5**) provides support for **Mellanox ConnectX-4** and **Mellanox ConnectX-4 Lx** families of 10/25/40/50/100 Gb/s adapters as well as their virtual functions (VF) in SR-IOV context.

Information and documentation about these adapters can be found on the Mellanox website. Help is also provided by the Mellanox community.

There is also a section dedicated to this poll mode driver.

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting `CONFIG_RTE_LIBRTE_MLX5_PMD=y` and recompiling DPDK.

Warning: `CONFIG_RTE_BUILD_COMBINE_LIBS` with `CONFIG_RTE_BUILD_SHARED_LIB` is not supported and thus the compilation will fail with this configuration.

5.9.1 Implementation details

Besides its dependency on libibverbs (that implies libmlx5 and associated kernel support), librte_pmd_mlx5 relies heavily on system calls for control operations such as querying/updating the MTU and flow control parameters.

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel combined with hardware specifications that allow it to handle virtual memory addresses directly ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

This capability allows the PMD to coexist with kernel network interfaces which remain functional, although they stop receiving unicast packets as long as they share the same MAC address.

Enabling librte_pmd_mlx5 causes DPDK applications to be linked against libibverbs.

5.9.2 Features

- Multiple TX and RX queues.
- Support for scattered TX and RX frames.
- IPv4, IPv6, TCPv4, TCPv6, UDPv4 and UDPv6 RSS on any number of queues.
- Several RSS hash keys, one for each flow type.
- Configurable RETA table.
- Support for multiple MAC addresses.
- VLAN filtering.
- Promiscuous mode.
- Multicast promiscuous mode.
- Hardware checksum offloads.

5.9.3 Limitations

- KVM and VMware ESX SR-IOV modes are not supported yet.
- Inner RSS for VXLAN frames is not supported yet.
- Port statistics through software counters only.
- Hardware checksum offloads for VXLAN inner header are not supported yet.
- Secondary processes are not supported yet.

5.9.4 Configuration

Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX5_PMD` (default **n**)

  Toggle compilation of librte_pmd_mlx5 itself.

- `CONFIG_RTE_LIBRTE_MLX5_DEBUG` (default **n**)

  Toggle debugging code and stricter compilation flags. Enabling this option adds additional run-time checks and debugging messages at the cost of lower performance.

- `CONFIG_RTE_LIBRTE_MLX5_SGE_WR_N` (default **4**)

  Number of scatter/gather elements (SGEs) per work request (WR). Lowering this number improves performance but also limits the ability to receive scattered packets (packets that do not fit a single mbuf). The default value is a safe tradeoff.

- `CONFIG_RTE_LIBRTE_MLX5_MAX_INLINE` (default **0**)

  Amount of data to be inlined during TX operations. Improves latency. Can improve PPS performance when PCI backpressure is detected and may be useful for scenarios involving heavy traffic on many queues.

  Since the additional software logic necessary to handle this mode can lower performance when there is no backpressure, it is not enabled by default.

- `CONFIG_RTE_LIBRTE_MLX5_TX_MP_CACHE` (default **8**)

  Maximum number of cached memory pools (MPs) per TX queue. Each MP from which buffers are to be transmitted must be associated to memory regions (MRs). This is a slow operation that must be cached.

  This value is always 1 for RX queues since they use a single MP.

Environment variables

- `MLX5_ENABLE_CQE_COMPRESSION`

  A nonzero value lets ConnectX-4 return smaller completion entries to improve performance when PCI backpressure is detected. It is most useful for scenarios involving heavy traffic on many queues.

  Since the additional software logic necessary to handle this mode can lower performance when there is no backpressure, it is not enabled by default.

Run-time configuration

- librte_pmd_mlx5 brings kernel network interfaces up during initialization because it is affected by their state. Forcing them down prevents packets reception.
- **ethtool** operations on related kernel interfaces also affect the PMD.

5.9.5 Prerequisites

This driver relies on external libraries and kernel drivers for resources allocations and initialization. The following dependencies are not part of DPDK and must be installed separately:

- **libibverbs**

  User space Verbs framework used by librte_pmd_mlx5. This library provides a generic interface between the kernel and low-level user space drivers such as libmlx5.

  It allows slow and privileged operations (context initialization, hardware resources allocations) to be managed by the kernel and fast operations to never leave user space.

- **libmlx5**

  Low-level user space driver library for Mellanox ConnectX-4 devices, it is automatically loaded by libibverbs.

  This library basically implements send/receive calls to the hardware queues.

- **Kernel modules** (mlnx-ofed-kernel)

  They provide the kernel-side Verbs API and low level device drivers that manage actual hardware initialization and resources sharing with user space processes.

  Unlike most other PMDs, these modules must remain loaded and bound to their devices:

  - mlx5_core: hardware driver managing Mellanox ConnectX-4 devices and related Ethernet kernel network devices.
  - mlx5_ib: InifiniBand device driver.
  - ib_uverbs: user space driver for Verbs (entry point for libibverbs).

- **Firmware update**

  Mellanox OFED releases include firmware updates for ConnectX-4 adapters.

  Because each release provides new features, these updates must be applied to match the kernel modules and libraries they come with.

Note: Both libraries are BSD and GPL licensed. Linux kernel modules are GPL licensed.

Currently supported by DPDK:

- Mellanox OFED **3.1-1.0.3** or **3.1-1.5.7.1** depending on usage.

  The following features are supported with version **3.1-1.5.7.1** and above only:

  - IPv6, UPDv6, TCPv6 RSS.
  - RX checksum offloads.
  - IBM POWER8.

- Minimum firmware version:

  With MLNX_OFED **3.1-1.0.3**:

  - ConnectX-4: **12.12.1240**
  - ConnectX-4 Lx: **14.12.1100**

With MLNX_OFED **3.1-1.5.7.1**:

- ConnectX-4: **12.13.0144**
- ConnectX-4 Lx: **14.13.0144**

Getting Mellanox OFED

While these libraries and kernel modules are available on OpenFabrics Alliance's website and provided by package managers on most distributions, this PMD requires Ethernet extensions that may not be supported at the moment (this is a work in progress).

Mellanox OFED includes the necessary support and should be used in the meantime. For DPDK, only libibverbs, libmlx5, mlnx-ofed-kernel packages and firmware updates are required from that distribution.

Note: Several versions of Mellanox OFED are available. Installing the version this DPDK release was developed and tested against is strongly recommended. Please check the *prerequisites*.

5.9.6 Notes for testpmd

Compared to librte_pmd_mlx4 that implements a single RSS configuration per port, librte_pmd_mlx5 supports per-protocol RSS configuration.

Since `testpmd` defaults to IP RSS mode and there is currently no command-line parameter to enable additional protocols (UDP and TCP as well as IP), the following commands must be entered from its CLI to get the same behavior as librte_pmd_mlx4:

```
> port stop all
> port config all rss all
> port start all
```

5.9.7 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-4 devices managed by librte_pmd_mlx5.

1. Load the kernel modules:

   ```
   modprobe -a ib_uverbs mlx5_core mlx5_ib
   ```

   Alternatively if MLNX_OFED is fully installed, the following script can be run:

   ```
   /etc/init.d/openibd restart
   ```

   Note: User space I/O kernel modules (uio and igb_uio) are not used and do not have to be loaded.

2. Make sure Ethernet interfaces are in working order and linked to kernel verbs. Related sysfs entries should be present:

   ```
   ls -d /sys/class/net/*/device/infiniband_verbs/uverbs* | cut -d / -f 5
   ```

Example output:

```
eth30
eth31
eth32
eth33
```

3. Optionally, retrieve their PCI bus addresses for whitelisting:

```
{
    for intf in eth2 eth3 eth4 eth5;
    do
        (cd "/sys/class/net/${intf}/device/" && pwd -P);
    done;
} |
sed -n 's,.*/\(.*\),-w \1,p'
```

Example output:

```
-w 0000:05:00.1
-w 0000:06:00.0
-w 0000:06:00.1
-w 0000:05:00.0
```

4. Request huge pages:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages/nr_hugepages
```

5. Start testpmd with basic parameters:

```
testpmd -c 0xff00 -n 4 -w 05:00.0 -w 05:00.1 -w 06:00.0 -w 06:00.1 -- --rxq=2 --txq=2 -i
```

Example output:

```
[...]
EAL: PCI device 0000:05:00.0 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_0" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fe
EAL: PCI device 0000:05:00.1 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_1" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:ff
EAL: PCI device 0000:06:00.0 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_2" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fa
EAL: PCI device 0000:06:00.1 on NUMA socket 0
EAL:   probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_3" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fb
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: librte_pmd_mlx5: 0x8cba80: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8cba80: RX queues number update: 0 -> 2
Port 0: E4:1D:2D:E7:0C:FE
Configuring Port 1 (socket 0)
PMD: librte_pmd_mlx5: 0x8ccac8: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8ccac8: RX queues number update: 0 -> 2
Port 1: E4:1D:2D:E7:0C:FF
Configuring Port 2 (socket 0)
PMD: librte_pmd_mlx5: 0x8cdb10: TX queues number update: 0 -> 2
```

```
PMD: librte_pmd_mlx5: 0x8cdb10: RX queues number update: 0 -> 2
Port 2: E4:1D:2D:E7:0C:FA
Configuring Port 3 (socket 0)
PMD: librte_pmd_mlx5: 0x8ceb58: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8ceb58: RX queues number update: 0 -> 2
Port 3: E4:1D:2D:E7:0C:FB
Checking link statuses...
Port 0 Link Up - speed 40000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex
Port 3 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

5.10 NFP poll mode driver library

Netronome's sixth generation of flow processors pack 216 programmable cores and over 100 hardware accelerators that uniquely combine packet, flow, security and content processing in a single device that scales up to 400 Gbps.

This document explains how to use DPDK with the Netronome Poll Mode Driver (PMD) supporting Netronome's Network Flow Processor 6xxx (NFP-6xxx).

Currently the driver supports virtual functions (VFs) only.

5.10.1 Dependencies

Before using the Netronome's DPDK PMD some NFP-6xxx configuration, which is not related to DPDK, is required. The system requires installation of **Netronome's BSP (Board Support Package)** which includes Linux drivers, programs and libraries.

If you have a NFP-6xxx device you should already have the code and documentation for doing this configuration. Contact **support@netronome.com** to obtain the latest available firmware.

The NFP Linux kernel drivers (including the required PF driver for the NFP) are available on Github at **https://github.com/Netronome/nfp-drv-kmods** along with build instructions.

DPDK runs in userspace and PMDs uses the Linux kernel UIO interface to allow access to physical devices from userspace. The NFP PMD requires a separate UIO driver, **nfp_uio**, to perform correct initialization. This driver is part of Netronome´s BSP and it is equivalent to Intel's igb_uio driver.

5.10.2 Building the software

Netronome's PMD code is provided in the **drivers/net/nfp** directory. Because Netronome´s BSP dependencies the driver is disabled by default in DPDK build using **common_linuxapp configuration** file. Enabling the driver or if you use another configuration file and want to have NFP support, this variable is needed:

- **CONFIG_RTE_LIBRTE_NFP_PMD=y**

Once DPDK is built all the DPDK apps and examples include support for the NFP PMD.

5.10.3 System configuration

Using the NFP PMD is not different to using other PMDs. Usual steps are:

1. **Configure hugepages:** All major Linux distributions have the hugepages functionality enabled by default. By default this allows the system uses for working with transparent hugepages. But in this case some hugepages need to be created/reserved for use with the DPDK through the hugetlbfs file system. First the virtual file system need to be mounted:

   ```
   mount -t hugetlbfs none /mnt/hugetlbfs
   ```

   The command uses the common mount point for this file system and it needs to be created if necessary.

   Configuring hugepages is performed via sysfs:

   ```
   /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

   This sysfs file is used to specify the number of hugepages to reserve. For example:

   ```
   echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

   This will reserve 2GB of memory using 1024 2MB hugepages. The file may be read to see if the operation was performed correctly:

   ```
   cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

   The number of unused hugepages may also be inspected.

   Before executing the DPDK app it should match the value of nr_hugepages.

   ```
   cat /sys/kernel/mm/hugepages/hugepages-2048kB/free_hugepages
   ```

   The hugepages reservation should be performed at system initialization and it is usual to use a kernel parameter for configuration. If the reservation is attempted on a busy system it will likely fail. Reserving memory for hugepages may be done adding the following to the grub kernel command line:

   ```
   default_hugepagesz=1M hugepagesz=2M hugepages=1024
   ```

   This will reserve 2GBytes of memory using 2Mbytes huge pages.

   Finally, for a NUMA system the allocation needs to be made on the correct NUMA node. In a DPDK app there is a master core which will (usually) perform memory allocation. It is important that some of the hugepages are reserved on the NUMA memory node where the network device is attached. This is because of a restriction in DPDK by which TX and RX descriptors rings must be created on the master code.

   Per-node allocation of hugepages may be inspected and controlled using sysfs. For example:

   ```
   cat /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
   ```

   For a NUMA system there will be a specific hugepage directory per node allowing control of hugepage reservation. A common problem may occur when hugepages reservation is performed after the system has been working for some time. Configuration using the global sysfs hugepage interface will succeed but the per-node allocations may be unsatisfactory.

   The number of hugepages that need to be reserved depends on how the app uses TX and RX descriptors, and packets mbufs.

2. **Enable SR-IOV on the NFP-6xxx device:** The current NFP PMD works with Virtual Functions (VFs) on a NFP device. Make sure that one of the Physical Function (PF) drivers from the above Github repository is installed and loaded.

   Virtual Functions need to be enabled before they can be used with the PMD. Before enabling the VFs it is useful to obtain information about the current NFP PCI device detected by the system:

   ```
   lspci -d19ee:
   ```

   Now, for example, configure two virtual functions on a NFP-6xxx device whose PCI system identity is "0000:03:00.0":

   ```
   echo 2 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
   ```

   The result of this command may be shown using lspci again:

   ```
   lspci -d19ee: -k
   ```

   Two new PCI devices should appear in the output of the above command. The -k option shows the device driver, if any, that devices are bound to. Depending on the modules loaded at this point the new PCI devices may be bound to nfp_netvf driver.

3. **To install the uio kernel module (manually):** All major Linux distributions have support for this kernel module so it is straightforward to install it:

   ```
   modprobe uio
   ```

   The module should now be listed by the lsmod command.

4. **To install the nfp_uio kernel module (manually):** This module supports NFP-6xxx devices through the UIO interface.

   This module is part of Netronome´s BSP and it should be available when the BSP is installed.

   ```
   modprobe nfp_uio.ko
   ```

   The module should now be listed by the lsmod command.

   Depending on which NFP modules are loaded, nfp_uio may be automatically bound to the NFP PCI devices by the system. Otherwise the binding needs to be done explicitly. This is the case when nfp_netvf, the Linux kernel driver for NFP VFs, was loaded when VFs were created. As described later in this document this configuration may also be performed using scripts provided by the Netronome´s BSP.

   First the device needs to be unbound, for example from the nfp_netvf driver:

   ```
   echo 0000:03:08.0 > /sys/bus/pci/devices/0000:03:08.0/driver/unbind

   lspci -d19ee: -k
   ```

   The output of lspci should now show that 0000:03:08.0 is not bound to any driver.

   The next step is to add the NFP PCI ID to the NFP UIO driver:

   ```
   echo 19ee 6003 > /sys/bus/pci/drivers/nfp_uio/new_id
   ```

   And then to bind the device to the nfp_uio driver:

   ```
   echo 0000:03:08.0 > /sys/bus/pci/drivers/nfp_uio/bind

   lspci -d19ee: -k
   ```

   lspci should show that device bound to nfp_uio driver.

5. **Using tools from Netronome´s BSP to install and bind modules:** DPDK provides scripts which are useful for installing the UIO modules and for binding the right device to those modules avoiding doing so manually. However, these scripts have not support for Netronome´s UIO driver. Along with drivers, the BSP installs those DPDK scripts slightly modified with support for Netronome´s UIO driver.

   Those specific scripts can be found in Netronome´s BSP installation directory. Refer to BSP documentation for more information.

   - **setup.sh**
   - **dpdk_nic_bind.py**

   Configuration may be performed by running setup.sh which invokes dpdk_nic_bind.py as needed. Executing setup.sh will display a menu of configuration options.

5.11 SZEDATA2 poll mode driver library

The SZEDATA2 poll mode driver library implements support for cards from COMBO family (**COMBO-80G**, **COMBO-100G**). The SZEDATA2 PMD is virtual PMD which uses interface provided by libsze2 library to communicate with COMBO cards over sze2 layer.

More information about family of COMBO cards and used technology (NetCOPE platform) can be found on the Liberouter website.

Note: This driver has external dependencies. Therefore it is disabled in default configuration files. It can be enabled by setting `CONFIG_RTE_LIBRTE_PMD_SZEDATA2=y` and recompiling.

Note: Currently the driver is supported only on x86_64 architectures. Only x86_64 versions of the external libraries are provided.

5.11.1 Prerequisites

This PMD requires kernel modules which are responsible for initialization and allocation of resources needed for sze2 layer function. Communication between PMD and kernel modules is mediated by libsze2 library. These kernel modules and library are not part of DPDK and must be installed separately:

- **libsze2 library**

  The library provides API for initialization of sze2 transfers, receiving and transmitting data segments.

- **Kernel modules**

  - combov3
  - szedata2_cv3

  Kernel modules manage initialization of hardware, allocation and sharing of resources for user space applications:

Information about getting the dependencies can be found here.

5.11.2 Using the SZEDATA2 PMD

SZEDATA2 PMD can be created by passing `--vdev=` option to EAL in the following format:

```
--vdev 'DEVICE,dev_path=PATH,rx_ifaces=RX_MASK,tx_ifaces=TX_MASK'
```

`DEVICE` and options `dev_path`, `rx_ifaces`, `tx_ifaces` are mandatory and must be separated by commas.

- `DEVICE`: contains prefix `eth_szedata2` followed by numbers or letters, must be unique for each virtual device
- `dev_path`: Defines path to szedata2 device. Value is valid path to szedata2 device. Example:

  ```
  dev_path=/dev/szedataII0
  ```

- `rx_ifaces`: Defines which receive channels will be used. For each channel is created one queue. Value is mask for selecting which receive channels are required. Example:

  ```
  rx_ifaces=0x3
  ```

- `tx_ifaces`: Defines which transmit channels will be used. For each channel is created one queue. Value is mask for selecting which transmit channels are required. Example:

  ```
  tx_ifaces=0x3
  ```

5.11.3 Example of usage

Read packets from 0. and 1. receive channel and write them to 0. and 1. transmit channel:

```
$RTE_TARGET/app/testpmd -c 0xf -n 2 \
--vdev 'eth_szedata20,dev_path=/dev/szedataII0,rx_ifaces=0x3,tx_ifaces=0x3' \
-- --port-topology=chained --rxq=2 --txq=2 --nb-cores=2
```

5.12 Poll Mode Driver for Emulated Virtio NIC

Virtio is a para-virtualization framework initiated by IBM, and supported by KVM hypervisor. In the Data Plane Development Kit (DPDK), we provide a virtio Poll Mode Driver (PMD) as a software solution, comparing to SRIOV hardware solution, for fast guest VM to guest VM communication and guest VM to host communication.

Vhost is a kernel acceleration module for virtio qemu backend. The DPDK extends kni to support vhost raw socket interface, which enables vhost to directly read/ write packets from/to a physical port. With this enhancement, virtio could achieve quite promising performance.

In future release, we will also make enhancement to vhost backend, releasing peak performance of virtio PMD driver.

For basic qemu-KVM installation and other Intel EM poll mode driver in guest VM, please refer to Chapter “Driver for VM Emulated Devices”.

In this chapter, we will demonstrate usage of virtio PMD driver with two backends, standard qemu vhost back end and vhost kni back end.

5.12.1 Virtio Implementation in DPDK

For details about the virtio spec, refer to Virtio PCI Card Specification written by Rusty Russell.

As a PMD, virtio provides packet reception and transmission callbacks virtio_recv_pkts and virtio_xmit_pkts.

In virtio_recv_pkts, index in range [vq->vq_used_cons_idx , vq->vq_ring.used->idx) in vring is available for virtio to burst out.

In virtio_xmit_pkts, same index range in vring is available for virtio to clean. Virtio will enqueue to be transmitted packets into vring, advance the vq->vq_ring.avail->idx, and then notify the host back end if necessary.

5.12.2 Features and Limitations of virtio PMD

In this release, the virtio PMD driver provides the basic functionality of packet reception and transmission.

- It supports merge-able buffers per packet when receiving packets and scattered buffer per packet when transmitting packets. The packet size supported is from 64 to 1518.
- It supports multicast packets and promiscuous mode.
- The descriptor number for the RX/TX queue is hard-coded to be 256 by qemu. If given a different descriptor number by the upper application, the virtio PMD generates a warning and fall back to the hard-coded value.
- Features of mac/vlan filter are supported, negotiation with vhost/backend are needed to support them. When backend can't support vlan filter, virtio app on guest should disable vlan filter to make sure the virtio port is configured correctly. E.g. specify '–disable-hw-vlan' in testpmd command line.
- RTE_PKTMBUF_HEADROOM should be defined larger than sizeof(struct virtio_net_hdr), which is 10 bytes.
- Virtio does not support runtime configuration.
- Virtio supports Link State interrupt.
- Virtio supports software vlan stripping and inserting.
- Virtio supports using port IO to get PCI resource when uio/igb_uio module is not available.

5.12.3 Prerequisites

The following prerequisites apply:

- In the BIOS, turn VT-x and VT-d on
- Linux kernel with KVM module; vhost module loaded and ioeventfd supported. Qemu standard backend without vhost support isn't tested, and probably isn't supported.

5.12.4 Virtio with kni vhost Back End

This section demonstrates kni vhost back end example setup for Phy-VM Communication.

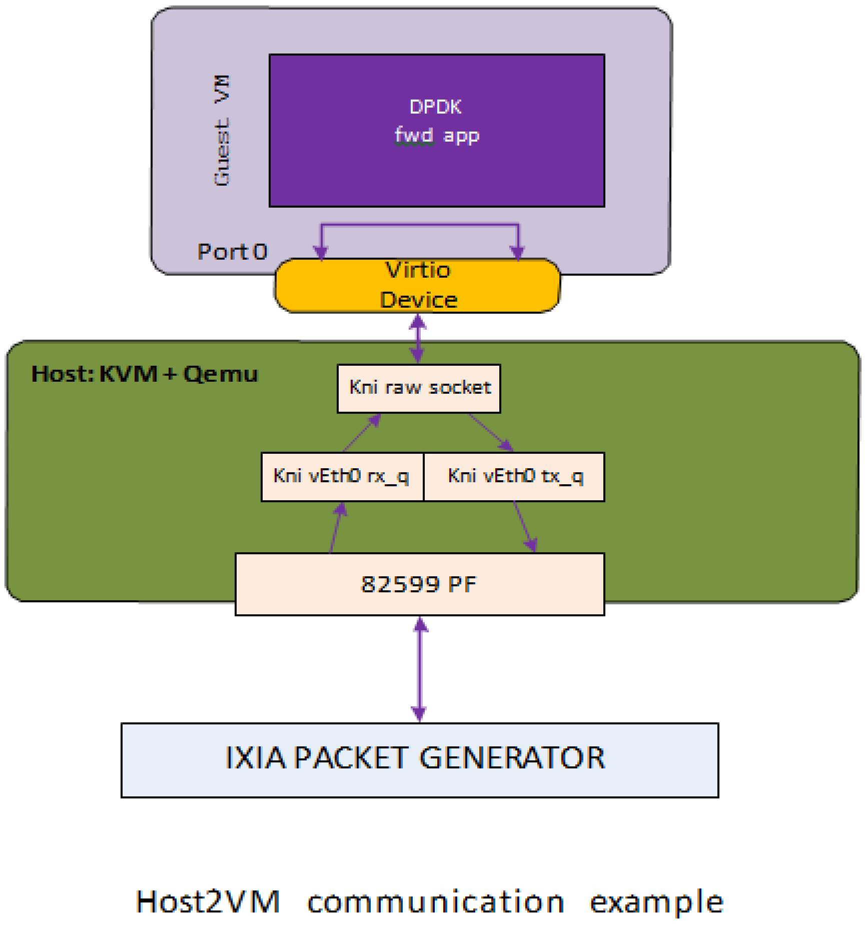


Fig. 5.5: Host2VM Communication Example Using kni vhost Back End

Host2VM communication example

1. Load the kni kernel module:

   ```
   insmod rte_kni.ko
   ```

   Other basic DPDK preparations like hugepage enabling, uio port binding are not listed here. Please refer to the *DPDK Getting Started Guide* for detailed instructions.

2. Launch the kni user application:

   ```
   examples/kni/build/app/kni -c 0xf -n 4 -- -p 0x1 -P --config="(0,1,3)"
   ```

   This command generates one network device vEth0 for physical port. If specify more physical ports, the generated network device will be vEth1, vEth2, and so on.

   For each physical port, kni creates two user threads. One thread loops to fetch packets from the physical NIC port into the kni receive queue. The other user thread loops to send packets in the kni transmit queue.

   For each physical port, kni also creates a kernel thread that retrieves packets from the kni receive queue, place them onto kni's raw socket's queue and wake up the vhost kernel thread to exchange packets with the virtio virt queue.

For more details about kni, please refer to Chapter 24 “Kernel NIC Interface”.

3. Enable the kni raw socket functionality for the specified physical NIC port, get the generated file descriptor and set it in the qemu command line parameter. Always remember to set ioeventfd_on and vhost_on.

   Example:

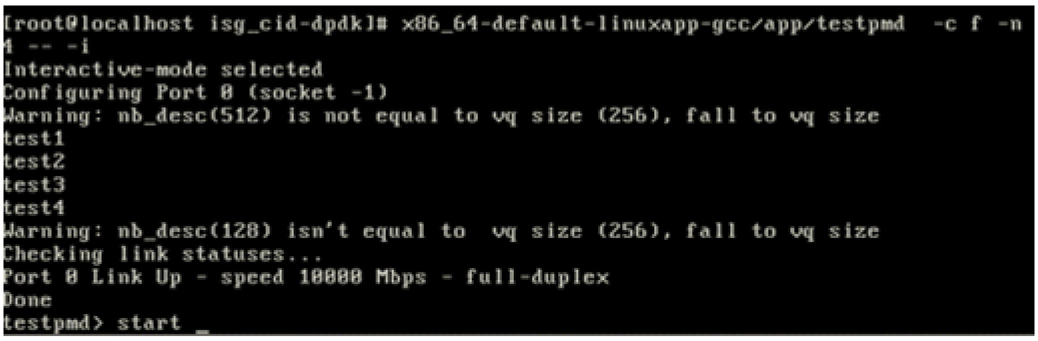
   ```
   echo 1 > /sys/class/net/vEth0/sock_en
   fd=`cat /sys/class/net/vEth0/sock_fd`
   exec qemu-system-x86_64 -enable-kvm -cpu host \
   -m 2048 -smp 4 -name dpdk-test1-vm1 \
   -drive file=/data/DPDKVMS/dpdk-vm.img \
   -netdev tap, fd=$fd,id=mynet_kni, script=no,vhost=on \
   -device virtio-net-pci,netdev=mynet_kni,bus=pci.0,addr=0x3,ioeventfd=on \
   -vnc:1 -daemonize
   ```

   In the above example, virtio port 0 in the guest VM will be associated with vEth0, which in turns corresponds to a physical port, which means received packets come from vEth0, and transmitted packets is sent to vEth0.

4. In the guest, bind the virtio device to the uio_pci_generic kernel module and start the forwarding application. When the virtio port in guest bursts rx, it is getting packets from the raw socket’s receive queue. When the virtio port bursts tx, it is sending packet to the tx_q.

   ```
   modprobe uio
   echo 512 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
   modprobe uio_pci_generic
   python tools/dpdk_nic_bind.py -b uio_pci_generic 00:03.0
   ```

   We use testpmd as the forwarding application in this example.

   ```
   [root@localhost isg_cid-dpdk]# x86_64-default-linuxapp-gcc/app/testpmd  -c f -n
   4 -- -i
   Interactive-mode selected
   Configuring Port 0 (socket -1)
   Warning: nb_desc(512) is not equal to vq size (256), fall to vq size
   test1
   test2
   test3
   test4
   Warning: nb_desc(128) isn't equal to  vq size (256), fall to vq size
   Checking link statuses...
   Port 0 Link Up - speed 10000 Mbps - full-duplex
   Done
   testpmd> start _
   ```

   Fig. 5.6: Running testpmd

5. Use IXIA packet generator to inject a packet stream into the KNI physical port.

   The packet reception and transmission flow path is:

   IXIA packet generator->82599 PF->KNI rx queue->KNI raw socket queue->Guest VM virtio port 0 rx burst->Guest VM virtio port 0 tx burst-> KNI tx queue->82599 PF-> IXIA packet generator

5.12.5 Virtio with qemu virtio Back End

```
qemu-system-x86_64 -enable-kvm -cpu host -m 2048 -smp 2 -mem-path /dev/
hugepages -mem-prealloc
-drive file=/data/DPDKVMS/dpdk-vm1
```

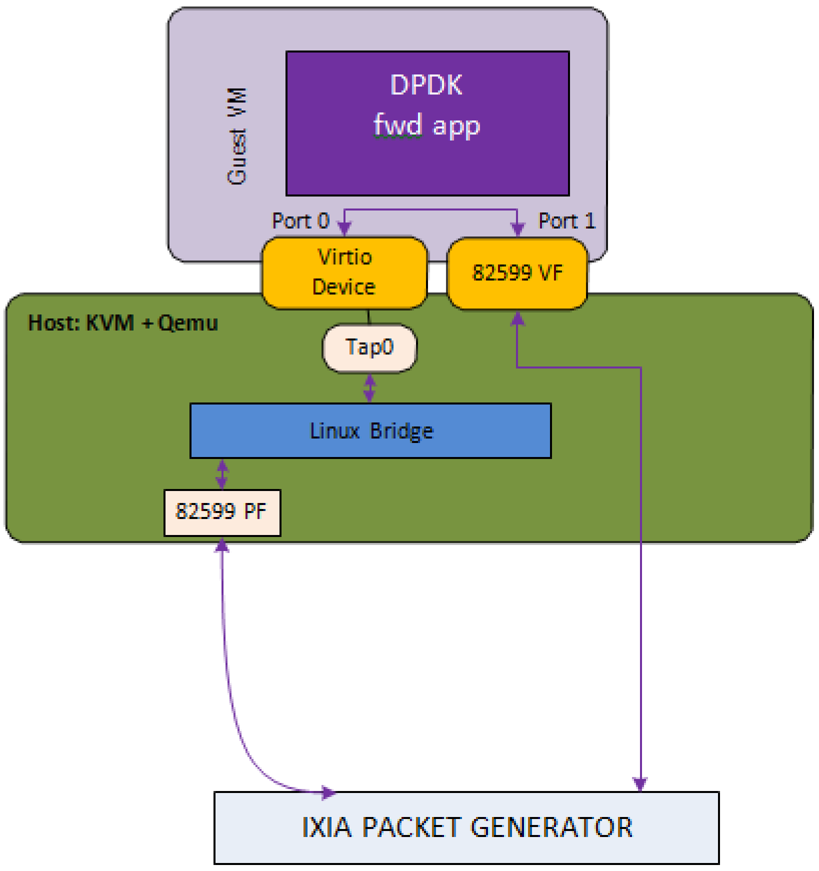


Fig. 5.7: Host2VM Communication Example Using qemu vhost Back End

```
-netdev tap,id=vm1_p1,ifname=tap0,script=no,vhost=on
-device virtio-net-pci,netdev=vm1_p1,bus=pci.0,addr=0x3,ioeventfd=on
-device pci-assign,host=04:10.1 \
```

In this example, the packet reception flow path is:

> IXIA packet generator->82599 PF->Linux Bridge->TAP0's socket queue-> Guest VM virtio port 0 rx burst-> Guest VM 82599 VF port1 tx burst-> IXIA packet generator

The packet transmission flow is:

> IXIA packet generator-> Guest VM 82599 VF port1 rx burst-> Guest VM virtio port 0 tx burst-> tap -> Linux Bridge->82599 PF-> IXIA packet generator

5.13 Poll Mode Driver for Paravirtual VMXNET3 NIC

The VMXNET3 adapter is the next generation of a paravirtualized NIC, introduced by VMware* ESXi. It is designed for performance and is not related to VMXNET or VMXENET2. It offers all the features available in VMXNET2, and adds several new features such as, multi-queue support (also known as Receive Side Scaling, RSS), IPv6 offloads, and MSI/MSI-X interrupt delivery. Because operating system vendors do not provide built-in drivers for this card, VMware Tools must be installed to have a driver for the VMXNET3 network adapter available. One can use the same device in a DPDK application with VMXNET3 PMD introduced in DPDK API.

Currently, the driver provides basic support for using the device in a DPDK application running on a guest OS. Optimization is needed on the backend, that is, the VMware* ESXi vmkernel switch, to achieve optimal performance end-to-end.

In this chapter, two setups with the use of the VMXNET3 PMD are demonstrated:

1. Vmxnet3 with a native NIC connected to a vSwitch
2. Vmxnet3 chaining VMs connected to a vSwitch

5.13.1 VMXNET3 Implementation in the DPDK

For details on the VMXNET3 device, refer to the VMXNET3 driver's vmxnet3 directory and support manual from VMware*.

For performance details, refer to the following link from VMware:

http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf

As a PMD, the VMXNET3 driver provides the packet reception and transmission callbacks, vmxnet3_recv_pkts and vmxnet3_xmit_pkts. It does not support scattered packet reception as part of vmxnet3_recv_pkts and vmxnet3_xmit_pkts. Also, it does not support scattered packet reception as part of the device operations supported.

The VMXNET3 PMD handles all the packet buffer memory allocation and resides in guest address space and it is solely responsible to free that memory when not needed. The packet buffers and features to be supported are made available to hypervisor via VMXNET3 PCI configuration space BARs. During RX/TX, the packet buffers are exchanged by their GPAs, and the hypervisor loads the buffers with packets in the RX case and sends packets to vSwitch in the TX case.

The VMXNET3 PMD is compiled with vmxnet3 device headers. The interface is similar to that of the other PMDs available in the DPDK API. The driver pre-allocates the packet buffers and loads the command ring descriptors in advance. The hypervisor fills those packet buffers on packet arrival and write completion ring descriptors, which are eventually pulled by the PMD. After reception, the DPDK application frees the descriptors and loads new packet buffers for the coming packets. The interrupts are disabled and there is no notification required. This keeps performance up on the RX side, even though the device provides a notification feature.

In the transmit routine, the DPDK application fills packet buffer pointers in the descriptors of the command ring and notifies the hypervisor. In response the hypervisor takes packets and passes them to the vSwitch. It writes into the completion descriptors ring. The rings are read by the PMD in the next transmit routine call and the buffers and descriptors are freed from memory.

5.13.2 Features and Limitations of VMXNET3 PMD

In release 1.6.0, the VMXNET3 PMD provides the basic functionality of packet reception and transmission. There are several options available for filtering packets at VMXNET3 device level including:

1. MAC Address based filtering:
    - Unicast, Broadcast, All Multicast modes - SUPPORTED BY DEFAULT
    - Multicast with Multicast Filter table - NOT SUPPORTED
    - Promiscuous mode - SUPPORTED
    - RSS based load balancing between queues - SUPPORTED
2. VLAN filtering:
    - VLAN tag based filtering without load balancing - SUPPORTED

Note:

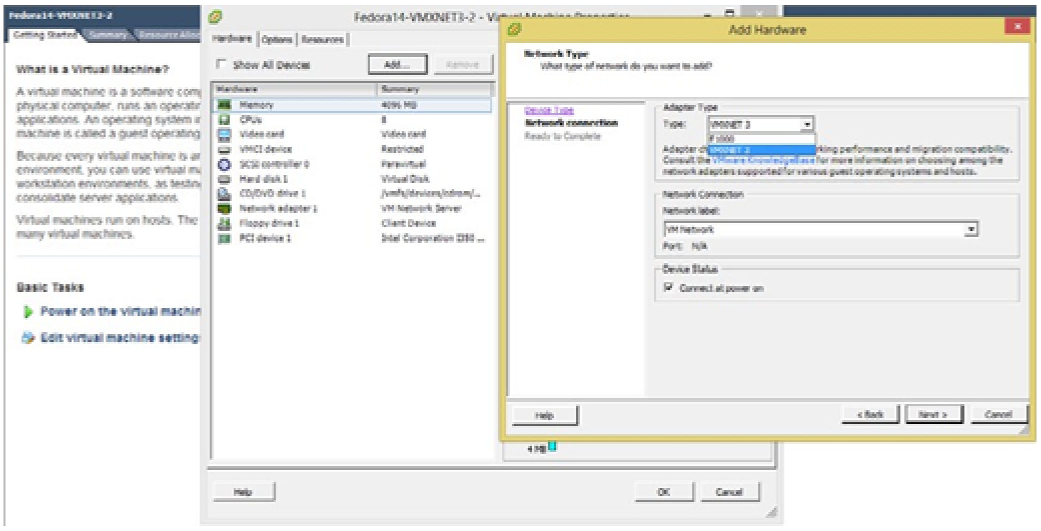
- Release 1.6.0 does not support separate headers and body receive cmd_ring and hence, multiple segment buffers are not supported. Only cmd_ring_0 is used for packet buffers, one for each descriptor.
- Receive and transmit of scattered packets is not supported.
- Multicast with Multicast Filter table is not supported.

5.13.3 Prerequisites

The following prerequisites apply:

- Before starting a VM, a VMXNET3 interface to a VM through VMware vSphere Client must be assigned. This is shown in the figure below.

Note: Depending on the Virtual Machine type, the VMware vSphere Client shows Ethernet adaptors while adding an Ethernet device. Ensure that the VM type used offers a VMXNET3


Add Hardware

Network Type
What type of network do you want to add?

Device Type
Network connection
Ready to Complete

Adapter Type
Type: VMXNET 3
E1000
VMXNET 3

Network Connection
Network label: VM Network
Port: N/A

Device Status
☑ Connect at power on

Fig. 5.8: Assigning a VMXNET3 interface to a VM using VMware vSphere Client

device. Refer to the VMware documentation for a listed of VMs.

Note: Follow the *DPDK Getting Started Guide* to setup the basic DPDK environment.

Note: Follow the *DPDK Sample Application's User Guide*, L2 Forwarding/L3 Forwarding and TestPMD for instructions on how to run a DPDK application using an assigned VMXNET3 device.

5.13.4 VMXNET3 with a Native NIC Connected to a vSwitch

This section describes an example setup for Phy-vSwitch-VM-Phy communication.

Note: Other instructions on preparing to use DPDK such as, hugepage enabling, uio port binding are not listed here. Please refer to *DPDK Getting Started Guide and DPDK Sample Application's User Guide* for detailed instructions.

The packet reception and transmission flow path is:

```
Packet generator -> 82576
                 -> VMware ESXi vSwitch
                 -> VMXNET3 device
                 -> Guest VM VMXNET3 port 0 rx burst
                 -> Guest VM 82599 VF port 0 tx burst
                 -> 82599 VF
                 -> Packet generator
```

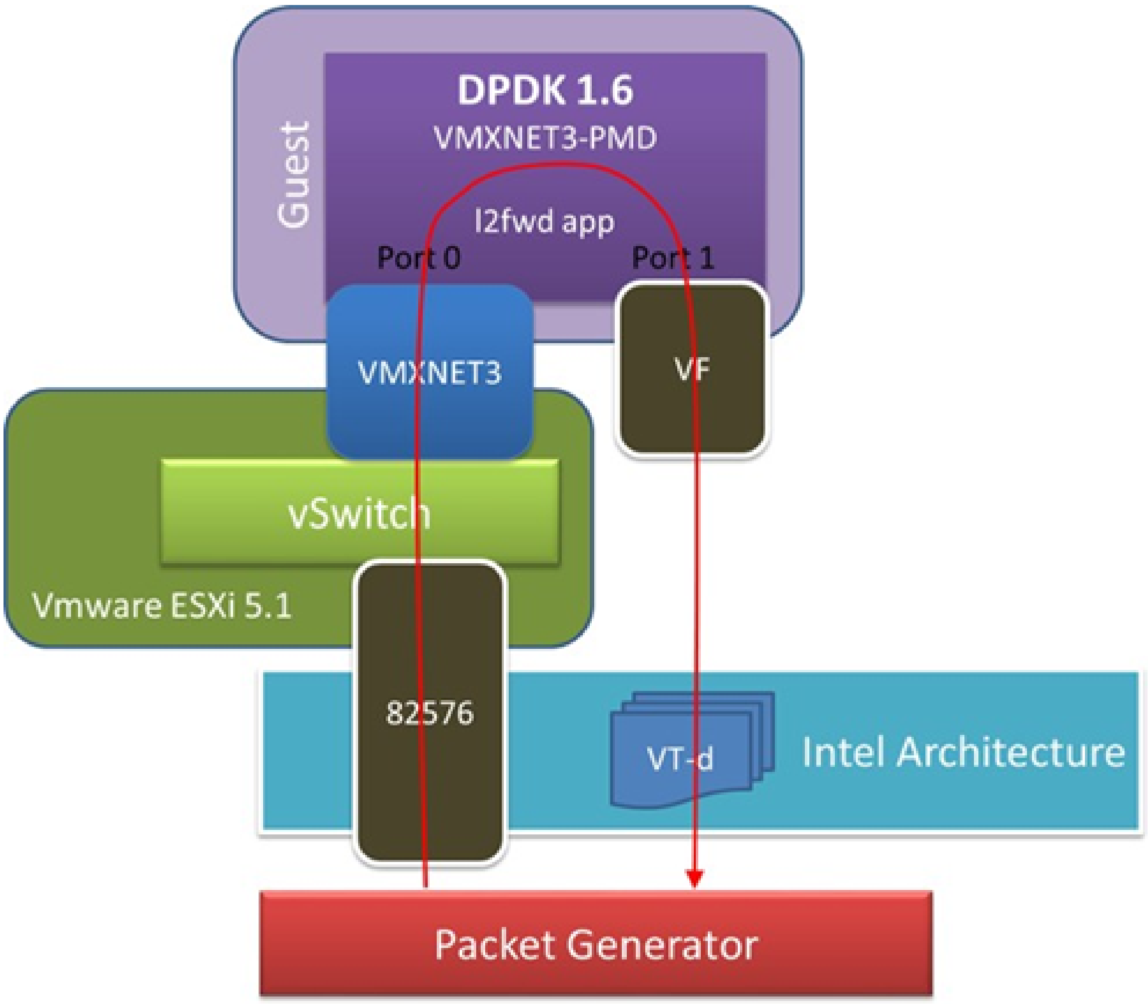


Fig. 5.9: VMXNET3 with a Native NIC Connected to a vSwitch

5.13.5 VMXNET3 Chaining VMs Connected to a vSwitch

The following figure shows an example VM-to-VM communication over a Phy-VM-vSwitch-VM-Phy communication channel.

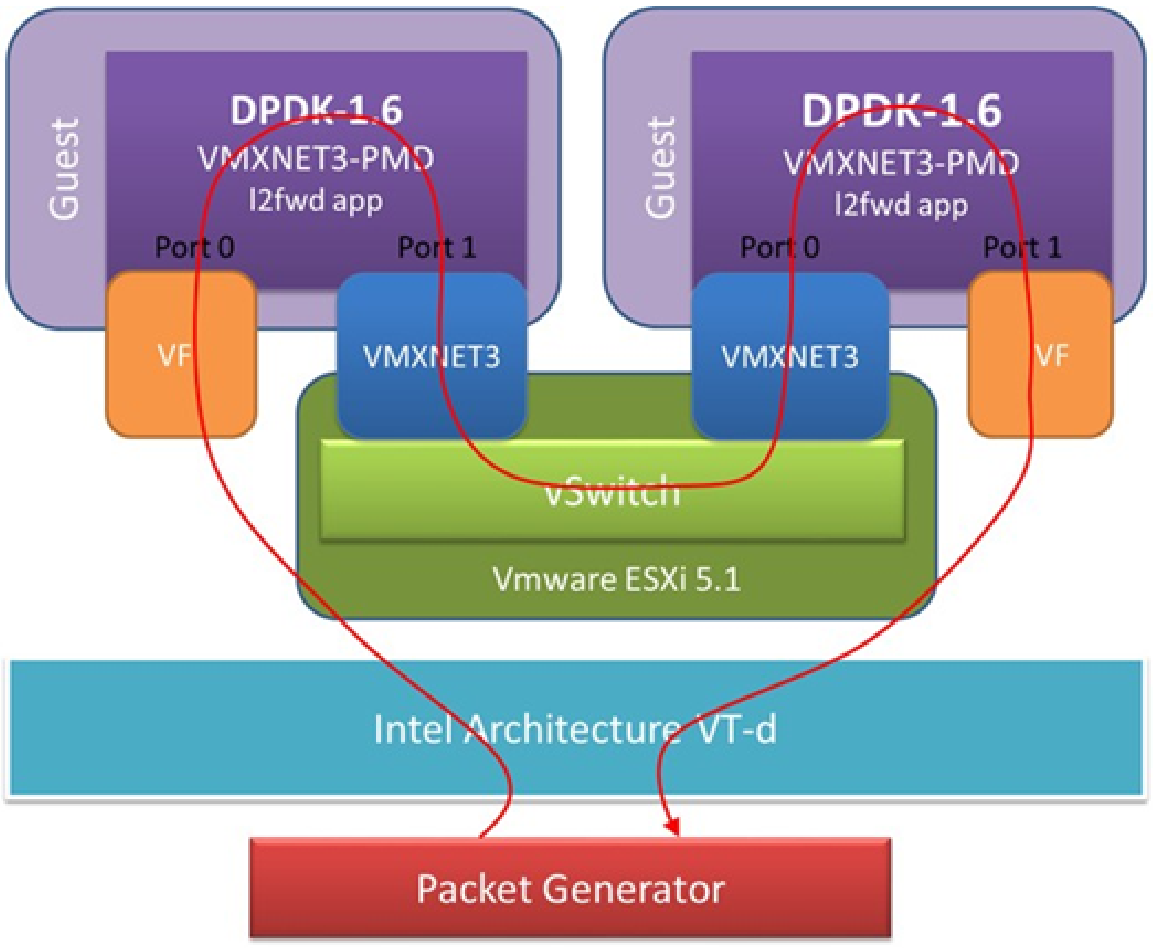


Fig. 5.10: VMXNET3 Chaining VMs Connected to a vSwitch

Note: When using the L2 Forwarding or L3 Forwarding applications, a destination MAC address needs to be written in packets to hit the other VM's VMXNET3 interface.

In this example, the packet flow path is:

```
Packet generator -> 82599 VF
                 -> Guest VM 82599 port 0 rx burst
                 -> Guest VM VMXNET3 port 1 tx burst
                 -> VMXNET3 device
                 -> VMware ESXi vSwitch
                 -> VMXNET3 device
                 -> Guest VM VMXNET3 port 0 rx burst
                 -> Guest VM 82599 VF port 1 tx burst
                 -> 82599 VF
                 -> Packet generator
```

5.14 Libpcap and Ring Based Poll Mode Drivers

In addition to Poll Mode Drivers (PMDs) for physical and virtual hardware, the DPDK also includes two pure-software PMDs. These two drivers are:

- A libpcap -based PMD (librte_pmd_pcap) that reads and writes packets using libpcap, - both from files on disk, as well as from physical NIC devices using standard Linux kernel drivers.
- A ring-based PMD (librte_pmd_ring) that allows a set of software FIFOs (that is, rte_ring) to be accessed using the PMD APIs, as though they were physical NICs.

Note: The libpcap -based PMD is disabled by default in the build configuration files, owing to an external dependency on the libpcap development files which must be installed on the board. Once the libpcap development files are installed, the library can be enabled by setting CONFIG_RTE_LIBRTE_PMD_PCAP=y and recompiling the Intel® DPDK.

5.14.1 Using the Drivers from the EAL Command Line

For ease of use, the DPDK EAL also has been extended to allow pseudo-Ethernet devices, using one or more of these drivers, to be created at application startup time during EAL initialization.

To do so, the –vdev= parameter must be passed to the EAL. This takes take options to allow ring and pcap-based Ethernet to be allocated and used transparently by the application. This can be used, for example, for testing on a virtual machine where there are no Ethernet ports.

Libpcap-based PMD

Pcap-based devices can be created using the virtual device –vdev option. The device name must start with the eth_pcap prefix followed by numbers or letters. The name is unique for each device. Each device can have multiple stream options and multiple devices can be used. Multiple device definitions can be arranged using multiple –vdev. Device name and stream options must be separated by commas as shown below:

```
$RTE_TARGET/app/testpmd -c f -n 4 --vdev  'eth_pcap0,stream_opt0=..,stream_opt1=..' --vdev='eth
```

Device Streams

Multiple ways of stream definitions can be assessed and combined as long as the following two rules are respected:

- A device is provided with two different streams - reception and transmission.
- A device is provided with one network interface name used for reading and writing packets.

The different stream types are:

- rx_pcap: Defines a reception stream based on a pcap file. The driver reads each packet within the given pcap file as if it was receiving it from the wire. The value is a path to a valid pcap file.

rx_pcap=/path/to/file.pcap

- tx_pcap: Defines a transmission stream based on a pcap file. The driver writes each received packet to the given pcap file. The value is a path to a pcap file. The file is overwritten if it already exists and it is created if it does not.

  tx_pcap=/path/to/file.pcap

- rx_iface: Defines a reception stream based on a network interface name. The driver reads packets coming from the given interface using the Linux kernel driver for that interface. The value is an interface name.

  rx_iface=eth0

- tx_iface: Defines a transmission stream based on a network interface name. The driver sends packets to the given interface using the Linux kernel driver for that interface. The value is an interface name.

  tx_iface=eth0

- iface: Defines a device mapping a network interface. The driver both reads and writes packets from and to the given interface. The value is an interface name.

  iface=eth0

Examples of Usage

Read packets from one pcap file and write them to another:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,rx_pcap=/path/to/ file_rx.pcap,tx_pcap=
```

Read packets from a network interface and write them to a pcap file:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,rx_iface=eth0,tx_pcap=/path/to/file_tx.
```

Read packets from a pcap file and write them to a network interface:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,rx_pcap=/path/to/ file_rx.pcap,tx_iface
```

Forward packets through two network interfaces:

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,iface=eth0' --vdev='eth_pcap1;iface=eth
```

Using libpcap-based PMD with the testpmd Application

One of the first things that testpmd does before starting to forward packets is to flush the RX streams by reading the first 512 packets on every RX stream and discarding them. When using a libpcap-based PMD this behavior can be turned off using the following command line option:

```
--no-flush-rx
```

It is also available in the runtime command line:

```
set flush_rx on/off
```

It is useful for the case where the rx_pcap is being used and no packets are meant to be discarded. Otherwise, the first 512 packets from the input pcap file will be discarded by the RX flushing operation.

```
$RTE_TARGET/app/testpmd -c '0xf' -n 4 --vdev 'eth_pcap0,rx_pcap=/path/to/ file_rx.pcap,tx_pcap=
```

Rings-based PMD

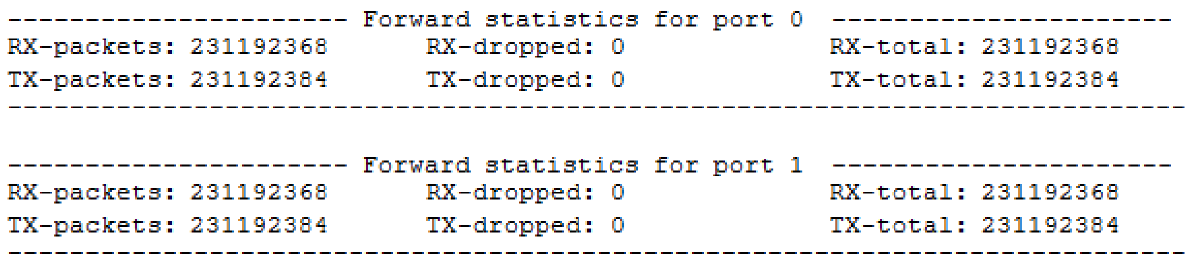
To run a DPDK application on a machine without any Ethernet devices, a pair of ring-based rte_ethdevs can be used as below. The device names passed to the –vdev option must start with eth_ring and take no additional parameters. Multiple devices may be specified, separated by commas.

```
./testpmd -c E -n 4 --vdev=eth_ring0 --vdev=eth_ring1 -- -i
EAL: Detected lcore 1 as core 1 on socket 0
...

Interactive-mode selected
Configuring Port 0 (socket 0)
Configuring Port 1 (socket 0)
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done

testpmd> start tx_first
io packet forwarding - CRC stripping disabled - packets/burst=16
nb forwarding cores=1 - nb forwarding ports=2
RX queues=1 - RX desc=128 - RX free threshold=0
RX threshold registers: pthresh=8 hthresh=8 wthresh=4
TX queues=1 - TX desc=512 - TX free threshold=0
TX threshold registers: pthresh=36 hthresh=0 wthresh=0
TX RS bit threshold=0 - TXQ flags=0x0

testpmd> stop
Telling cores to stop...
Waiting for lcores to finish...
```



```
----------------------- Forward statistics for port 0  ----------------------
RX-packets: 231192368      RX-dropped: 0            RX-total: 231192368
TX-packets: 231192384      TX-dropped: 0            TX-total: 231192384
----------------------------------------------------------------------------

----------------------- Forward statistics for port 1  ----------------------
RX-packets: 231192368      RX-dropped: 0            RX-total: 231192368
TX-packets: 231192384      TX-dropped: 0            TX-total: 231192384
----------------------------------------------------------------------------
```

```
+++++++++++++++ Accumulated forward statistics for allports++++++++++
RX-packets: 462384736  RX-dropped: 0 RX-total: 462384736
TX-packets: 462384768  TX-dropped: 0 TX-total: 462384768
+++++++++++++++++++++++++++++++++++++++++++++++++++++

Done.
```

Using the Poll Mode Driver from an Application

Both drivers can provide similar APIs to allow the user to create a PMD, that is, rte_ethdev structure, instances at run-time in the end-application, for example, using rte_eth_from_rings() or rte_eth_from_pcaps() APIs. For the rings-based PMD, this functionality could be used, for example, to allow data exchange between cores using rings to be done in exactly the same way as sending or receiving packets from an Ethernet device. For the libpcap-based PMD, it allows an application to open one or more pcap files and use these as a source of packet input to the application.

Usage Examples

To create two pseudo-Ethernet ports where all traffic sent to a port is looped back for reception on the same port (error handling omitted for clarity):

```
#define RING_SIZE 256
#define NUM_RINGS 2
#define SOCKET0 0

struct rte_ring *ring[NUM_RINGS];
int port0, port1;

ring[0] = rte_ring_create("R0", RING_SIZE, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);
ring[1] = rte_ring_create("R1", RING_SIZE, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);

/* create two ethdev's */

port0 = rte_eth_from_rings("eth_ring0", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);
port1 = rte_eth_from_rings("eth_ring1", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);
```

To create two pseudo-Ethernet ports where the traffic is switched between them, that is, traffic sent to port 0 is read back from port 1 and vice-versa, the final two lines could be changed as below:

```
port0 = rte_eth_from_rings("eth_ring0", &ring[0], 1, &ring[1], 1, SOCKET0);
port1 = rte_eth_from_rings("eth_ring1", &ring[1], 1, &ring[0], 1, SOCKET0);
```

This type of configuration could be useful in a pipeline model, for example, where one may want to have inter-core communication using pseudo Ethernet devices rather than raw rings, for reasons of API consistency.

Enqueuing and dequeuing items from an rte_ring using the rings-based PMD may be slower than using the native rings API. This is because DPDK Ethernet drivers make use of function pointers to call the appropriate enqueue or dequeue functions, while the rte_ring specific functions are direct function calls in the code and are often inlined by the compiler.

> Once an ethdev has been created, for either a ring or a pcap-based PMD, it should be configured and started in the same way as a regular Ethernet device, that is, by calling rte_eth_dev_configure() to set the number of receive and transmit queues, then calling rte_eth_rx_queue_setup() / tx_queue_setup() for each of those queues and finally calling rte_eth_dev_start() to allow transmission and reception of packets to begin.

Figures

CHAPTER 6

Crypto Device Drivers

6.1 AESN-NI Multi Buffer Crytpo Poll Mode Driver

The AESNI MB PMD (**librte_pmd_aesni_mb**) provides poll mode crypto driver support for utilizing Intel multi buffer library, see the white paper Fast Multi-buffer IPsec Implementations on Intel® Architecture Processors.

The AES-NI MB PMD has current only been tested on Fedora 21 64-bit with gcc.

6.1.1 Features

AESNI MB PMD has support for:

Cipher algorithms:

- RTE_CRYPTO_SYM_CIPHER_AES128_CBC
- RTE_CRYPTO_SYM_CIPHER_AES256_CBC
- RTE_CRYPTO_SYM_CIPHER_AES512_CBC

Hash algorithms:

- RTE_CRYPTO_SYM_HASH_SHA1_HMAC
- RTE_CRYPTO_SYM_HASH_SHA256_HMAC
- RTE_CRYPTO_SYM_HASH_SHA512_HMAC

6.1.2 Limitations

- Chained mbufs are not supported.
- Hash only is not supported.
- Cipher only is not supported.
- Only in-place is currently supported (destination address is the same as source address).
- Only supports session-oriented API implementation (session-less APIs are not supported).
- Not performance tuned.

6.1.3 Installation

To build DPDK with the AESNI_MB_PMD the user is required to download the mult- buffer library from here and compile it on their user system before building DPDK. When building the multi-buffer library it is necessary to have YASM package installed and also requires the overriding of YASM path when building, as a path is hard coded in the Makefile of the release package.

```
make YASM=/usr/bin/yasm
```

The environmental variable AESNI_MULTI_BUFFER_LIB_PATH must be exported with the path where you extracted and built the multi buffer library and finally set CONFIG_RTE_LIBRTE_PMD_AESNI_MB=y in config/common_linuxapp.

6.2 Quick Assist Crypto Poll Mode Driver

The QAT PMD provides poll mode crypto driver support for **Intel QuickAssist Technology DH895xxC** hardware accelerator.

The QAT PMD has been tested on Fedora 21 64-bit with gcc and on the 4.3 kernel.org Linux kernel.

6.2.1 Features

The QAT PMD has support for:

Cipher algorithms:

- `RTE_CRYPTO_SYM_CIPHER_AES128_CBC`
- `RTE_CRYPTO_SYM_CIPHER_AES192_CBC`
- `RTE_CRYPTO_SYM_CIPHER_AES256_CBC`

Hash algorithms:

- `RTE_CRYPTO_AUTH_SHA1_HMAC`
- `RTE_CRYPTO_AUTH_SHA256_HMAC`
- `RTE_CRYPTO_AUTH_SHA512_HMAC`
- `RTE_CRYPTO_AUTH_AES_XCBC_MAC`

6.2.2 Limitations

- Chained mbufs are not supported.
- Hash only is not supported.
- Cipher only is not supported.
- Only in-place is currently supported (destination address is the same as source address).
- Only supports the session-oriented API implementation (session-less APIs are not supported).

- Not performance tuned.

6.2.3 Installation

To use the DPDK QAT PMD an SRIOV-enabled QAT kernel driver is required. The VF devices exposed by this driver will be used by QAT PMD.

If you are running on kernel 4.3 or greater, see instructions for *Installation using kernel.org driver* below. If you are on a kernel earlier than 4.3, see *Installation using 01.org QAT driver*.

6.2.4 Installation using 01.org QAT driver

Download the latest QuickAssist Technology Driver from 01.org Consult the *Getting Started Guide* at the same URL for further information.

The steps below assume you are:

- Building on a platform with one `DH895xCC` device.
- Using package `qatmux.l.2.3.0-34.tgz`.
- On Fedora21 kernel `3.17.4-301.fc21.x86_64`.

In the BIOS ensure that SRIOV is enabled and VT-d is disabled.

Uninstall any existing QAT driver, for example by running:

- `./installer.sh uninstall` in the directory where originally installed.
- or `rmmod qat_dh895xcc; rmmod intel_qat`.

Build and install the SRIOV-enabled QAT driver:

```
mkdir /QAT
cd /QAT
# copy qatmux.l.2.3.0-34.tgz to this location
tar zxof qatmux.l.2.3.0-34.tgz

export ICP_WITHOUT_IOMMU=1
./installer.sh install QAT1.6 host
```

You can use `cat /proc/icp_dh895xcc_dev0/version` to confirm the driver is correctly installed. You can use `lspci -d:443` to confirm the bdf of the 32 VF devices are available per `DH895xCC` device.

To complete the installation - follow instructions in *Binding the available VFs to the DPDK UIO driver*.

Note: If using a later kernel and the build fails with an error relating to `strict_stroul` not being available apply the following patch:

```
/QAT/QAT1.6/quickassist/utilities/downloader/Target_CoreLibs/uclo/include/linux/uclo_platform.h
+ #if LINUX_VERSION_CODE >= KERNEL_VERSION(3,18,5)
+ #define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; if (kstrtoul((str), (base), (num))) p
+ #else
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,38)
#define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; if (strict_strtoull((str), (base), (num
#else
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,25)
#define STR_TO_64(str, base, num, endPtr) {endPtr=NULL; strict_strtoll((str), (base), (num));}
#else
```

```
#define STR_TO_64(str, base, num, endPtr)                                \
     do {                                                               \
           if (str[0] == '-')                                           \
           {                                                            \
                *(num) = -(simple_strtoull((str+1), &(endPtr), (base))); \
           }else {                                                      \
                *(num) = simple_strtoull((str), &(endPtr), (base));      \
           }                                                            \
     } while(0)
+ #endif
#endif
#endif
```

If the build fails due to missing header files you may need to do following:

- `sudo yum install zlib-devel`
- `sudo yum install openssl-devel`

If the build or install fails due to mismatching kernel sources you may need to do the following:

- `sudo yum install kernel-headers-`uname -r``
- `sudo yum install kernel-src-`uname -r``
- `sudo yum install kernel-devel-`uname -r``

6.2.5 Installation using kernel.org driver

Assuming you are running on at least a 4.3 kernel, you can use the stock kernel.org QAT driver to start the QAT hardware.

The steps below assume you are:

- Running DPDK on a platform with one `DH895xCC` device.
- On a kernel at least version 4.3.

In BIOS ensure that SRIOV is enabled and VT-d is disabled.

Ensure the QAT driver is loaded on your system, by executing:

```
lsmod | grep qat
```

You should see the following output:

```
qat_dh895xcc            5626  0
intel_qat              82336  1 qat_dh895xcc
```

Next, you need to expose the VFs using the sysfs file system.

First find the bdf of the DH895xCC device:

```
lspci -d : 435
```

You should see output similar to:

```
03:00.0 Co-processor: Intel Corporation Coleto Creek PCIe Endpoint
```

Using the sysfs, enable the VFs:

```
echo 32 > /sys/bus/pci/drivers/dh895xcc/0000\:03\:00.0/sriov_numvfs
```

If you get an error, it's likely you're using a QAT kernel driver earlier than kernel 4.3.

To verify that the VFs are available for use - use `lspci -d:443` to confirm the bdf of the 32 VF devices are available per `DH895xCC` device.

To complete the installation - follow instructions in *Binding the available VFs to the DPDK UIO driver*.

6.2.6 Binding the available VFs to the DPDK UIO driver

The unbind command below assumes `bdfs` of `03:01.00-03:04.07`, if yours are different adjust the unbind command below:

```
cd $RTE_SDK
modprobe uio
insmod ./build/kmod/igb_uio.ko

for device in $(seq 1 4); do \
    for fn in $(seq 0 7); do \
        echo -n 0000:03:0${device}.${fn} > \
        /sys/bus/pci/devices/0000\:03\:0${device}.${fn}/driver/unbind; \
    done; \
done

echo "8086 0443" > /sys/bus/pci/drivers/igb_uio/new_id
```

You can use `lspci -vvd:443` to confirm that all devices are now in use by igb_uio kernel driver.

CHAPTER 7

Sample Applications User Guide

7.1 Introduction

This document describes the sample applications that are included in the Data Plane Development Kit (DPDK). Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

7.1.1 Documentation Roadmap

The following is a list of DPDK documents in suggested reading order:

- **Release Notes** : Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guides** : Describes how to install and configure the DPDK software for your operating system; designed to get users up and running quickly with the software.
- **Programmer's Guide:** Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linuxapp) environment.
  - The content of the DPDK, the build system (including the commands that can be used in the root DPDK Makefile to build the development kit and an application) and guidelines for porting an application.
  - Optimizations used in the software and those that should be considered for new development

A glossary of terms is also provided.

- **API Reference** : Provides detailed information about DPDK functions, data structures and other programming constructs.
- **Sample Applications User Guide** : Describes a set of sample applications. Each chapter describes a sample application that showcases specific functionality and provides instructions on how to compile, run and use the sample application.

7.2 Command Line Sample Application

This chapter describes the Command Line sample application that is part of the Data Plane Development Kit (DPDK).

7.2.1 Overview

The Command Line sample application is a simple application that demonstrates the use of the command line interface in the DPDK. This application is a readline-like interface that can be used to debug a DPDK application, in a Linux* application environment.

Note: The rte_cmdline library should not be used in production code since it is not validated to the same standard as other Intel® DPDK libraries. See also the “rte_cmdline library should not be used in production code due to limited testing” item in the “Known Issues” section of the Release Notes.

The Command Line sample application supports some of the features of the GNU readline library such as, completion, cut/paste and some other special bindings that make configuration and debug faster and easier.

The application shows how the rte_cmdline application can be extended to handle a list of objects. There are three simple commands:

- add obj_name IP: Add a new object with an IP/IPv6 address associated to it.
- del obj_name: Delete the specified object.
- show obj_name: Show the IP associated with the specified object.

Note: To terminate the application, use **Ctrl-d**.

7.2.2 Compiling the Application

1. Go to example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/cmdline
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   Refer to the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.2.3 Running the Application

To run the application in linuxapp environment, issue the following command:

```
$ ./build/cmdline -c f -n 4
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.2.4 Explanation

The following sections provide some explanation of the code.

EAL Initialization and cmdline Start

The first task is the initialization of the Environment Abstraction Layer (EAL). This is achieved as follows:

```
int main(int argc, char **argv)
{
    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic("Cannot init EAL\n");
```

Then, a new command line object is created and started to interact with the user through the console:

```
cl = cmdline_stdin_new(main_ctx, "example> ");
cmdline_interact(cl);
cmdline_stdin_exit(cl);
```

The cmd line_interact() function returns when the user types **Ctrl-d** and in this case, the application exits.

Defining a cmdline Context

A cmdline context is a list of commands that are listed in a NULL-terminated table, for example:

```
cmdline_parse_ctx_t main_ctx[] = {
    (cmdline_parse_inst_t *) &cmd_obj_del_show,
    (cmdline_parse_inst_t *) &cmd_obj_add,
    (cmdline_parse_inst_t *) &cmd_help,
     NULL,
};
```

Each command (of type cmdline_parse_inst_t) is defined statically. It contains a pointer to a callback function that is executed when the command is parsed, an opaque pointer, a help string and a list of tokens in a NULL-terminated table.

The rte_cmdline application provides a list of pre-defined token types:

- String Token: Match a static string, a list of static strings or any string.
- Number Token: Match a number that can be signed or unsigned, from 8-bit to 32-bit.
- IP Address Token: Match an IPv4 or IPv6 address or network.
- Ethernet* Address Token: Match a MAC address.

In this example, a new token type obj_list is defined and implemented in the parse_obj_list.c and parse_obj_list.h files.

For example, the cmd_obj_del_show command is defined as shown below:

```
struct cmd_obj_add_result {
    cmdline_fixed_string_t action;
    cmdline_fixed_string_t name;
    struct object *obj;
};

static void cmd_obj_del_show_parsed(void *parsed_result, struct cmdline *cl, attribute ((unusec
{
   /* ... */
}

cmdline_parse_token_string_t cmd_obj_action = TOKEN_STRING_INITIALIZER(struct cmd_obj_del_show_

parse_token_obj_list_t cmd_obj_obj = TOKEN_OBJ_LIST_INITIALIZER(struct cmd_obj_del_show_result,

cmdline_parse_inst_t cmd_obj_del_show = {
    .f = cmd_obj_del_show_parsed, /* function to call */
    .data = NULL,  /* 2nd arg of func */
    .help_str = "Show/del an object",
    .tokens = { /* token list, NULL terminated */
        (void *)&cmd_obj_action,
        (void *)&cmd_obj_obj,
         NULL,
    },
};
```

This command is composed of two tokens:

- The first token is a string token that can be show or del.
- The second token is an object that was previously added using the add command in the global_obj_list variable.

Once the command is parsed, the rte_cmdline application fills a cmd_obj_del_show_result structure. A pointer to this structure is given as an argument to the callback function and can be used in the body of this function.

7.3 Ethtool Sample Application

The Ethtool sample application shows an implementation of an ethtool-like API and provides a console environment that allows its use to query and change Ethernet card parameters. The sample is based upon a simple L2 frame reflector.

7.3.1 Compiling the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SD}/examples/ethtool
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

```
make
```

7.3.2 Running the Application

The application requires an available core for each port, plus one. The only available options are the standard ones for the EAL:

```
./ethtool-app/ethtool-app/${RTE_TARGET}/ethtool [EAL options]
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.3.3 Using the application

The application is console-driven using the cmdline DPDK interface:

```
EthApp>
```

From this interface the available commands and descriptions of what they do as as follows:

- `drvinfo`: Print driver info
- `eeprom`: Dump EEPROM to file
- `link`: Print port link states
- `macaddr`: Gets/sets MAC address
- `mtu`: Set NIC MTU
- `open`: Open port
- `pause`: Get/set port pause state
- `portstats`: Print port statistics
- `regs`: Dump port register(s) to file
- `ringparam`: Get/set ring parameters
- `rxmode`: Toggle port Rx mode
- `stop`: Stop port
- `validate`: Check that given MAC address is valid unicast address
- `vlan`: Add/remove VLAN id
- `quit`: Exit program

7.3.4 Explanation

The sample program has two parts: A background *packet reflector* that runs on a slave core, and a foreground *Ethtool Shell* that runs on the master core. These are described below.

Packet Reflector

The background packet reflector is intended to demonstrate basic packet processing on NIC ports controlled by the Ethtool shim. Each incoming MAC frame is rewritten so that it is returned to the sender, using the port in question's own MAC address as the source address, and is then sent out on the same port.

Ethtool Shell

The foreground part of the Ethtool sample is a console-based interface that accepts commands as described in *using the application*. Individual call-back functions handle the detail associated with each command, which make use of the functions defined in the *Ethtool interface* to the DPDK functions.

7.3.5 Ethtool interface

The Ethtool interface is built as a separate library, and implements the following functions:

- `rte_ethtool_get_drvinfo()`
- `rte_ethtool_get_regs_len()`
- `rte_ethtool_get_regs()`
- `rte_ethtool_get_link()`
- `rte_ethtool_get_eeprom_len()`
- `rte_ethtool_get_eeprom()`
- `rte_ethtool_set_eeprom()`
- `rte_ethtool_get_pauseparam()`
- `rte_ethtool_set_pauseparam()`
- `rte_ethtool_net_open()`
- `rte_ethtool_net_stop()`
- `rte_ethtool_net_get_mac_addr()`
- `rte_ethtool_net_set_mac_addr()`
- `rte_ethtool_net_validate_addr()`
- `rte_ethtool_net_change_mtu()`
- `rte_ethtool_net_get_stats64()`
- `rte_ethtool_net_vlan_rx_add_vid()`
- `rte_ethtool_net_vlan_rx_kill_vid()`
- `rte_ethtool_net_set_rx_mode()`
- `rte_ethtool_get_ringparam()`
- `rte_ethtool_set_ringparam()`

7.4 Exception Path Sample Application

The Exception Path sample application is a simple example that demonstrates the use of the DPDK to set up an exception path for packets to go through the Linux* kernel. This is done by using virtual TAP network interfaces. These can be read from and written to by the DPDK application and appear to the kernel as a standard network interface.

7.4.1 Overview

The application creates two threads for each NIC port being used. One thread reads from the port and writes the data unmodified to a thread-specific TAP interface. The second thread reads from a TAP interface and writes the data unmodified to the NIC port.

The packet flow through the exception path application is as shown in the following figure.

Fig. 7.1: Packet Flow

To make throughput measurements, kernel bridges must be setup to forward data between the bridges appropriately.

7.4.2 Compiling the Application

1. Go to example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/exception_path
```

2. Set the target (a default target will be used if not specified). For example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

This application is intended as a linuxapp only. See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

1. Build the application:

```
make
```

7.4.3 Running the Application

The application requires a number of command line options:

```
.build/exception_path [EAL options] -- -p PORTMASK -i IN_CORES -o OUT_CORES
```

where:

- -p PORTMASK: A hex bitmask of ports to use
- -i IN_CORES: A hex bitmask of cores which read from NIC
- -o OUT_CORES: A hex bitmask of cores which write to NIC

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The number of bits set in each bitmask must be the same. The coremask -c parameter of the EAL options should include IN_CORES and OUT_CORES. The same bit must not be set in IN_CORES and OUT_CORES. The affinities between ports and cores are set beginning with the least significant bit of each mask, that is, the port represented by the lowest bit in PORTMASK is read from by the core represented by the lowest bit in IN_CORES, and written to by the core represented by the lowest bit in OUT_CORES.

For example to run the application with two ports and four cores:

```
./build/exception_path -c f -n 4 -- -p 3 -i 3 -o c
```

Getting Statistics

While the application is running, statistics on packets sent and received can be displayed by sending the SIGUSR1 signal to the application from another terminal:

```
killall -USR1 exception_path
```

The statistics can be reset by sending a SIGUSR2 signal in a similar way.

7.4.4 Explanation

The following sections provide some explanation of the code.

Initialization

Setup of the mbuf pool, driver and queues is similar to the setup done in the L2 Forwarding sample application (see Chapter 9 "L2 forwarding Sample Application (in Real and Virtualized Environments" for details). In addition, the TAP interfaces must also be created. A TAP interface is created for each lcore that is being used. The code for creating the TAP interface is as follows:

```
/*
 *   Create a tap network interface, or use existing one with same name.
 *   If name[0]='\0' then a name is automatically assigned and returned in name.
 */

static int tap_create(char *name)
{
    struct ifreq ifr;
    int fd, ret;

    fd = open("/dev/net/tun", O_RDWR);
    if (fd < 0)
        return fd;

    memset(&ifr, 0, sizeof(ifr));

    /* TAP device without packet information */

    ifr.ifr_flags = IFF_TAP | IFF_NO_PI;
    if (name && *name)
        rte_snprinf(ifr.ifr_name, IFNAMSIZ, name);

    ret = ioctl(fd, TUNSETIFF, (void *) &ifr);

    if (ret < 0) {
        close(fd);
        return ret;

    }

    if (name)
        snprintf(name, IFNAMSIZ, ifr.ifr_name);

    return fd;
}
```

The other step in the initialization process that is unique to this sample application is the association of each port with two cores:

- One core to read from the port and write to a TAP interface
- A second core to read from a TAP interface and write to the port

This is done using an array called port_ids[], which is indexed by the lcore IDs. The population of this array is shown below:

```
tx_port = 0;
rx_port = 0;

RTE_LCORE_FOREACH(i) {
    if (input_cores_mask & (1ULL << i)) {
        /* Skip ports that are not enabled */
        while ((ports_mask & (1 << rx_port)) == 0) {
            rx_port++;
            if (rx_port > (sizeof(ports_mask) * 8))
                goto fail; /* not enough ports */
        }
        port_ids[i] = rx_port++;
    } else if (output_cores_mask & (1ULL << i)) {
        /* Skip ports that are not enabled */
        while ((ports_mask & (1 << tx_port)) == 0) {
            tx_port++;
            if (tx_port > (sizeof(ports_mask) * 8))
               goto fail; /* not enough ports */
        }
        port_ids[i] = tx_port++;
    }
}
```

Packet Forwarding

After the initialization steps are complete, the main_loop() function is run on each lcore. This function first checks the lcore_id against the user provided input_cores_mask and output_cores_mask to see if this core is reading from or writing to a TAP interface.

For the case that reads from a NIC port, the packet reception is the same as in the L2 Forwarding sample application (see Section 9.4.6, “Receive, Process and Transmit Packets”). The packet transmission is done by calling write() with the file descriptor of the appropriate TAP interface and then explicitly freeing the mbuf back to the pool.

```
/* Loop forever reading from NIC and writing to tap */

for (;;) {
    struct rte_mbuf *pkts_burst[PKT_BURST_SZ];
    unsigned i;
```

```
        const unsigned nb_rx = rte_eth_rx_burst(port_ids[lcore_id], 0, pkts_burst, PKT_BURST_SZ);

        lcore_stats[lcore_id].rx += nb_rx;

        for (i = 0; likely(i < nb_rx); i++) {
            struct rte_mbuf *m = pkts_burst[i];
            int ret = write(tap_fd, rte_pktmbuf_mtod(m, void*),

            rte_pktmbuf_data_len(m));
            rte_pktmbuf_free(m);
            if (unlikely(ret<0))
                lcore_stats[lcore_id].dropped++;
            else
                lcore_stats[lcore_id].tx++;
        }
    }
```

For the other case that reads from a TAP interface and writes to a NIC port, packets are retrieved by doing a read() from the file descriptor of the appropriate TAP interface. This fills in the data into the mbuf, then other fields are set manually. The packet can then be transmitted as normal.

```
/* Loop forever reading from tap and writing to NIC */

for (;;) {
    int ret;
    struct rte_mbuf *m = rte_pktmbuf_alloc(pktmbuf_pool);

    if (m == NULL)
        continue;

    ret = read(tap_fd, m->pkt.data, MAX_PACKET_SZ); lcore_stats[lcore_id].rx++;
    if (unlikely(ret < 0)) {
        FATAL_ERROR("Reading from %s interface failed", tap_name);
    }

    m->pkt.nb_segs = 1;
    m->pkt.next = NULL;
    m->pkt.data_len = (uint16_t)ret;

    ret = rte_eth_tx_burst(port_ids[lcore_id], 0, &m, 1);
    if (unlikely(ret < 1)) {
        rte_pktmuf_free(m);
        lcore_stats[lcore_id].dropped++;
    }
    else {
        lcore_stats[lcore_id].tx++;
    }
}
```

To set up loops for measuring throughput, TAP interfaces can be connected using bridging. The steps to do this are described in the section that follows.

Managing TAP Interfaces and Bridges

The Exception Path sample application creates TAP interfaces with names of the format tap_dpdk_nn, where nn is the lcore ID. These TAP interfaces need to be configured for use:

```
ifconfig tap_dpdk_00 up
```

To set up a bridge between two interfaces so that packets sent to one interface can be read from another, use the brctl tool:

```
brctl addbr "br0"
brctl addif br0 tap_dpdk_00
brctl addif br0 tap_dpdk_03
ifconfig br0 up
```

The TAP interfaces created by this application exist only when the application is running, so the steps above need to be repeated each time the application is run. To avoid this, persistent TAP interfaces can be created using openvpn:

```
openvpn --mktun --dev tap_dpdk_00
```

If this method is used, then the steps above have to be done only once and the same TAP interfaces can be reused each time the application is run. To remove bridges and persistent TAP interfaces, the following commands are used:

```
ifconfig br0 down
brctl delbr br0
openvpn --rmtun --dev tap_dpdk_00
```

7.5 Hello World Sample Application

The Hello World sample application is an example of the simplest DPDK application that can be written. The application simply prints an “helloworld” message on every enabled lcore.

7.5.1 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/helloworld
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started* Guide for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.5.2 Running the Application

To run the example in a linuxapp environment:

```
$ ./build/helloworld -c f -n 4
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.5.3 Explanation

The following sections provide some explanation of code.

EAL Initialization

The first task is to initialize the Environment Abstraction Layer (EAL). This is done in the main() function using the following code:

```
int

main(int argc, char **argv)

{
    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic("Cannot init EAL\n");
```

This call finishes the initialization process that was started before main() is called (in case of a Linuxapp environment). The argc and argv arguments are provided to the rte_eal_init() function. The value returned is the number of parsed arguments.

Starting Application Unit Lcores

Once the EAL is initialized, the application is ready to launch a function on an lcore. In this example, lcore_hello() is called on every available lcore. The following is the definition of the function:

```
static int
lcore_hello( attribute ((unused)) void *arg)
{
    unsigned lcore_id;

    lcore_id = rte_lcore_id();
    printf("hello from core %u\n", lcore_id);
    return 0;
}
```

The code that launches the function on each lcore is as follows:

```
/* call lcore_hello() on every slave lcore */

RTE_LCORE_FOREACH_SLAVE(lcore_id) {
   rte_eal_remote_launch(lcore_hello, NULL, lcore_id);
}

/* call it on master lcore too */

lcore_hello(NULL);
```

The following code is equivalent and simpler:

```
rte_eal_mp_remote_launch(lcore_hello, NULL, CALL_MASTER);
```

Refer to the *DPDK API Reference* for detailed information on the rte_eal_mp_remote_launch() function.

7.6 Basic Forwarding Sample Application

The Basic Forwarding sample application is a simple *skeleton* example of a forwarding application.

It is intended as a demonstration of the basic components of a DPDK forwarding application. For more detailed implementations see the L2 and L3 forwarding sample applications.

7.6.1 Compiling the Application

To compile the application export the path to the DPDK source tree and go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk

cd ${RTE_SDK}/examples/skeleton
```

Set the target, for example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started* Guide for possible `RTE_TARGET` values.

Build the application as follows:

```
make
```

7.6.2 Running the Application

To run the example in a `linuxapp` environment:

```
./build/basicfwd -c 2 -n 4
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.6.3 Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_mempool_create("MBUF_POOL",
                               NUM_MBUFS * nb_ports,
                               MBUF_SIZE,
                               MBUF_CACHE_SIZE,
                               sizeof(struct rte_pktmbuf_pool_private),
                               rte_pktmbuf_pool_init, NULL,
                               rte_pktmbuf_init,      NULL,
                               rte_socket_id(),
                               0);
```

Mbufs are the packet buffer structure used by DPDK. They are explained in detail in the "Mbuf Library" section of the *DPDK Programmer's Guide*.

The `main()` function also initializes all the ports using the user defined `port_init()` function which is explained in the next section:

```
for (portid = 0; portid < nb_ports; portid++) {
    if (port_init(portid, mbuf_pool) != 0) {
        rte_exit(EXIT_FAILURE,
                 "Cannot init port %" PRIu8 "\n", portid);
    }
}
```

Once the initialization is complete, the application is ready to launch a function on an lcore. In this example `lcore_main()` is called on a single lcore.

```
lcore_main();
```

The `lcore_main()` function is explained below.

The Port Initialization Function

The main functional part of the port initialization used in the Basic Forwarding application is shown below:

```
static inline int
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    struct ether_addr addr;
    int retval;
    uint16_t q;

    if (port >= rte_eth_dev_count())
        return -1;

    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
        return retval;

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0)
            return retval;
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }
    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    /* Enable RX in promiscuous mode for the Ethernet device. */
    rte_eth_promiscuous_enable(port);

    return 0;
}
```

The Ethernet ports are configured with default settings using the `rte_eth_dev_configure()` function and the `port_conf_default` struct:

```
static const struct rte_eth_conf port_conf_default = {
    .rxmode = { .max_rx_pkt_len = ETHER_MAX_LEN }
};
```

For this example the ports are set up with 1 RX and 1 TX queue using the `rte_eth_rx_queue_setup()` and `rte_eth_tx_queue_setup()` functions.

The Ethernet port is then started:

```
retval  = rte_eth_dev_start(port);
```

Finally the RX port is set in promiscuous mode:

```
rte_eth_promiscuous_enable(port);
```

The Lcores Main

As we saw above the `main()` function calls an application function on the available lcores. For the Basic Forwarding application the lcore function looks like the following:

```
static __attribute__((noreturn)) void
lcore_main(void)
{
    const uint8_t nb_ports = rte_eth_dev_count();
    uint8_t port;

    /*
     * Check that the port is on the same NUMA node as the polling thread
     * for best performance.
     */
    for (port = 0; port < nb_ports; port++)
        if (rte_eth_dev_socket_id(port) > 0 &&
                rte_eth_dev_socket_id(port) !=
                        (int)rte_socket_id())
            printf("WARNING, port %u is on remote NUMA node to "
                    "polling thread.\n\tPerformance will "
                    "not be optimal.\n", port);

    printf("\nCore %u forwarding packets. [Ctrl+C to quit]\n",
            rte_lcore_id());

    /* Run until the application is quit or killed. */
    for (;;) {
        /*
         * Receive packets on a port and forward them on the paired
         * port. The mapping is 0 -> 1, 1 -> 0, 2 -> 3, 3 -> 2, etc.
         */
        for (port = 0; port < nb_ports; port++) {
            /* Get burst of RX packets, from first port of pair. */
            struct rte_mbuf *bufs[BURST_SIZE];
            const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                    bufs, BURST_SIZE);

            if (unlikely(nb_rx == 0))
                continue;

            /* Send burst of TX packets, to second port of pair. */
            const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                    bufs, nb_rx);

            /* Free any unsent packets. */
            if (unlikely(nb_tx < nb_rx)) {
                uint16_t buf;
                for (buf = nb_tx; buf < nb_rx; buf++)
                    rte_pktmbuf_free(bufs[buf]);
            }
        }
    }
}
```

The main work of the application is done within the loop:

```
for (;;) {
    for (port = 0; port < nb_ports; port++) {

        /* Get burst of RX packets, from first port of pair. */
        struct rte_mbuf *bufs[BURST_SIZE];
        const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                bufs, BURST_SIZE);

        if (unlikely(nb_rx == 0))
            continue;

        /* Send burst of TX packets, to second port of pair. */
        const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                bufs, nb_rx);

        /* Free any unsent packets. */
        if (unlikely(nb_tx < nb_rx)) {
            uint16_t buf;
            for (buf = nb_tx; buf < nb_rx; buf++)
                rte_pktmbuf_free(bufs[buf]);
        }
    }
}
```

Packets are received in bursts on the RX ports and transmitted in bursts on the TX ports. The ports are grouped in pairs with a simple mapping scheme using the an XOR on the port number:

```
0 -> 1
1 -> 0

2 -> 3
3 -> 2

etc.
```

The `rte_eth_tx_burst()` function frees the memory buffers of packets that are transmitted. If packets fail to transmit, `(nb_tx < nb_rx)`, then they must be freed explicitly using `rte_pktmbuf_free()`.

The forwarding loop can be interrupted and the application closed using `Ctrl-C`.

7.7 RX/TX Callbacks Sample Application

The RX/TX Callbacks sample application is a packet forwarding application that demonstrates the use of user defined callbacks on received and transmitted packets. The application performs a simple latency check, using callbacks, to determine the time packets spend within the application.

In the sample application a user defined callback is applied to all received packets to add a timestamp. A separate callback is applied to all packets prior to transmission to calculate the elapsed time, in CPU cycles.

7.7.1 Compiling the Application

To compile the application export the path to the DPDK source tree and go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk

cd ${RTE_SDK}/examples/rxtx_callbacks
```

Set the target, for example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started* Guide for possible `RTE_TARGET` values.

The callbacks feature requires that the `CONFIG_RTE_ETHDEV_RXTX_CALLBACKS` setting is on in the `config/common_` config file that applies to the target. This is generally on by default:

```
CONFIG_RTE_ETHDEV_RXTX_CALLBACKS=y
```

Build the application as follows:

```
make
```

7.7.2 Running the Application

To run the example in a `linuxapp` environment:

```
./build/rxtx_callbacks -c 2 -n 4
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.7.3 Explanation

The `rxtx_callbacks` application is mainly a simple forwarding application based on the Basic Forwarding Sample Application. See that section of the documentation for more details of the forwarding part of the application.

The sections below explain the additional RX/TX callback code.

The Main Function

The `main()` function performs the application initialization and calls the execution threads for each lcore. This function is effectively identical to the `main()` function explained in Basic Forwarding Sample Application.

The `lcore_main()` function is also identical.

The main difference is in the user defined `port_init()` function where the callbacks are added. This is explained in the next section:

The Port Initialization Function

The main functional part of the port initialization is shown below with comments:

```
static inline int
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    struct ether_addr addr;
    int retval;
    uint16_t q;

    if (port >= rte_eth_dev_count())
        return -1;

    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
        return retval;

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0)
            return retval;
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }

    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    /* Enable RX in promiscuous mode for the Ethernet device. */
    rte_eth_promiscuous_enable(port);


    /* Add the callbacks for RX and TX.*/
    rte_eth_add_rx_callback(port, 0, add_timestamps, NULL);
    rte_eth_add_tx_callback(port, 0, calc_latency, NULL);
    return 0;
}
```

The RX and TX callbacks are added to the ports/queues as function pointers:

```
rte_eth_add_rx_callback(port, 0, add_timestamps, NULL);
rte_eth_add_tx_callback(port, 0, calc_latency,   NULL);
```

More than one callback can be added and additional information can be passed to callback function pointers as a `void*`. In the examples above `NULL` is used.

The `add_timestamps()` and `calc_latency()` functions are explained below.

The add_timestamps() Callback

The `add_timestamps()` callback is added to the RX port and is applied to all packets received:

```
static uint16_t
add_timestamps(uint8_t port __rte_unused, uint16_t qidx __rte_unused,
        struct rte_mbuf **pkts, uint16_t nb_pkts, void *_ __rte_unused)
{
    unsigned i;
    uint64_t now = rte_rdtsc();

    for (i = 0; i < nb_pkts; i++)
        pkts[i]->udata64 = now;

    return nb_pkts;
}
```

The DPDK function `rte_rdtsc()` is used to add a cycle count timestamp to each packet (see the *cycles* section of the *DPDK API Documentation* for details).

The calc_latency() Callback

The `calc_latency()` callback is added to the TX port and is applied to all packets prior to transmission:

```
static uint16_t
calc_latency(uint8_t port __rte_unused, uint16_t qidx __rte_unused,
        struct rte_mbuf **pkts, uint16_t nb_pkts, void *_ __rte_unused)
{
    uint64_t cycles = 0;
    uint64_t now = rte_rdtsc();
    unsigned i;

    for (i = 0; i < nb_pkts; i++)
        cycles += now - pkts[i]->udata64;

    latency_numbers.total_cycles += cycles;
    latency_numbers.total_pkts   += nb_pkts;

    if (latency_numbers.total_pkts > (100 * 1000 * 1000ULL)) {
        printf("Latency = %"PRIu64" cycles\n",
                latency_numbers.total_cycles / latency_numbers.total_pkts);

        latency_numbers.total_cycles = latency_numbers.total_pkts = 0;
    }
```

```
        return nb_pkts;
    }
```

The `calc_latency()` function accumulates the total number of packets and the total number of cycles used. Once more than 100 million packets have been transmitted the average cycle count per packet is printed out and the counters are reset.

7.8 IP Fragmentation Sample Application

The IPv4 Fragmentation application is a simple example of packet processing using the Data Plane Development Kit (DPDK). The application does L3 forwarding with IPv4 and IPv6 packet fragmentation.

7.8.1 Overview

The application demonstrates the use of zero-copy buffers for packet fragmentation. The initialization and run-time paths are very similar to those of the L2 forwarding application (see Chapter 9 “L2 Forwarding Simple Application (in Real and Virtualized Environments)” for more information). This guide highlights the differences between the two applications.

There are three key differences from the L2 Forwarding sample application:

- The first difference is that the IP Fragmentation sample application makes use of indirect buffers.
- The second difference is that the forwarding decision is taken based on information read from the input packet’s IP header.
- The third difference is that the application differentiates between IP and non-IP traffic by means of offload flags.

The Longest Prefix Match (LPM for IPv4, LPM6 for IPv6) table is used to store/lookup an outgoing port number, associated with that IP address. Any unmatched packets are forwarded to the originating port.

By default, input frame sizes up to 9.5 KB are supported. Before forwarding, the input IP packet is fragmented to fit into the “standard” Ethernet* v2 MTU (1500 bytes).

7.8.2 Building the Application

To build the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/ip_fragmentation
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

1. Build the application:

   ```
   make
   ```

7.8.3 Running the Application

The LPM object is created and loaded with the pre-configured entries read from global l3fwd_ipv4_route_array and l3fwd_ipv6_route_array tables. For each input packet, the packet forwarding decision (that is, the identification of the output interface for the packet) is taken as a result of LPM lookup. If the IP packet size is greater than default output MTU, then the input packet is fragmented and several fragments are sent via the output interface.

Application usage:

```
./build/ip_fragmentation [EAL options] -- -p PORTMASK [-q NQ]
```

where:

- -p PORTMASK is a hexadecimal bitmask of ports to configure
- -q NQ is the number of queue (=ports) per lcore (the default is 1)

To run the example in linuxapp environment with 2 lcores (2,4) over 2 ports(0,2) with 1 RX queue per lcore:

```
./build/ip_fragmentation -c 0x14 -n 3 -- -p 5
EAL: coremask set to 14
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 1
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 1
EAL: Detected lcore 4 on socket 0
...

Initializing port 0 on lcore 2... Address:00:1B:21:76:FA:2C, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 1
Initializing port 2 on lcore 4... Address:00:1B:21:5C:FF:54, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 3IP_FRAG: Socket 0: adding route 100.10.0.0/16 (port 0)
IP_FRAG: Socket 0: adding route 100.20.0.0/16 (port 1)
...
IP_FRAG: Socket 0: adding route 0101:0101:0101:0101:0101:0101:0101:0101/48 (port 0)
IP_FRAG: Socket 0: adding route 0201:0101:0101:0101:0101:0101:0101:0101/48 (port 1)
...
IP_FRAG: entering main loop on lcore 4
IP_FRAG: -- lcoreid=4 portid=2
IP_FRAG: entering main loop on lcore 2
IP_FRAG: -- lcoreid=2 portid=0
```

To run the example in linuxapp environment with 1 lcore (4) over 2 ports(0,2) with 2 RX queues per lcore:

```
./build/ip_fragmentation -c 0x10 -n 3 -- -p 5 -q 2
```

To test the application, flows should be set up in the flow generator that match the values in the l3fwd_ipv4_route_array and/or l3fwd_ipv6_route_array table.

The default l3fwd_ipv4_route_array table is:

```
struct l3fwd_ipv4_route l3fwd_ipv4_route_array[] = {
    {IPv4(100, 10, 0, 0), 16, 0},
    {IPv4(100, 20, 0, 0), 16, 1},
    {IPv4(100, 30, 0, 0), 16, 2},
    {IPv4(100, 40, 0, 0), 16, 3},
    {IPv4(100, 50, 0, 0), 16, 4},
    {IPv4(100, 60, 0, 0), 16, 5},
    {IPv4(100, 70, 0, 0), 16, 6},
        {IPv4(100, 80, 0, 0), 16, 7},
};
```

The default l3fwd_ipv6_route_array table is:

```
struct l3fwd_ipv6_route l3fwd_ipv6_route_array[] = {
    {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 0},
    {{2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 1},
    {{3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 2},
    {{4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 3},
    {{5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 4},
    {{6, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 5},
    {{7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 6},
    {{8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 7},
};
```

For example, for the input IPv4 packet with destination address: 100.10.1.1 and packet length 9198 bytes, seven IPv4 packets will be sent out from port #0 to the destination address 100.10.1.1: six of those packets will have length 1500 bytes and one packet will have length 318 bytes. IP Fragmentation sample application provides basic NUMA support in that all the memory structures are allocated on all sockets that have active lcores on them.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.9 IPv4 Multicast Sample Application

The IPv4 Multicast application is a simple example of packet processing using the Data Plane Development Kit (DPDK). The application performs L3 multicasting.

7.9.1 Overview

The application demonstrates the use of zero-copy buffers for packet forwarding. The initialization and run-time paths are very similar to those of the L2 forwarding application (see Chapter 9 "L2 Forwarding Sample Application (in Real and Virtualized Environments)" for details more information). This guide highlights the differences between the two applications. There are two key differences from the L2 Forwarding sample application:

- The IPv4 Multicast sample application makes use of indirect buffers.
- The forwarding decision is taken based on information read from the input packet's IPv4 header.

The lookup method is the Four-byte Key (FBK) hash-based method. The lookup table is composed of pairs of destination IPv4 address (the FBK) and a port mask associated with that IPv4 address.

For convenience and simplicity, this sample application does not take IANA-assigned multicast addresses into account, but instead equates the last four bytes of the multicast group (that is, the last four bytes of the destination IP address) with the mask of ports to multicast packets to. Also, the application does not consider the Ethernet addresses; it looks only at the IPv4 destination address for any given packet.

7.9.2 Building the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/ipv4_multicast
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

1. Build the application:

   ```
   make
   ```

Note: The compiled application is written to the build subdirectory. To have the application written to a different location, the O=/path/to/build/directory option may be specified in the make command.

7.9.3 Running the Application

The application has a number of command line options:

```
./build/ipv4_multicast [EAL options] -- -p PORTMASK [-q NQ]
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -q NQ: determines the number of queues per lcore

Note: Unlike the basic L2/L3 Forwarding sample applications, NUMA support is not provided in the IPv4 Multicast sample application.

Typically, to run the IPv4 Multicast sample application, issue the following command (as root):

```
./build/ipv4_multicast -c 0x00f -n 3 -- -p 0x3 -q 1
```

In this command:

- The -c option enables cores 0, 1, 2 and 3
- The -n option specifies 3 memory channels
- The -p option enables ports 0 and 1
- The -q option assigns 1 queue to each lcore

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.9.4 Explanation

The following sections provide some explanation of the code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the L2 Forwarding sample application (see Chapter 9 “L2 Forwarding Sample Application in Real and Virtualized Environments” for more information). The following sections describe aspects that are specific to the IPv4 Multicast sample application.

Memory Pool Initialization

The IPv4 Multicast sample application uses three memory pools. Two of the pools are for indirect buffers used for packet duplication purposes. Memory pools for indirect buffers are initialized differently from the memory pool for direct buffers:

```
packet_pool = rte_mempool_create("packet_pool", NB_PKT_MBUF, PKT_MBUF_SIZE, 32, sizeof(struct r
                                 rte_pktmbuf_pool_init, NULL, rte_pktmbuf_init, NULL, rte_socke

header_pool = rte_mempool_create("header_pool", NB_HDR_MBUF, HDR_MBUF_SIZE, 32, 0, NULL, NULL,
clone_pool = rte_mempool_create("clone_pool", NB_CLONE_MBUF,
CLONE_MBUF_SIZE, 32, 0, NULL, NULL, rte_pktmbuf_init, NULL, rte_socket_id(), 0);
```

The reason for this is because indirect buffers are not supposed to hold any packet data and therefore can be initialized with lower amount of reserved memory for each buffer.

Hash Initialization

The hash object is created and loaded with the pre-configured entries read from a global array:

```
static int

init_mcast_hash(void)
{
    uint32_t i;
    mcast_hash_params.socket_id = rte_socket_id();

    mcast_hash = rte_fbk_hash_create(&mcast_hash_params);
    if (mcast_hash == NULL){
        return -1;
    }

    for (i = 0; i < N_MCAST_GROUPS; i ++){
        if (rte_fbk_hash_add_key(mcast_hash, mcast_group_table[i].ip, mcast_group_table[i].port
                    return -1;
        }
    }
    return 0;
}
```

Forwarding

All forwarding is done inside the mcast_forward() function. Firstly, the Ethernet* header is removed from the packet and the IPv4 address is extracted from the IPv4 header:

```
/* Remove the Ethernet header from the input packet */

iphdr = (struct ipv4_hdr *)rte_pktmbuf_adj(m, sizeof(struct ether_hdr));
```

```
RTE_MBUF_ASSERT(iphdr != NULL);
dest_addr = rte_be_to_cpu_32(iphdr->dst_addr);
```

Then, the packet is checked to see if it has a multicast destination address and if the routing table has any ports assigned to the destination address:

```
if (!IS_IPV4_MCAST(dest_addr) ||
    (hash = rte_fbk_hash_lookup(mcast_hash, dest_addr)) <= 0 ||
    (port_mask = hash & enabled_port_mask) == 0) {
        rte_pktmbuf_free(m);
        return;
}
```

Then, the number of ports in the destination portmask is calculated with the help of the bitcnt() function:

```
/* Get number of bits set. */

static inline uint32_t bitcnt(uint32_t v)
{
    uint32_t n;

    for (n = 0; v != 0; v &= v - 1, n++)
        ;
    return (n);
}
```

This is done to determine which forwarding algorithm to use. This is explained in more detail in the next section.

Thereafter, a destination Ethernet address is constructed:

```
/* construct destination Ethernet address */

dst_eth_addr = ETHER_ADDR_FOR_IPV4_MCAST(dest_addr);
```

Since Ethernet addresses are also part of the multicast process, each outgoing packet carries the same destination Ethernet address. The destination Ethernet address is constructed from the lower 23 bits of the multicast group OR-ed with the Ethernet address 01:00:5e:00:00:00, as per RFC 1112:

```
#define ETHER_ADDR_FOR_IPV4_MCAST(x) \
    (rte_cpu_to_be_64(0x01005e000000ULL | ((x) & 0x7fffff)) >> 16)
```

Then, packets are dispatched to the destination ports according to the portmask associated with a multicast group:

```
for (port = 0; use_clone != port_mask; port_mask >>= 1, port++) {
    /* Prepare output packet and send it out. */

    if ((port_mask & 1) != 0) {
        if (likely ((mc = mcast_out_pkt(m, use_clone)) != NULL))
            mcast_send_pkt(mc, &dst_eth_addr.as_addr, qconf, port);
        else if (use_clone == 0)
             rte_pktmbuf_free(m);
    }
}
```

The actual packet transmission is done in the mcast_send_pkt() function:

```
static inline void mcast_send_pkt(struct rte_mbuf *pkt, struct ether_addr *dest_addr, struct lc
{
    struct ether_hdr *ethdr;
    uint16_t len;
```

```
        /* Construct Ethernet header. */

        ethdr = (struct ether_hdr *)rte_pktmbuf_prepend(pkt, (uint16_t) sizeof(*ethdr));

        RTE_MBUF_ASSERT(ethdr != NULL);

        ether_addr_copy(dest_addr, &ethdr->d_addr);
        ether_addr_copy(&ports_eth_addr[port], &ethdr->s_addr);
        ethdr->ether_type = rte_be_to_cpu_16(ETHER_TYPE_IPv4);

        /* Put new packet into the output queue */

        len = qconf->tx_mbufs[port].len;
        qconf->tx_mbufs[port].m_table[len] = pkt;
        qconf->tx_mbufs[port].len = ++len;

        /* Transmit packets */

        if (unlikely(MAX_PKT_BURST == len))
            send_burst(qconf, port);
    }
```

Buffer Cloning

This is the most important part of the application since it demonstrates the use of zero- copy buffer cloning. There are two approaches for creating the outgoing packet and although both are based on the data zero-copy idea, there are some differences in the detail.

The first approach creates a clone of the input packet, for example, walk though all segments of the input packet and for each of segment, create a new buffer and attach that new buffer to the segment (refer to rte_pktmbuf_clone() in the rte_mbuf library for more details). A new buffer is then allocated for the packet header and is prepended to the cloned buffer.

The second approach does not make a clone, it just increments the reference counter for all input packet segment, allocates a new buffer for the packet header and prepends it to the input packet.

Basically, the first approach reuses only the input packet's data, but creates its own copy of packet's metadata. The second approach reuses both input packet's data and metadata.

The advantage of first approach is that each outgoing packet has its own copy of the metadata, so we can safely modify the data pointer of the input packet. That allows us to skip creation if the output packet is for the last destination port and instead modify input packet's header in place. For example, for N destination ports, we need to invoke mcast_out_pkt() (N-1) times.

The advantage of the second approach is that there is less work to be done for each outgoing packet, that is, the "clone" operation is skipped completely. However, there is a price to pay. The input packet's metadata must remain intact, so for N destination ports, we need to invoke mcast_out_pkt() (N) times.

Therefore, for a small number of outgoing ports (and segments in the input packet), first approach is faster. As the number of outgoing ports (and/or input segments) grows, the second approach becomes more preferable.

Depending on the number of segments or the number of ports in the outgoing portmask, either the first (with cloning) or the second (without cloning) approach is taken:

```
use_clone = (port_num <= MCAST_CLONE_PORTS && m->pkt.nb_segs <= MCAST_CLONE_SEGS);
```

It is the mcast_out_pkt() function that performs the packet duplication (either with or without actually cloning the buffers):

```
static inline struct rte_mbuf *mcast_out_pkt(struct rte_mbuf *pkt, int use_clone)
{
    struct rte_mbuf *hdr;

    /* Create new mbuf for the header. */

    if (unlikely ((hdr = rte_pktmbuf_alloc(header_pool)) == NULL))
        return (NULL);

    /* If requested, then make a new clone packet. */

    if (use_clone != 0 && unlikely ((pkt = rte_pktmbuf_clone(pkt, clone_pool)) == NULL)) {
        rte_pktmbuf_free(hdr);
        return (NULL);
    }

    /* prepend new header */

    hdr->pkt.next = pkt;

    /* update header's fields */

    hdr->pkt.pkt_len = (uint16_t)(hdr->pkt.data_len + pkt->pkt.pkt_len);
    hdr->pkt.nb_segs = (uint8_t)(pkt->pkt.nb_segs + 1);

    /* copy metadata from source packet */

    hdr->pkt.in_port = pkt->pkt.in_port;
    hdr->pkt.vlan_macip = pkt->pkt.vlan_macip;
    hdr->pkt.hash = pkt->pkt.hash;
    hdr->ol_flags = pkt->ol_flags;
    rte_mbuf_sanity_check(hdr, RTE_MBUF_PKT, 1);

    return (hdr);
}
```

7.10 IP Reassembly Sample Application

The L3 Forwarding application is a simple example of packet processing using the DPDK. The application performs L3 forwarding with reassembly for fragmented IPv4 and IPv6 packets.

7.10.1 Overview

The application demonstrates the use of the DPDK libraries to implement packet forwarding with reassembly for IPv4 and IPv6 fragmented packets. The initialization and run- time paths are very similar to those of the L2 forwarding application (see Chapter 9 "L2 Forwarding Sample Application" for more information). The main difference from the L2 Forwarding sample application is that it reassembles fragmented IPv4 and IPv6 packets before forwarding. The maximum allowed size of reassembled packet is 9.5 KB.

There are two key differences from the L2 Forwarding sample application:

- The first difference is that the forwarding decision is taken based on information read from the input packet's IP header.

- The second difference is that the application differentiates between IP and non-IP traffic by means of offload flags.

7.10.2 The Longest Prefix Match (LPM for IPv4, LPM6 for IPv6) table is used to store/lookup an outgoing port number, associated with that IPv4 address. Any unmatched packets are forwarded to the originating port.Compiling the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/ip_reassembly
   ```

1. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

1. Build the application:

   ```
   make
   ```

7.10.3 Running the Application

The application has a number of command line options:

```
./build/ip_reassembly [EAL options] -- -p PORTMASK [-q NQ] [--maxflows=FLOWS>] [--flowttl=TTL[(
```

where:

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -q NQ: Number of RX queues per lcore
- –maxflows=FLOWS: determines maximum number of active fragmented flows (1-65535). Default value: 4096.
- –flowttl=TTL[(s|ms)]: determines maximum Time To Live for fragmented packet. If all fragments of the packet wouldn't appear within given time-out, then they are considered as invalid and will be dropped. Valid range is 1ms - 3600s. Default value: 1s.

To run the example in linuxapp environment with 2 lcores (2,4) over 2 ports(0,2) with 1 RX queue per lcore:

```
./build/ip_reassembly -c 0x14 -n 3 -- -p 5
EAL: coremask set to 14
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 1
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 1
EAL: Detected lcore 4 on socket 0
...

Initializing port 0 on lcore 2... Address:00:1B:21:76:FA:2C, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
Skipping disabled port 1
Initializing port 2 on lcore 4... Address:00:1B:21:5C:FF:54, rxq=0 txq=2,0 txq=4,1
done: Link Up - speed 10000 Mbps - full-duplex
```

```
Skipping disabled port 3IP_FRAG: Socket 0: adding route 100.10.0.0/16 (port 0)
IP_RSMBL: Socket 0: adding route 100.20.0.0/16 (port 1)
...

IP_RSMBL: Socket 0: adding route 0101:0101:0101:0101:0101:0101:0101:0101/48 (port 0)
IP_RSMBL: Socket 0: adding route 0201:0101:0101:0101:0101:0101:0101:0101/48 (port 1)
...

IP_RSMBL: entering main loop on lcore 4
IP_RSMBL: -- lcoreid=4 portid=2
IP_RSMBL: entering main loop on lcore 2
IP_RSMBL: -- lcoreid=2 portid=0
```

To run the example in linuxapp environment with 1 lcore (4) over 2 ports(0,2) with 2 RX queues per lcore:

```
./build/ip_reassembly -c 0x10 -n 3 -- -p 5 -q 2
```

To test the application, flows should be set up in the flow generator that match the values in the l3fwd_ipv4_route_array and/or l3fwd_ipv6_route_array table.

Please note that in order to test this application, the traffic generator should be generating valid fragmented IP packets. For IPv6, the only supported case is when no other extension headers other than fragment extension header are present in the packet.

The default l3fwd_ipv4_route_array table is:

```
struct l3fwd_ipv4_route l3fwd_ipv4_route_array[] = {
    {IPv4(100, 10, 0, 0), 16, 0},
    {IPv4(100, 20, 0, 0), 16, 1},
    {IPv4(100, 30, 0, 0), 16, 2},
    {IPv4(100, 40, 0, 0), 16, 3},
    {IPv4(100, 50, 0, 0), 16, 4},
    {IPv4(100, 60, 0, 0), 16, 5},
    {IPv4(100, 70, 0, 0), 16, 6},
    {IPv4(100, 80, 0, 0), 16, 7},
};
```

The default l3fwd_ipv6_route_array table is:

```
struct l3fwd_ipv6_route l3fwd_ipv6_route_array[] = {
    {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 0},
    {{2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 1},
    {{3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 2},
    {{4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 3},
    {{5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 4},
    {{6, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 5},
    {{7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 6},
    {{8, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 48, 7},
};
```

For example, for the fragmented input IPv4 packet with destination address: 100.10.1.1, a reassembled IPv4 packet be sent out from port #0 to the destination address 100.10.1.1 once all the fragments are collected.

7.10.4 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the L2 forwarding application (see Chapter 9 “L2 Forwarding Sample Application” for more information). The following sections describe aspects that are specific to the IP reassemble sample application.

IPv4 Fragment Table Initialization

This application uses the rte_ip_frag library. Please refer to Programmer's Guide for more detailed explanation of how to use this library. Fragment table maintains information about already received fragments of the packet. Each IP packet is uniquely identified by triple <Source IP address>, <Destination IP address>, <ID>. To avoid lock contention, each RX queue has its own Fragment Table, e.g. the application can't handle the situation when different fragments of the same packet arrive through different RX queues. Each table entry can hold information about packet consisting of up to RTE_LIBRTE_IP_FRAG_MAX_FRAGS fragments.

```
frag_cycles = (rte_get_tsc_hz() + MS_PER_S - 1) / MS_PER_S * max_flow_ttl;

if ((qconf->frag_tbl[queue] = rte_ip_frag_tbl_create(max_flow_num, IPV4_FRAG_TBL_BUCKET_ENTRIES
{
    RTE_LOG(ERR, IP_RSMBL, "ip_frag_tbl_create(%u) on " "lcore: %u for queue: %u failed\n",  ma
    return -1;
}
```

Mempools Initialization

The reassembly application demands a lot of mbuf's to be allocated. At any given time up to (2 * max_flow_num * RTE_LIBRTE_IP_FRAG_MAX_FRAGS * <maximum number of mbufs per packet>) can be stored inside Fragment Table waiting for remaining fragments. To keep mempool size under reasonable limits and to avoid situation when one RX queue can starve other queues, each RX queue uses its own mempool.

```
nb_mbuf = RTE_MAX(max_flow_num, 2UL * MAX_PKT_BURST) * RTE_LIBRTE_IP_FRAG_MAX_FRAGS;
nb_mbuf *= (port_conf.rxmode.max_rx_pkt_len + BUF_SIZE - 1) / BUF_SIZE;
nb_mbuf *= 2; /* ipv4 and ipv6 */
nb_mbuf += RTE_TEST_RX_DESC_DEFAULT + RTE_TEST_TX_DESC_DEFAULT;
nb_mbuf = RTE_MAX(nb_mbuf, (uint32_t)NB_MBUF);

snprintf(buf, sizeof(buf), "mbuf_pool_%u_%u", lcore, queue);

if ((rxq->pool = rte_mempool_create(buf, nb_mbuf, MBUF_SIZE, 0, sizeof(struct rte_pktmbuf_pool_
    rte_pktmbuf_init, NULL, socket, MEMPOOL_F_SP_PUT | MEMPOOL_F_SC_GET)) == NULL) {

        RTE_LOG(ERR, IP_RSMBL, "mempool_create(%s) failed", buf);
        return -1;
}
```

Packet Reassembly and Forwarding

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() function. If the packet is an IPv4 or IPv6 fragment, then it calls rte_ipv4_reassemble_packet() for IPv4 packets, or rte_ipv6_reassemble_packet() for IPv6 packets. These functions either return a pointer to valid mbuf that contains reassembled packet, or NULL (if the packet can't be reassembled for some reason). Then l3fwd_simple_forward() continues with the code for the packet forwarding decision (that is, the identification of the output interface for the packet) and actual transmit of the packet.

The rte_ipv4_reassemble_packet() or rte_ipv6_reassemble_packet() are responsible for:

1. Searching the Fragment Table for entry with packet's <IP Source Address, IP Destination Address, Packet ID>
2. If the entry is found, then check if that entry already timed-out. If yes, then free all previously received fragments, and remove information about them from the entry.
3. If no entry with such key is found, then try to create a new one by one of two ways:
   (a) Use as empty entry
   (b) Delete a timed-out entry, free mbufs associated with it mbufs and store a new entry with specified key in it.
4. Update the entry with new fragment information and check if a packet can be reassembled (the packet's entry contains all fragments).
   (a) If yes, then, reassemble the packet, mark table's entry as empty and return the reassembled mbuf to the caller.
   (b) If no, then just return a NULL to the caller.

If at any stage of packet processing a reassembly function encounters an error (can't insert new entry into the Fragment table, or invalid/timed-out fragment), then it will free all associated with the packet fragments, mark the table entry as invalid and return NULL to the caller.

Debug logging and Statistics Collection

The RTE_LIBRTE_IP_FRAG_TBL_STAT controls statistics collection for the IP Fragment Table. This macro is disabled by default. To make ip_reassembly print the statistics to the standard output, the user must send either an USR1, INT or TERM signal to the process. For all of these signals, the ip_reassembly process prints Fragment table statistics for each RX queue, plus the INT and TERM will cause process termination as usual.

7.11 Kernel NIC Interface Sample Application

The Kernel NIC Interface (KNI) is a DPDK control plane solution that allows userspace applications to exchange packets with the kernel networking stack. To accomplish this, DPDK userspace applications use an IOCTL call to request the creation of a KNI virtual device in the Linux* kernel. The IOCTL call provides interface information and the DPDK's physical address space, which is re-mapped into the kernel address space by the KNI kernel loadable module that saves the information to a virtual device context. The DPDK creates FIFO queues for packet ingress and egress to the kernel module for each device allocated.

The KNI kernel loadable module is a standard net driver, which upon receiving the IOCTL call access the DPDK's FIFO queue to receive/transmit packets from/to the DPDK userspace application. The FIFO queues contain pointers to data packets in the DPDK. This:

- Provides a faster mechanism to interface with the kernel net stack and eliminates system calls
- Facilitates the DPDK using standard Linux* userspace net tools (tcpdump, ftp, and so on)
- Eliminate the copy_to_user and copy_from_user operations on packets.

The Kernel NIC Interface sample application is a simple example that demonstrates the use of the DPDK to create a path for packets to go through the Linux* kernel. This is done by creating one or more kernel net devices for each of the DPDK ports. The application allows the use of standard Linux tools (ethtool, ifconfig, tcpdump) with the DPDK ports and also the exchange of packets between the DPDK application and the Linux* kernel.

7.11.1 Overview

The Kernel NIC Interface sample application uses two threads in user space for each physical NIC port being used, and allocates one or more KNI device for each physical NIC port with kernel module's support. For a physical NIC port, one thread reads from the port and writes to KNI devices, and another thread reads from KNI devices and writes the data unmodified to the physical NIC port. It is recommended to configure one KNI device for each physical NIC port. If configured with more than one KNI devices for a physical NIC port, it is just for performance testing, or it can work together with VMDq support in future.

The packet flow through the Kernel NIC Interface application is as shown in the following figure.

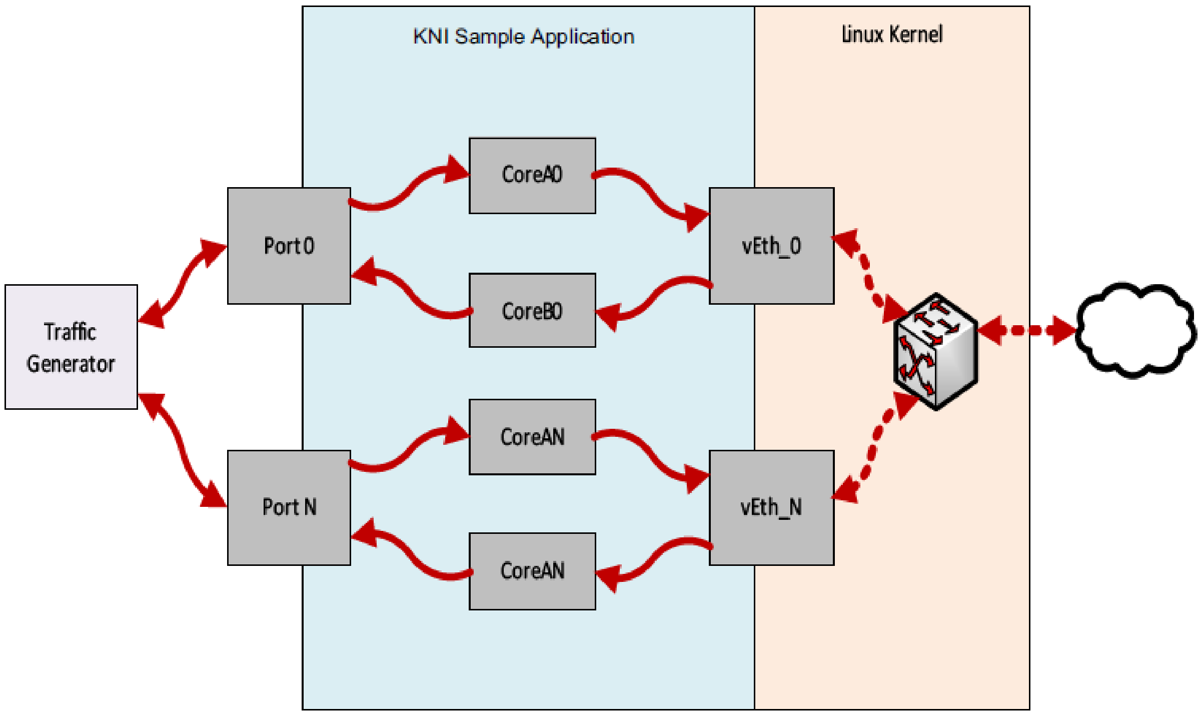


Fig. 7.2: Kernel NIC Application Packet Flow

7.11.2 Compiling the Application

Compile the application as follows:

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/kni
```

2. Set the target (a default target is used if not specified)

Note: This application is intended as a linuxapp only.

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

3. Build the application:

```
make
```

7.11.3 Loading the Kernel Module

Loading the KNI kernel module without any parameter is the typical way a DPDK application gets packets into and out of the kernel net stack. This way, only one kernel thread is created for all KNI devices for packet receiving in kernel side:

```
#insmod rte_kni.ko
```

Pinning the kernel thread to a specific core can be done using a taskset command such as following:

```
#taskset -p 100000 `pgrep --fl kni_thread | awk '{print $1}'`
```

This command line tries to pin the specific kni_thread on the 20th lcore (lcore numbering starts at 0), which means it needs to check if that lcore is available on the board. This command must be sent after the application has been launched, as insmod does not start the kni thread.

For optimum performance, the lcore in the mask must be selected to be on the same socket as the lcores used in the KNI application.

To provide flexibility of performance, the kernel module of the KNI, located in the kmod sub-directory of the DPDK target directory, can be loaded with parameter of kthread_mode as follows:

- #insmod rte_kni.ko kthread_mode=single

  This mode will create only one kernel thread for all KNI devices for packet receiving in kernel side. By default, it is in this single kernel thread mode. It can set core affinity for this kernel thread by using Linux command taskset.

- #insmod rte_kni.ko kthread_mode =multiple

  This mode will create a kernel thread for each KNI device for packet receiving in kernel side. The core affinity of each kernel thread is set when creating the KNI device. The lcore ID for each kernel thread is provided in the command line of launching the application. Multiple kernel thread mode can provide scalable higher performance.

To measure the throughput in a loopback mode, the kernel module of the KNI, located in the kmod sub-directory of the DPDK target directory, can be loaded with parameters as follows:

- #insmod rte_kni.ko lo_mode=lo_mode_fifo

  This loopback mode will involve ring enqueue/dequeue operations in kernel space.

- #insmod rte_kni.ko lo_mode=lo_mode_fifo_skb

  This loopback mode will involve ring enqueue/dequeue operations and sk buffer copies in kernel space.

7.11.4 Running the Application

The application requires a number of command line options:

```
kni [EAL options] -- -P -p PORTMASK --config="(port,lcore_rx,lcore_tx[,lcore_kthread,...])[,por
```

Where:

- -P: Set all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- -p PORTMASK: Hexadecimal bitmask of ports to configure.
- –config="(port,lcore_rx, lcore_tx[,lcore_kthread, ...]) [, port,lcore_rx, lcore_tx[,lcore_kthread, ...]]": Determines which lcores of RX, TX, kernel thread are mapped to which ports.

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The -c coremask parameter of the EAL options should include the lcores indicated by the lcore_rx and lcore_tx, but does not need to include lcores indicated by lcore_kthread as they are used to pin the kernel thread on. The -p PORTMASK parameter should include the ports indicated by the port in –config, neither more nor less.

The lcore_kthread in –config can be configured none, one or more lcore IDs. In multiple kernel thread mode, if configured none, a KNI device will be allocated for each port, while no specific lcore affinity will be set for its kernel thread. If configured one or more lcore IDs, one or more KNI devices will be allocated for each port, while specific lcore affinity will be set for its kernel thread. In single kernel thread mode, if configured none, a KNI device will be allocated for each port. If configured one or more lcore IDs, one or more KNI devices will be allocated for each port while no lcore affinity will be set as there is only one kernel thread for all KNI devices.

For example, to run the application with two ports served by six lcores, one lcore of RX, one lcore of TX, and one lcore of kernel thread for each port:

```
./build/kni -c 0xf0 -n 4 -- -P -p 0x3 -config="(0,4,6,8),(1,5,7,9)"
```

7.11.5 KNI Operations

Once the KNI application is started, one can use different Linux* commands to manage the net interfaces. If more than one KNI devices configured for a physical port, only the first KNI device will be paired to the physical device. Operations on other KNI devices will not affect the physical port handled in user space application.

Assigning an IP address:

```
#ifconfig vEth0_0 192.168.0.1
```

Displaying the NIC registers:

```
#ethtool -d vEth0_0
```

Dumping the network traffic:

```
#tcpdump -i vEth0_0
```

When the DPDK userspace application is closed, all the KNI devices are deleted from Linux*.

7.11.6 Explanation

The following sections provide some explanation of code.

Initialization

Setup of mbuf pool, driver and queues is similar to the setup done in the L2 Forwarding sample application (see Chapter 9 “L2 Forwarding Sample Application (in Real and Virtualized Environments” for details). In addition, one or more kernel NIC interfaces are allocated for each of the configured ports according to the command line parameters.

The code for allocating the kernel NIC interfaces for a specific port is as follows:

```
static int
kni_alloc(uint8_t port_id)
{
    uint8_t i;
    struct rte_kni *kni;
    struct rte_kni_conf conf;
    struct kni_port_params **params = kni_port_params_array;

    if (port_id >= RTE_MAX_ETHPORTS || !params[port_id])
        return -1;

    params[port_id]->nb_kni = params[port_id]->nb_lcore_k ? params[port_id]->nb_lcore_k : 1;

    for (i = 0; i < params[port_id]->nb_kni; i++) {

        /* Clear conf at first */

        memset(&conf, 0, sizeof(conf));
        if (params[port_id]->nb_lcore_k) {
            snprintf(conf.name, RTE_KNI_NAMESIZE, "vEth%u_%u", port_id, i);
            conf.core_id = params[port_id]->lcore_k[i];
            conf.force_bind = 1;
        } else
            snprintf(conf.name, RTE_KNI_NAMESIZE, "vEth%u", port_id);
            conf.group_id = (uint16_t)port_id;
            conf.mbuf_size = MAX_PACKET_SZ;

            /*
             *   The first KNI device associated to a port
             *   is the master, for multiple kernel thread
             *   environment.
             */

            if (i == 0) {
                struct rte_kni_ops ops;
                struct rte_eth_dev_info dev_info;

                memset(&dev_info, 0, sizeof(dev_info)); rte_eth_dev_info_get(port_id, &dev_inf

                conf.addr = dev_info.pci_dev->addr;
                conf.id = dev_info.pci_dev->id;

                memset(&ops, 0, sizeof(ops));

                ops.port_id = port_id;
                ops.change_mtu = kni_change_mtu;
                ops.config_network_if = kni_config_network_interface;
```

```
                    kni = rte_kni_alloc(pktmbuf_pool, &conf, &ops);
                } else
                    kni = rte_kni_alloc(pktmbuf_pool, &conf, NULL);

                if (!kni)
                    rte_exit(EXIT_FAILURE, "Fail to create kni for "
                            "port: %d\n", port_id);

                params[port_id]->kni[i] = kni;
            }
        return 0;
}
```

The other step in the initialization process that is unique to this sample application is the association of each port with lcores for RX, TX and kernel threads.

- One lcore to read from the port and write to the associated one or more KNI devices
- Another lcore to read from one or more KNI devices and write to the port
- Other lcores for pinning the kernel threads on one by one

This is done by using the‘kni_port_params_array[]‘ array, which is indexed by the port ID. The code is as follows:

```
static int
parse_config(const char *arg)
{
    const char *p, *p0 = arg;
    char s[256], *end;
    unsigned size;
    enum fieldnames {
        FLD_PORT = 0,
        FLD_LCORE_RX,
        FLD_LCORE_TX,
        _NUM_FLD = KNI_MAX_KTHREAD + 3,
    };
    int i, j, nb_token;
    char *str_fld[_NUM_FLD];
    unsigned long int_fld[_NUM_FLD];
    uint8_t port_id, nb_kni_port_params = 0;

    memset(&kni_port_params_array, 0, sizeof(kni_port_params_array));

    while (((p = strchr(p0, '(')) != NULL) && nb_kni_port_params < RTE_MAX_ETHPORTS) {
        p++;
        if ((p0 = strchr(p, ')')) == NULL)
            goto fail;

        size = p0 - p;

        if (size >= sizeof(s)) {
            printf("Invalid config parameters\n");
            goto fail;
        }

        snprintf(s, sizeof(s), "%.*s", size, p);
        nb_token = rte_strsplit(s, sizeof(s), str_fld, _NUM_FLD, ',');

        if (nb_token <= FLD_LCORE_TX) {
            printf("Invalid config parameters\n");
            goto fail;
        }
        for (i = 0; i < nb_token; i++) {
            errno = 0;
            int_fld[i] = strtoul(str_fld[i], &end, 0);
            if (errno != 0 || end == str_fld[i]) {
                printf("Invalid config parameters\n");
                goto fail;
            }
        }

        i = 0;
        port_id = (uint8_t)int_fld[i++];

        if (port_id >= RTE_MAX_ETHPORTS) {
            printf("Port ID %u could not exceed the maximum %u\n", port_id, RTE_MAX_ETHPORTS);
            goto fail;
        }

        if (kni_port_params_array[port_id]) {
            printf("Port %u has been configured\n", port_id);
            goto fail;
        }

        kni_port_params_array[port_id] = (struct kni_port_params*)rte_zmalloc("KNI_port_params"
        kni_port_params_array[port_id]->port_id = port_id;
        kni_port_params_array[port_id]->lcore_rx = (uint8_t)int_fld[i++];
        kni_port_params_array[port_id]->lcore_tx = (uint8_t)int_fld[i++];

        if (kni_port_params_array[port_id]->lcore_rx >= RTE_MAX_LCORE || kni_port_params_array[
            printf("lcore_rx %u or lcore_tx %u ID could not "
                    "exceed the maximum %u\n",
                    kni_port_params_array[port_id]->lcore_rx, kni_port_params_array[port_id]->l
            goto fail;
       }

    for (j = 0; i < nb_token && j < KNI_MAX_KTHREAD; i++, j++)
        kni_port_params_array[port_id]->lcore_k[j] = (uint8_t)int_fld[i];
        kni_port_params_array[port_id]->nb_lcore_k = j;
    }

    print_config();

    return 0;

fail:

    for (i = 0; i < RTE_MAX_ETHPORTS; i++) {
        if (kni_port_params_array[i]) {
            rte_free(kni_port_params_array[i]);
            kni_port_params_array[i] = NULL;
        }
    }

    return -1;

}
```

Packet Forwarding

After the initialization steps are completed, the main_loop() function is run on each lcore. This function first checks the lcore_id against the user provided lcore_rx and lcore_tx to see if this lcore is reading from or writing to kernel NIC interfaces.

For the case that reads from a NIC port and writes to the kernel NIC interfaces, the packet reception is the same as in L2 Forwarding sample application (see Section 9.4.6 “Receive, Process and Transmit Packets”). The packet transmission is done by sending mbufs into the kernel NIC interfaces by rte_kni_tx_burst(). The KNI library automatically frees the mbufs after the kernel successfully copied the mbufs.

```
/**
 *   Interface to burst rx and enqueue mbufs into rx_q
 */

static void
kni_ingress(struct kni_port_params *p)
{
    uint8_t i, nb_kni, port_id;
    unsigned nb_rx, num;
    struct rte_mbuf *pkts_burst[PKT_BURST_SZ];

    if (p == NULL)
        return;

    nb_kni = p->nb_kni;
    port_id = p->port_id;

    for (i = 0; i < nb_kni; i++) {
        /* Burst rx from eth */
        nb_rx = rte_eth_rx_burst(port_id, 0, pkts_burst, PKT_BURST_SZ);
        if (unlikely(nb_rx > PKT_BURST_SZ)) {
            RTE_LOG(ERR, APP, "Error receiving from eth\n");
            return;
        }

        /* Burst tx to kni */
        num = rte_kni_tx_burst(p->kni[i], pkts_burst, nb_rx);
        kni_stats[port_id].rx_packets += num;
        rte_kni_handle_request(p->kni[i]);

        if (unlikely(num < nb_rx)) {
            /* Free mbufs not tx to kni interface */
            kni_burst_free_mbufs(&pkts_burst[num], nb_rx - num);
            kni_stats[port_id].rx_dropped += nb_rx - num;
        }
    }
}
```

For the other case that reads from kernel NIC interfaces and writes to a physical NIC port, packets are retrieved by reading mbufs from kernel NIC interfaces by *rte_kni_rx_burst()*. The packet transmission is the same as in the L2 Forwarding sample application (see Section 9.4.6 “Receive, Process and Transmit Packet’s”).

```
/**
 *   Interface to dequeue mbufs from tx_q and burst tx
 */

static void

kni_egress(struct kni_port_params *p)
{
    uint8_t i, nb_kni, port_id;
    unsigned nb_tx, num;
    struct rte_mbuf *pkts_burst[PKT_BURST_SZ];

    if (p == NULL)
```

```
            return;

        nb_kni = p->nb_kni;
        port_id = p->port_id;

        for (i = 0; i < nb_kni; i++) {
            /* Burst rx from kni */
            num = rte_kni_rx_burst(p->kni[i], pkts_burst, PKT_BURST_SZ);
            if (unlikely(num > PKT_BURST_SZ)) {
                RTE_LOG(ERR, APP, "Error receiving from KNI\n");
                return;
            }

            /* Burst tx to eth */

            nb_tx = rte_eth_tx_burst(port_id, 0, pkts_burst, (uint16_t)num);

            kni_stats[port_id].tx_packets += nb_tx;

            if (unlikely(nb_tx < num)) {
                /* Free mbufs not tx to NIC */
                kni_burst_free_mbufs(&pkts_burst[nb_tx], num - nb_tx);
                kni_stats[port_id].tx_dropped += num - nb_tx;
            }
        }
    }
```

Callbacks for Kernel Requests

To execute specific PMD operations in user space requested by some Linux* commands, callbacks must be implemented and filled in the struct rte_kni_ops structure. Currently, setting a new MTU and configuring the network interface (up/ down) are supported.

```
static struct rte_kni_ops kni_ops = {
    .change_mtu = kni_change_mtu,
    .config_network_if = kni_config_network_interface,
};

/* Callback for request of changing MTU */

static int
kni_change_mtu(uint8_t port_id, unsigned new_mtu)
{
    int ret;
    struct rte_eth_conf conf;

    if (port_id >= rte_eth_dev_count()) {
        RTE_LOG(ERR, APP, "Invalid port id %d\n", port_id);
        return -EINVAL;
    }

    RTE_LOG(INFO, APP, "Change MTU of port %d to %u\n", port_id, new_mtu);

    /* Stop specific port */

    rte_eth_dev_stop(port_id);

    memcpy(&conf, &port_conf, sizeof(conf));

    /* Set new MTU */
    if (new_mtu > ETHER_MAX_LEN)
        conf.rxmode.jumbo_frame = 1;
    else
        conf.rxmode.jumbo_frame = 0;

    /* mtu + length of header + length of FCS = max pkt length */

    conf.rxmode.max_rx_pkt_len = new_mtu + KNI_ENET_HEADER_SIZE + KNI_ENET_FCS_SIZE;

    ret = rte_eth_dev_configure(port_id, 1, 1, &conf);
    if (ret < 0) {
        RTE_LOG(ERR, APP, "Fail to reconfigure port %d\n", port_id);
        return ret;
    }

    /* Restart specific port */

    ret = rte_eth_dev_start(port_id);
    if (ret < 0) {
         RTE_LOG(ERR, APP, "Fail to restart port %d\n", port_id);
        return ret;
    }

    return 0;
}

/* Callback for request of configuring network interface up/down */

static int
kni_config_network_interface(uint8_t port_id, uint8_t if_up)
{
    int ret = 0;

    if (port_id >= rte_eth_dev_count() || port_id >= RTE_MAX_ETHPORTS) {
        RTE_LOG(ERR, APP, "Invalid port id %d\n", port_id);
        return -EINVAL;
    }

    RTE_LOG(INFO, APP, "Configure network interface of %d %s\n",

    port_id, if_up ? "up" : "down");

    if (if_up != 0) {
        /* Configure network interface up */
        rte_eth_dev_stop(port_id);
        ret = rte_eth_dev_start(port_id);
    } else /* Configure network interface down */
        rte_eth_dev_stop(port_id);

    if (ret < 0)
        RTE_LOG(ERR, APP, "Failed to start port %d\n", port_id);
    return ret;
}
```

7.12 Keep Alive Sample Application

The Keep Alive application is a simple example of a heartbeat/watchdog for packet processing cores. It demonstrates how to detect 'failed' DPDK cores and notify a fault management entity of this failure. Its purpose is to ensure the failure of the core does not result in a fault that is not detectable by a management entity.

7.12.1 Overview

The application demonstrates how to protect against 'silent outages' on packet processing cores. A Keep Alive Monitor Agent Core (master) monitors the state of packet processing cores (worker cores) by dispatching pings at a regular time interval (default is 5ms) and monitoring the state of the cores. Cores states are: Alive, MIA, Dead or Buried. MIA indicates a missed ping, and Dead indicates two missed pings within the specified time interval. When a core is Dead, a callback function is invoked to restart the packet processing core; A real life application might use this callback function to notify a higher level fault management entity of the core failure in order to take the appropriate corrective action.

Note: Only the worker cores are monitored. A local (on the host) mechanism or agent to supervise the Keep Alive Monitor Agent Core DPDK core is required to detect its failure.

Note: This application is based on the L2 forwarding application. As such, the initialization and run-time paths are very similar to those of the L2 forwarding application.

7.12.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk cd ${RTE_SDK}/examples/keep_alive
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.12.3 Running the Application

The application has a number of command line options:

```
./build/l2fwd-keepalive [EAL options] \
        -- -p PORTMASK [-q NQ] [-K PERIOD] [-T PERIOD]
```

where,

- `p PORTMASK`: A hexadecimal bitmask of the ports to configure
- `q NQ`: A number of queues (=ports) per lcore (default is 1)
- `K PERIOD`: Heartbeat check period in ms(5ms default; 86400 max)
- `T PERIOD`: statistics will be refreshed each PERIOD seconds (0 to disable, 10 default, 86400 maximum).

To run the application in linuxapp environment with 4 lcores, 16 ports 8 RX queues per lcore and a ping interval of 10ms, issue the command:

```
./build/l2fwd-keepalive -c f -n 4 -- -q 8 -p ffff -K 10
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.12.4 Explanation

The following sections provide some explanation of the The Keep-Alive/'Liveliness' conceptual scheme. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the L2 forwarding application (see Chapter 9 "L2 Forwarding Sample Application (in Real and Virtualized Environments)" for more information).

The Keep-Alive/'Liveliness' conceptual scheme:

- A Keep- Alive Agent Runs every N Milliseconds.
- DPDK Cores respond to the keep-alive agent.
- If keep-alive agent detects time-outs, it notifies the fault management entity through a callback function.

The following sections provide some explanation of the code aspects that are specific to the Keep Alive sample application.

The heartbeat functionality is initialized with a struct rte_heartbeat and the callback function to invoke in the case of a timeout.

```
rte_global_keepalive_info = rte_keepalive_create(&dead_core, NULL);
if (rte_global_hbeat_info == NULL)
    rte_exit(EXIT_FAILURE, "keepalive_create() failed");
```

The function that issues the pings hbeat_dispatch_pings() is configured to run every check_period milliseconds.

```
if (rte_timer_reset(&hb_timer,
        (check_period * rte_get_timer_hz()) / 1000,
        PERIODICAL,
        rte_lcore_id(),
        &hbeat_dispatch_pings, rte_global_keepalive_info
        ) != 0 )
    rte_exit(EXIT_FAILURE, "Keepalive setup failure.\n");
```

The rest of the initialization and run-time path follows the same paths as the the L2 forwarding application. The only addition to the main processing loop is the mark alive functionality and the example random failures.

```
rte_keepalive_mark_alive(&rte_global_hbeat_info);
cur_tsc = rte_rdtsc();

/* Die randomly within 7 secs for demo purposes.. */
if (cur_tsc - tsc_initial > tsc_lifetime)
break;
```

The rte_keepalive_mark_alive function simply sets the core state to alive.

```
static inline void
rte_keepalive_mark_alive(struct rte_heartbeat *keepcfg)
{
    keepcfg->state_flags[rte_lcore_id()] = 1;
}
```

7.13 L2 Forwarding Sample Application (in Real and Virtualized Environments) with core load statistics.

The L2 Forwarding sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) which also takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

Note: This application is a variation of L2 Forwarding sample application. It demonstrate possible scheme of job stats library usage therefore some parts of this document is identical with original L2 forwarding application.

7.13.1 Overview

The L2 Forwarding sample application, which can operate in real and virtualized environments, performs L2 forwarding for each packet that is received. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 1 and 2 forward into each other, and ports 3 and 4 forward into each other. Also, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX port MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to benchmark performance using a traffic-generator, as shown in the Fig. 7.3.

The application can also be used in a virtualized environment as shown in Fig. 7.4.

The L2 Forwarding application can also be used as a starting point for developing a new application based on the DPDK.

Fig. 7.3: Performance Benchmark Setup (Basic Environment)

Virtual Function Setup Instructions

This application can use the virtual function available in the system and therefore can be used in a virtual machine without passing through the whole Network Device into a guest machine in a virtualized scenario. The virtual functions can be enabled in the host machine or the hypervisor with the respective physical function driver.

For example, in a Linux* host machine, it is possible to enable a virtual function using the following command:

```
modprobe ixgbe max_vfs=2,2
```

This command enables two Virtual Functions on each of Physical Function of the NIC, with two physical ports in the PCI configuration space. It is important to note that enabled Virtual Function 0 and 2 would belong to Physical Function 0 and Virtual Function 1 and 3 would belong to Physical Function 1, in this case enabling a total of four Virtual Functions.

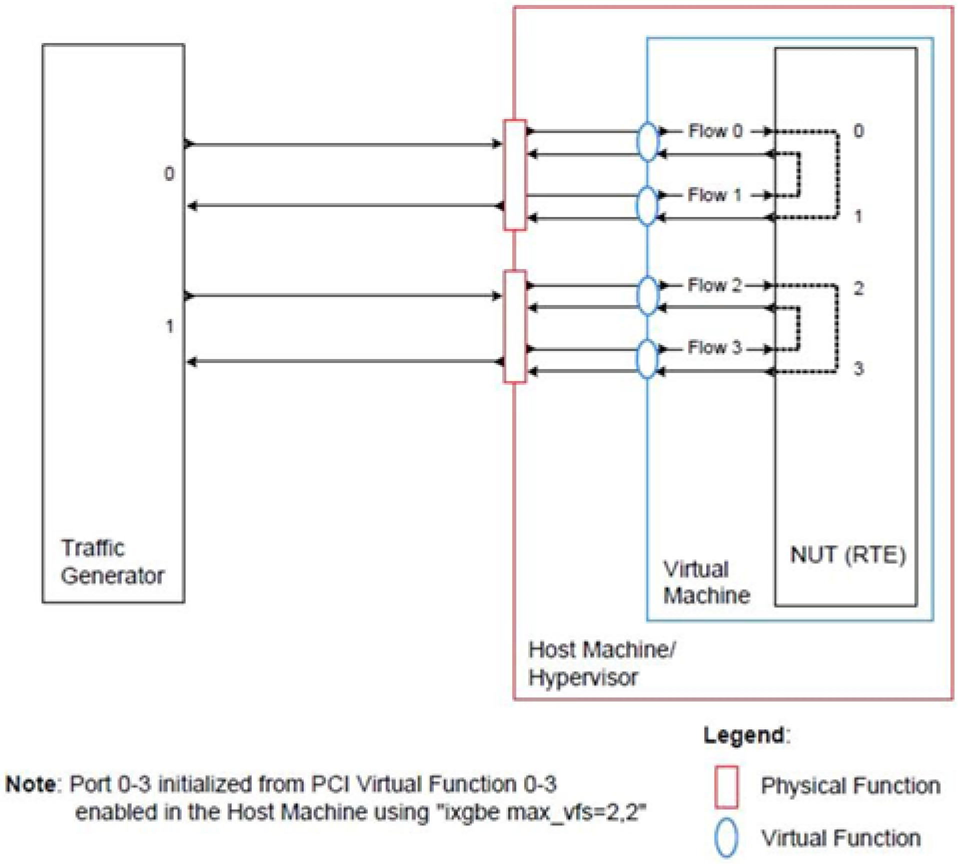


Fig. 7.4: Performance Benchmark Setup (Virtualized Environment)

7.13.2 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/l2fwd-jobstats
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.13.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd-jobstats [EAL options] -- -p PORTMASK [-q NQ] [-l]
```

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure
- q NQ: A number of queues (=ports) per lcore (default is 1)
- l: Use locale thousands separator when formatting big numbers.

To run the application in linuxapp environment with 4 lcores, 16 ports, 8 RX queues per lcore and thousands separator printing, issue the command:

```
$ ./build/l2fwd-jobstats -c f -n 4 -- -q 8 -p ffff -l
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.13.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The L2 Forwarding sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments (see Section 9.3). The preferred way to parse parameters is to use the getopt() function, since it is part of a well-defined and portable library.

The parsing of arguments is done in the l2fwd_parse_args() function. The method of argument parsing is not described here. Refer to the *glibc getopt(3)* man page for details.

EAL arguments are parsed first, then application-specific arguments. This is done at the beginning of the main() function:

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");
```

```
argc -= ret;
argv += ret;

/* parse application arguments (after the EAL ones) */

ret = l2fwd_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid L2FWD arguments\n");
```

Mbuf Pool Initialization

Once the arguments are parsed, the mbuf pool is created. The mbuf pool contains a set of mbuf objects that will be used by the driver and the application to store network packet data:

```
/* create the mbuf pool */
l2fwd_pktmbuf_pool =
    rte_mempool_create("mbuf_pool", NB_MBUF,
               MBUF_SIZE, 32,
               sizeof(struct rte_pktmbuf_pool_private),
               rte_pktmbuf_pool_init, NULL,
               rte_pktmbuf_init, NULL,
               rte_socket_id(), 0);

if (l2fwd_pktmbuf_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot init mbuf pool\n");
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver, which expects to have some reserved space in the mempool structure, sizeof(struct rte_pktmbuf_pool_private) bytes. The number of allocated pkt mbufs is NB_MBUF, with a size of MBUF_SIZE each. A per-lcore cache of 32 mbufs is kept. The memory is allocated in rte_socket_id() socket, but it is possible to extend this code to allocate one mbuf pool per socket.

Two callback pointers are also given to the rte_mempool_create() function:

- The first callback pointer is to rte_pktmbuf_pool_init() and is used to initialize the private data of the mempool, which is needed by the driver. This function is provided by the mbuf API, but can be copied and extended by the developer.
- The second callback pointer given to rte_mempool_create() is the mbuf initializer. The default is used, that is, rte_pktmbuf_init(), which is provided in the rte_mbuf library. If a more complex application wants to extend the rte_pktmbuf structure for its own needs, a new function derived from rte_pktmbuf_init( ) can be created.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer's Guide* and the *DPDK API Reference*.

```
nb_ports = rte_eth_dev_count();

if (nb_ports == 0)
    rte_exit(EXIT_FAILURE, "No Ethernet ports - bye\n");

if (nb_ports > RTE_MAX_ETHPORTS)
    nb_ports = RTE_MAX_ETHPORTS;
```

```
/* reset l2fwd_dst_ports */

for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++)
    l2fwd_dst_ports[portid] = 0;

last_port = 0;

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */
for (portid = 0; portid < nb_ports; portid++) {
    /* skip ports that are not enabled */
    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
       continue;

    if (nb_ports_in_mask % 2) {
        l2fwd_dst_ports[portid] = last_port;
        l2fwd_dst_ports[last_port] = portid;
    }
    else
       last_port = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t)portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: "
        "err=%d, port=%u\n",
        ret, portid);
```

The global configuration is stored in a static structure:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .split_hdr_size = 0,
        .header_split = 0,   /**< Header Split disabled */
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
        .jumbo_frame = 0,    /**< Jumbo Frame Support disabled */
        .hw_strip_crc= 0,    /**< CRC stripped by hardware */
    },

    .txmode = {
        .mq_mode = ETH_DCB_NONE
    },
};
```

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore.

If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup(portid, 0, nb_rxd,
            rte_eth_dev_socket_id(portid),
            NULL,
            l2fwd_pktmbuf_pool);

if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup:err=%d, port=%u\n",
            ret, (unsigned) portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE];
    truct mbuf_table tx_mbufs[RTE_MAX_ETHPORTS];

    struct rte_timer rx_timers[MAX_RX_QUEUE_PER_LCORE];
    struct rte_jobstats port_fwd_jobs[MAX_RX_QUEUE_PER_LCORE];

    struct rte_timer flush_timer;
    struct rte_jobstats flush_job;
    struct rte_jobstats idle_job;
    struct rte_jobstats_context jobs_context;

    rte_atomic16_t stats_read_pending;
    rte_spinlock_t lock;
} __rte_cache_aligned;
```

Values of struct lcore_queue_conf:

- n_rx_port and rx_port_list[] are used in the main packet processing loop (see Section 9.4.6 “Receive, Process and Transmit Packets” later in this chapter).
- rx_timers and flush_timer are used to ensure forced TX on low packet rate.
- flush_job, idle_job and jobs_context are librte_jobstats objects used for managing l2fwd jobs.
- stats_read_pending and lock are used during job stats read phase.

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue on each port */

fflush(stdout);
ret = rte_eth_tx_queue_setup(portid, 0, nb_txd,
        rte_eth_dev_socket_id(portid),
        NULL);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup:err=%d, port=%u\n",
            ret, (unsigned) portid);
```

Jobs statistics initialization

There are several statistics objects available:

- Flush job statistics

```
rte_jobstats_init(&qconf->flush_job, "flush", drain_tsc, drain_tsc,
        drain_tsc, 0);

rte_timer_init(&qconf->flush_timer);
ret = rte_timer_reset(&qconf->flush_timer, drain_tsc, PERIODICAL,
            lcore_id, &l2fwd_flush_job, NULL);

if (ret < 0) {
    rte_exit(1, "Failed to reset flush job timer for lcore %u: %s",
                lcore_id, rte_strerror(-ret));
}
```

- Statistics per RX port

```
rte_jobstats_init(job, name, 0, drain_tsc, 0, MAX_PKT_BURST);
rte_jobstats_set_update_period_function(job, l2fwd_job_update_cb);

rte_timer_init(&qconf->rx_timers[i]);
ret = rte_timer_reset(&qconf->rx_timers[i], 0, PERIODICAL, lcore_id,
        l2fwd_fwd_job, (void *)(uintptr_t)i);

if (ret < 0) {
    rte_exit(1, "Failed to reset lcore %u port %u job timer: %s",
                lcore_id, qconf->rx_port_list[i], rte_strerror(-ret));
}
```

Following parameters are passed to rte_jobstats_init():

- 0 as minimal poll period
- drain_tsc as maximum poll period
- MAX_PKT_BURST as desired target value (RX burst size)

Main loop

The forwarding path is reworked comparing to original L2 Forwarding application. In the l2fwd_main_loop() function three loops are placed.

```
for (;;) {
    rte_spinlock_lock(&qconf->lock);

    do {
        rte_jobstats_context_start(&qconf->jobs_context);

        /* Do the Idle job:
         * - Read stats_read_pending flag
         * - check if some real job need to be executed
         */
        rte_jobstats_start(&qconf->jobs_context, &qconf->idle_job);

        do {
            uint8_t i;
            uint64_t now = rte_get_timer_cycles();

            need_manage = qconf->flush_timer.expire < now;
            /* Check if we was esked to give a stats. */
```

```
                stats_read_pending =
                        rte_atomic16_read(&qconf->stats_read_pending);
                need_manage |= stats_read_pending;

                for (i = 0; i < qconf->n_rx_port && !need_manage; i++)
                    need_manage = qconf->rx_timers[i].expire < now;

            } while (!need_manage);
            rte_jobstats_finish(&qconf->idle_job, qconf->idle_job.target);

            rte_timer_manage();
            rte_jobstats_context_finish(&qconf->jobs_context);
        } while (likely(stats_read_pending == 0));

        rte_spinlock_unlock(&qconf->lock);
        rte_pause();
    }
```

First infinite for loop is to minimize impact of stats reading. Lock is only locked/unlocked when asked.

Second inner while loop do the whole jobs management. When any job is ready, the use rte_timer_manage() is used to call the job handler. In this place functions l2fwd_fwd_job() and l2fwd_flush_job() are called when needed. Then rte_jobstats_context_finish() is called to mark loop end - no other jobs are ready to execute. By this time stats are ready to be read and if stats_read_pending is set, loop breaks allowing stats to be read.

Third do-while loop is the idle job (idle stats counter). Its only purpose is monitoring if any job is ready or stats job read is pending for this lcore. Statistics from this part of code is considered as the headroom available for additional processing.

Receive, Process and Transmit Packets

The main task of l2fwd_fwd_job() function is to read ingress packets from the RX queue of particular port and forward it. This is done using the following code:

```
total_nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst,
        MAX_PKT_BURST);

for (j = 0; j < total_nb_rx; j++) {
    m = pkts_burst[j];
    rte_prefetch0(rte_pktmbuf_mtod(m, void *));
    l2fwd_simple_forward(m, portid);
}
```

Packets are read in a burst of size MAX_PKT_BURST. Then, each mbuf in the table is processed by the l2fwd_simple_forward() function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses.

The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

After first read second try is issued.

```
if (total_nb_rx == MAX_PKT_BURST) {
    const uint16_t nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst,
            MAX_PKT_BURST);

    total_nb_rx += nb_rx;
    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
```

```
            rte_prefetch0(rte_pktmbuf_mtod(m, void *));
            l2fwd_simple_forward(m, portid);
        }
    }
```

This second read is important to give job stats library a feedback how many packets was processed.

```
    /* Adjust period time in which we are running here. */
    if (rte_jobstats_finish(job, total_nb_rx) != 0) {
        rte_timer_reset(&qconf->rx_timers[port_idx], job->period, PERIODICAL,
                lcore_id, l2fwd_fwd_job, arg);
    }
```

To maximize performance exactly MAX_PKT_BURST is expected (the target value) to be read for each l2fwd_fwd_job() call. If total_nb_rx is smaller than target value job->period will be increased. If it is greater the period will be decreased.

Note: In the following code, one line for getting the output port requires some explanation.

During the initialization process, a static array of destination ports (l2fwd_dst_ports[]) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. Naturally, the number of ports in the portmask must be even, otherwise, the application exits.

```
    static void
    l2fwd_simple_forward(struct rte_mbuf *m, unsigned portid)
    {
        struct ether_hdr *eth;
        void *tmp;
        unsigned dst_port;

        dst_port = l2fwd_dst_ports[portid];

        eth = rte_pktmbuf_mtod(m, struct ether_hdr *);

        /* 02:00:00:00:00:xx */

        tmp = &eth->d_addr.addr_bytes[0];

        *((uint64_t *)tmp) = 0x000000000002 + ((uint64_t) dst_port << 40);

        /* src addr */

        ether_addr_copy(&l2fwd_ports_eth_addr[dst_port], &eth->s_addr);

        l2fwd_send_packet(m, (uint8_t) dst_port);
    }
```

Then, the packet is sent using the l2fwd_send_packet (m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the l2fwd_send_burst() function directly from the main loop to send all the received packets on the same TX port, using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that, so the same approach can be reused in a more complex application.

The l2fwd_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the l2fwd_send_burst() function:

```
/* Send the packet on an output interface */

static int
l2fwd_send_packet(struct rte_mbuf *m, uint8_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        l2fwd_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }

    qconf->tx_mbufs[port].len = len; return 0;
}
```

To ensure that no packets remain in the tables, the flush job exists. The l2fwd_flush_job() is called periodically to for each lcore draining TX queue of each port. This technique introduces some latency when there are not many packets to send, however it improves performance:

```
static void
l2fwd_flush_job(__rte_unused struct rte_timer *timer, __rte_unused void *arg)
{
    uint64_t now;
    unsigned lcore_id;
    struct lcore_queue_conf *qconf;
    struct mbuf_table *m_table;
    uint8_t portid;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];

    rte_jobstats_start(&qconf->jobs_context, &qconf->flush_job);

    now = rte_get_timer_cycles();
    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
        m_table = &qconf->tx_mbufs[portid];
        if (m_table->len == 0 || m_table->next_flush_time <= now)
            continue;

        l2fwd_send_burst(qconf, portid);
    }


    /* Pass target to indicate that this job is happy of time interval
     * in which it was called. */
    rte_jobstats_finish(&qconf->flush_job, qconf->flush_job.target);
}
```

7.14 L2 Forwarding Sample Application (in Real and Virtualized Environments)

The L2 Forwarding sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) which also takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

Note: Please note that previously a separate L2 Forwarding in Virtualized Environments sample application was used, however, in later DPDK versions these sample applications have been merged.

7.14.1 Overview

The L2 Forwarding sample application, which can operate in real and virtualized environments, performs L2 forwarding for each packet that is received on an RX_PORT. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 1 and 2 forward into each other, and ports 3 and 4 forward into each other. Also, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX_PORT MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to benchmark performance using a traffic-generator, as shown in the Fig. 7.5.

The application can also be used in a virtualized environment as shown in Fig. 7.6.

The L2 Forwarding application can also be used as a starting point for developing a new application based on the DPDK.

Fig. 7.5: Performance Benchmark Setup (Basic Environment)

Virtual Function Setup Instructions

This application can use the virtual function available in the system and therefore can be used in a virtual machine without passing through the whole Network Device into a guest machine in a virtualized scenario. The virtual functions can be enabled in the host machine or the hypervisor with the respective physical function driver.

For example, in a Linux* host machine, it is possible to enable a virtual function using the following command:

```
modprobe ixgbe max_vfs=2,2
```

This command enables two Virtual Functions on each of Physical Function of the NIC, with two physical ports in the PCI configuration space. It is important to note that enabled Virtual Function 0 and 2 would belong to Physical Function 0 and Virtual Function 1 and 3 would belong to Physical Function 1, in this case enabling a total of four Virtual Functions.

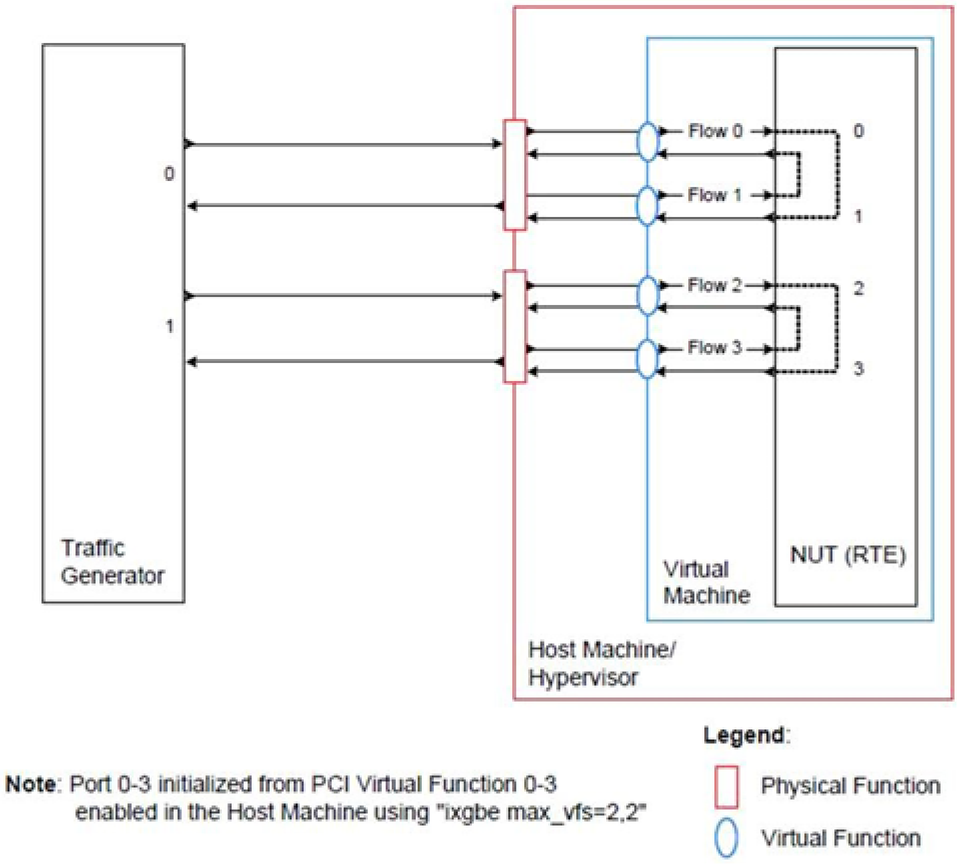


Fig. 7.6: Performance Benchmark Setup (Virtualized Environment)

7.14.2 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/l2fwd
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.14.3 Running the Application

The application requires a number of command line options:

```
./build/l2fwd [EAL options] -- -p PORTMASK [-q NQ]
```

where,

- p PORTMASK: A hexadecimal bitmask of the ports to configure
- q NQ: A number of queues (=ports) per lcore (default is 1)

To run the application in linuxapp environment with 4 lcores, 16 ports and 8 RX queues per lcore, issue the command:

```
$ ./build/l2fwd -c f -n 4 -- -q 8 -p ffff
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.14.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The L2 Forwarding sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments (see Section 9.3). The preferred way to parse parameters is to use the getopt() function, since it is part of a well-defined and portable library.

The parsing of arguments is done in the l2fwd_parse_args() function. The method of argument parsing is not described here. Refer to the *glibc getopt(3)* man page for details.

EAL arguments are parsed first, then application-specific arguments. This is done at the beginning of the main() function:

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

argc -= ret;
```

```
argv += ret;

/* parse application arguments (after the EAL ones) */

ret = l2fwd_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid L2FWD arguments\n");
```

Mbuf Pool Initialization

Once the arguments are parsed, the mbuf pool is created. The mbuf pool contains a set of mbuf objects that will be used by the driver and the application to store network packet data:

```
/* create the mbuf pool */

l2fwd_pktmbuf_pool = rte_mempool_create("mbuf_pool", NB_MBUF, MBUF_SIZE, 32, sizeof(struct rte_
    rte_pktmbuf_pool_init, NULL, rte_pktmbuf_init, NULL, SOCKET0, 0);

if (l2fwd_pktmbuf_pool == NULL)
    rte_panic("Cannot init mbuf pool\n");
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver, which expects to have some reserved space in the mempool structure, sizeof(struct rte_pktmbuf_pool_private) bytes. The number of allocated pkt mbufs is NB_MBUF, with a size of MBUF_SIZE each. A per-lcore cache of 32 mbufs is kept. The memory is allocated in NUMA socket 0, but it is possible to extend this code to allocate one mbuf pool per socket.

Two callback pointers are also given to the rte_mempool_create() function:

- The first callback pointer is to rte_pktmbuf_pool_init() and is used to initialize the private data of the mempool, which is needed by the driver. This function is provided by the mbuf API, but can be copied and extended by the developer.
- The second callback pointer given to rte_mempool_create() is the mbuf initializer. The default is used, that is, rte_pktmbuf_init(), which is provided in the rte_mbuf library. If a more complex application wants to extend the rte_pktmbuf structure for its own needs, a new function derived from rte_pktmbuf_init( ) can be created.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer's Guide* - Rel 1.4 EAR and the *DPDK API Reference*.

```
if (rte_eal_pci_probe() < 0)
    rte_exit(EXIT_FAILURE, "Cannot probe PCI\n");

nb_ports = rte_eth_dev_count();

if (nb_ports == 0)
    rte_exit(EXIT_FAILURE, "No Ethernet ports - bye\n");

if (nb_ports > RTE_MAX_ETHPORTS)
    nb_ports = RTE_MAX_ETHPORTS;

/* reset l2fwd_dst_ports */
```

```
for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++)
    l2fwd_dst_ports[portid] = 0;

last_port = 0;

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */

for (portid = 0; portid < nb_ports; portid++) {
    /* skip ports that are not enabled */

    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
       continue;

    if (nb_ports_in_mask % 2) {
        l2fwd_dst_ports[portid] = last_port;
        l2fwd_dst_ports[last_port] = portid;
    }
    else
       last_port = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

Observe that:

- rte_igb_pmd_init_all() simultaneously registers the driver as a PCI driver and as an Ethernet* Poll Mode Driver.
- rte_eal_pci_probe() parses the devices on the PCI bus and initializes recognized devices.

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t)portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: "
        "err=%d, port=%u\n",
        ret, portid);
```

The global configuration is stored in a static structure:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .split_hdr_size = 0,
        .header_split = 0,   /**< Header Split disabled */
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
        .jumbo_frame = 0,    /**< Jumbo Frame Support disabled */
        .hw_strip_crc= 0,    /**< CRC stripped by hardware */
    },

    .txmode = {
        .mq_mode = ETH_DCB_NONE
    },
};
```

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup((uint8_t) portid, 0, nb_rxd, SOCKET0, &rx_conf, l2fwd_pktmbuf_pool
if (ret < 0)

    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup: "
        "err=%d, port=%u\n",
        ret, portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE];
    struct mbuf_table tx_mbufs[L2FWD_MAX_PORTS];
} rte_cache_aligned;

struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
```

The values n_rx_port and rx_port_list[] are used in the main packet processing loop (see Section 9.4.6 “Receive, Process and Transmit Packets” later in this chapter).

The global configuration for the RX queues is stored in a static structure:

```
static const struct rte_eth_rxconf rx_conf = {
    .rx_thresh = {
        .pthresh = RX_PTHRESH,
        .hthresh = RX_HTHRESH,
        .wthresh = RX_WTHRESH,
    },
};
```

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue on each port */

fflush(stdout);

ret = rte_eth_tx_queue_setup((uint8_t) portid, 0, nb_txd, rte_eth_dev_socket_id(portid), &tx_co
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup:err=%d, port=%u\n", ret, (unsigned) portid);
```

The global configuration for TX queues is stored in a static structure:

```
static const struct rte_eth_txconf tx_conf = {
    .tx_thresh = {
        .pthresh = TX_PTHRESH,
        .hthresh = TX_HTHRESH,
        .wthresh = TX_WTHRESH,
    },
    .tx_free_thresh = RTE_TEST_TX_DESC_DEFAULT + 1, /* disable feature */
};
```

Receive, Process and Transmit Packets

In the l2fwd_main_loop() function, the main task is to read ingress packets from the RX queues. This is done using the following code:

```
/*
 * Read packet from RX queues
 */

for (i = 0; i < qconf->n_rx_port; i++) {
    portid = qconf->rx_port_list[i];
    nb_rx = rte_eth_rx_burst((uint8_t) portid, 0,  pkts_burst, MAX_PKT_BURST);

    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0[rte_pktmbuf_mtod(m, void *)); l2fwd_simple_forward(m, portid);
    }
}
```

Packets are read in a burst of size MAX_PKT_BURST. The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the l2fwd_simple_forward() function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses.

Note: In the following code, one line for getting the output port requires some explanation.

During the initialization process, a static array of destination ports (l2fwd_dst_ports[]) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. Naturally, the number of ports in the portmask must be even, otherwise, the application exits.

```
static void
l2fwd_simple_forward(struct rte_mbuf *m, unsigned portid)
{
    struct ether_hdr *eth;
    void *tmp;
    unsigned dst_port;

    dst_port = l2fwd_dst_ports[portid];

    eth = rte_pktmbuf_mtod(m, struct ether_hdr *);

    /* 02:00:00:00:00:xx */

    tmp = &eth->d_addr.addr_bytes[0];

    *((uint64_t *)tmp) = 0x000000000002 + ((uint64_t) dst_port << 40);

    /* src addr */

    ether_addr_copy(&l2fwd_ports_eth_addr[dst_port], &eth->s_addr);

    l2fwd_send_packet(m, (uint8_t) dst_port);
}
```

Then, the packet is sent using the l2fwd_send_packet (m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the l2fwd_send_burst() function directly from the main loop to send all the received packets on the same TX port, using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that, so the same approach can be reused in a more complex application.

The l2fwd_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the l2fwd_send_burst() function:

```
/* Send the packet on an output interface */

static int
l2fwd_send_packet(struct rte_mbuf *m, uint8_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf \*qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        l2fwd_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }

    qconf->tx_mbufs[port].len = len; return 0;
}
```

To ensure that no packets remain in the tables, each lcore does a draining of TX queue in its main loop. This technique introduces some latency when there are not many packets to send, however it improves performance:

```
cur_tsc = rte_rdtsc();

/*
 *   TX burst queue drain
 */

diff_tsc = cur_tsc - prev_tsc;

if (unlikely(diff_tsc > drain_tsc)) {
    for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
        if (qconf->tx_mbufs[portid].len == 0)
            continue;

        l2fwd_send_burst(&lcore_queue_conf[lcore_id], qconf->tx_mbufs[portid].len, (uint8_t) po

        qconf->tx_mbufs[portid].len = 0;
    }

    /* if timer is enabled */

    if (timer_period > 0) {
        /* advance the timer */

        timer_tsc += diff_tsc;

        /* if timer has reached its timeout */

        if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
            /* do this only on master core */

            if (lcore_id == rte_get_master_lcore()) {
                print_stats();

                /* reset the timer */
                timer_tsc = 0;
            }
        }
    }

    prev_tsc = cur_tsc;
}
```

7.15 L3 Forwarding Sample Application

The L3 Forwarding application is a simple example of packet processing using the DPDK. The application performs L3 forwarding.

7.15.1 Overview

The application demonstrates the use of the hash and LPM libraries in the DPDK to implement packet forwarding. The initialization and run-time paths are very similar to those of the L2 forwarding application (see Chapter 9 "L2 Forwarding Sample Application (in Real and Virtualized Environments)" for more information). The main difference from the L2 Forwarding sample application is that the forwarding decision is made based on information read from the input packet.

The lookup method is either hash-based or LPM-based and is selected at compile time. When the selected lookup method is hash-based, a hash object is used to emulate the flow classification stage. The hash object is used in correlation with a flow table to map each input packet to its flow at runtime.

The hash lookup key is represented by a DiffServ 5-tuple composed of the following fields read from the input packet: Source IP Address, Destination IP Address, Protocol, Source Port and Destination Port. The ID of the output interface for the input packet is read from the identified flow table entry. The set of flows used by the application is statically configured and loaded into the hash at initialization time. When the selected lookup method is LPM based, an LPM object is used to emulate the forwarding stage for IPv4 packets. The LPM object is used as the routing table to identify the next hop for each input packet at runtime.

The LPM lookup key is represented by the Destination IP Address field read from the input packet. The ID of the output interface for the input packet is the next hop returned by the LPM lookup. The set of LPM rules used by the application is statically configured and loaded into the LPM object at initialization time.

In the sample application, hash-based forwarding supports IPv4 and IPv6. LPM-based forwarding supports IPv4 only.

7.15.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/l3fwd
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.15.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd [EAL options] -- -p PORTMASK [-P]  --config(port,queue,lcore)[,(port,queue,lcore)
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -P: optional, sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- –config (port,queue,lcore)[,(port,queue,lcore)]: determines which queues from which ports are mapped to which cores
- –enable-jumbo: optional, enables jumbo frames
- –max-pkt-len: optional, maximum packet length in decimal (64-9600)
- –no-numa: optional, disables numa awareness
- –hash-entry-num: optional, specifies the hash entry number in hexadecimal to be setup
- –ipv6: optional, set it if running ipv6 packets

For example, consider a dual processor socket platform where cores 0-7 and 16-23 appear on socket 0, while cores 8-15 and 24-31 appear on socket 1. Let's say that the programmer wants to use memory from both NUMA nodes, the platform has only two ports, one connected to each NUMA node, and the programmer wants to use two cores from each processor socket to do the packet processing.

To enable L3 forwarding between two ports, using two cores, cores 1 and 2, from each processor, while also taking advantage of local memory access by optimizing around NUMA, the programmer must enable two queues from each port, pin to the appropriate cores and allocate memory from the appropriate NUMA node. This is achieved using the following command:

```
./build/l3fwd -c 606 -n 4 -- -p 0x3 --config="(0,0,1),(0,1,2),(1,0,9),(1,1,10)"
```

In this command:

- The -c option enables cores 0, 1, 2, 3

- The -p option enables ports 0 and 1
- The –config option enables two queues on each port and maps each (port,queue) pair to a specific core. Logic to enable multiple RX queues using RSS and to allocate memory from the correct NUMA nodes is included in the application and is done transparently. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	0	Map queue 0 from port 0 to lcore 0.
0	1	2	Map queue 1 from port 0 to lcore 2.
1	0	1	Map queue 0 from port 1 to lcore 1.
1	1	3	Map queue 1 from port 1 to lcore 3.

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.15.4 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the L2 forwarding application (see Chapter 9 “L2 Forwarding Sample Application (in Real and Virtualized Environments)” for more information). The following sections describe aspects that are specific to the L3 Forwarding sample application.

Hash Initialization

The hash object is created and loaded with the pre-configured entries read from a global array, and then generate the expected 5-tuple as key to keep consistence with those of real flow for the convenience to execute hash performance test on 4M/8M/16M flows.

Note: The Hash initialization will setup both ipv4 and ipv6 hash table, and populate the either table depending on the value of variable ipv6. To support the hash performance test with up to 8M single direction flows/16M bi-direction flows, populate_ipv4_many_flow_into_table() function will populate the hash table with specified hash table entry number(default 4M).

Note: Value of global variable ipv6 can be specified with –ipv6 in the command line. Value of global variable hash_entry_number, which is used to specify the total hash entry number for all used ports in hash performance test, can be specified with –hash-entry-num VALUE in command line, being its default value 4.

```
#if (APP_LOOKUP_METHOD == APP_LOOKUP_EXACT_MATCH)

    static void
    setup_hash(int socketid)
    {
        // ...

        if (hash_entry_number != HASH_ENTRY_NUMBER_DEFAULT) {
            if (ipv6 == 0) {
                /* populate the ipv4 hash */
                populate_ipv4_many_flow_into_table(ipv4_l3fwd_lookup_struct[socketid], hash_ent
            } else {
                /* populate the ipv6 hash */
                populate_ipv6_many_flow_into_table( ipv6_l3fwd_lookup_struct[socketid], hash_er
            }
        } else
            if (ipv6 == 0) {
                /* populate the ipv4 hash */
                populate_ipv4_few_flow_into_table(ipv4_l3fwd_lookup_struct[socketid]);
            } else {
                /* populate the ipv6 hash */
                populate_ipv6_few_flow_into_table(ipv6_l3fwd_lookup_struct[socketid]);
            }
        }
    }
#endif
```

LPM Initialization

The LPM object is created and loaded with the pre-configured entries read from a global array.

```
#if (APP_LOOKUP_METHOD == APP_LOOKUP_LPM)

static void
setup_lpm(int socketid)
{
    unsigned i;
    int ret;
    char s[64];

    /* create the LPM table */

    snprintf(s, sizeof(s), "IPV4_L3FWD_LPM_%d", socketid);

    ipv4_l3fwd_lookup_struct[socketid] = rte_lpm_create(s, socketid, IPV4_L3FWD_LPM_MAX_RULES,

    if (ipv4_l3fwd_lookup_struct[socketid] == NULL)
        rte_exit(EXIT_FAILURE, "Unable to create the l3fwd LPM table"
            " on socket %d\n", socketid);

    /* populate the LPM table */

    for (i = 0; i < IPV4_L3FWD_NUM_ROUTES; i++) {
        /* skip unused ports */

        if ((1 << ipv4_l3fwd_route_array[i].if_out & enabled_port_mask) == 0)
            continue;

        ret = rte_lpm_add(ipv4_l3fwd_lookup_struct[socketid], ipv4_l3fwd_route_array[i].ip,
                                ipv4_l3fwd_route_array[i].depth, ipv4_l3fwd_route_array[i].if_o

        if (ret < 0) {
            rte_exit(EXIT_FAILURE, "Unable to add entry %u to the "
                    "l3fwd LPM table on socket %d\n", i, socketid);
        }

        printf("LPM: Adding route 0x%08x / %d (%d)\n",
            (unsigned)ipv4_l3fwd_route_array[i].ip, ipv4_l3fwd_route_array[i].depth, ipv4_l3fwc
    }
}
#endif
```

Packet Forwarding for Hash-based Lookups

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() or simple_ipv4_fwd_4pkts() function for IPv4 packets or the simple_ipv6_fwd_4pkts() function for IPv6 packets. The l3fwd_simple_forward() function provides the basic functionality for both IPv4 and IPv6 packet forwarding for any number of burst packets received, and the packet forwarding decision (that is, the identification of the output interface for the packet) for hash-based lookups is done by the get_ipv4_dst_port() or get_ipv6_dst_port() function. The get_ipv4_dst_port() function is shown below:

```
static inline uint8_t
get_ipv4_dst_port(void *ipv4_hdr, uint8_t portid, lookup_struct_t *ipv4_l3fwd_lookup_struct)
{
    int ret = 0;
    union ipv4_5tuple_host key;

    ipv4_hdr = (uint8_t \*)ipv4_hdr + offsetof(struct ipv4_hdr, time_to_live);

    m128i data = _mm_loadu_si128(( m128i*)(ipv4_hdr));

    /* Get 5 tuple: dst port, src port, dst IP address, src IP address and protocol */

    key.xmm = _mm_and_si128(data, mask0);

    /* Find destination port */

    ret = rte_hash_lookup(ipv4_l3fwd_lookup_struct, (const void *)&key);

    return (uint8_t)((ret < 0)? portid : ipv4_l3fwd_out_if[ret]);
}
```

The get_ipv6_dst_port() function is similar to the get_ipv4_dst_port() function.

The simple_ipv4_fwd_4pkts() and simple_ipv6_fwd_4pkts() function are optimized for continuous 4 valid ipv4 and ipv6 packets, they leverage the multiple buffer optimization to boost the performance of forwarding packets with the exact match on hash table. The key code snippet of simple_ipv4_fwd_4pkts() is shown below:

```
static inline void
simple_ipv4_fwd_4pkts(struct rte_mbuf* m[4], uint8_t portid, struct lcore_conf *qconf)
{
    // ...

    data[0] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[0], unsigned char *) + sizeof(struct
    data[1] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[1], unsigned char *) + sizeof(struct
    data[2] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[2], unsigned char *) + sizeof(struct
    data[3] = _mm_loadu_si128(( m128i*)(rte_pktmbuf_mtod(m[3], unsigned char *) + sizeof(struct

    key[0].xmm = _mm_and_si128(data[0], mask0);
    key[1].xmm = _mm_and_si128(data[1], mask0);
    key[2].xmm = _mm_and_si128(data[2], mask0);
    key[3].xmm = _mm_and_si128(data[3], mask0);

    const void *key_array[4] = {&key[0], &key[1], &key[2],&key[3]};

    rte_hash_lookup_multi(qconf->ipv4_lookup_struct, &key_array[0], 4, ret);

    dst_port[0] = (ret[0] < 0)? portid:ipv4_l3fwd_out_if[ret[0]];
    dst_port[1] = (ret[1] < 0)? portid:ipv4_l3fwd_out_if[ret[1]];
    dst_port[2] = (ret[2] < 0)? portid:ipv4_l3fwd_out_if[ret[2]];
    dst_port[3] = (ret[3] < 0)? portid:ipv4_l3fwd_out_if[ret[3]];
```

```
        // ...
    }
```

The simple_ipv6_fwd_4pkts() function is similar to the simple_ipv4_fwd_4pkts() function.

Packet Forwarding for LPM-based Lookups

For each input packet, the packet forwarding operation is done by the l3fwd_simple_forward() function, but the packet forwarding decision (that is, the identification of the output interface for the packet) for LPM-based lookups is done by the get_ipv4_dst_port() function below:

```
static inline uint8_t
get_ipv4_dst_port(struct ipv4_hdr *ipv4_hdr, uint8_t portid, lookup_struct_t *ipv4_l3fwd_lookup
{
    uint8_t next_hop;

    return (uint8_t) ((rte_lpm_lookup(ipv4_l3fwd_lookup_struct, rte_be_to_cpu_32(ipv4_hdr->dst_
}
```

7.16 L3 Forwarding with Power Management Sample Application

7.16.1 Introduction

The L3 Forwarding with Power Management application is an example of power-aware packet processing using the DPDK. The application is based on existing L3 Forwarding sample application, with the power management algorithms to control the P-states and C-states of the Intel processor via a power management library.

7.16.2 Overview

The application demonstrates the use of the Power libraries in the DPDK to implement packet forwarding. The initialization and run-time paths are very similar to those of the L3 forwarding sample application (see Chapter 10 “L3 Forwarding Sample Application” for more information). The main difference from the L3 Forwarding sample application is that this application introduces power-aware optimization algorithms by leveraging the Power library to control P-state and C-state of processor based on packet load.

The DPDK includes poll-mode drivers to configure Intel NIC devices and their receive (Rx) and transmit (Tx) queues. The design principle of this PMD is to access the Rx and Tx descriptors directly without any interrupts to quickly receive, process and deliver packets in the user space.

In general, the DPDK executes an endless packet processing loop on dedicated IA cores that include the following steps:

- Retrieve input packets through the PMD to poll Rx queue
- Process each received packet or provide received packets to other processing cores through software queues
- Send pending output packets to Tx queue through the PMD

In this way, the PMD achieves better performance than a traditional interrupt-mode driver, at the cost of keeping cores active and running at the highest frequency, hence consuming the maximum power all the time. However, during the period of processing light network traffic, which happens regularly in communication infrastructure systems due to well-known “tidal effect”, the PMD is still busy waiting for network packets, which wastes a lot of power.

Processor performance states (P-states) are the capability of an Intel processor to switch between different supported operating frequencies and voltages. If configured correctly, according to system workload, this feature provides power savings. CPUFreq is the infrastructure provided by the Linux* kernel to control the processor performance state capability. CPUFreq supports a user space governor that enables setting frequency via manipulating the virtual file device from a user space application. The Power library in the DPDK provides a set of APIs for manipulating a virtual file device to allow user space application to set the CPUFreq governor and set the frequency of specific cores.

This application includes a P-state power management algorithm to generate a frequency hint to be sent to CPUFreq. The algorithm uses the number of received and available Rx packets on recent polls to make a heuristic decision to scale frequency up/down. Specifically, some thresholds are checked to see whether a specific core running an DPDK polling thread needs to increase frequency a step up based on the near to full trend of polled Rx queues. Also, it decreases frequency a step if packet processed per loop is far less than the expected threshold or the thread’s sleeping time exceeds a threshold.

C-States are also known as sleep states. They allow software to put an Intel core into a low power idle state from which it is possible to exit via an event, such as an interrupt. However, there is a tradeoff between the power consumed in the idle state and the time required to wake up from the idle state (exit latency). Therefore, as you go into deeper C-states, the power consumed is lower but the exit latency is increased. Each C-state has a target residency. It is essential that when entering into a C-state, the core remains in this C-state for at least as long as the target residency in order to fully realize the benefits of entering the C-state. CPUIdle is the infrastructure provide by the Linux kernel to control the processor C-state capability. Unlike CPUFreq, CPUIdle does not provide a mechanism that allows the application to change C-state. It actually has its own heuristic algorithms in kernel space to select target C-state to enter by executing privileged instructions like HLT and MWAIT, based on the speculative sleep duration of the core. In this application, we introduce a heuristic algorithm that allows packet processing cores to sleep for a short period if there is no Rx packet received on recent polls. In this way, CPUIdle automatically forces the corresponding cores to enter deeper C-states instead of always running to the C0 state waiting for packets.

Note: To fully demonstrate the power saving capability of using C-states, it is recommended to enable deeper C3 and C6 states in the BIOS during system boot up.

7.16.3 Compiling the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/l3fwd-power
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.16.4 Running the Application

The application has a number of command line options:

```
./build/l3fwd_power [EAL options] -- -p PORTMASK [-P]  --config(port,queue,lcore)[,(port,queue,
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -P: Sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- –config (port,queue,lcore)[,(port,queue,lcore)]: determines which queues from which ports are mapped to which cores.
- –enable-jumbo: optional, enables jumbo frames
- –max-pkt-len: optional, maximum packet length in decimal (64-9600)
- –no-numa: optional, disables numa awareness

See Chapter 10 "L3 Forwarding Sample Application" for details. The L3fwd-power example reuses the L3fwd command line options.

7.16.5 Explanation

The following sections provide some explanation of the sample application code. As mentioned in the overview section, the initialization and run-time paths are identical to those of the L3 forwarding application. The following sections describe aspects that are specific to the L3 Forwarding with Power Management sample application.

Power Library Initialization

The Power library is initialized in the main routine. It changes the P-state governor to userspace for specific cores that are under control. The Timer library is also initialized and several timers are created later on, responsible for checking if it needs to scale down frequency at run time by checking CPU utilization statistics.

Note: Only the power management related initialization is shown.

```
int main(int argc, char **argv)
{
    struct lcore_conf *qconf;
    int ret;
    unsigned nb_ports;
    uint16_t queueid;
    unsigned lcore_id;
    uint64_t hz;
    uint32_t n_tx_queue, nb_lcores;
    uint8_t portid, nb_rx_queue, queue, socketid;
```

```
        // ...

        /* init RTE timer library to be used to initialize per-core timers */

        rte_timer_subsystem_init();

        // ...


        /* per-core initialization */

        for (lcore_id = 0; lcore_id < RTE_MAX_LCORE; lcore_id++) {
            if (rte_lcore_is_enabled(lcore_id) == 0)
                continue;

            /* init power management library for a specified core */

            ret = rte_power_init(lcore_id);
            if (ret)
                rte_exit(EXIT_FAILURE, "Power management library "
                    "initialization failed on core%d\n", lcore_id);

            /* init timer structures for each enabled lcore */

            rte_timer_init(&power_timers[lcore_id]);

            hz = rte_get_hpet_hz();

            rte_timer_reset(&power_timers[lcore_id], hz/TIMER_NUMBER_PER_SECOND, SINGLE, lcore_id,

            // ...
        }

        // ...
    }
```

Monitoring Loads of Rx Queues

In general, the polling nature of the DPDK prevents the OS power management subsystem from knowing if the network load is actually heavy or light. In this sample, sampling network load work is done by monitoring received and available descriptors on NIC Rx queues in recent polls. Based on the number of returned and available Rx descriptors, this example implements algorithms to generate frequency scaling hints and speculative sleep duration, and use them to control P-state and C-state of processors via the power management library. Frequency (P-state) control and sleep state (C-state) control work individually for each logical core, and the combination of them contributes to a power efficient packet processing solution when serving light network loads.

The rte_eth_rx_burst() function and the newly-added rte_eth_rx_queue_count() function are used in the endless packet processing loop to return the number of received and available Rx descriptors. And those numbers of specific queue are passed to P-state and C-state heuristic algorithms to generate hints based on recent network load trends.

Note: Only power control related code is shown.

```
    static
```

```
attribute ((noreturn)) int main_loop( attribute ((unused)) void *dummy)
{
    // ...

    while (1) {
    // ...

    /**
     * Read packet from RX queues
     */

    lcore_scaleup_hint = FREQ_CURRENT;
    lcore_rx_idle_count = 0;

    for (i = 0; i < qconf->n_rx_queue; ++i)
    {
        rx_queue = &(qconf->rx_queue_list[i]);
        rx_queue->idle_hint = 0;
        portid = rx_queue->port_id;
        queueid = rx_queue->queue_id;

        nb_rx = rte_eth_rx_burst(portid, queueid, pkts_burst, MAX_PKT_BURST);
        stats[lcore_id].nb_rx_processed += nb_rx;

        if (unlikely(nb_rx == 0)) {
            /**
             * no packet received from rx queue, try to
             * sleep for a while forcing CPU enter deeper
             * C states.
             */

            rx_queue->zero_rx_packet_count++;

            if (rx_queue->zero_rx_packet_count <= MIN_ZERO_POLL_COUNT)
                continue;

            rx_queue->idle_hint = power_idle_heuristic(rx_queue->zero_rx_packet_count);
            lcore_rx_idle_count++;
        } else {
            rx_ring_length = rte_eth_rx_queue_count(portid, queueid);

            rx_queue->zero_rx_packet_count = 0;

            /**
             * do not scale up frequency immediately as
             * user to kernel space communication is costly
             * which might impact packet I/O for received
             * packets.
             */

            rx_queue->freq_up_hint = power_freq_scaleup_heuristic(lcore_id, rx_ring_length);
        }

        /* Prefetch and forward packets */

        // ...
    }

    if (likely(lcore_rx_idle_count != qconf->n_rx_queue)) {
        for (i = 1, lcore_scaleup_hint = qconf->rx_queue_list[0].freq_up_hint; i < qconf->n_rx_
            x_queue = &(qconf->rx_queue_list[i]);

            if (rx_queue->freq_up_hint > lcore_scaleup_hint)
```

```
                    lcore_scaleup_hint = rx_queue->freq_up_hint;
            }

            if (lcore_scaleup_hint == FREQ_HIGHEST)

                rte_power_freq_max(lcore_id);

            else if (lcore_scaleup_hint == FREQ_HIGHER)
                rte_power_freq_up(lcore_id);
            } else {
                /**
                 *  All Rx queues empty in recent consecutive polls,
                 *  sleep in a conservative manner, meaning sleep as
                 * less as possible.
                 */

                for (i = 1, lcore_idle_hint = qconf->rx_queue_list[0].idle_hint; i < qconf->n_rx_qu
                    rx_queue = &(qconf->rx_queue_list[i]);
                    if (rx_queue->idle_hint < lcore_idle_hint)
                        lcore_idle_hint = rx_queue->idle_hint;
                }

                if ( lcore_idle_hint < SLEEP_GEAR1_THRESHOLD)
                    /**
                     *   execute "pause" instruction to avoid context
                     *   switch for short sleep.
                     */
                    rte_delay_us(lcore_idle_hint);
                else
                    /* long sleep force ruining thread to suspend */
                    usleep(lcore_idle_hint);

               stats[lcore_id].sleep_time += lcore_idle_hint;
            }
        }
    }
```

P-State Heuristic Algorithm

The power_freq_scaleup_heuristic() function is responsible for generating a frequency hint for the specified logical core according to available descriptor number returned from rte_eth_rx_queue_count(). On every poll for new packets, the length of available descriptor on an Rx queue is evaluated, and the algorithm used for frequency hinting is as follows:

- If the size of available descriptors exceeds 96, the maximum frequency is hinted.
- If the size of available descriptors exceeds 64, a trend counter is incremented by 100.
- If the length of the ring exceeds 32, the trend counter is incremented by 1.
- When the trend counter reached 10000 the frequency hint is changed to the next higher frequency.

Note: The assumption is that the Rx queue size is 128 and the thresholds specified above must be adjusted accordingly based on actual hardware Rx queue size, which are configured via the rte_eth_rx_queue_setup() function.

In general, a thread needs to poll packets from multiple Rx queues. Most likely, different queue have different load, so they would return different frequency hints. The algorithm evaluates all the hints and then scales up frequency in an aggressive manner by scaling up to highest frequency as long as one Rx queue requires. In this way, we can minimize any negative performance impact.

On the other hand, frequency scaling down is controlled in the timer callback function. Specifically, if the sleep times of a logical core indicate that it is sleeping more than 25% of the sampling period, or if the average packet per iteration is less than expectation, the frequency is decreased by one step.

C-State Heuristic Algorithm

Whenever recent rte_eth_rx_burst() polls return 5 consecutive zero packets, an idle counter begins incrementing for each successive zero poll. At the same time, the function power_idle_heuristic() is called to generate speculative sleep duration in order to force logical to enter deeper sleeping C-state. There is no way to control C- state directly, and the CPUIdle subsystem in OS is intelligent enough to select C-state to enter based on actual sleep period time of giving logical core. The algorithm has the following sleeping behavior depending on the idle counter:

- If idle count less than 100, the counter value is used as a microsecond sleep value through rte_delay_us() which execute pause instructions to avoid costly context switch but saving power at the same time.
- If idle count is between 100 and 999, a fixed sleep interval of 100 μs is used. A 100 μs sleep interval allows the core to enter the C1 state while keeping a fast response time in case new traffic arrives.
- If idle count is greater than 1000, a fixed sleep value of 1 ms is used until the next timer expiration is used. This allows the core to enter the C3/C6 states.

Note: The thresholds specified above need to be adjusted for different Intel processors and traffic profiles.

If a thread polls multiple Rx queues and different queue returns different sleep duration values, the algorithm controls the sleep time in a conservative manner by sleeping for the least possible time in order to avoid a potential performance impact.

7.17 L3 Forwarding with Access Control Sample Application

The L3 Forwarding with Access Control application is a simple example of packet processing using the DPDK. The application performs a security check on received packets. Packets that are in the Access Control List (ACL), which is loaded during initialization, are dropped. Others are forwarded to the correct port.

7.17.1 Overview

The application demonstrates the use of the ACL library in the DPDK to implement access control and packet L3 forwarding. The application loads two types of rules at initialization:

- Route information rules, which are used for L3 forwarding
- Access Control List (ACL) rules that blacklist (or block) packets with a specific characteristic

When packets are received from a port, the application extracts the necessary information from the TCP/IP header of the received packet and performs a lookup in the rule database to figure out whether the packets should be dropped (in the ACL range) or forwarded to desired ports. The initialization and run-time paths are similar to those of the L3 forwarding application (see Chapter 10, “L3 Forwarding Sample Application” for more information). However, there are significant differences in the two applications. For example, the original L3 forwarding application uses either LPM or an exact match algorithm to perform forwarding port lookup, while this application uses the ACL library to perform both ACL and route entry lookup. The following sections provide more detail.

Classification for both IPv4 and IPv6 packets is supported in this application. The application also assumes that all the packets it processes are TCP/UDP packets and always extracts source/destination port information from the packets.

Tuple Packet Syntax

The application implements packet classification for the IPv4/IPv6 5-tuple syntax specifically. The 5-tuple syntax consist of a source IP address, a destination IP address, a source port, a destination port and a protocol identifier. The fields in the 5-tuple syntax have the following formats:

- **Source IP address and destination IP address** : Each is either a 32-bit field (for IPv4), or a set of 4 32-bit fields (for IPv6) represented by a value and a mask length. For example, an IPv4 range of 192.168.1.0 to 192.168.1.255 could be represented by a value = [192, 168, 1, 0] and a mask length = 24.
- **Source port and destination port** : Each is a 16-bit field, represented by a lower start and a higher end. For example, a range of ports 0 to 8192 could be represented by lower = 0 and higher = 8192.
- **Protocol identifier** : An 8-bit field, represented by a value and a mask, that covers a range of values. To verify that a value is in the range, use the following expression: “(VAL & mask) == value”

The trick in how to represent a range with a mask and value is as follows. A range can be enumerated in binary numbers with some bits that are never changed and some bits that are dynamically changed. Set those bits that dynamically changed in mask and value with 0. Set those bits that never changed in the mask with 1, in value with number expected. For example, a range of 6 to 7 is enumerated as 0b110 and 0b111. Bit 1-7 are bits never changed and bit 0 is the bit dynamically changed. Therefore, set bit 0 in mask and value with 0, set bits 1-7 in mask with 1, and bits 1-7 in value with number 0b11. So, mask is 0xfe, value is 0x6.

Note: The library assumes that each field in the rule is in LSB or Little Endian order when creating the database. It internally converts them to MSB or Big Endian order. When performing a lookup, the library assumes the input is in MSB or Big Endian order.

Access Rule Syntax

In this sample application, each rule is a combination of the following:

- 5-tuple field: This field has a format described in Section.
- priority field: A weight to measure the priority of the rules. The rule with the higher priority will ALWAYS be returned if the specific input has multiple matches in the rule database. Rules with lower priority will NEVER be returned in any cases.
- userdata field: A user-defined field that could be any value. It can be the forwarding port number if the rule is a route table entry or it can be a pointer to a mapping address if the rule is used for address mapping in the NAT application. The key point is that it is a useful reserved field for user convenience.

ACL and Route Rules

The application needs to acquire ACL and route rules before it runs. Route rules are mandatory, while ACL rules are optional. To simplify the complexity of the priority field for each rule, all ACL and route entries are assumed to be in the same file. To read data from the specified file successfully, the application assumes the following:

- Each rule occupies a single line.
- Only the following four rule line types are valid in this application:
- ACL rule line, which starts with a leading character '@'
- Route rule line, which starts with a leading character 'R'
- Comment line, which starts with a leading character '#'
- Empty line, which consists of a space, form-feed ('f'), newline ('n'), carriage return ('r'), horizontal tab ('t'), or vertical tab ('v').

Other lines types are considered invalid.

- Rules are organized in descending order of priority, which means rules at the head of the file always have a higher priority than those further down in the file.
- A typical IPv4 ACL rule line should have a format as shown below:

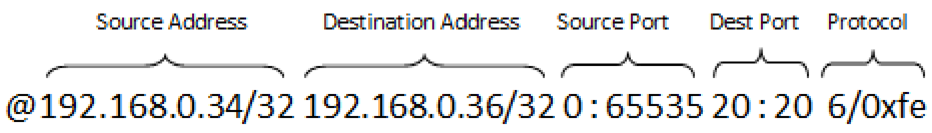


Fig. 7.7: A typical IPv4 ACL rule

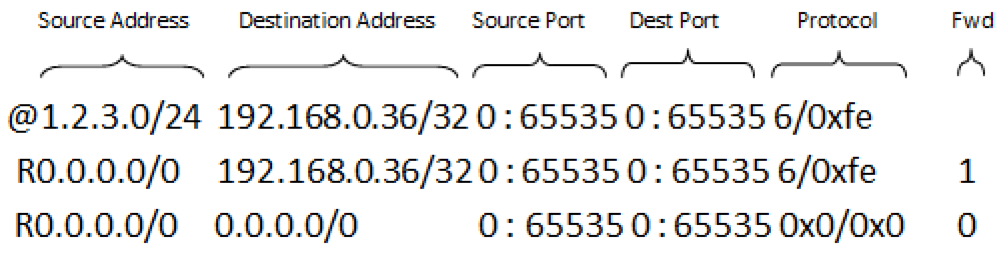
IPv4 addresses are specified in CIDR format as specified in RFC 4632. They consist of the dot notation for the address and a prefix length separated by '/'. For example, 192.168.0.34/32, where the address is 192.168.0.34 and the prefix length is 32.

Ports are specified as a range of 16-bit numbers in the format MIN:MAX, where MIN and MAX are the inclusive minimum and maximum values of the range. The range 0:65535 represents all possible ports in a range. When MIN and MAX are the same value, a single port is represented, for example, 20:20.

The protocol identifier is an 8-bit value and a mask separated by '/'. For example: 6/0xfe matches protocol values 6 and 7.

- Route rules start with a leading character 'R' and have the same format as ACL rules except an extra field at the tail that indicates the forwarding port number.

Rules File Example



Source Address	Destination Address	Source Port	Dest Port	Protocol	Fwd
@1.2.3.0/24	192.168.0.36/32	0 : 65535	0 : 65535	6/0xfe	
R0.0.0.0/0	192.168.0.36/32	0 : 65535	0 : 65535	6/0xfe	1
R0.0.0.0/0	0.0.0.0/0	0 : 65535	0 : 65535	0x0/0x0	0

Fig. 7.8: Rules example

Each rule is explained as follows:

- Rule 1 (the first line) tells the application to drop those packets with source IP address = [1.2.3.*], destination IP address = [192.168.0.36], protocol = [6]/[7]
- Rule 2 (the second line) is similar to Rule 1, except the source IP address is ignored. It tells the application to forward packets with destination IP address = [192.168.0.36], protocol = [6]/[7], destined to port 1.
- Rule 3 (the third line) tells the application to forward all packets to port 0. This is something like a default route entry.

As described earlier, the application assume rules are listed in descending order of priority, therefore Rule 1 has the highest priority, then Rule 2, and finally, Rule 3 has the lowest priority.

Consider the arrival of the following three packets:

- Packet 1 has source IP address = [1.2.3.4], destination IP address = [192.168.0.36], and protocol = [6]
- Packet 2 has source IP address = [1.2.4.4], destination IP address = [192.168.0.36], and protocol = [6]
- Packet 3 has source IP address = [1.2.3.4], destination IP address = [192.168.0.36], and protocol = [8]

Observe that:

- Packet 1 matches all of the rules
- Packet 2 matches Rule 2 and Rule 3
- Packet 3 only matches Rule 3

For priority reasons, Packet 1 matches Rule 1 and is dropped. Packet 2 matches Rule 2 and is forwarded to port 1. Packet 3 matches Rule 3 and is forwarded to port 0.

For more details on the rule file format, please refer to rule_ipv4.db and rule_ipv6.db files (inside <RTE_SDK>/examples/l3fwd-acl/).

Application Phases

Once the application starts, it transitions through three phases:

- **Initialization Phase** - Perform the following tasks:
- Parse command parameters. Check the validity of rule file(s) name(s), number of logical cores, receive and transmit queues. Bind ports, queues and logical cores. Check ACL search options, and so on.
- Call Environmental Abstraction Layer (EAL) and Poll Mode Driver (PMD) functions to initialize the environment and detect possible NICs. The EAL creates several threads and sets affinity to a specific hardware thread CPU based on the configuration specified by the command line arguments.
- Read the rule files and format the rules into the representation that the ACL library can recognize. Call the ACL library function to add the rules into the database and compile them as a trie of pattern sets. Note that application maintains a separate AC contexts for IPv4 and IPv6 rules.
- **Runtime Phase** - Process the incoming packets from a port. Packets are processed in three steps:
  - Retrieval: Gets a packet from the receive queue. Each logical core may process several queues for different ports. This depends on the configuration specified by command line arguments.
  - Lookup: Checks that the packet type is supported (IPv4/IPv6) and performs a 5-tuple lookup over corresponding AC context. If an ACL rule is matched, the packets will be dropped and return back to step 1. If a route rule is matched, it indicates the packet is not in the ACL list and should be forwarded. If there is no matches for the packet, then the packet is dropped.
  - Forwarding: Forwards the packet to the corresponding port.
- **Final Phase** - Perform the following tasks:

  Calls the EAL, PMD driver and ACL library to free resource, then quits.

7.17.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/l3fwd-acl
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK IPL Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.17.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd-acl [EAL options] -- -p PORTMASK [-P] --config(port,queue,lcore)[,(port,queue,lco
```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- -P: Sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- –config (port,queue,lcore)[,(port,queue,lcore)]: determines which queues from which ports are mapped to which cores
- –rule_ipv4 FILENAME: Specifies the IPv4 ACL and route rules file
- –rule_ipv6 FILENAME: Specifies the IPv6 ACL and route rules file
- –scalar: Use a scalar function to perform rule lookup
- –enable-jumbo: optional, enables jumbo frames
- –max-pkt-len: optional, maximum packet length in decimal (64-9600)
- –no-numa: optional, disables numa awareness

As an example, consider a dual processor socket platform where cores 0, 2, 4, 6, 8 and 10 appear on socket 0, while cores 1, 3, 5, 7, 9 and 11 appear on socket 1. Let's say that the user wants to use memory from both NUMA nodes, the platform has only two ports and the user wants to use two cores from each processor socket to do the packet processing.

To enable L3 forwarding between two ports, using two cores from each processor, while also taking advantage of local memory access by optimizing around NUMA, the user must enable two queues from each port, pin to the appropriate cores and allocate memory from the appropriate NUMA node. This is achieved using the following command:

```
./build/l3fwd-acl -c f -n 4 -- -p 0x3 --config="(0,0,0),(0,1,2),(1,0,1),(1,1,3)" --rule_ipv4=".
```

In this command:

- The -c option enables cores 0, 1, 2, 3
- The -p option enables ports 0 and 1
- The –config option enables two queues on each port and maps each (port,queue) pair to a specific core. Logic to enable multiple RX queues using RSS and to allocate memory from the correct NUMA nodes is included in the application and is done transparently. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	0	Map queue 0 from port 0 to lcore 0.
0	1	2	Map queue 1 from port 0 to lcore 2.
1	0	1	Map queue 0 from port 1 to lcore 1.
1	1	3	Map queue 1 from port 1 to lcore 3.

- The –rule_ipv4 option specifies the reading of IPv4 rules sets from the ./ rule_ipv4.db file.
- The –rule_ipv6 option specifies the reading of IPv6 rules sets from the ./ rule_ipv6.db file.

- The –scalar option specifies the performing of rule lookup with a scalar function.

7.17.4 Explanation

The following sections provide some explanation of the sample application code. The aspects of port, device and CPU configuration are similar to those of the L3 forwarding application (see Chapter 10, “L3 Forwarding Sample Application” for more information). The following sections describe aspects that are specific to L3 forwarding with access control.

Parse Rules from File

As described earlier, both ACL and route rules are assumed to be saved in the same file. The application parses the rules from the file and adds them to the database by calling the ACL library function. It ignores empty and comment lines, and parses and validates the rules it reads. If errors are detected, the application exits with messages to identify the errors encountered.

The application needs to consider the userdata and priority fields. The ACL rules save the index to the specific rules in the userdata field, while route rules save the forwarding port number. In order to differentiate the two types of rules, ACL rules add a signature in the userdata field. As for the priority field, the application assumes rules are organized in descending order of priority. Therefore, the code only decreases the priority number with each rule it parses.

Setting Up the ACL Context

For each supported AC rule format (IPv4 5-tuple, IPv6 6-tuple) application creates a separate context handler from the ACL library for each CPU socket on the board and adds parsed rules into that context.

Note, that for each supported rule type, application needs to calculate the expected offset of the fields from the start of the packet. That’s why only packets with fixed IPv4/ IPv6 header are supported. That allows to perform ACL classify straight over incoming packet buffer - no extra protocol field retrieval need to be performed.

Subsequently, the application checks whether NUMA is enabled. If it is, the application records the socket IDs of the CPU cores involved in the task.

Finally, the application creates contexts handler from the ACL library, adds rules parsed from the file into the database and build an ACL trie. It is important to note that the application creates an independent copy of each database for each socket CPU involved in the task to reduce the time for remote memory access.

7.18 L3 Forwarding in a Virtualization Environment Sample Application

The L3 Forwarding in a Virtualization Environment sample application is a simple example of packet processing using the DPDK. The application performs L3 forwarding that takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

7.18.1 Overview

The application demonstrates the use of the hash and LPM libraries in the DPDK to implement packet forwarding. The initialization and run-time paths are very similar to those of the L3 forwarding application (see Chapter 10 “L3 Forwarding Sample Application” for more information). The forwarding decision is taken based on information read from the input packet.

The lookup method is either hash-based or LPM-based and is selected at compile time. When the selected lookup method is hash-based, a hash object is used to emulate the flow classification stage. The hash object is used in correlation with the flow table to map each input packet to its flow at runtime.

The hash lookup key is represented by the DiffServ 5-tuple composed of the following fields read from the input packet: Source IP Address, Destination IP Address, Protocol, Source Port and Destination Port. The ID of the output interface for the input packet is read from the identified flow table entry. The set of flows used by the application is statically configured and loaded into the hash at initialization time. When the selected lookup method is LPM based, an LPM object is used to emulate the forwarding stage for IPv4 packets. The LPM object is used as the routing table to identify the next hop for each input packet at runtime.

The LPM lookup key is represented by the Destination IP Address field read from the input packet. The ID of the output interface for the input packet is the next hop returned by the LPM lookup. The set of LPM rules used by the application is statically configured and loaded into the LPM object at the initialization time.

Note: Please refer to Section 9.1.1 “Virtual Function Setup Instructions” for virtualized test case setup.

7.18.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/l3fwd-vf
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

Note: The compiled application is written to the build subdirectory. To have the application written to a different location, the O=/path/to/build/directory option may be specified in the make command.

7.18.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd-vf [EAL options] -- -p PORTMASK  --config(port,queue,lcore)[,(port,queue,lcore)]
```

where,

- –p PORTMASK: Hexadecimal bitmask of ports to configure
- –config (port,queue,lcore)[,(port,queue,lcore]: determines which queues from which ports are mapped to which cores
- –no-numa: optional, disables numa awareness

For example, consider a dual processor socket platform where cores 0,2,4,6, 8, and 10 appear on socket 0, while cores 1,3,5,7,9, and 11 appear on socket 1. Let's say that the programmer wants to use memory from both NUMA nodes, the platform has only two ports and the programmer wants to use one core from each processor socket to do the packet processing since only one Rx/Tx queue pair can be used in virtualization mode.

To enable L3 forwarding between two ports, using one core from each processor, while also taking advantage of local memory accesses by optimizing around NUMA, the programmer can pin to the appropriate cores and allocate memory from the appropriate NUMA node. This is achieved using the following command:

```
./build/l3fwd-vf -c 0x03 -n 3 -- -p 0x3 --config="(0,0,0),(1,0,1)"
```

In this command:

- The -c option enables cores 0 and 1
- The -p option enables ports 0 and 1
- The –config option enables one queue on each port and maps each (port,queue) pair to a specific core. Logic to enable multiple RX queues using RSS and to allocate memory from the correct NUMA nodes is included in the application and is done transparently. The following table shows the mapping in this example:

Port	Queue	lcore	Description
0	0	0	Map queue 0 from port 0 to lcore 0
1	1	1	Map queue 0 from port 1 to lcore 1

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.18.4 Explanation

The operation of this application is similar to that of the basic L3 Forwarding Sample Application. See Section 10.4 "Explanation" for more information.

7.19 Link Status Interrupt Sample Application

The Link Status Interrupt sample application is a simple example of packet processing using the Data Plane Development Kit (DPDK) that demonstrates how network link status changes for a network port can be captured and used by a DPDK application.

7.19.1 Overview

The Link Status Interrupt sample application registers a user space callback for the link status interrupt of each port and performs L2 forwarding for each packet that is received on an RX_PORT. The following operations are performed:

- RX_PORT and TX_PORT are paired with available ports one-by-one according to the core mask
- The source MAC address is replaced by the TX_PORT MAC address
- The destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID

This application can be used to demonstrate the usage of link status interrupt and its user space callbacks and the behavior of L2 forwarding each time the link status changes.

7.19.2 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/link_status_interrupt
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

Note: The compiled application is written to the build subdirectory. To have the application written to a different location, the O=/path/to/build/directory option may be specified on the make command line.

7.19.3 Running the Application

The application requires a number of command line options:

```
./build/link_status_interrupt [EAL options] -- -p PORTMASK [-q NQ][-T PERIOD]
```

where,

- -p PORTMASK: A hexadecimal bitmask of the ports to configure
- -q NQ: A number of queues (=ports) per lcore (default is 1)
- -T PERIOD: statistics will be refreshed each PERIOD seconds (0 to disable, 10 default)

To run the application in a linuxapp environment with 4 lcores, 4 memory channels, 16 ports and 8 RX queues per lcore, issue the command:

```
$ ./build/link_status_interrupt -c f -n 4-- -q 8 -p ffff
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.19.4 Explanation

The following sections provide some explanation of the code.

Command Line Arguments

The Link Status Interrupt sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments (see Section 13.3).

Command line parsing is done in the same way as it is done in the L2 Forwarding Sample Application. See Section 9.4.1, “Command Line Arguments” for more information.

Mbuf Pool Initialization

Mbuf pool initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See Section 9.4.2, “Mbuf Pool Initialization” for more information.

Driver Initialization

The main part of the code in the main() function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the Poll Mode Driver in the *DPDK Programmer’s Guide and the DPDK API Reference*.

```
if (rte_eal_pci_probe() < 0)
    rte_exit(EXIT_FAILURE, "Cannot probe PCI\n");

nb_ports = rte_eth_dev_count();
if (nb_ports == 0)
    rte_exit(EXIT_FAILURE, "No Ethernet ports - bye\n");

if (nb_ports > RTE_MAX_ETHPORTS)
    nb_ports = RTE_MAX_ETHPORTS;

/*
 * Each logical core is assigned a dedicated TX queue on each port.
 */

for (portid = 0; portid < nb_ports; portid++) {
    /* skip ports that are not enabled */

    if ((lsi_enabled_port_mask & (1 << portid)) == 0)
        continue;

    /* save the destination port id */

    if (nb_ports_in_mask % 2) {
        lsi_dst_ports[portid] = portid_last;
        lsi_dst_ports[portid_last] = portid;
    }
    else
        portid_last = portid;

    nb_ports_in_mask++;

    rte_eth_dev_info_get((uint8_t) portid, &dev_info);
}
```

Observe that:

- rte_eal_pci_probe() parses the devices on the PCI bus and initializes recognized devices.

The next step is to configure the RX and TX queues. For each port, there is only one RX queue (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. The rte_eth_dev_configure() function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t) portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: err=%d, port=%u\n", ret, portid);
```

The global configuration is stored in a static structure:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .split_hdr_size = 0,
        .header_split = 0,   /**< Header Split disabled */
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
        .hw_strip_crc= 0,    /**< CRC stripped by hardware */
    },
    .txmode = {},
    .intr_conf = {
        .lsc = 1, /**< link status interrupt feature enabled */
    },
};
```

Configuring lsc to 0 (the default) disables the generation of any link status change interrupts in kernel space and no user space interrupt event is received. The public interface rte_eth_link_get() accesses the NIC registers directly to update the link status. Configuring lsc to non-zero enables the generation of link status change interrupts in kernel space when a link status change is present and calls the user space callbacks registered by the application. The public interface rte_eth_link_get() just reads the link status in a global structure that would be updated in the interrupt host thread only.

Interrupt Callback Registration

The application can register one or more callbacks to a specific port and interrupt event. An example callback function that has been written as indicated below.

```
static void
lsi_event_callback(uint8_t port_id, enum rte_eth_event_type type, void *param)
{
    struct rte_eth_link link;

    RTE_SET_USED(param);

    printf("\n\nIn registered callback...\n");

    printf("Event type: %s\n", type == RTE_ETH_EVENT_INTR_LSC ? "LSC interrupt" : "unknown ever

    rte_eth_link_get_nowait(port_id, &link);

    if (link.link_status) {
        printf("Port %d Link Up - speed %u Mbps - %s\n\n", port_id, (unsigned)link.link_speed,
               (link.link_duplex == ETH_LINK_FULL_DUPLEX) ? ("full-duplex") : ("half-duplex"));
    } else
        printf("Port %d Link Down\n\n", port_id);
}
```

This function is called when a link status interrupt is present for the right port. The port_id indicates which port the interrupt applies to. The type parameter identifies the interrupt event type, which currently can be RTE_ETH_EVENT_INTR_LSC only, but other types can be added in the future. The param parameter is the address of the parameter for the callback. This function should be implemented with care since it will be called in the interrupt host thread, which is different from the main thread of its caller.

The application registers the lsi_event_callback and a NULL parameter to the link status interrupt event on each port:

```
rte_eth_dev_callback_register((uint8_t)portid, RTE_ETH_EVENT_INTR_LSC, lsi_event_callback, NULL
```

This registration can be done only after calling the rte_eth_dev_configure() function and before calling any other function. If lsc is initialized with 0, the callback is never called since no interrupt event would ever be present.

RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the -q option, which specifies the number of queues per lcore.

For example, if the user specifies -q 4, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is -p ffff), the application will need four lcores to poll all the ports.

```
ret = rte_eth_rx_queue_setup((uint8_t) portid, 0, nb_rxd, SOCKET0, &rx_conf, lsi_pktmbuf_pool);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_rx_queue_setup: err=%d, port=%u\n", ret, portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called struct lcore_queue_conf.

```
struct lcore_queue_conf {
    unsigned n_rx_port;
    unsigned rx_port_list[MAX_RX_QUEUE_PER_LCORE]; unsigned tx_queue_id;
    struct mbuf_table tx_mbufs[LSI_MAX_PORTS];
} rte_cache_aligned;

struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
```

The n_rx_port and rx_port_list[] fields are used in the main packet processing loop (see Section 13.4.7, "Receive, Process and Transmit Packets" later in this chapter).

The global configuration for the RX queues is stored in a static structure:

```
static const struct rte_eth_rxconf rx_conf = {
    .rx_thresh = {
        .pthresh = RX_PTHRESH,
        .hthresh = RX_HTHRESH,
        .wthresh = RX_WTHRESH,
    },
};
```

TX Queue Initialization

Each lcore should be able to transmit on any port. For every port, a single TX queue is initialized.

```
/* init one TX queue logical core on each port */

fflush(stdout);

ret = rte_eth_tx_queue_setup(portid, 0, nb_txd, rte_eth_dev_socket_id(portid), &tx_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup: err=%d,port=%u\n", ret, (unsigned) portid);
```

The global configuration for TX queues is stored in a static structure:

```
static const struct rte_eth_txconf tx_conf = {
    .tx_thresh = {
        .pthresh = TX_PTHRESH,
        .hthresh = TX_HTHRESH,
        .wthresh = TX_WTHRESH,
    },
    .tx_free_thresh = RTE_TEST_TX_DESC_DEFAULT + 1, /* disable feature */
};
```

Receive, Process and Transmit Packets

In the lsi_main_loop() function, the main task is to read ingress packets from the RX queues. This is done using the following code:

```
/*
 *   Read packet from RX queues
 */

for (i = 0; i < qconf->n_rx_port; i++) {
    portid = qconf->rx_port_list[i];
    nb_rx = rte_eth_rx_burst((uint8_t) portid, 0, pkts_burst, MAX_PKT_BURST);
    port_statistics[portid].rx += nb_rx;

    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        rte_prefetch0(rte_pktmbuf_mtod(m, void *));
        lsi_simple_forward(m, portid);
    }
}
```

Packets are read in a burst of size MAX_PKT_BURST. The rte_eth_rx_burst() function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the lsi_simple_forward() function. The processing is very simple: processes the TX port from the RX port and then replaces the source and destination MAC addresses.

Note: In the following code, the two lines for calculating the output port require some explanation. If portId is even, the first line does nothing (as portid & 1 will be 0), and the second line adds 1. If portId is odd, the first line subtracts one and the second line does nothing. Therefore, 0 goes to 1, and 1 to 0, 2 goes to 3 and 3 to 2, and so on.

```
static void
lsi_simple_forward(struct rte_mbuf *m, unsigned portid)
{
    struct ether_hdr *eth;
    void *tmp;
    unsigned dst_port = lsi_dst_ports[portid];
```

```
        eth = rte_pktmbuf_mtod(m, struct ether_hdr *);

        /* 02:00:00:00:00:xx */

        tmp = &eth->d_addr.addr_bytes[0];

        *((uint64_t *)tmp) = 0x000000000002 + (dst_port << 40);

        /* src addr */
        ether_addr_copy(&lsi_ports_eth_addr[dst_port], &eth->s_addr);

        lsi_send_packet(m, dst_port);
    }
```

Then, the packet is sent using the lsi_send_packet(m, dst_port) function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the lsi_send_burst() function directly from the main loop to send all the received packets on the same TX port using the burst-oriented send function, which is more efficient.

However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that so the same approach can be reused in a more complex application.

The lsi_send_packet() function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the lsi_send_burst() function:

```
/* Send the packet on an output interface */

static int
lsi_send_packet(struct rte_mbuf *m, uint8_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();
    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */

    if (unlikely(len == MAX_PKT_BURST)) {
        lsi_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }
    qconf->tx_mbufs[port].len = len;

    return 0;
}
```

To ensure that no packets remain in the tables, each lcore does a draining of the TX queue in its main loop. This technique introduces some latency when there are not many packets to send. However, it improves performance:

```
cur_tsc = rte_rdtsc();

/*
 *    TX burst queue drain
 */
```

```
    diff_tsc = cur_tsc - prev_tsc;

    if (unlikely(diff_tsc > drain_tsc)) {
        /* this could be optimized (use queueid instead of * portid), but it is not called so ofte

        for (portid = 0; portid < RTE_MAX_ETHPORTS; portid++) {
            if (qconf->tx_mbufs[portid].len == 0)
                continue;

            lsi_send_burst(&lcore_queue_conf[lcore_id],
            qconf->tx_mbufs[portid].len, (uint8_t) portid);
            qconf->tx_mbufs[portid].len = 0;
        }

        /* if timer is enabled */

        if (timer_period > 0) {
            /* advance the timer */

            timer_tsc += diff_tsc;

            /* if timer has reached its timeout */

            if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
                /* do this only on master core */

                if (lcore_id == rte_get_master_lcore()) {
                    print_stats();

                    /* reset the timer */
                    timer_tsc = 0;
                }
            }
        }
        prev_tsc = cur_tsc;
    }
```

7.20 Load Balancer Sample Application

The Load Balancer sample application demonstrates the concept of isolating the packet I/O task from the application-specific workload. Depending on the performance target, a number of logical cores (lcores) are dedicated to handle the interaction with the NIC ports (I/O lcores), while the rest of the lcores are dedicated to performing the application processing (worker lcores). The worker lcores are totally oblivious to the intricacies of the packet I/O activity and use the NIC-agnostic interface provided by software rings to exchange packets with the I/O cores.

7.20.1 Overview

The architecture of the Load Balance application is presented in the following figure.

For the sake of simplicity, the diagram illustrates a specific case of two I/O RX and two I/O TX lcores off loading the packet I/O overhead incurred by four NIC ports from four worker cores, with each I/O lcore handling RX/TX for two NIC ports.

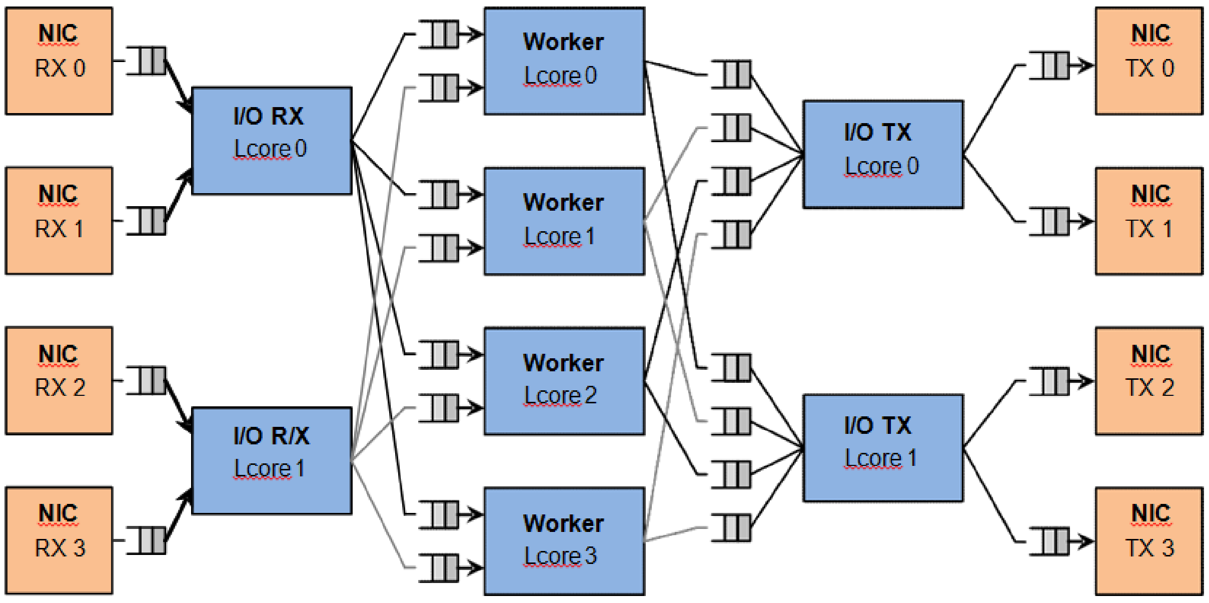


Fig. 7.9: Load Balancer Application Architecture

I/O RX Logical Cores

Each I/O RX lcore performs packet RX from its assigned NIC RX rings and then distributes the received packets to the worker threads. The application allows each I/O RX lcore to communicate with any of the worker threads, therefore each (I/O RX lcore, worker lcore) pair is connected through a dedicated single producer - single consumer software ring.

The worker lcore to handle the current packet is determined by reading a predefined 1-byte field from the input packet:

worker_id = packet[load_balancing_field] % n_workers

Since all the packets that are part of the same traffic flow are expected to have the same value for the load balancing field, this scheme also ensures that all the packets that are part of the same traffic flow are directed to the same worker lcore (flow affinity) in the same order they enter the system (packet ordering).

I/O TX Logical Cores

Each I/O lcore owns the packet TX for a predefined set of NIC ports. To enable each worker thread to send packets to any NIC TX port, the application creates a software ring for each (worker lcore, NIC TX port) pair, with each I/O TX core handling those software rings that are associated with NIC ports that it handles.

Worker Logical Cores

Each worker lcore reads packets from its set of input software rings and routes them to the NIC ports for transmission by dispatching them to output software rings. The routing logic is LPM based, with all the worker threads sharing the same LPM rules.

7.20.2 Compiling the Application

The sequence of steps used to build the application is:

1. Export the required environment variables:

   ```
   export RTE_SDK=<Path to the DPDK installation folder>
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

2. Build the application executable file:

   ```
   cd ${RTE_SDK}/examples/load_balancer
   make
   ```

   For more details on how to build the DPDK libraries and sample applications, please refer to the *DPDK Getting Started Guide.*

7.20.3 Running the Application

To successfully run the application, the command line used to start the application has to be in sync with the traffic flows configured on the traffic generator side.

For examples of application command lines and traffic generator flows, please refer to the DPDK Test Report. For more details on how to set up and run the sample applications provided with DPDK package, please refer to the *DPDK Getting Started Guide*.

7.20.4 Explanation

Application Configuration

The application run-time configuration is done through the application command line parameters. Any parameter that is not specified as mandatory is optional, with the default value hard-coded in the main.h header file from the application folder.

The list of application command line parameters is listed below:

1. –rx "(PORT, QUEUE, LCORE), ...": The list of NIC RX ports and queues handled by the I/O RX lcores. This parameter also implicitly defines the list of I/O RX lcores. This is a mandatory parameter.
2. –tx "(PORT, LCORE), ... ": The list of NIC TX ports handled by the I/O TX lcores. This parameter also implicitly defines the list of I/O TX lcores. This is a mandatory parameter.
3. –w "LCORE, ...": The list of the worker lcores. This is a mandatory parameter.
4. –lpm "IP / PREFIX => PORT; ...": The list of LPM rules used by the worker lcores for packet forwarding. This is a mandatory parameter.
5. –rsz "A, B, C, D": Ring sizes:
   (a) A = The size (in number of buffer descriptors) of each of the NIC RX rings read by the I/O RX lcores.
   (b) B = The size (in number of elements) of each of the software rings used by the I/O RX lcores to send packets to worker lcores.
   (c) C = The size (in number of elements) of each of the software rings used by the worker lcores to send packets to I/O TX lcores.

   (d) D = The size (in number of buffer descriptors) of each of the NIC TX rings written by I/O TX lcores.

6. –bsz “(A, B), (C, D), (E, F)”: Burst sizes:
   (a) A = The I/O RX lcore read burst size from NIC RX.
   (b) B = The I/O RX lcore write burst size to the output software rings.
   (c) C = The worker lcore read burst size from the input software rings.
   (d) D = The worker lcore write burst size to the output software rings.
   (e) E = The I/O TX lcore read burst size from the input software rings.
   (f) F = The I/O TX lcore write burst size to the NIC TX.
7. –pos-lb POS: The position of the 1-byte field within the input packet used by the I/O RX lcores to identify the worker lcore for the current packet. This field needs to be within the first 64 bytes of the input packet.

The infrastructure of software rings connecting I/O lcores and worker lcores is built by the application as a result of the application configuration provided by the user through the application command line parameters.

A specific lcore performing the I/O RX role for a specific set of NIC ports can also perform the I/O TX role for the same or a different set of NIC ports. A specific lcore cannot perform both the I/O role (either RX or TX) and the worker role during the same session.

Example:

```
./load_balancer -c 0xf8 -n 4 -- --rx "(0,0,3),(1,0,3)" --tx "(0,3),(1,3)" --w "4,5,6,7" --lpm "
```

There is a single I/O lcore (lcore 3) that handles RX and TX for two NIC ports (ports 0 and 1) that handles packets to/from four worker lcores (lcores 4, 5, 6 and 7) that are assigned worker IDs 0 to 3 (worker ID for lcore 4 is 0, for lcore 5 is 1, for lcore 6 is 2 and for lcore 7 is 3).

Assuming that all the input packets are IPv4 packets with no VLAN label and the source IP address of the current packet is A.B.C.D, the worker lcore for the current packet is determined by byte D (which is byte 29). There are two LPM rules that are used by each worker lcore to route packets to the output NIC ports.

The following table illustrates the packet flow through the system for several possible traffic flows:

Flow #	Source IP Address	Destination IP Address	Worker ID (Worker lcore)	Output NIC Port
1	0.0.0.0	1.0.0.1	0 (4)	0
2	0.0.0.1	1.0.1.2	1 (5)	1
3	0.0.0.14	1.0.0.3	2 (6)	0
4	0.0.0.15	1.0.1.4	3 (7)	1

NUMA Support

The application has built-in performance enhancements for the NUMA case:

1. One buffer pool per each CPU socket.
2. One LPM table per each CPU socket.

3. Memory for the NIC RX or TX rings is allocated on the same socket with the lcore handling the respective ring.

In the case where multiple CPU sockets are used in the system, it is recommended to enable at least one lcore to fulfill the I/O role for the NIC ports that are directly attached to that CPU socket through the PCI Express* bus. It is always recommended to handle the packet I/O with lcores from the same CPU socket as the NICs.

Depending on whether the I/O RX lcore (same CPU socket as NIC RX), the worker lcore and the I/O TX lcore (same CPU socket as NIC TX) handling a specific input packet, are on the same or different CPU sockets, the following run-time scenarios are possible:

1. AAA: The packet is received, processed and transmitted without going across CPU sockets.
2. AAB: The packet is received and processed on socket A, but as it has to be transmitted on a NIC port connected to socket B, the packet is sent to socket B through software rings.
3. ABB: The packet is received on socket A, but as it has to be processed by a worker lcore on socket B, the packet is sent to socket B through software rings. The packet is transmitted by a NIC port connected to the same CPU socket as the worker lcore that processed it.
4. ABC: The packet is received on socket A, it is processed by an lcore on socket B, then it has to be transmitted out by a NIC connected to socket C. The performance price for crossing the CPU socket boundary is paid twice for this packet.

7.21 Multi-process Sample Application

This chapter describes the example applications for multi-processing that are included in the DPDK.

7.21.1 Example Applications

Building the Sample Applications

The multi-process example applications are built in the same way as other sample applications, and as documented in the *DPDK Getting Started Guide*. To build all the example applications:

1. Set RTE_SDK and go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/multi_process
   ```

2. Set the target (a default target will be used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the applications:

   ```
   make
   ```

Note: If just a specific multi-process application needs to be built, the final make command can be run just in that application's directory, rather than at the top-level multi-process directory.

Basic Multi-process Example

The examples/simple_mp folder in the DPDK release contains a basic example application to demonstrate how two DPDK processes can work together using queues and memory pools to share information.

Running the Application

To run the application, start one copy of the simple_mp binary in one terminal, passing at least two cores in the coremask, as follows:

```
./build/simple_mp -c 3 -n 4 --proc-type=primary
```

For the first DPDK process run, the proc-type flag can be omitted or set to auto, since all DPDK processes will default to being a primary instance, meaning they have control over the hugepage shared memory regions. The process should start successfully and display a command prompt as follows:

```
$ ./build/simple_mp -c 3 -n 4 --proc-type=primary
EAL: coremask set to 3
EAL: Detected lcore 0 on socket 0
EAL: Detected lcore 1 on socket 0
EAL: Detected lcore 2 on socket 0
EAL: Detected lcore 3 on socket 0
...

EAL: Requesting 2 pages of size 1073741824
EAL: Requesting 768 pages of size 2097152
EAL: Ask a virtual area of 0x40000000 bytes
EAL: Virtual area found at 0x7ff200000000 (size = 0x40000000)
...

EAL: check igb_uio module
EAL: check module finished
EAL: Master core 0 is ready (tid=54e41820)
EAL: Core 1 is ready (tid=53b32700)

Starting core 1

simple_mp >
```

To run the secondary process to communicate with the primary process, again run the same binary setting at least two cores in the coremask:

```
./build/simple_mp -c C -n 4 --proc-type=secondary
```

When running a secondary process such as that shown above, the proc-type parameter can again be specified as auto. However, omitting the parameter altogether will cause the process to try and start as a primary rather than secondary process.

Once the process type is specified correctly, the process starts up, displaying largely similar status messages to the primary instance as it initializes. Once again, you will be presented with a command prompt.

Once both processes are running, messages can be sent between them using the send command. At any stage, either process can be terminated using the quit command.

```
EAL: Master core 10 is ready (tid=b5f89820)           EAL: Master core 8 is ready (tid=864a3820
EAL: Core 11 is ready (tid=84ffe700)                  EAL: Core 9 is ready (tid=85995700)
Starting core 11                                      Starting core 9
simple_mp > send hello_secondary                      simple_mp > core 9: Received 'hello_secor
simple_mp > core 11: Received 'hello_primary'         simple_mp > send hello_primary
simple_mp > quit                                      simple_mp > quit
```

Note: If the primary instance is terminated, the secondary instance must also be shut-down and restarted after the primary. This is necessary because the primary instance will clear and reset the shared memory regions on startup, invalidating the secondary process's pointers. The secondary process can be stopped and restarted without affecting the primary process.

How the Application Works

The core of this example application is based on using two queues and a single memory pool in shared memory. These three objects are created at startup by the primary process, since the secondary process cannot create objects in memory as it cannot reserve memory zones, and the secondary process then uses lookup functions to attach to these objects as it starts up.

```
if (rte_eal_process_type() == RTE_PROC_PRIMARY){
    send_ring = rte_ring_create(_PRI_2_SEC, ring_size, SOCKET0, flags);
    recv_ring = rte_ring_create(_SEC_2_PRI, ring_size, SOCKET0, flags);
    message_pool = rte_mempool_create(_MSG_POOL, pool_size, string_size, pool_cache, priv_data_
} else {
    recv_ring = rte_ring_lookup(_PRI_2_SEC);
    send_ring = rte_ring_lookup(_SEC_2_PRI);
    message_pool = rte_mempool_lookup(_MSG_POOL);
}
```

Note, however, that the named ring structure used as send_ring in the primary process is the recv_ring in the secondary process.

Once the rings and memory pools are all available in both the primary and secondary processes, the application simply dedicates two threads to sending and receiving messages respectively. The receive thread simply dequeues any messages on the receive ring, prints them, and frees the buffer space used by the messages back to the memory pool. The send thread makes use of the command-prompt library to interactively request user input for messages to send. Once a send command is issued by the user, a buffer is allocated from the memory pool, filled in with the message contents, then enqueued on the appropriate rte_ring.

Symmetric Multi-process Example

The second example of DPDK multi-process support demonstrates how a set of processes can run in parallel, with each process performing the same set of packet- processing operations. (Since each process is identical in functionality to the others, we refer to this as symmetric multi-processing, to differentiate it from asymmetric multi- processing - such as a client-server mode of operation seen in the next example, where different processes perform different tasks, yet co-operate to form a packet-processing system.) The following diagram shows the data-flow through the application, using two processes.

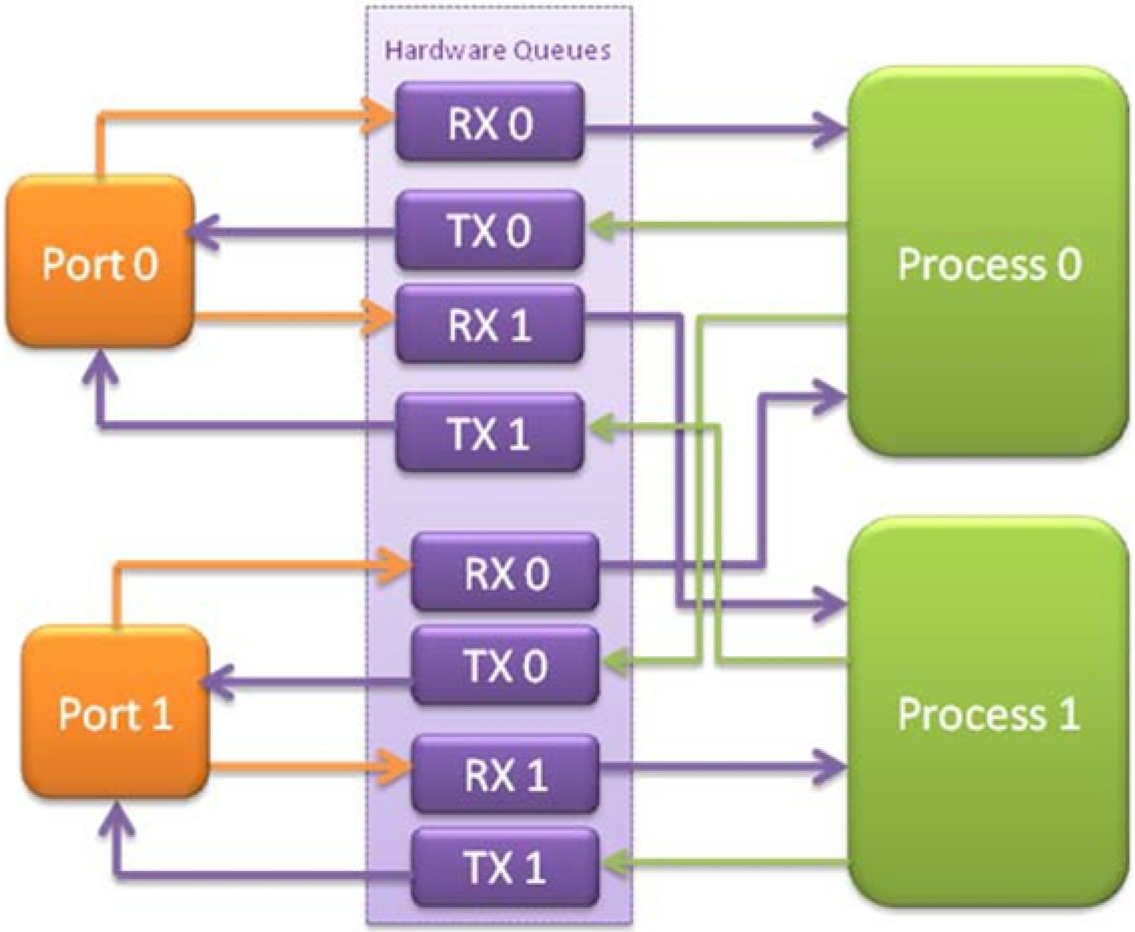


Fig. 7.10: Example Data Flow in a Symmetric Multi-process Application

As the diagram shows, each process reads packets from each of the network ports in use. RSS is used to distribute incoming packets on each port to different hardware RX queues. Each process reads a different RX queue on each port and so does not contend with any other process for that queue access. Similarly, each process writes outgoing packets to a different TX queue on each port.

Running the Application

As with the simple_mp example, the first instance of the symmetric_mp process must be run as the primary instance, though with a number of other application- specific parameters also provided after the EAL arguments. These additional parameters are:

- -p <portmask>, where portmask is a hexadecimal bitmask of what ports on the system are to be used. For example: -p 3 to use ports 0 and 1 only.
- –num-procs <N>, where N is the total number of symmetric_mp instances that will be run side-by-side to perform packet processing. This parameter is used to configure the appropriate number of receive queues on each network port.
- –proc-id <n>, where n is a numeric value in the range 0 <= n < N (number of processes, specified above). This identifies which symmetric_mp instance is being run, so that each process can read a unique receive queue on each network port.

The secondary symmetric_mp instances must also have these parameters specified, and the first two must be the same as those passed to the primary instance, or errors result.

For example, to run a set of four symmetric_mp instances, running on lcores 1-4, all performing level-2 forwarding of packets between ports 0 and 1, the following commands can be used (assuming run as root):

```
# ./build/symmetric_mp -c 2 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=0
# ./build/symmetric_mp -c 4 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=1
# ./build/symmetric_mp -c 8 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=2
# ./build/symmetric_mp -c 10 -n 4 --proc-type=auto -- -p 3 --num-procs=4 --proc-id=3
```

Note: In the above example, the process type can be explicitly specified as primary or secondary, rather than auto. When using auto, the first process run creates all the memory structures needed for all processes - irrespective of whether it has a proc-id of 0, 1, 2 or 3.

Note: For the symmetric multi-process example, since all processes work in the same manner, once the hugepage shared memory and the network ports are initialized, it is not necessary to restart all processes if the primary instance dies. Instead, that process can be restarted as a secondary, by explicitly setting the proc-type to secondary on the command line. (All subsequent instances launched will also need this explicitly specified, as auto-detection will detect no primary processes running and therefore attempt to re-initialize shared memory.)

How the Application Works

The initialization calls in both the primary and secondary instances are the same for the most part, calling the rte_eal_init(), 1 G and 10 G driver initialization and then rte_eal_pci_probe()

functions. Thereafter, the initialization done depends on whether the process is configured as a primary or secondary instance.

In the primary instance, a memory pool is created for the packet mbufs and the network ports to be used are initialized - the number of RX and TX queues per port being determined by the num-procs parameter passed on the command-line. The structures for the initialized network ports are stored in shared memory and therefore will be accessible by the secondary process as it initializes.

```
if (num_ports & 1)
   rte_exit(EXIT_FAILURE, "Application must use an even number of ports\n");

for(i = 0; i < num_ports; i++){
    if(proc_type == RTE_PROC_PRIMARY)
        if (smp_port_init(ports[i], mp, (uint16_t)num_procs) < 0)
            rte_exit(EXIT_FAILURE, "Error initializing ports\n");
}
```

In the secondary instance, rather than initializing the network ports, the port information exported by the primary process is used, giving the secondary process access to the hardware and software rings for each network port. Similarly, the memory pool of mbufs is accessed by doing a lookup for it by name:

```
mp = (proc_type == RTE_PROC_SECONDARY) ? rte_mempool_lookup(_SMP_MBUF_POOL) : rte_mempool_creat
```

Once this initialization is complete, the main loop of each process, both primary and secondary, is exactly the same - each process reads from each port using the queue corresponding to its proc-id parameter, and writes to the corresponding transmit queue on the output port.

Client-Server Multi-process Example

The third example multi-process application included with the DPDK shows how one can use a client-server type multi-process design to do packet processing. In this example, a single server process performs the packet reception from the ports being used and distributes these packets using round-robin ordering among a set of client processes, which perform the actual packet processing. In this case, the client applications just perform level-2 forwarding of packets by sending each packet out on a different network port.

The following diagram shows the data-flow through the application, using two client processes.

Running the Application

The server process must be run initially as the primary process to set up all memory structures for use by the clients. In addition to the EAL parameters, the application- specific parameters are:

- -p <portmask >, where portmask is a hexadecimal bitmask of what ports on the system are to be used. For example: -p 3 to use ports 0 and 1 only.
- -n <num-clients>, where the num-clients parameter is the number of client processes that will process the packets received by the server application.

Note: In the server process, a single thread, the master thread, that is, the lowest numbered lcore in the coremask, performs all packet I/O. If a coremask is specified with more than a

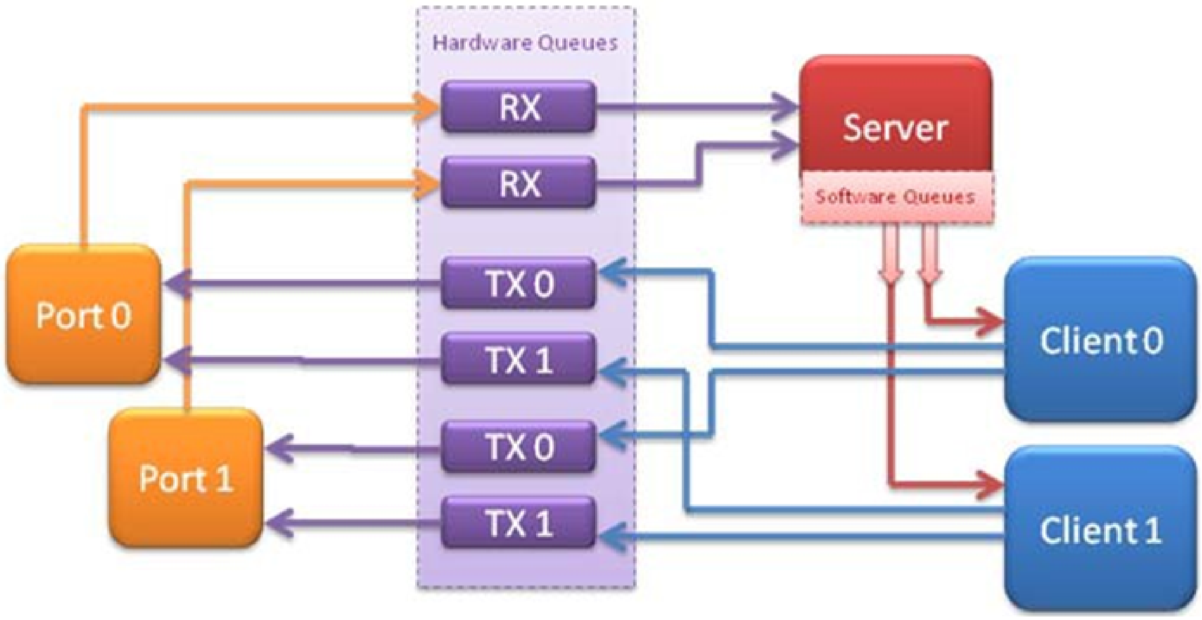


Fig. 7.11: Example Data Flow in a Client-Server Symmetric Multi-process Application

single lcore bit set in it, an additional lcore will be used for a thread to periodically print packet count statistics.

Since the server application stores configuration data in shared memory, including the network ports to be used, the only application parameter needed by a client process is its client instance ID. Therefore, to run a server application on lcore 1 (with lcore 2 printing statistics) along with two client processes running on lcores 3 and 4, the following commands could be used:

```
# ./mp_server/build/mp_server -c 6 -n 4 -- -p 3 -n 2
# ./mp_client/build/mp_client -c 8 -n 4 --proc-type=auto -- -n 0
# ./mp_client/build/mp_client -c 10 -n 4 --proc-type=auto -- -n 1
```

Note: If the server application dies and needs to be restarted, all client applications also need to be restarted, as there is no support in the server application for it to run as a secondary process. Any client processes that need restarting can be restarted without affecting the server process.

How the Application Works

The server process performs the network port and data structure initialization much as the symmetric multi-process application does when run as primary. One additional enhancement in this sample application is that the server process stores its port configuration data in a memory zone in hugepage shared memory. This eliminates the need for the client processes to have the portmask parameter passed into them on the command line, as is done for the symmetric multi-process application, and therefore eliminates mismatched parameters as a potential source of errors.

In the same way that the server process is designed to be run as a primary process instance only, the client processes are designed to be run as secondary instances only. They have no code to attempt to create shared memory objects. Instead, handles to all needed rings and memory pools are obtained via calls to rte_ring_lookup() and rte_mempool_lookup(). The network ports for use by the processes are obtained by loading the network port drivers and probing the PCI bus, which will, as in the symmetric multi-process example, automatically get access to the network ports using the settings already configured by the primary/server process.

Once all applications are initialized, the server operates by reading packets from each network port in turn and distributing those packets to the client queues (software rings, one for each client process) in round-robin order. On the client side, the packets are read from the rings in as big of bursts as possible, then routed out to a different network port. The routing used is very simple. All packets received on the first NIC port are transmitted back out on the second port and vice versa. Similarly, packets are routed between the 3rd and 4th network ports and so on. The sending of packets is done by writing the packets directly to the network ports; they are not transferred back via the server process.

In both the server and the client processes, outgoing packets are buffered before being sent, so as to allow the sending of multiple packets in a single burst to improve efficiency. For example, the client process will buffer packets to send, until either the buffer is full or until we receive no further packets from the server.

Master-slave Multi-process Example

The fourth example of DPDK multi-process support demonstrates a master-slave model that provide the capability of application recovery if a slave process crashes or meets unexpected conditions. In addition, it also demonstrates the floating process, which can run among different cores in contrast to the traditional way of binding a process/thread to a specific CPU core, using the local cache mechanism of mempool structures.

This application performs the same functionality as the L2 Forwarding sample application, therefore this chapter does not cover that part but describes functionality that is introduced in this multi-process example only. Please refer to Chapter 9, “L2 Forwarding Sample Application (in Real and Virtualized Environments)” for more information.

Unlike previous examples where all processes are started from the command line with input arguments, in this example, only one process is spawned from the command line and that process creates other processes. The following section describes this in more detail.

Master-slave Process Models

The process spawned from the command line is called the *master process* in this document. A process created by the master is called a *slave process*. The application has only one master process, but could have multiple slave processes.

Once the master process begins to run, it tries to initialize all the resources such as memory, CPU cores, driver, ports, and so on, as the other examples do. Thereafter, it creates slave processes, as shown in the following figure.

The master process calls the rte_eal_mp_remote_launch() EAL function to launch an application function for each pinned thread through the pipe. Then, it waits to check if any slave processes have exited. If so, the process tries to re-initialize the resources that belong to that slave and launch them in the pinned thread entry again. The following section describes the recovery procedures in more detail.

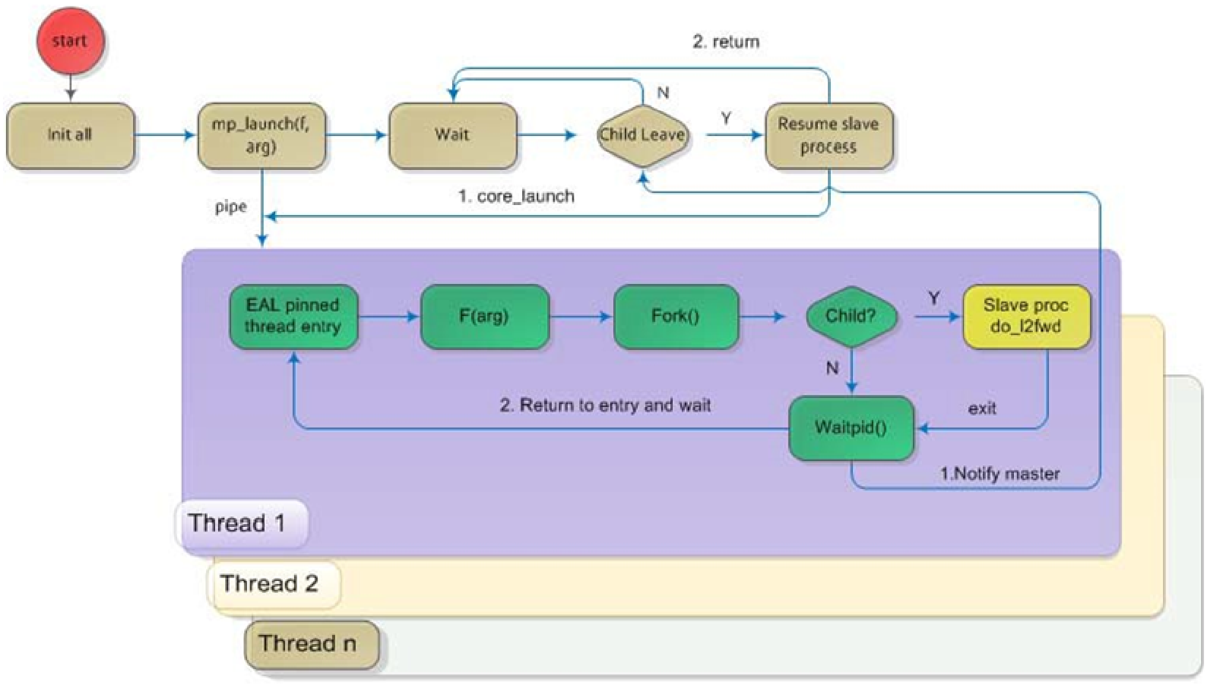


Fig. 7.12: Master-slave Process Workflow

For each pinned thread in EAL, after reading any data from the pipe, it tries to call the function that the application specified. In this master specified function, a fork() call creates a slave process that performs the L2 forwarding task. Then, the function waits until the slave exits, is killed or crashes. Thereafter, it notifies the master of this event and returns. Finally, the EAL pinned thread waits until the new function is launched.

After discussing the master-slave model, it is necessary to mention another issue, global and static variables.

For multiple-thread cases, all global and static variables have only one copy and they can be accessed by any thread if applicable. So, they can be used to sync or share data among threads.

In the previous examples, each process has separate global and static variables in memory and are independent of each other. If it is necessary to share the knowledge, some communication mechanism should be deployed, such as, memzone, ring, shared memory, and so on. The global or static variables are not a valid approach to share data among processes. For variables in this example, on the one hand, the slave process inherits all the knowledge of these variables after being created by the master. On the other hand, other processes cannot know if one or more processes modifies them after slave creation since that is the nature of a multiple process address space. But this does not mean that these variables cannot be used to share or sync data; it depends on the use case. The following are the possible use cases:

1. The master process starts and initializes a variable and it will never be changed after slave processes created. This case is OK.
2. After the slave processes are created, the master or slave cores need to change a variable, but other processes do not need to know the change. This case is also OK.
3. After the slave processes are created, the master or a slave needs to change a variable. In the meantime, one or more other process needs to be aware of the change. In this case, global and static variables cannot be used to share knowledge. Another communi-

cation mechanism is needed. A simple approach without lock protection can be a heap buffer allocated by rte_malloc or mem zone.

Slave Process Recovery Mechanism

Before talking about the recovery mechanism, it is necessary to know what is needed before a new slave instance can run if a previous one exited.

When a slave process exits, the system returns all the resources allocated for this process automatically. However, this does not include the resources that were allocated by the DPDK. All the hardware resources are shared among the processes, which include memzone, mempool, ring, a heap buffer allocated by the rte_malloc library, and so on. If the new instance runs and the allocated resource is not returned, either resource allocation failed or the hardware resource is lost forever.

When a slave process runs, it may have dependencies on other processes. They could have execution sequence orders; they could share the ring to communicate; they could share the same port for reception and forwarding; they could use lock structures to do exclusive access in some critical path. What happens to the dependent process(es) if the peer leaves? The consequence are varied since the dependency cases are complex. It depends on what the processed had shared. However, it is necessary to notify the peer(s) if one slave exited. Then, the peer(s) will be aware of that and wait until the new instance begins to run.

Therefore, to provide the capability to resume the new slave instance if the previous one exited, it is necessary to provide several mechanisms:

1. Keep a resource list for each slave process. Before a slave process run, the master should prepare a resource list. After it exits, the master could either delete the allocated resources and create new ones, or re-initialize those for use by the new instance.
2. Set up a notification mechanism for slave process exit cases. After the specific slave leaves, the master should be notified and then help to create a new instance. This mechanism is provided in Section 15.1.5.1, “Master-slave Process Models”.
3. Use a synchronization mechanism among dependent processes. The master should have the capability to stop or kill slave processes that have a dependency on the one that has exited. Then, after the new instance of exited slave process begins to run, the dependency ones could resume or run from the start. The example sends a STOP command to slave processes dependent on the exited one, then they will exit. Thereafter, the master creates new instances for the exited slave processes.

The following diagram describes slave process recovery.

Floating Process Support

When the DPDK application runs, there is always a -c option passed in to indicate the cores that are enabled. Then, the DPDK creates a thread for each enabled core. By doing so, it creates a 1:1 mapping between the enabled core and each thread. The enabled core always has an ID, therefore, each thread has a unique core ID in the DPDK execution environment. With the ID, each thread can easily access the structures or resources exclusively belonging to it without using function parameter passing. It can easily use the rte_lcore_id() function to get the value in every function that is called.

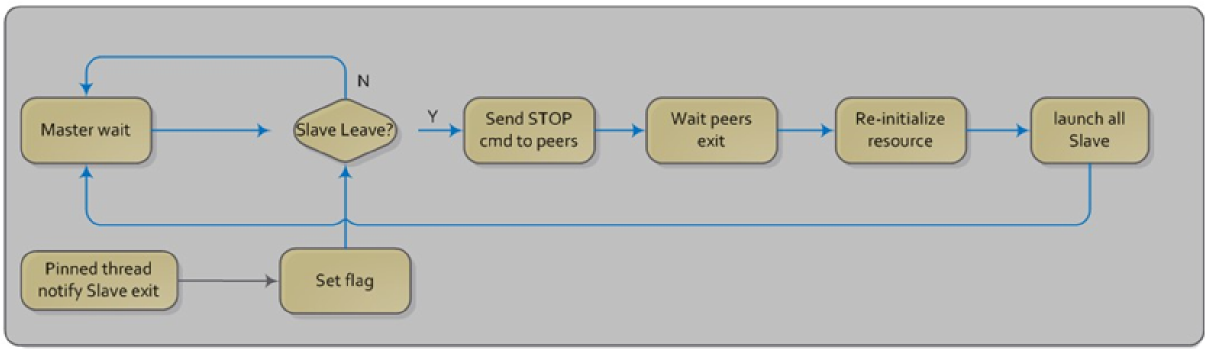


Fig. 7.13: Slave Process Recovery Process Flow

For threads/processes not created in that way, either pinned to a core or not, they will not own a unique ID and the rte_lcore_id() function will not work in the correct way. However, sometimes these threads/processes still need the unique ID mechanism to do easy access on structures or resources. For example, the DPDK mempool library provides a local cache mechanism (refer to *DPDK Programmer's Guide* , Section 6.4, "Local Cache") for fast element allocation and freeing. If using a non-unique ID or a fake one, a race condition occurs if two or more threads/ processes with the same core ID try to use the local cache.

Therefore, unused core IDs from the passing of parameters with the -c option are used to organize the core ID allocation array. Once the floating process is spawned, it tries to allocate a unique core ID from the array and release it on exit.

A natural way to spawn a floating process is to use the fork() function and allocate a unique core ID from the unused core ID array. However, it is necessary to write new code to provide a notification mechanism for slave exit and make sure the process recovery mechanism can work with it.

To avoid producing redundant code, the Master-Slave process model is still used to spawn floating processes, then cancel the affinity to specific cores. Besides that, clear the core ID assigned to the DPDK spawning a thread that has a 1:1 mapping with the core mask. Thereafter, get a new core ID from the unused core ID allocation array.

Run the Application

This example has a command line similar to the L2 Forwarding sample application with a few differences.

To run the application, start one copy of the l2fwd_fork binary in one terminal. Unlike the L2 Forwarding example, this example requires at least three cores since the master process will wait and be accountable for slave process recovery. The command is as follows:

```
#./build/l2fwd_fork -c 1c -n 4 -- -p 3 -f
```

This example provides another -f option to specify the use of floating process. If not specified, the example will use a pinned process to perform the L2 forwarding task.

To verify the recovery mechanism, proceed as follows: First, check the PID of the slave processes:

```
#ps -fe | grep l2fwd_fork
root 5136 4843 29 11:11 pts/1 00:00:05 ./build/l2fwd_fork
root 5145 5136 98 11:11 pts/1 00:00:11 ./build/l2fwd_fork
root 5146 5136 98 11:11 pts/1 00:00:11 ./build/l2fwd_fork
```

Then, kill one of the slaves:

```
#kill -9 5145
```

After 1 or 2 seconds, check whether the slave has resumed:

```
#ps -fe | grep l2fwd_fork
root 5136 4843 3 11:11 pts/1 00:00:06 ./build/l2fwd_fork
root 5247 5136 99 11:14 pts/1 00:00:01 ./build/l2fwd_fork
root 5248 5136 99 11:14 pts/1 00:00:01 ./build/l2fwd_fork
```

It can also monitor the traffic generator statics to see whether slave processes have resumed.

Explanation

As described in previous sections, not all global and static variables need to change to be accessible in multiple processes; it depends on how they are used. In this example, the statics info on packets dropped/forwarded/received count needs to be updated by the slave process, and the master needs to see the update and print them out. So, it needs to allocate a heap buffer using rte_zmalloc. In addition, if the -f option is specified, an array is needed to store the allocated core ID for the floating process so that the master can return it after a slave has exited accidentally.

```
static int
l2fwd_malloc_shared_struct(void)
{
    port_statistics = rte_zmalloc("port_stat", sizeof(struct l2fwd_port_statistics) * RTE_MAX_F

    if (port_statistics == NULL)
        return -1;

    /* allocate mapping_id array */

    if (float_proc) {
        int i;

        mapping_id = rte_malloc("mapping_id", sizeof(unsigned) * RTE_MAX_LCORE, 0);
        if (mapping_id == NULL)
            return -1;

        for (i = 0 ;i < RTE_MAX_LCORE; i++)
            mapping_id[i] = INVALID_MAPPING_ID;

    }
    return 0;
}
```

For each slave process, packets are received from one port and forwarded to another port that another slave is operating on. If the other slave exits accidentally, the port it is operating on may not work normally, so the first slave cannot forward packets to that port. There is a dependency on the port in this case. So, the master should recognize the dependency. The following is the code to detect this dependency:

```
for (portid = 0; portid < nb_ports; portid++) {
    /* skip ports that are not enabled */

    if ((l2fwd_enabled_port_mask & (1 << portid)) == 0)
        continue;
    /* Find pair ports' lcores */

    find_lcore = find_pair_lcore = 0;
    pair_port = l2fwd_dst_ports[portid];

    for (i = 0; i < RTE_MAX_LCORE; i++) {
        if (!rte_lcore_is_enabled(i))
            continue;

        for (j = 0; j < lcore_queue_conf[i].n_rx_port;j++) {
            if (lcore_queue_conf[i].rx_port_list[j] == portid) {
                lcore = i;
                find_lcore = 1;
                break;
            }

            if (lcore_queue_conf[i].rx_port_list[j] == pair_port) {
                pair_lcore = i;
                find_pair_lcore = 1;
                break;
            }
        }

        if (find_lcore && find_pair_lcore)
            break;
    }

    if (!find_lcore || !find_pair_lcore)
        rte_exit(EXIT_FAILURE, "Not find port=%d pair\\n", portid);

    printf("lcore %u and %u paired\\n", lcore, pair_lcore);

    lcore_resource[lcore].pair_id = pair_lcore;
    lcore_resource[pair_lcore].pair_id = lcore;
}
```

Before launching the slave process, it is necessary to set up the communication channel between the master and slave so that the master can notify the slave if its peer process with the dependency exited. In addition, the master needs to register a callback function in the case where a specific slave exited.

```
    for (i = 0; i < RTE_MAX_LCORE; i++) {
        if (lcore_resource[i].enabled) {
            /* Create ring for master and slave communication */

            ret = create_ms_ring(i);
            if (ret != 0)
                rte_exit(EXIT_FAILURE, "Create ring for lcore=%u failed",i);

            if (flib_register_slave_exit_notify(i,slave_exit_cb) != 0)
                rte_exit(EXIT_FAILURE, "Register master_trace_slave_exit failed");
        }
    }
```

After launching the slave process, the master waits and prints out the port statics periodically. If an event indicating that a slave process exited is detected, it sends the STOP command to the peer and waits until it has also exited. Then, it tries to clean up the execution environment and prepare new resources. Finally, the new slave instance is launched.

```
    while (1) {
        sleep(1);
        cur_tsc = rte_rdtsc();
        diff_tsc = cur_tsc - prev_tsc;

        /* if timer is enabled */

        if (timer_period > 0) {
            /* advance the timer */
            timer_tsc += diff_tsc;

            /* if timer has reached its timeout */
            if (unlikely(timer_tsc >= (uint64_t) timer_period)) {
                print_stats();

                /* reset the timer */
                timer_tsc = 0;
            }
        }

        prev_tsc = cur_tsc;

        /* Check any slave need restart or recreate */

        rte_spinlock_lock(&res_lock);

        for (i = 0; i < RTE_MAX_LCORE; i++) {
            struct lcore_resource_struct *res  = &lcore_resource[i];
            struct lcore_resource_struct *pair = &lcore_resource[res->pair_id];

            /* If find slave exited, try to reset pair */

            if (res->enabled && res->flags && pair->enabled) {
                if (!pair->flags) {
                    master_sendcmd_with_ack(pair->lcore_id, CMD_STOP);
                    rte_spinlock_unlock(&res_lock);
                    sleep(1);
                    rte_spinlock_lock(&res_lock);
                    if (pair->flags)
                        continue;
                }

                if (reset_pair(res->lcore_id, pair->lcore_id) != 0)
                    rte_exit(EXIT_FAILURE, "failed to reset slave");

                res->flags  = 0;
                pair->flags = 0;
            }
        }
        rte_spinlock_unlock(&res_lock);
    }
```

When the slave process is spawned and starts to run, it checks whether the floating process option is applied. If so, it clears the affinity to a specific core and also sets the unique core ID to 0. Then, it tries to allocate a new core ID. Since the core ID has changed, the resource allocated by the master cannot work, so it remaps the resource to the new core ID slot.

```
static int
l2fwd_launch_one_lcore( attribute ((unused)) void *dummy)
{
    unsigned lcore_id = rte_lcore_id();

    if (float_proc) {
        unsigned flcore_id;

        /* Change it to floating process, also change it's lcore_id */
```

```
            clear_cpu_affinity();

            RTE_PER_LCORE(_lcore_id) = 0;

            /* Get a lcore_id */

            if (flib_assign_lcore_id() < 0 ) {
                printf("flib_assign_lcore_id failed\n");
                return -1;
            }

            flcore_id = rte_lcore_id();

            /* Set mapping id, so master can return it after slave exited */

            mapping_id[lcore_id] = flcore_id;
            printf("Org lcore_id = %u, cur lcore_id = %u\n",lcore_id, flcore_id);
            remapping_slave_resource(lcore_id, flcore_id);
        }

        l2fwd_main_loop();

        /* return lcore_id before return */
        if (float_proc) {
            flib_free_lcore_id(rte_lcore_id());
            mapping_id[lcore_id] = INVALID_MAPPING_ID;
        }
        return 0;
    }
```

7.22 QoS Metering Sample Application

The QoS meter sample application is an example that demonstrates the use of DPDK to provide QoS marking and metering, as defined by RFC2697 for Single Rate Three Color Marker (srTCM) and RFC 2698 for Two Rate Three Color Marker (trTCM) algorithm.

7.22.1 Overview

The application uses a single thread for reading the packets from the RX port, metering, marking them with the appropriate color (green, yellow or red) and writing them to the TX port.

A policing scheme can be applied before writing the packets to the TX port by dropping or changing the color of the packet in a static manner depending on both the input and output colors of the packets that are processed by the meter.

The operation mode can be selected as compile time out of the following options:

- Simple forwarding
- srTCM color blind
- srTCM color aware
- srTCM color blind
- srTCM color aware

Please refer to RFC2697 and RFC2698 for details about the srTCM and trTCM configurable parameters (CIR, CBS and EBS for srTCM; CIR, PIR, CBS and PBS for trTCM).

The color blind modes are functionally equivalent with the color-aware modes when all the incoming packets are colored as green.

7.22.2 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/qos_meter
   ```

2. Set the target (a default target is used if not specified):

   Note: This application is intended as a linuxapp only.

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

3. Build the application:

   ```
   make
   ```

7.22.3 Running the Application

The application execution command line is as below:

```
./qos_meter [EAL options] -- -p PORTMASK
```

The application is constrained to use a single core in the EAL core mask and 2 ports only in the application port mask (first port from the port mask is used for RX and the other port in the core mask is used for TX).

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.22.4 Explanation

Selecting one of the metering modes is done with these defines:

```
#define APP_MODE_FWD    0
#define APP_MODE_SRTCM_COLOR_BLIND  1
#define APP_MODE_SRTCM_COLOR_AWARE  2
#define APP_MODE_TRTCM_COLOR_BLIND  3
#define APP_MODE_TRTCM_COLOR_AWARE  4

#define APP_MODE  APP_MODE_SRTCM_COLOR_BLIND
```

To simplify debugging (for example, by using the traffic generator RX side MAC address based packet filtering feature), the color is defined as the LSB byte of the destination MAC address.

The traffic meter parameters are configured in the application source code with following default values:

```
struct rte_meter_srtcm_params app_srtcm_params[] = {

    {.cir = 1000000 * 46, .cbs = 2048, .ebs = 2048},
```

```
};

struct rte_meter_trtcm_params app_trtcm_params[] = {

    {.cir = 1000000 * 46, .pir = 1500000 * 46, .cbs = 2048, .pbs = 2048},

};
```

Assuming the input traffic is generated at line rate and all packets are 64 bytes Ethernet frames (IPv4 packet size of 46 bytes) and green, the expected output traffic should be marked as shown in the following table:

Table 7.1: Output Traffic Marking

Mode	Green (Mpps)	Yellow (Mpps)	Red (Mpps)
srTCM blind	1	1	12.88
srTCM color	1	1	12.88
trTCM blind	1	0.5	13.38
trTCM color	1	0.5	13.38
FWD	14.88	0	0

To set up the policing scheme as desired, it is necessary to modify the main.h source file, where this policy is implemented as a static structure, as follows:

```
int policer_table[e_RTE_METER_COLORS][e_RTE_METER_COLORS] =
{
   { GREEN, RED, RED},
   { DROP, YELLOW, RED},
   { DROP, DROP, RED}
};
```

Where rows indicate the input color, columns indicate the output color, and the value that is stored in the table indicates the action to be taken for that particular case.

There are four different actions:

- GREEN: The packet's color is changed to green.
- YELLOW: The packet's color is changed to yellow.
- RED: The packet's color is changed to red.
- DROP: The packet is dropped.

In this particular case:

- Every packet which input and output color are the same, keeps the same color.
- Every packet which color has improved is dropped (this particular case can't happen, so these values will not be used).
- For the rest of the cases, the color is changed to red.

7.23 QoS Scheduler Sample Application

The QoS sample application demonstrates the use of the DPDK to provide QoS scheduling.

7.23.1 Overview

The architecture of the QoS scheduler application is shown in the following figure.

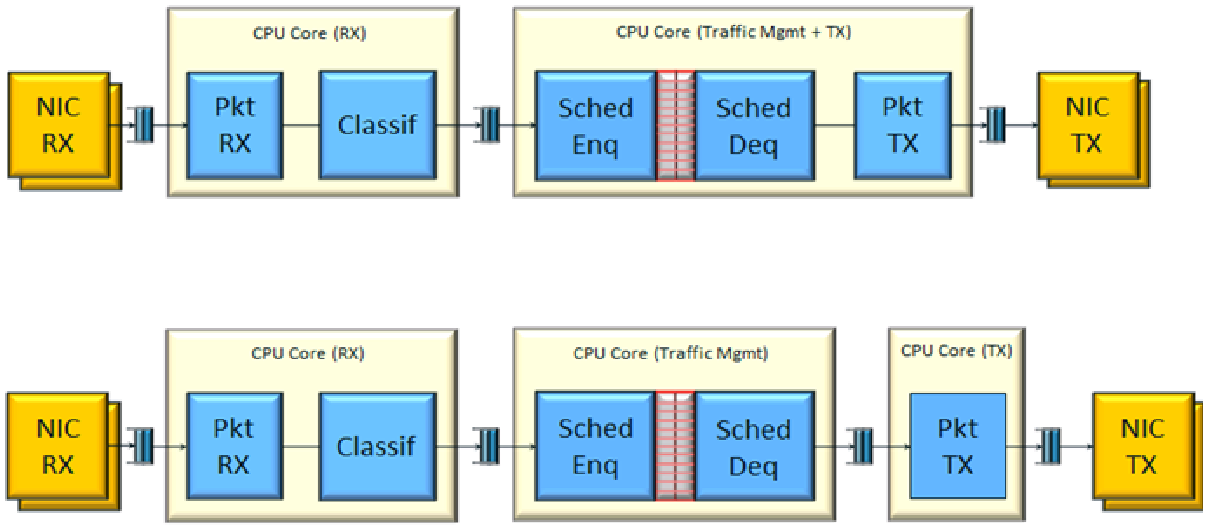


Fig. 7.14: QoS Scheduler Application Architecture

There are two flavors of the runtime execution for this application, with two or three threads per each packet flow configuration being used. The RX thread reads packets from the RX port, classifies the packets based on the double VLAN (outer and inner) and the lower two bytes of the IP destination address and puts them into the ring queue. The worker thread dequeues the packets from the ring and calls the QoS scheduler enqueue/dequeue functions. If a separate TX core is used, these are sent to the TX ring. Otherwise, they are sent directly to the TX port. The TX thread, if present, reads from the TX ring and write the packets to the TX port.

7.23.2 Compiling the Application

To compile the application:

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/qos_sched
   ```

2. Set the target (a default target is used if not specified). For example:

   Note: This application is intended as a linuxapp only.

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

3. Build the application:

   ```
   make
   ```

Note: To get statistics on the sample app using the command line interface as described in the next section, DPDK must be compiled defining *CONFIG_RTE_SCHED_COLLECT_STATS*, which can be done by changing the configuration file for the specific target to be compiled.

7.23.3 Running the Application

Note: In order to run the application, a total of at least 4 G of huge pages must be set up for each of the used sockets (depending on the cores in use).

The application has a number of command line options:

```
./qos_sched [EAL options] -- <APP PARAMS>
```

Mandatory application parameters include:

- –pfc “RX PORT, TX PORT, RX LCORE, WT LCORE, TX CORE”: Packet flow configuration. Multiple pfc entities can be configured in the command line, having 4 or 5 items (if TX core defined or not).

Optional application parameters include:

- -i: It makes the application to start in the interactive mode. In this mode, the application shows a command line that can be used for obtaining statistics while scheduling is taking place (see interactive mode below for more information).
- –mst n: Master core index (the default value is 1).
- –rsz “A, B, C”: Ring sizes:
- A = Size (in number of buffer descriptors) of each of the NIC RX rings read by the I/O RX lcores (the default value is 128).
- B = Size (in number of elements) of each of the software rings used by the I/O RX lcores to send packets to worker lcores (the default value is 8192).
- C = Size (in number of buffer descriptors) of each of the NIC TX rings written by worker lcores (the default value is 256)
- –bsz “A, B, C, D”: Burst sizes
- A = I/O RX lcore read burst size from the NIC RX (the default value is 64)
- B = I/O RX lcore write burst size to the output software rings, worker lcore read burst size from input software rings,QoS enqueue size (the default value is 64)
- C = QoS dequeue size (the default value is 32)
- D = Worker lcore write burst size to the NIC TX (the default value is 64)
- –msz M: Mempool size (in number of mbufs) for each pfc (default 2097152)
- –rth “A, B, C”: The RX queue threshold parameters
- A = RX prefetch threshold (the default value is 8)
- B = RX host threshold (the default value is 8)
- C = RX write-back threshold (the default value is 4)
- –tth “A, B, C”: TX queue threshold parameters
- A = TX prefetch threshold (the default value is 36)
- B = TX host threshold (the default value is 0)
- C = TX write-back threshold (the default value is 0)

- –cfg FILE: Profile configuration to load

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

The profile configuration file defines all the port/subport/pipe/traffic class/queue parameters needed for the QoS scheduler configuration.

The profile file has the following format:

```
; port configuration [port]

frame overhead = 24
number of subports per port = 1
number of pipes per subport = 4096
queue sizes = 64 64 64 64

; Subport configuration

[subport 0]
tb rate = 1250000000; Bytes per second
tb size = 1000000; Bytes
tc 0 rate = 1250000000;     Bytes per second
tc 1 rate = 1250000000;     Bytes per second
tc 2 rate = 1250000000;     Bytes per second
tc 3 rate = 1250000000;     Bytes per second
tc period = 10;             Milliseconds
tc oversubscription period = 10;     Milliseconds

pipe 0-4095 = 0;        These pipes are configured with pipe profile 0

; Pipe configuration

[pipe profile 0]
tb rate = 305175; Bytes per second
tb size = 1000000; Bytes

tc 0 rate = 305175; Bytes per second
tc 1 rate = 305175; Bytes per second
tc 2 rate = 305175; Bytes per second
tc 3 rate = 305175; Bytes per second
tc period = 40; Milliseconds

tc 0 oversubscription weight = 1
tc 1 oversubscription weight = 1
tc 2 oversubscription weight = 1
tc 3 oversubscription weight = 1

tc 0 wrr weights = 1 1 1 1
tc 1 wrr weights = 1 1 1 1
tc 2 wrr weights = 1 1 1 1
tc 3 wrr weights = 1 1 1 1

; RED params per traffic class and color (Green / Yellow / Red)

[red]
tc 0 wred min = 48 40 32
tc 0 wred max = 64 64 64
tc 0 wred inv prob = 10 10 10
tc 0 wred weight = 9 9 9

tc 1 wred min = 48 40 32
tc 1 wred max = 64 64 64
tc 1 wred inv prob = 10 10 10
tc 1 wred weight = 9 9 9

tc 2 wred min = 48 40 32
tc 2 wred max = 64 64 64
tc 2 wred inv prob = 10 10 10
tc 2 wred weight = 9 9 9

tc 3 wred min = 48 40 32
tc 3 wred max = 64 64 64
tc 3 wred inv prob = 10 10 10
tc 3 wred weight = 9 9 9
```

Interactive mode

These are the commands that are currently working under the command line interface:

- Control Commands
- –quit: Quits the application.
- General Statistics
  - stats app: Shows a table with in-app calculated statistics.
  - stats port X subport Y: For a specific subport, it shows the number of packets that went through the scheduler properly and the number of packets that were dropped. The same information is shown in bytes. The information is displayed in a table separating it in different traffic classes.
  - stats port X subport Y pipe Z: For a specific pipe, it shows the number of packets that went through the scheduler properly and the number of packets that were dropped. The same information is shown in bytes. This information is displayed in a table separating it in individual queues.
- Average queue size

All of these commands work the same way, averaging the number of packets throughout a specific subset of queues.

Two parameters can be configured for this prior to calling any of these commands:

- qavg n X: n is the number of times that the calculation will take place. Bigger numbers provide higher accuracy. The default value is 10.
- qavg period X: period is the number of microseconds that will be allowed between each calculation. The default value is 100.

The commands that can be used for measuring average queue size are:

- qavg port X subport Y: Show average queue size per subport.
- qavg port X subport Y tc Z: Show average queue size per subport for a specific traffic class.
- qavg port X subport Y pipe Z: Show average queue size per pipe.
- qavg port X subport Y pipe Z tc A: Show average queue size per pipe for a specific traffic class.
- qavg port X subport Y pipe Z tc A q B: Show average queue size of a specific queue.

Example

The following is an example command with a single packet flow configuration:

```
./qos_sched -c a2 -n 4 -- --pfc "3,2,5,7" --cfg ./profile.cfg
```

This example uses a single packet flow configuration which creates one RX thread on lcore 5 reading from port 3 and a worker thread on lcore 7 writing to port 2.

Another example with 2 packet flow configurations using different ports but sharing the same core for QoS scheduler is given below:

```
./qos_sched -c c6 -n 4 -- --pfc "3,2,2,6,7" --pfc "1,0,2,6,7" --cfg ./profile.cfg
```

Note that independent cores for the packet flow configurations for each of the RX, WT and TX thread are also supported, providing flexibility to balance the work.

The EAL coremask is constrained to contain the default mastercore 1 and the RX, WT and TX cores only.

7.23.4 Explanation

The Port/Subport/Pipe/Traffic Class/Queue are the hierarchical entities in a typical QoS application:

- A subport represents a predefined group of users.
- A pipe represents an individual user/subscriber.
- A traffic class is the representation of a different traffic type with a specific loss rate, delay and jitter requirements; such as data voice, video or data transfers.
- A queue hosts packets from one or multiple connections of the same type belonging to the same user.

The traffic flows that need to be configured are application dependent. This application classifies based on the QinQ double VLAN tags and the IP destination address as indicated in the following table.

Table 7.2: Entity Types

Level Name	**Siblings per Parent**	**QoS Functional Description**	**Selected By**
Port	•	Ethernet port	Physical port
Subport	Config (8)	Traffic shaped (token bucket)	Outer VLAN tag
Pipe	Config (4k)	Traffic shaped (token bucket)	Inner VLAN tag
Traffic Class	4	TCs of the same pipe services in strict priority	Destination IP address (0.0.X.0)
Queue	4	Queue of the same TC serviced in WRR	Destination IP address (0.0.0.X)

Please refer to the “QoS Scheduler” chapter in the *DPDK Programmer’s Guide* for more information about these parameters.

7.24 Intel® QuickAssist Technology Sample Application

This sample application demonstrates the use of the cryptographic operations provided by the Intel® QuickAssist Technology from within the DPDK environment. Therefore, building and running this application requires having both the DPDK and the QuickAssist Technology Software Library installed, as well as at least one Intel® QuickAssist Technology hardware device present in the system.

For this sample application, there is a dependency on either of:

- Intel® Communications Chipset 8900 to 8920 Series Software for Linux* package
- Intel® Communications Chipset 8925 to 8955 Series Software for Linux* package

7.24.1 Overview

An overview of the application is provided in Fig. 7.15. For simplicity, only two NIC ports and one Intel® QuickAssist Technology device are shown in this diagram, although the number of NIC ports and Intel® QuickAssist Technology devices can be different.

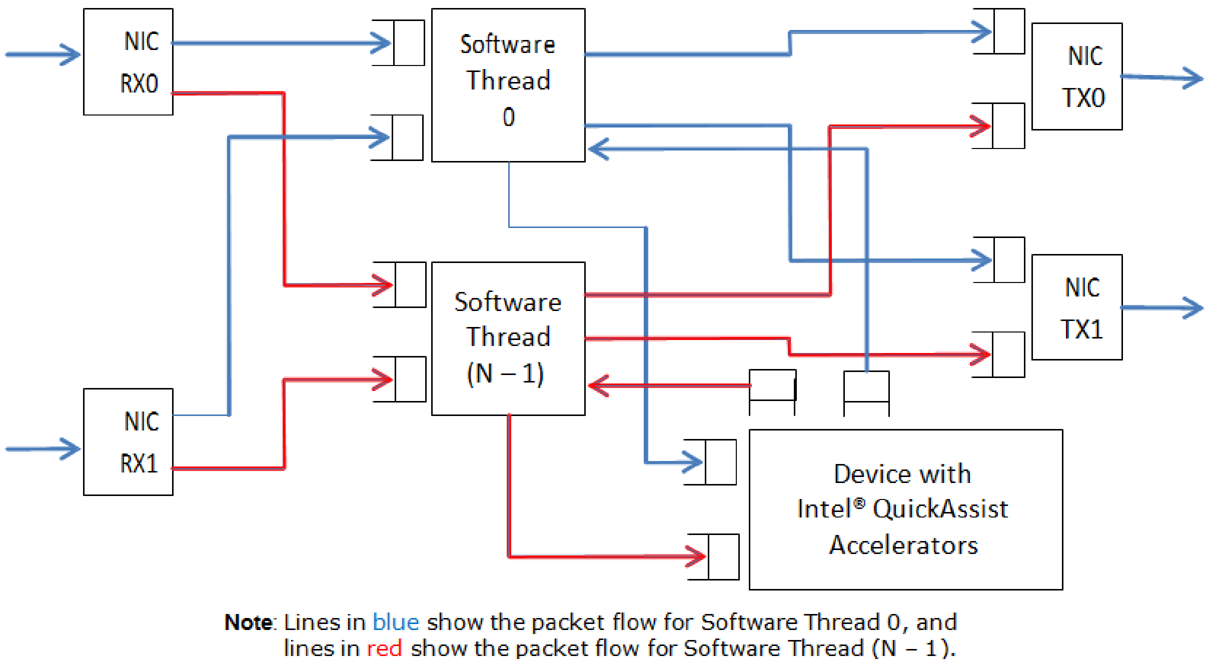

NIC RX0
Software Thread 0
NIC TX0
Software Thread (N – 1)
NIC TX1
NIC RX1
Device with Intel® QuickAssist Accelerators
Note: Lines in blue show the packet flow for Software Thread 0, and lines in red show the packet flow for Software Thread (N – 1).

Fig. 7.15: Intel® QuickAssist Technology Application Block Diagram

The application allows the configuration of the following items:

- Number of NIC ports
- Number of logical cores (lcores)
- Mapping of NIC RX queues to logical cores

Each lcore communicates with every cryptographic acceleration engine in the system through a pair of dedicated input - output queues. Each lcore has a dedicated NIC TX queue with every NIC port in the system. Therefore, each lcore reads packets from its NIC RX queues and cryptographic accelerator output queues and writes packets to its NIC TX queues and cryptographic accelerator input queues.

Each incoming packet that is read from a NIC RX queue is either directly forwarded to its destination NIC TX port (forwarding path) or first sent to one of the Intel® QuickAssist Technology devices for either encryption or decryption before being sent out on its destination NIC TX port (cryptographic path).

The application supports IPv4 input packets only. For each input packet, the decision between the forwarding path and the cryptographic path is taken at the classification stage based on the value of the IP source address field read from the input packet. Assuming that the IP source address is A.B.C.D, then if:

- D = 0: the forwarding path is selected (the packet is forwarded out directly)
- D = 1: the cryptographic path for encryption is selected (the packet is first encrypted and then forwarded out)
- D = 2: the cryptographic path for decryption is selected (the packet is first decrypted and then forwarded out)

For the cryptographic path cases (D = 1 or D = 2), byte C specifies the cipher algorithm and byte B the cryptographic hash algorithm to be used for the current packet. Byte A is not used and can be any value. The cipher and cryptographic hash algorithms supported by this application are listed in the crypto.h header file.

For each input packet, the destination NIC TX port is decided at the forwarding stage (executed after the cryptographic stage, if enabled for the packet) by looking at the RX port index of the dst_ports[ ] array, which was initialized at startup, being the outport the adjacent enabled port. For example, if ports 1,3,5 and 6 are enabled, for input port 1, outport port will be 3 and vice versa, and for input port 5, output port will be 6 and vice versa.

For the cryptographic path, it is the payload of the IPv4 packet that is encrypted or decrypted.

Setup

Building and running this application requires having both the DPDK package and the QuickAssist Technology Software Library installed, as well as at least one Intel® QuickAssist Technology hardware device present in the system.

For more details on how to build and run DPDK and Intel® QuickAssist Technology applications, please refer to the following documents:

- *DPDK Getting Started Guide*
- Intel® Communications Chipset 8900 to 8920 Series Software for Linux* Getting Started Guide (440005)
- Intel® Communications Chipset 8925 to 8955 Series Software for Linux* Getting Started Guide (523128)

For more details on the actual platforms used to validate this application, as well as performance numbers, please refer to the Test Report, which is accessible by contacting your Intel representative.

7.24.2 Building the Application

Steps to build the application:

1. Set up the following environment variables:

   ```
   export RTE_SDK=<Absolute path to the DPDK installation folder>
   export ICP_ROOT=<Absolute path to the Intel QAT installation folder>
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   Refer to the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   cd ${RTE_SDK}/examples/dpdk_qat
   make
   ```

7.24.3 Running the Application

Intel® QuickAssist Technology Configuration Files

The Intel® QuickAssist Technology configuration files used by the application are located in the config_files folder in the application folder. There following sets of configuration files are included in the DPDK package:

- Stargo CRB (single CPU socket): located in the stargo folder
  - dh89xxcc_qa_dev0.conf
- Shumway CRB (dual CPU socket): located in the shumway folder
  - dh89xxcc_qa_dev0.conf
  - dh89xxcc_qa_dev1.conf
- Coleto Creek: located in the coleto folder
  - dh895xcc_qa_dev0.conf

The relevant configuration file(s) must be copied to the /etc/ directory.

Please note that any change to these configuration files requires restarting the Intel® QuickAssist Technology driver using the following command:

```
# service qat_service restart
```

Refer to the following documents for information on the Intel® QuickAssist Technology configuration files:

- Intel® Communications Chipset 8900 to 8920 Series Software Programmer's Guide
- Intel® Communications Chipset 8925 to 8955 Series Software Programmer's Guide
- Intel® Communications Chipset 8900 to 8920 Series Software for Linux* Getting Started Guide.
- Intel® Communications Chipset 8925 to 8955 Series Software for Linux* Getting Started Guide.

Traffic Generator Setup and Application Startup

The application has a number of command line options:

> dpdk_qat [EAL options] – -p PORTMASK [–no-promisc] [–config ‘(port,queue,lcore)[,(port,queue,lcore)]’]

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure
- –no-promisc: Disables promiscuous mode for all ports, so that only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted. By default promiscuous mode is enabled so that packets are accepted regardless of the packet’s Ethernet MAC destination address.
- –config’(port,queue,lcore)[,(port,queue,lcore)]’: determines which queues from which ports are mapped to which cores.

Refer to Chapter 10 , “L3 Forwarding Sample Application” for more detailed descriptions of the –config command line option.

As an example, to run the application with two ports and two cores, which are using different Intel® QuickAssist Technology execution engines, performing AES-CBC-128 encryption with AES-XCBC-MAC-96 hash, the following settings can be used:

- Traffic generator source IP address: 0.9.6.1
- Command line:

  ```
  ./build/dpdk_qat -c 0xff -n 2 -- -p 0x3 --config '(0,0,1),(1,0,2)'
  ```

Refer to the *DPDK Test Report* for more examples of traffic generator setup and the application startup command lines. If no errors are generated in response to the startup commands, the application is running correctly.

7.25 Quota and Watermark Sample Application

The Quota and Watermark sample application is a simple example of packet processing using Data Plane Development Kit (DPDK) that showcases the use of a quota as the maximum number of packets enqueue/dequeue at a time and low and high watermarks to signal low and high ring usage respectively.

Additionally, it shows how ring watermarks can be used to feedback congestion notifications to data producers by temporarily stopping processing overloaded rings and sending Ethernet flow control frames.

This sample application is split in two parts:

- qw - The core quota and watermark sample application
- qwctl - A command line tool to alter quota and watermarks while qw is running

7.25.1 Overview

The Quota and Watermark sample application performs forwarding for each packet that is received on a given port. The destination port is the adjacent port from the enabled port mask, that is, if the first four ports are enabled (port mask 0xf), ports 0 and 1 forward into each other, and ports 2 and 3 forward into each other. The MAC addresses of the forwarded Ethernet frames are not affected.

Internally, packets are pulled from the ports by the master logical core and put on a variable length processing pipeline, each stage of which being connected by rings, as shown in Fig. 7.16.

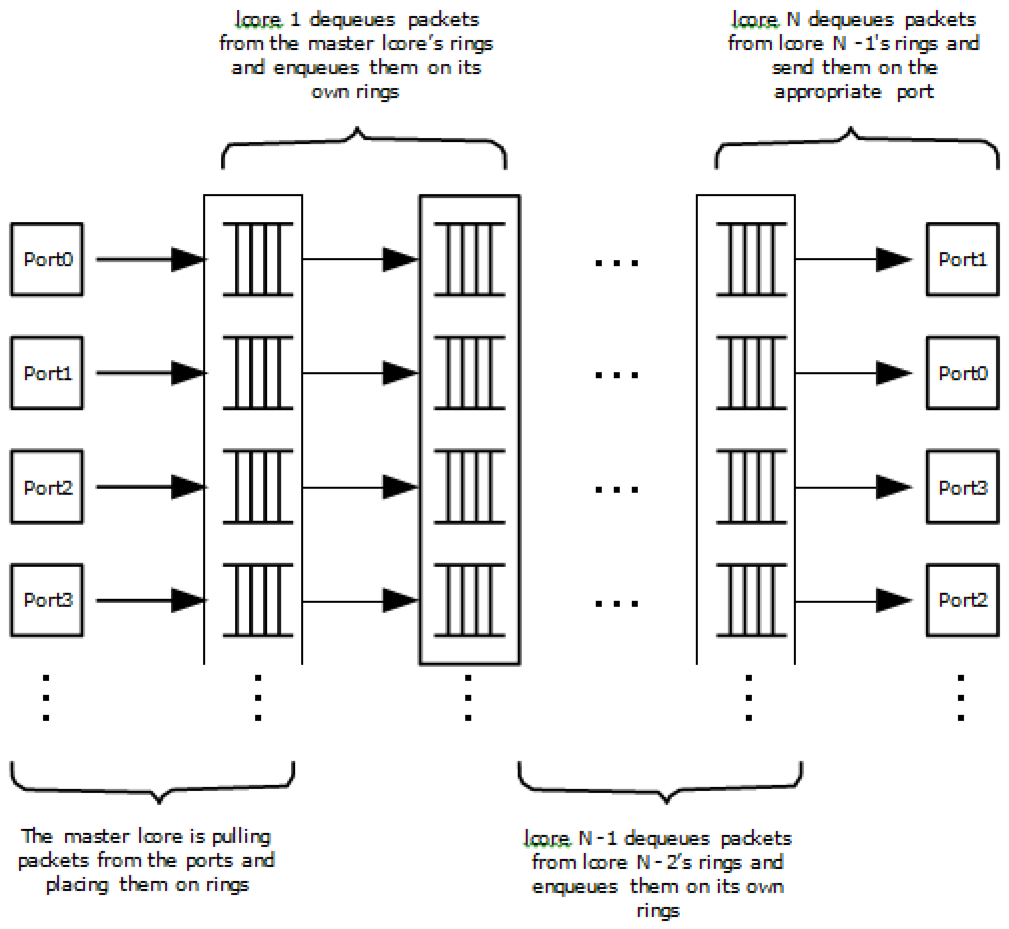

lcore 1 dequeues packets from the master lcore's rings and enqueues them on its own rings
lcore N dequeues packets from lcore N -1's rings and send them on the appropriate port
Port0
Port1
Port2
Port3
Port1
Port0
Port3
Port2
The master lcore is pulling packets from the ports and placing them on rings
lcore N-1 dequeues packets from lcore N-2's rings and enqueues them on its own rings

Fig. 7.16: Pipeline Overview

An adjustable quota value controls how many packets are being moved through the pipeline per enqueue and dequeue. Adjustable watermark values associated with the rings control a back-off mechanism that tries to prevent the pipeline from being overloaded by:

- Stopping enqueuing on rings for which the usage has crossed the high watermark threshold
- Sending Ethernet pause frames
- Only resuming enqueuing on a ring once its usage goes below a global low watermark threshold

This mechanism allows congestion notifications to go up the ring pipeline and eventually lead to an Ethernet flow control frame being send to the source.

On top of serving as an example of quota and watermark usage, this application can be used to benchmark ring based processing pipelines performance using a traffic- generator, as shown in Fig. 7.17.

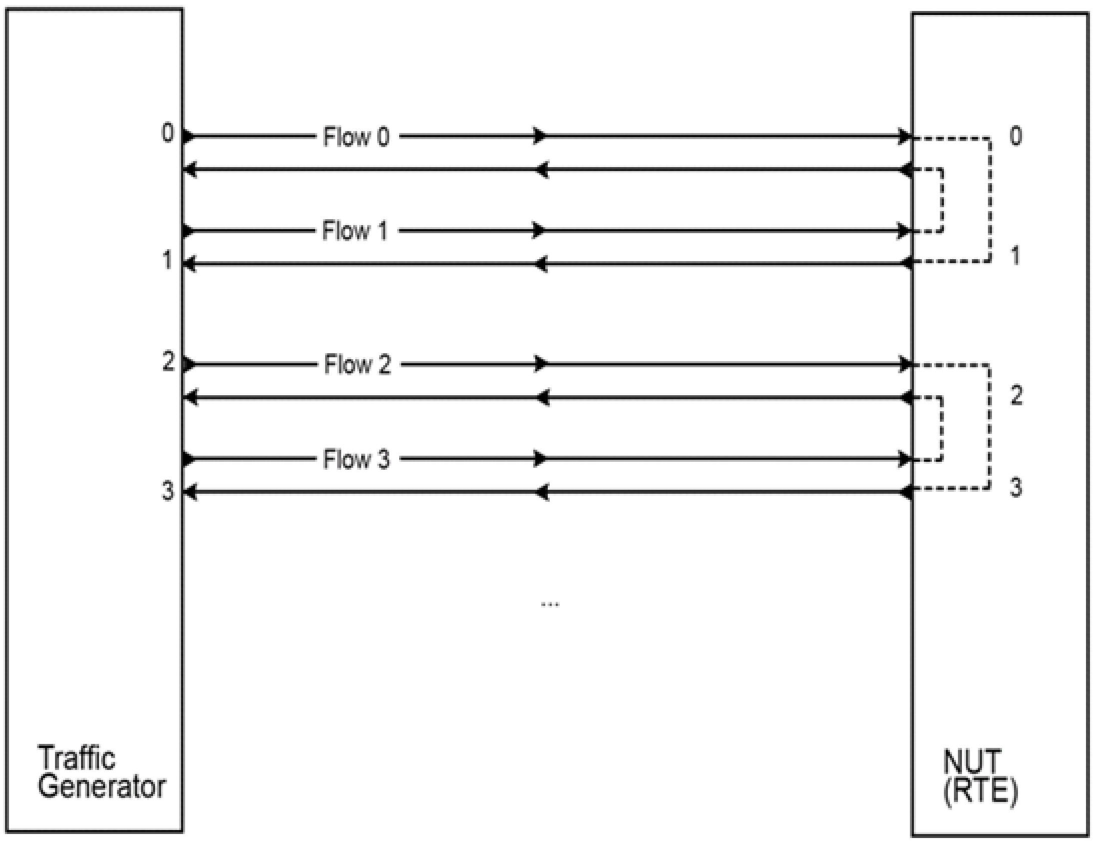


Fig. 7.17: Ring-based Processing Pipeline Performance Setup

7.25.2 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/quota_watermark
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.25.3 Running the Application

The core application, qw, has to be started first.

Once it is up and running, one can alter quota and watermarks while it runs using the control application, qwctl.

Running the Core Application

The application requires a single command line option:

```
./qw/build/qw [EAL options] -- -p PORTMASK
```

where,

-p PORTMASK: A hexadecimal bitmask of the ports to configure

To run the application in a linuxapp environment with four logical cores and ports 0 and 2, issue the following command:

```
./qw/build/qw -c f -n 4 -- -p 5
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

Running the Control Application

The control application requires a number of command line options:

```
./qwctl/build/qwctl [EAL options] --proc-type=secondary
```

The –proc-type=secondary option is necessary for the EAL to properly initialize the control application to use the same huge pages as the core application and thus be able to access its rings.

To run the application in a linuxapp environment on logical core 0, issue the following command:

```
./qwctl/build/qwctl -c 1 -n 4 --proc-type=secondary
```

Refer to the *DPDK Getting Started* Guide for general information on running applications and the Environment Abstraction Layer (EAL) options.

qwctl is an interactive command line that let the user change variables in a running instance of qw. The help command gives a list of available commands:

```
$ qwctl > help
```

7.25.4 Code Overview

The following sections provide a quick guide to the application's source code.

Core Application - qw

EAL and Drivers Setup

The EAL arguments are parsed at the beginning of the main() function:

```
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot initialize EAL\n");

argc -= ret;
argv += ret;
```

Then, a call to init_dpdk(), defined in init.c, is made to initialize the poll mode drivers:

```
void
init_dpdk(void)
{
    int ret;
```

```
        /* Bind the drivers to usable devices */

        ret = rte_eal_pci_probe();
        if (ret < 0)
            rte_exit(EXIT_FAILURE, "rte_eal_pci_probe(): error %d\n", ret);

        if (rte_eth_dev_count() < 2)
            rte_exit(EXIT_FAILURE, "Not enough Ethernet port available\n");
    }
```

To fully understand this code, it is recommended to study the chapters that relate to the *Poll Mode Driver* in the *DPDK Getting Started Guide* and the *DPDK API Reference*.

Shared Variables Setup

The quota and low_watermark shared variables are put into an rte_memzone using a call to setup_shared_variables():

```
    void
    setup_shared_variables(void)
    {
        const struct rte_memzone *qw_memzone;

        qw_memzone = rte_memzone_reserve(QUOTA_WATERMARK_MEMZONE_NAME, 2 * sizeof(int), rte_socket

        if (qw_memzone == NULL)
            rte_exit(EXIT_FAILURE, "%s\n", rte_strerror(rte_errno));

        quota = qw_memzone->addr;
        low_watermark = (unsigned int *) qw_memzone->addr + sizeof(int);
    }
```

These two variables are initialized to a default value in main() and can be changed while qw is running using the qwctl control program.

Application Arguments

The qw application only takes one argument: a port mask that specifies which ports should be used by the application. At least two ports are needed to run the application and there should be an even number of ports given in the port mask.

The port mask parsing is done in parse_qw_args(), defined in args.c.

Mbuf Pool Initialization

Once the application's arguments are parsed, an mbuf pool is created. It contains a set of mbuf objects that are used by the driver and the application to store network packets:

```
/* Create a pool of mbuf to store packets */

mbuf_pool = rte_mempool_create("mbuf_pool", MBUF_PER_POOL, MBUF_SIZE, 32, sizeof(struct rte_pkt
    rte_pktmbuf_pool_init, NULL, rte_pktmbuf_init, NULL, rte_socket_id(), 0);

if (mbuf_pool == NULL)
    rte_panic("%s\n", rte_strerror(rte_errno));
```

The rte_mempool is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver, which expects to have some reserved space in the mempool structure, sizeof(struct rte_pktmbuf_pool_private) bytes.

The number of allocated pkt mbufs is MBUF_PER_POOL, with a size of MBUF_SIZE each. A per-lcore cache of 32 mbufs is kept. The memory is allocated in on the master lcore's socket, but it is possible to extend this code to allocate one mbuf pool per socket.

Two callback pointers are also given to the rte_mempool_create() function:

- The first callback pointer is to rte_pktmbuf_pool_init() and is used to initialize the private data of the mempool, which is needed by the driver. This function is provided by the mbuf API, but can be copied and extended by the developer.
- The second callback pointer given to rte_mempool_create() is the mbuf initializer.

The default is used, that is, rte_pktmbuf_init(), which is provided in the rte_mbuf library. If a more complex application wants to extend the rte_pktmbuf structure for its own needs, a new function derived from rte_pktmbuf_init() can be created.

Ports Configuration and Pairing

Each port in the port mask is configured and a corresponding ring is created in the master lcore's array of rings. This ring is the first in the pipeline and will hold the packets directly coming from the port.

```
for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++)
    if (is_bit_set(port_id, portmask)) {
        configure_eth_port(port_id);
        init_ring(master_lcore_id, port_id);
    }

pair_ports();
```

The configure_eth_port() and init_ring() functions are used to configure a port and a ring respectively and are defined in init.c. They make use of the DPDK APIs defined in rte_eth.h and rte_ring.h.

pair_ports() builds the port_pairs[] array so that its key-value pairs are a mapping between reception and transmission ports. It is defined in init.c.

Logical Cores Assignment

The application uses the master logical core to poll all the ports for new packets and enqueue them on a ring associated with the port.

Each logical core except the last runs pipeline_stage() after a ring for each used port is initialized on that core. pipeline_stage() on core X dequeues packets from core X-1's rings and enqueue them on its own rings. See Fig. 7.18.

```
/* Start pipeline_stage() on all the available slave lcore but the last */

for (lcore_id = 0 ; lcore_id < last_lcore_id; lcore_id++) {
    if (rte_lcore_is_enabled(lcore_id) && lcore_id != master_lcore_id) {
        for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++)
            if (is_bit_set(port_id, portmask))
                init_ring(lcore_id, port_id);
            rte_eal_remote_launch(pipeline_stage, NULL, lcore_id);
    }
}
```

The last available logical core runs send_stage(), which is the last stage of the pipeline dequeuing packets from the last ring in the pipeline and sending them out on the destination port setup by pair_ports().

```
/* Start send_stage() on the last slave core */

rte_eal_remote_launch(send_stage, NULL, last_lcore_id);
```

Receive, Process and Transmit Packets

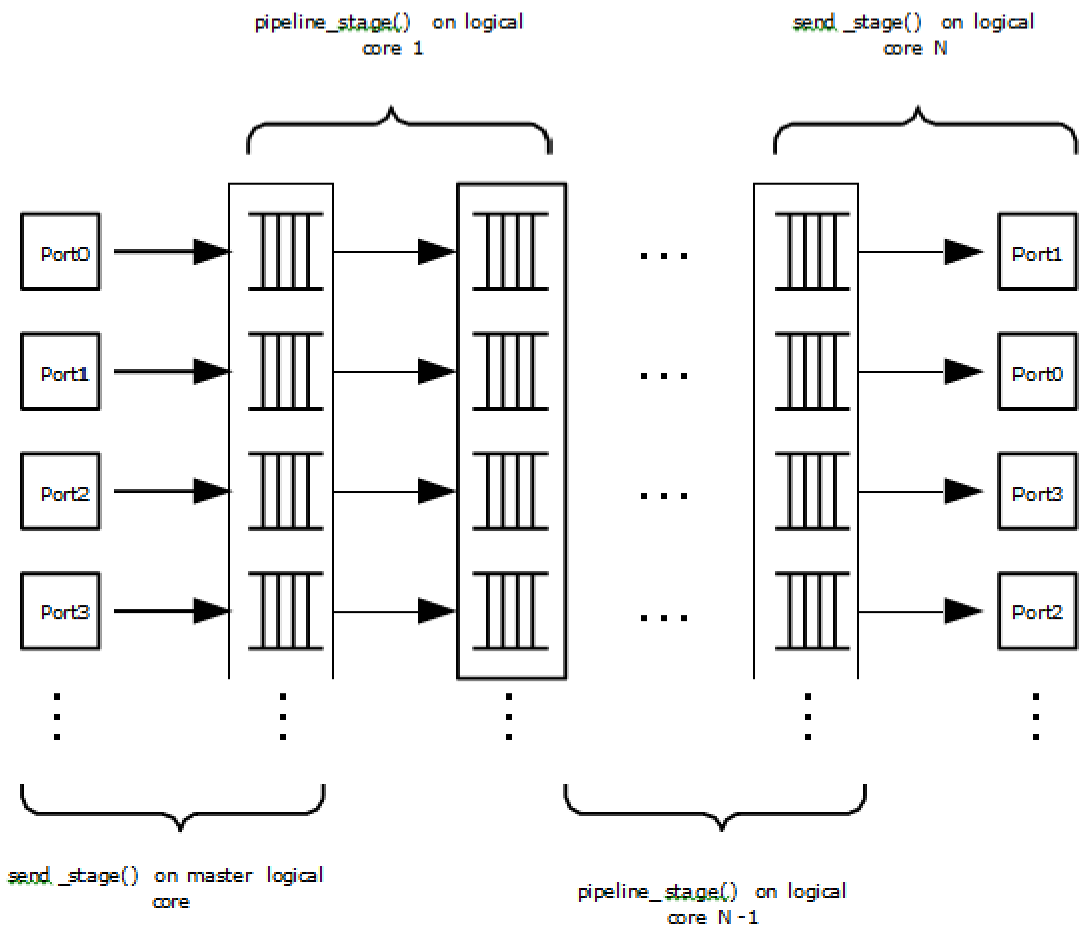

pipeline_stage() on logical core 1
send_stage() on logical core N
Port0
Port1
Port2
Port3
Port1
Port0
Port3
Port2
send_stage() on master logical core
pipeline_stage() on logical core N -1

Fig. 7.18: Threads and Pipelines

In the receive_stage() function running on the master logical core, the main task is to read ingress packets from the RX ports and enqueue them on the port's corresponding first ring in the pipeline. This is done using the following code:

```
lcore_id = rte_lcore_id();

/* Process each port round robin style */

for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++) {
    if (!is_bit_set(port_id, portmask))
        continue;
    ring = rings[lcore_id][port_id];

    if (ring_state[port_id] != RING_READY) {
        if (rte_ring_count(ring) > *low_watermark)
            continue;
    else
        ring_state[port_id] = RING_READY;
    }

    /* Enqueue received packets on the RX ring */

    nb_rx_pkts = rte_eth_rx_burst(port_id, 0, pkts, *quota);

    ret = rte_ring_enqueue_bulk(ring, (void *) pkts, nb_rx_pkts);
    if (ret == -EDQUOT) {
        ring_state[port_id] = RING_OVERLOADED;
        send_pause_frame(port_id, 1337);
    }
}
```

For each port in the port mask, the corresponding ring's pointer is fetched into ring and that ring's state is checked:

- If it is in the RING_READY state, *quota packets are grabbed from the port and put on the ring. Should this operation make the ring's usage cross its high watermark, the ring is marked as overloaded and an Ethernet flow control frame is sent to the source.
- If it is not in the RING_READY state, this port is ignored until the ring's usage crosses the *low_watermark value.

The pipeline_stage() function's task is to process and move packets from the preceding pipeline stage. This thread is running on most of the logical cores to create and arbitrarily long pipeline.

```
    lcore_id = rte_lcore_id();

    previous_lcore_id = get_previous_lcore_id(lcore_id);

    for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++) {
        if (!is_bit_set(port_id, portmask))
            continue;

        tx = rings[lcore_id][port_id];
        rx = rings[previous_lcore_id][port_id];
        if (ring_state[port_id] != RING_READY) {
            if (rte_ring_count(tx) > *low_watermark)
                continue;
        else
            ring_state[port_id] = RING_READY;
        }

        /* Dequeue up to quota mbuf from rx */

        nb_dq_pkts = rte_ring_dequeue_burst(rx, pkts, *quota);

        if (unlikely(nb_dq_pkts < 0))
            continue;

        /* Enqueue them on tx */

        ret = rte_ring_enqueue_bulk(tx, pkts, nb_dq_pkts);
        if (ret == -EDQUOT)
            ring_state[port_id] = RING_OVERLOADED;
```

```
}
```

The thread's logic works mostly like receive_stage(), except that packets are moved from ring to ring instead of port to ring.

In this example, no actual processing is done on the packets, but pipeline_stage() is an ideal place to perform any processing required by the application.

Finally, the send_stage() function's task is to read packets from the last ring in a pipeline and send them on the destination port defined in the port_pairs[] array. It is running on the last available logical core only.

```
lcore_id = rte_lcore_id();

previous_lcore_id = get_previous_lcore_id(lcore_id);

for (port_id = 0; port_id < RTE_MAX_ETHPORTS; port_id++) {
    if (!is_bit_set(port_id, portmask)) continue;

    dest_port_id = port_pairs[port_id];
    tx = rings[previous_lcore_id][port_id];

    if (rte_ring_empty(tx)) continue;

    /* Dequeue packets from tx and send them */

    nb_dq_pkts = rte_ring_dequeue_burst(tx, (void *) tx_pkts, *quota);
    nb_tx_pkts = rte_eth_tx_burst(dest_port_id, 0, tx_pkts, nb_dq_pkts);
}
```

For each port in the port mask, up to *quota packets are pulled from the last ring in its pipeline and sent on the destination port paired with the current port.

Control Application - qwctl

The qwctl application uses the rte_cmdline library to provide the user with an interactive command line that can be used to modify and inspect parameters in a running qw application. Those parameters are the global quota and low_watermark value as well as each ring's built-in high watermark.

Command Definitions

The available commands are defined in commands.c.

It is advised to use the cmdline sample application user guide as a reference for everything related to the rte_cmdline library.

Accessing Shared Variables

The setup_shared_variables() function retrieves the shared variables quota and low_watermark from the rte_memzone previously created by qw.

```
static void
setup_shared_variables(void)
{
    const struct rte_memzone *qw_memzone;
```

```
        qw_memzone = rte_memzone_lookup(QUOTA_WATERMARK_MEMZONE_NAME);
        if (qw_memzone == NULL)
            rte_exit(EXIT_FAILURE, "Couldn't find memzone\n");

        quota = qw_memzone->addr;

        low_watermark = (unsigned int *) qw_memzone->addr + sizeof(int);
    }
```

7.26 Timer Sample Application

The Timer sample application is a simple application that demonstrates the use of a timer in a DPDK application. This application prints some messages from different lcores regularly, demonstrating the use of timers.

7.26.1 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/timer
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible *RTE_TARGET* values.

3. Build the application:

   ```
   make
   ```

7.26.2 Running the Application

To run the example in linuxapp environment:

```
$ ./build/timer -c f -n 4
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.26.3 Explanation

The following sections provide some explanation of the code.

Initialization and Main Loop

In addition to EAL initialization, the timer subsystem must be initialized, by calling the rte_timer_subsystem_init() function.

```
/* init EAL */

ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_panic("Cannot init EAL\n");
```

```
/* init RTE timer library */

rte_timer_subsystem_init();
```

After timer creation (see the next paragraph), the main loop is executed on each slave lcore using the well-known rte_eal_remote_launch() and also on the master.

```
/* call lcore_mainloop() on every slave lcore  */

RTE_LCORE_FOREACH_SLAVE(lcore_id) {
    rte_eal_remote_launch(lcore_mainloop, NULL, lcore_id);
}

/* call it on master lcore too */

(void) lcore_mainloop(NULL);
```

The main loop is very simple in this example:

```
while (1) {
    /*
     *   Call the timer handler on each core: as we don't
     *   need a very precise timer, so only call
     *   rte_timer_manage() every ~10ms (at 2 GHz). In a real
     *   application, this will enhance performances as
     *   reading the HPET timer is not efficient.
    */

    cur_tsc = rte_rdtsc();

    diff_tsc = cur_tsc - prev_tsc;

    if (diff_tsc > TIMER_RESOLUTION_CYCLES) {
        rte_timer_manage();
        prev_tsc = cur_tsc;
    }
}
```

As explained in the comment, it is better to use the TSC register (as it is a per-lcore register) to check if the rte_timer_manage() function must be called or not. In this example, the resolution of the timer is 10 milliseconds.

Managing Timers

In the main() function, the two timers are initialized. This call to rte_timer_init() is necessary before doing any other operation on the timer structure.

```
/* init timer structures */

rte_timer_init(&timer0);
rte_timer_init(&timer1);
```

Then, the two timers are configured:

- The first timer (timer0) is loaded on the master lcore and expires every second. Since the PERIODICAL flag is provided, the timer is reloaded automatically by the timer subsystem. The callback function is timer0_cb().
- The second timer (timer1) is loaded on the next available lcore every 333 ms. The SINGLE flag means that the timer expires only once and must be reloaded manually if required. The callback function is timer1_cb().

```
/* load timer0, every second, on master lcore, reloaded automatically */

hz = rte_get_hpet_hz();

lcore_id = rte_lcore_id();

rte_timer_reset(&timer0, hz, PERIODICAL, lcore_id, timer0_cb, NULL);

/* load timer1, every second/3, on next lcore, reloaded manually */

lcore_id = rte_get_next_lcore(lcore_id, 0, 1);

rte_timer_reset(&timer1, hz/3, SINGLE, lcore_id, timer1_cb, NULL);
```

The callback for the first timer (timer0) only displays a message until a global counter reaches 20 (after 20 seconds). In this case, the timer is stopped using the rte_timer_stop() function.

```
/* timer0 callback */

static void
timer0_cb( attribute ((unused)) struct rte_timer *tim, __attribute ((unused)) void *arg)
{
    static unsigned counter = 0;

    unsigned lcore_id = rte_lcore_id();

    printf("%s() on lcore %u\n", FUNCTION , lcore_id);

    /* this timer is automatically reloaded until we decide to stop it, when counter reaches 2

    if ((counter ++) == 20)
        rte_timer_stop(tim);
}
```

The callback for the second timer (timer1) displays a message and reloads the timer on the next lcore, using the rte_timer_reset() function:

```
/* timer1 callback */

static void
timer1_cb( attribute ((unused)) struct rte_timer *tim, _attribute ((unused)) void *arg)
{
    unsigned lcore_id = rte_lcore_id();
    uint64_t hz;

    printf("%s() on lcore %u\\n", FUNCTION , lcore_id);

    /* reload it on another lcore */

    hz = rte_get_hpet_hz();

    lcore_id = rte_get_next_lcore(lcore_id, 0, 1);

    rte_timer_reset(&timer1, hz/3, SINGLE, lcore_id, timer1_cb, NULL);
}
```

7.27 Packet Ordering Application

The Packet Ordering sample app simply shows the impact of reordering a stream. It's meant to stress the library with different configurations for performance.

7.27.1 Overview

The application uses at least three CPU cores:

- RX core (maser core) receives traffic from the NIC ports and feeds Worker cores with traffic through SW queues.
- Worker core (slave core) basically do some light work on the packet. Currently it modifies the output port of the packet for configurations with more than one port enabled.
- TX Core (slave core) receives traffic from Worker cores through software queues, inserts out-of-order packets into reorder buffer, extracts ordered packets from the reorder buffer and sends them to the NIC ports for transmission.

7.27.2 Compiling the Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/helloworld
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started* Guide for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.27.3 Running the Application

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

Application Command Line

The application execution command line is:

```
./test-pipeline [EAL options] -- -p PORTMASK [--disable-reorder]
```

The -c EAL CPU_COREMASK option has to contain at least 3 CPU cores. The first CPU core in the core mask is the master core and would be assigned to RX core, the last to TX core and the rest to Worker cores.

The PORTMASK parameter must contain either 1 or even enabled port numbers. When setting more than 1 port, traffic would be forwarded in pairs. For example, if we enable 4 ports, traffic from port 0 to 1 and from 1 to 0, then the other pair from 2 to 3 and from 3 to 2, having [0,1] and [2,3] pairs.

The disable-reorder long option does, as its name implies, disable the reordering of traffic, which should help evaluate reordering performance impact.

7.28 VMDQ and DCB Forwarding Sample Application

The VMDQ and DCB Forwarding sample application is a simple example of packet processing using the DPDK. The application performs L2 forwarding using VMDQ and DCB to divide the incoming traffic into 128 queues. The traffic splitting is performed in hardware by the VMDQ and DCB features of the Intel® 82599 10 Gigabit Ethernet Controller.

7.28.1 Overview

This sample application can be used as a starting point for developing a new application that is based on the DPDK and uses VMDQ and DCB for traffic partitioning.

The VMDQ and DCB filters work on VLAN traffic to divide the traffic into 128 input queues on the basis of the VLAN ID field and VLAN user priority field. VMDQ filters split the traffic into 16 or 32 groups based on the VLAN ID. Then, DCB places each packet into one of either 4 or 8 queues within that group, based upon the VLAN user priority field.

In either case, 16 groups of 8 queues, or 32 groups of 4 queues, the traffic can be split into 128 hardware queues on the NIC, each of which can be polled individually by a DPDK application.

All traffic is read from a single incoming port (port 0) and output on port 1, without any processing being performed. The traffic is split into 128 queues on input, where each thread of the application reads from multiple queues. For example, when run with 8 threads, that is, with the -c FF option, each thread receives and forwards packets from 16 queues.

As supplied, the sample application configures the VMDQ feature to have 16 pools with 8 queues each as indicated in Fig. 7.19. The Intel® 82599 10 Gigabit Ethernet Controller NIC also supports the splitting of traffic into 32 pools of 4 queues each and this can be used by changing the NUM_POOLS parameter in the supplied code. The NUM_POOLS parameter can be passed on the command line, after the EAL parameters:

```
./build/vmdq_dcb [EAL options] -- -p PORTMASK --nb-pools NP
```

where, NP can be 16 or 32.

Fig. 7.19: Packet Flow Through the VMDQ and DCB Sample Application

In Linux* user space, the application can display statistics with the number of packets received on each queue. To have the application display the statistics, send a SIGHUP signal to the running application process, as follows:

where, <pid> is the process id of the application process.

The VMDQ and DCB Forwarding sample application is in many ways simpler than the L2 Forwarding application (see Chapter 9 , “L2 Forwarding Sample Application (in Real and Virtualized Environments)”) as it performs unidirectional L2 forwarding of packets from one port to a second port. No command-line options are taken by this application apart from the standard EAL command-line options.

Note: Since VMD queues are being used for VMM, this application works correctly when VTd is disabled in the BIOS or Linux* kernel (intel_iommu=off).

7.28.2 Compiling the Application

1. Go to the examples directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/vmdq_dcb
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.28.3 Running the Application

To run the example in a linuxapp environment:

```
user@target:~$ ./build/vmdq_dcb -c f -n 4 -- -p 0x3 --nb-pools 16
```

Refer to the *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.28.4 Explanation

The following sections provide some explanation of the code.

Initialization

The EAL, driver and PCI configuration is performed largely as in the L2 Forwarding sample application, as is the creation of the mbuf pool. See Chapter 9, “L2 Forwarding Sample Application (in Real and Virtualized Environments)”. Where this example application differs is in the configuration of the NIC port for RX.

The VMDQ and DCB hardware feature is configured at port initialization time by setting the appropriate values in the rte_eth_conf structure passed to the rte_eth_dev_configure() API. Initially in the application, a default structure is provided for VMDQ and DCB configuration to be filled in later by the application.

```
/* empty vmdq+dcb configuration structure. Filled in programmatically */

static const struct rte_eth_conf vmdq_dcb_conf_default = {
    .rxmode = {
        .mq_mode = ETH_VMDQ_DCB,
        .split_hdr_size = 0,
        .header_split = 0,   /**< Header Split disabled */
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
       .jumbo_frame = 0,     /**< Jumbo Frame Support disabled */
    },

    .txmode = {
        .mq_mode = ETH_DCB_NONE,
    },
```

```
    .rx_adv_conf = {
        /*
         *    should be overridden separately in code with
         *    appropriate values
         */

        .vmdq_dcb_conf = {
            .nb_queue_pools = ETH_16_POOLS,
            .enable_default_pool = 0,
            .default_pool = 0,
            .nb_pool_maps = 0,
            .pool_map = {{0, 0},},
            .dcb_queue = {0},
        },
    },
};
```

The get_eth_conf() function fills in an rte_eth_conf structure with the appropriate values, based on the global vlan_tags array, and dividing up the possible user priority values equally among the individual queues (also referred to as traffic classes) within each pool, that is, if the number of pools is 32, then the user priority fields are allocated two to a queue. If 16 pools are used, then each of the 8 user priority fields is allocated to its own queue within the pool. For the VLAN IDs, each one can be allocated to possibly multiple pools of queues, so the pools parameter in the rte_eth_vmdq_dcb_conf structure is specified as a bitmask value.

```
const uint16_t vlan_tags[] = {
    0, 1, 2, 3, 4, 5, 6, 7,
    8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26, 27, 28, 29, 30, 31
};


/* Builds up the correct configuration for vmdq+dcb based on the vlan tags array
 * given above, and the number of traffic classes available for use. */

static inline int
get_eth_conf(struct rte_eth_conf *eth_conf, enum rte_eth_nb_pools num_pools)
{
    struct rte_eth_vmdq_dcb_conf conf;
    unsigned i;

    if (num_pools != ETH_16_POOLS && num_pools != ETH_32_POOLS ) return -1;

    conf.nb_queue_pools = num_pools;
    conf.enable_default_pool = 0;
    conf.default_pool = 0; /* set explicit value, even if not used */
    conf.nb_pool_maps = sizeof( vlan_tags )/sizeof( vlan_tags[ 0 ]);

    for (i = 0; i < conf.nb_pool_maps; i++){
        conf.pool_map[i].vlan_id = vlan_tags[ i ];
        conf.pool_map[i].pools = 1 << (i % num_pools);
    }

    for (i = 0; i < ETH_DCB_NUM_USER_PRIORITIES; i++){
        conf.dcb_queue[i] = (uint8_t)(i % (NUM_QUEUES/num_pools));
    }

    (void) rte_memcpy(eth_conf, &vmdq_dcb_conf_default, sizeof(\*eth_conf));
    (void) rte_memcpy(&eth_conf->rx_adv_conf.vmdq_dcb_conf, &conf, sizeof(eth_conf->rx_adv_conf

    return 0;
}
```

Once the network port has been initialized using the correct VMDQ and DCB values, the initialization of the port's RX and TX hardware rings is performed similarly to that in the L2 Forwarding sample application. See Chapter 9, "L2 Forwarding Sample Application (in Real and Virtualized Environments)" for more information.

Statistics Display

When run in a linuxapp environment, the VMDQ and DCB Forwarding sample application can display statistics showing the number of packets read from each RX queue. This is provided by way of a signal handler for the SIGHUP signal, which simply prints to standard output the packet counts in grid form. Each row of the output is a single pool with the columns being the queue number within that pool.

To generate the statistics output, use the following command:

```
user@host$ sudo killall -HUP vmdq_dcb_app
```

Please note that the statistics output will appear on the terminal where the vmdq_dcb_app is running, rather than the terminal from which the HUP signal was sent.

7.29 Vhost Sample Application

The vhost sample application demonstrates integration of the Data Plane Development Kit (DPDK) with the Linux* KVM hypervisor by implementing the vhost-net offload API. The sample application performs simple packet switching between virtual machines based on Media Access Control (MAC) address or Virtual Local Area Network (VLAN) tag. The splitting of Ethernet traffic from an external switch is performed in hardware by the Virtual Machine Device Queues (VMDQ) and Data Center Bridging (DCB) features of the Intel® 82599 10 Gigabit Ethernet Controller.

7.29.1 Background

Virtio networking (virtio-net) was developed as the Linux* KVM para-virtualized method for communicating network packets between host and guest. It was found that virtio-net performance was poor due to context switching and packet copying between host, guest, and QEMU. The following figure shows the system architecture for a virtio-based networking (virtio-net).

The Linux* Kernel vhost-net module was developed as an offload mechanism for virtio-net. The vhost-net module enables KVM (QEMU) to offload the servicing of virtio-net devices to the vhost-net kernel module, reducing the context switching and packet copies in the virtual dataplane.

This is achieved by QEMU sharing the following information with the vhost-net module through the vhost-net API:

- The layout of the guest memory space, to enable the vhost-net module to translate addresses.
- The locations of virtual queues in QEMU virtual address space, to enable the vhost module to read/write directly to and from the virtqueues.

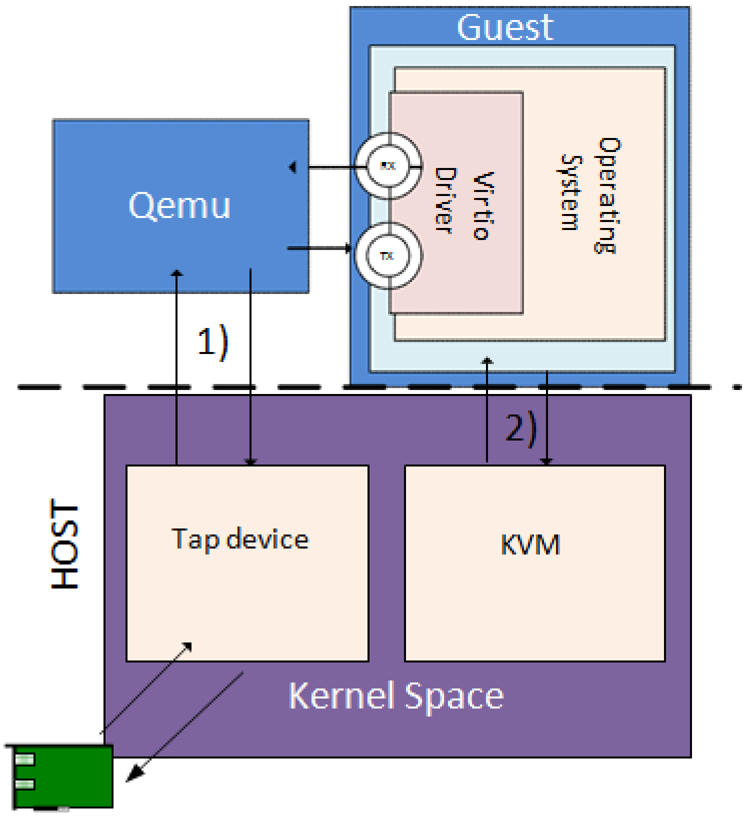


Fig. 7.20: System Architecture for Virtio-based Networking (virtio-net).

- An event file descriptor (eventfd) configured in KVM to send interrupts to the virtio- net device driver in the guest. This enables the vhost-net module to notify (call) the guest.
- An eventfd configured in KVM to be triggered on writes to the virtio-net device's Peripheral Component Interconnect (PCI) config space. This enables the vhost-net module to receive notifications (kicks) from the guest.

The following figure shows the system architecture for virtio-net networking with vhost-net offload.

7.29.2 Sample Code Overview

The DPDK vhost-net sample code demonstrates KVM (QEMU) offloading the servicing of a Virtual Machine's (VM's) virtio-net devices to a DPDK-based application in place of the kernel's vhost-net module.

The DPDK vhost-net sample code is based on vhost library. Vhost library is developed for user space Ethernet switch to easily integrate with vhost functionality.

The vhost library implements the following features:

- Management of virtio-net device creation/destruction events.
- Mapping of the VM's physical memory into the DPDK vhost-net's address space.
- Triggering/receiving notifications to/from VMs via eventfds.
- A virtio-net back-end implementation providing a subset of virtio-net features.

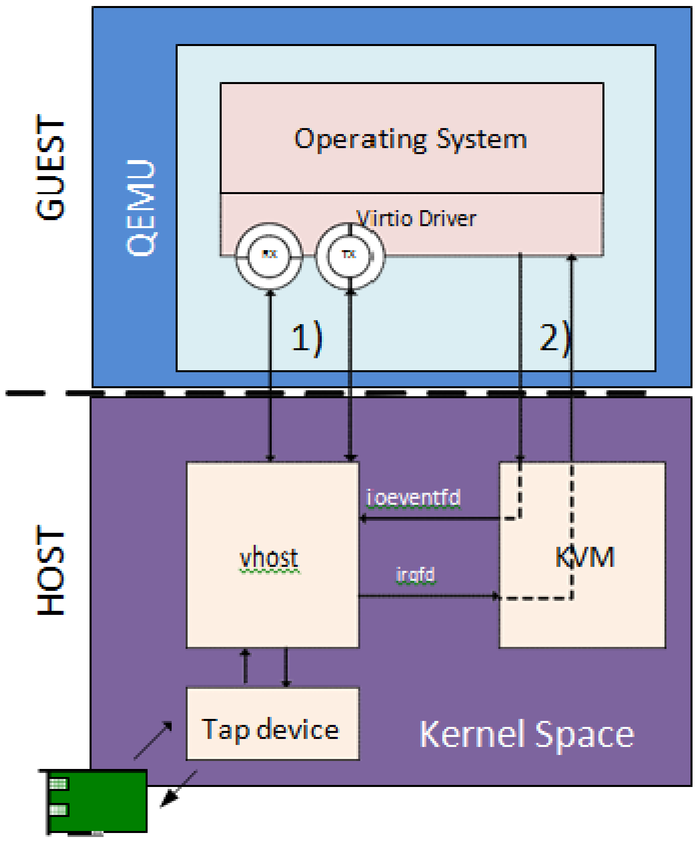


Fig. 7.21: Virtio with Linux

There are two vhost implementations in vhost library, vhost cuse and vhost user. In vhost cuse, a character device driver is implemented to receive and process vhost requests through ioctl messages. In vhost user, a socket server is created to received vhost requests through socket messages. Most of the messages share the same handler routine.

Note: Any vhost cuse specific requirement in the following sections will be emphasized.

Two implementations are turned on and off statically through configure file. Only one implementation could be turned on. They don't co-exist in current implementation.

The vhost sample code application is a simple packet switching application with the following feature:

- Packet switching between virtio-net devices and the network interface card, including using VMDQs to reduce the switching that needs to be performed in software.

The following figure shows the architecture of the Vhost sample application based on vhost-cuse.

The following figure shows the flow of packets through the vhost-net sample application.

7.29.3 Supported Distributions

The example in this section have been validated with the following distributions:

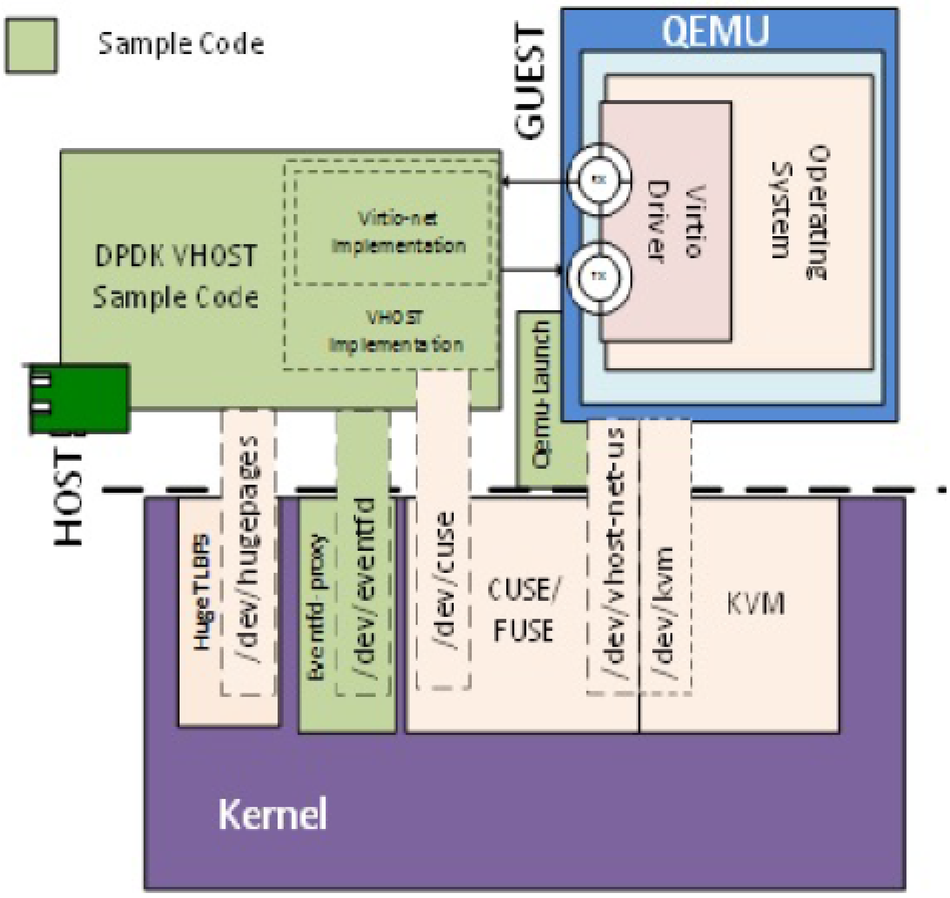


Fig. 7.22: Vhost-net Architectural Overview

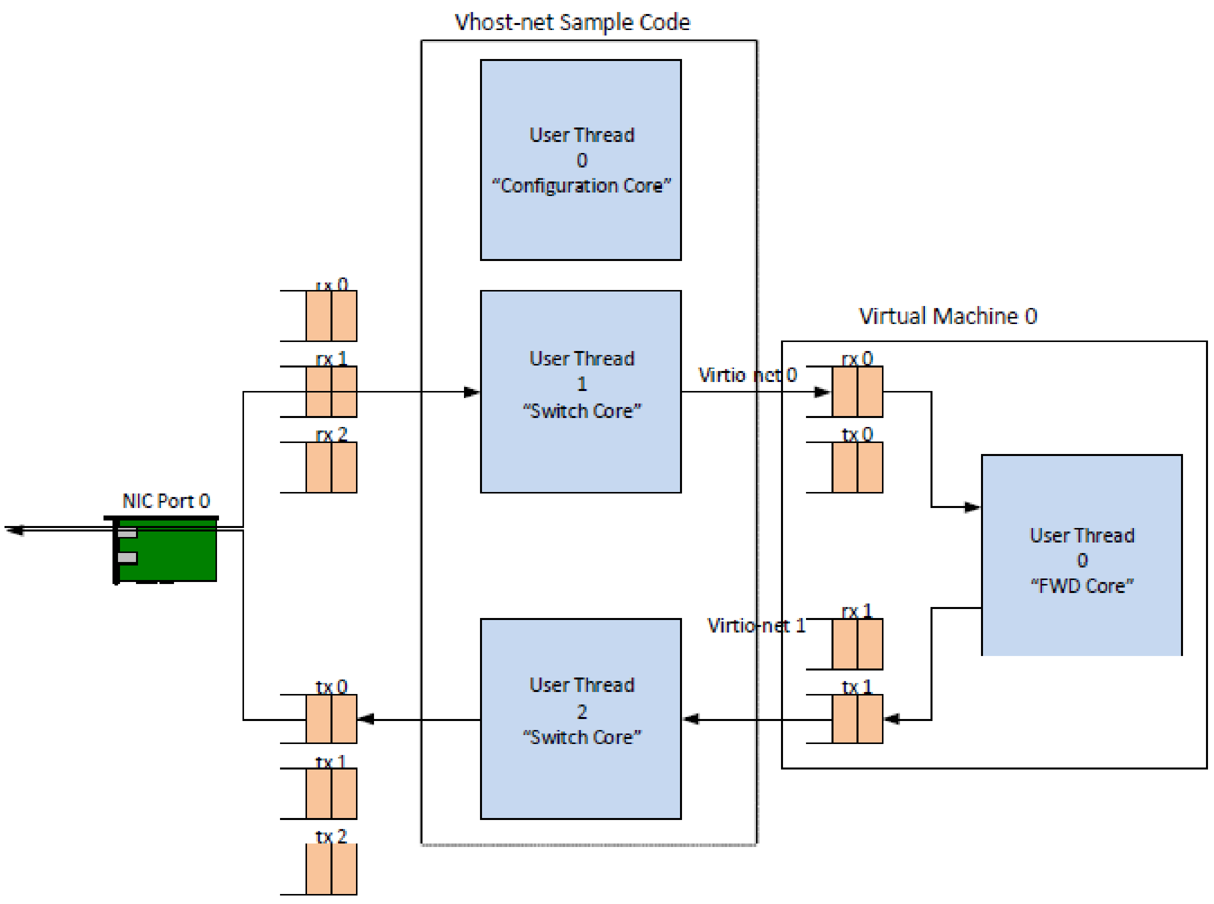


Fig. 7.23: Packet Flow Through the vhost-net Sample Application

- Fedora* 18
- Fedora* 19
- Fedora* 20

7.29.4 Prerequisites

This section lists prerequisite packages that must be installed.

Installing Packages on the Host(vhost cuse required)

The vhost cuse code uses the following packages; fuse, fuse-devel, and kernel-modules-extra. The vhost user code don't rely on those modules as eventfds are already installed into vhost process through Unix domain socket.

1. Install Fuse Development Libraries and headers:

   ```
   yum -y install fuse fuse-devel
   ```

2. Install the Cuse Kernel Module:

   ```
   yum -y install kernel-modules-extra
   ```

QEMU simulator

For vhost user, qemu 2.2 is required.

Setting up the Execution Environment

The vhost sample code requires that QEMU allocates a VM's memory on the hugetlbfs file system. As the vhost sample code requires hugepages, the best practice is to partition the system into separate hugepage mount points for the VMs and the vhost sample code.

Note: This is best-practice only and is not mandatory. For systems that only support 2 MB page sizes, both QEMU and vhost sample code can use the same hugetlbfs mount point without issue.

QEMU

VMs with gigabytes of memory can benefit from having QEMU allocate their memory from 1 GB huge pages. 1 GB huge pages must be allocated at boot time by passing kernel parameters through the grub boot loader.

1. Calculate the maximum memory usage of all VMs to be run on the system. Then, round this value up to the nearest Gigabyte the execution environment will require.

2. Edit the /etc/default/grub file, and add the following to the GRUB_CMDLINE_LINUX entry:

   ```
   GRUB_CMDLINE_LINUX="... hugepagesz=1G hugepages=<Number of hugepages required> default_hug
   ```

3. Update the grub boot loader:

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

4. Reboot the system.
5. The hugetlbfs mount point (/dev/hugepages) should now default to allocating gigabyte pages.

Note: Making the above modification will change the system default hugepage size to 1 GB for all applications.

Vhost Sample Code

In this section, we create a second hugetlbs mount point to allocate hugepages for the DPDK vhost sample code.

1. Allocate sufficient 2 MB pages for the DPDK vhost sample code:

```
echo 256 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

2. Mount hugetlbs at a separate mount point for 2 MB pages:

```
mount -t hugetlbfs nodev /mnt/huge -o pagesize=2M
```

The above steps can be automated by doing the following:

1. Edit /etc/fstab to add an entry to automatically mount the second hugetlbfs mount point:

```
hugetlbfs <tab> /mnt/huge <tab> hugetlbfs defaults,pagesize=1G 0 0
```

2. Edit the /etc/default/grub file, and add the following to the GRUB_CMDLINE_LINUX entry:

```
GRUB_CMDLINE_LINUX="... hugepagesz=2M hugepages=256 ... default_hugepagesz=1G"
```

3. Update the grub bootloader:

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

4. Reboot the system.

Note: Ensure that the default hugepage size after this setup is 1 GB.

Setting up the Guest Execution Environment

It is recommended for testing purposes that the DPDK testpmd sample application is used in the guest to forward packets, the reasons for this are discussed in Section 22.7, “Running the Virtual Machine (QEMU)”.

The testpmd application forwards packets between pairs of Ethernet devices, it requires an even number of Ethernet devices (virtio or otherwise) to execute. It is therefore recommended to create multiples of two virtio-net devices for each Virtual Machine either through libvirt or at the command line as follows.

Note: Observe that in the example, “-device” and “-netdev” are repeated for two virtio-net devices.

For vhost cuse:

```
qemu-system-x86_64 ... \
-netdev tap,id=hostnet1,vhost=on,vhostfd=<open fd> \
-device virtio-net-pci, netdev=hostnet1,id=net1 \
-netdev tap,id=hostnet2,vhost=on,vhostfd=<open fd> \
-device virtio-net-pci, netdev=hostnet2,id=net1
```

For vhost user:

```
qemu-system-x86_64 ... \
-chardev socket,id=char1,path=<sock_path> \
-netdev type=vhost-user,id=hostnet1,chardev=char1 \
-device virtio-net-pci,netdev=hostnet1,id=net1 \
-chardev socket,id=char2,path=<sock_path> \
-netdev type=vhost-user,id=hostnet2,chardev=char2 \
-device virtio-net-pci,netdev=hostnet2,id=net2
```

sock_path is the path for the socket file created by vhost.

7.29.5 Compiling the Sample Code

1. Compile vhost lib:

   To enable vhost, turn on vhost library in the configure file config/common_linuxapp.

   ```
   CONFIG_RTE_LIBRTE_VHOST=n
   ```

   vhost user is turned on by default in the configure file config/common_linuxapp. To enable vhost cuse, disable vhost user.

   ```
      CONFIG_RTE_LIBRTE_VHOST_USER=y

   After vhost is enabled and the implementation is selected, build the vhost library.
   ```

2. Go to the examples directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/vhost
   ```

3. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

4. Build the application:

   ```
   cd ${RTE_SDK}
   make config ${RTE_TARGET}
   make install ${RTE_TARGET}
   cd ${RTE_SDK}/examples/vhost
   make
   ```

5. Go to the eventfd_link directory(vhost cuse required):

   ```
   cd ${RTE_SDK}/lib/librte_vhost/eventfd_link
   ```

6. Build the eventfd_link kernel module(vhost cuse required):

   ```
   make
   ```

7.29.6 Running the Sample Code

1. Install the cuse kernel module(vhost cuse required):

   ```
   modprobe cuse
   ```

2. Go to the eventfd_link directory(vhost cuse required):

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/lib/librte_vhost/eventfd_link
   ```

3. Install the eventfd_link module(vhost cuse required):

   ```
   insmod ./eventfd_link.ko
   ```

4. Go to the examples directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/vhost/build/app
   ```

5. Run the vhost-switch sample code:

   vhost cuse:

   ```
   ./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
    -- -p 0x1 --dev-basename usvhost
   ```

   vhost user: a socket file named usvhost will be created under current directory. Use its path as the socket path in guest's qemu commandline.

   ```
   ./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
    -- -p 0x1 --dev-basename usvhost
   ```

Note: Please note the huge-dir parameter instructs the DPDK to allocate its memory from the 2 MB page hugetlbfs.

Note: The number used with the –socket-mem parameter may need to be more than 1024. The number required depends on the number of mbufs allocated by vhost-switch.

Parameters

Basename. vhost cuse uses a Linux* character device to communicate with QEMU. The basename is used to generate the character devices name.

/dev/<basename>

For compatibility with the QEMU wrapper script, a base name of "usvhost" should be used:

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- -p 0x1 --dev-basename usvhost
```

vm2vm. The vm2vm parameter disable/set mode of packet switching between guests in the host. Value of "0" means disabling vm2vm implies that on virtual machine packet transmission will always go to the Ethernet port; Value of "1" means software mode packet forwarding between guests, it needs packets copy in vHOST, so valid only in one-copy implementation, and invalid for zero copy implementation; value of "2" means hardware mode packet forwarding between guests, it allows packets go to the Ethernet port, hardware L2 switch will determine which guest the packet should forward to or need send to external, which bases on the packet destination MAC address and VLAN tag.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --vm2vm [0,1,2]
```

Mergeable Buffers. The mergeable buffers parameter controls how virtio-net descriptors are used for virtio-net headers. In a disabled state, one virtio-net header is used per packet buffer; in an enabled state one virtio-net header is used for multiple packets. The default value is 0 or disabled since recent kernels virtio-net drivers show performance degradation with this feature is enabled.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --mergeable [0,1]
```

Stats. The stats parameter controls the printing of virtio-net device statistics. The parameter specifies an interval second to print statistics, with an interval of 0 seconds disabling statistics.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
-- --stats [0,n]
```

RX Retry. The rx-retry option enables/disables enqueue retries when the guests RX queue is full. This feature resolves a packet loss that is observed at high data-rates, by allowing it to delay and retry in the receive path. This option is enabled by default.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --rx-retry [0,1]
```

RX Retry Number. The rx-retry-num option specifies the number of retries on an RX burst, it takes effect only when rx retry is enabled. The default value is 4.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --rx-retry 1 --rx-retry-num 5
```

RX Retry Delay Time. The rx-retry-delay option specifies the timeout (in micro seconds) between retries on an RX burst, it takes effect only when rx retry is enabled. The default value is 15.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --rx-retry 1 --rx-retry-delay 20
```

Zero copy. The zero copy option enables/disables the zero copy mode for RX/TX packet, in the zero copy mode the packet buffer address from guest translate into host physical address and then set directly as DMA address. If the zero copy mode is disabled, then one copy mode is utilized in the sample. This option is disabled by default.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --zero-copy [0,1]
```

RX descriptor number. The RX descriptor number option specify the Ethernet RX descriptor number, Linux legacy virtio-net has different behavior in how to use the vring descriptor from DPDK based virtio-net PMD, the former likely allocate half for virtio header, another half for frame buffer, while the latter allocate all for frame buffer, this lead to different number for available frame buffer in vring, and then lead to different Ethernet RX descriptor number could be used in zero copy mode. So it is valid only in zero copy mode is enabled. The value is 32 by default.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --zero-copy 1 --rx-desc-num [0, n]
```

TX descriptor number. The TX descriptor number option specify the Ethernet TX descriptor number, it is valid only in zero copy mode is enabled. The value is 64 by default.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --zero-copy 1 --tx-desc-num [0, n]
```

VLAN strip. The VLAN strip option enable/disable the VLAN strip on host, if disabled, the guest will receive the packets with VLAN tag. It is enabled by default.

```
./vhost-switch -c f -n 4 --socket-mem 1024 --huge-dir /mnt/huge \
 -- --vlan-strip [0, 1]
```

7.29.7 Running the Virtual Machine (QEMU)

QEMU must be executed with specific parameters to:

- Ensure the guest is configured to use virtio-net network adapters.

  ```
  qemu-system-x86_64 ... -device virtio-net-pci,netdev=hostnet1, \
  id=net1 ...
  ```

- Ensure the guest's virtio-net network adapter is configured with offloads disabled.

  ```
  qemu-system-x86_64 ... -device virtio-net-pci,netdev=hostnet1, \
  id=net1, csum=off,gso=off,guest_tso4=off,guest_tso6=off,guest_ecn=off
  ```

- Redirect QEMU to communicate with the DPDK vhost-net sample code in place of the vhost-net kernel module(vhost cuse).

  ```
  qemu-system-x86_64 ... -netdev tap,id=hostnet1,vhost=on, \
  vhostfd=<open fd> ...
  ```

- Enable the vhost-net sample code to map the VM's memory into its own process address space.

  ```
  qemu-system-x86_64 ... -mem-prealloc -mem-path /dev/hugepages ...
  ```

Note: The QEMU wrapper (qemu-wrap.py) is a Python script designed to automate the QEMU configuration described above. It also facilitates integration with libvirt, although the script may also be used standalone without libvirt.

Redirecting QEMU to vhost-net Sample Code(vhost cuse)

To redirect QEMU to the vhost-net sample code implementation of the vhost-net API, an open file descriptor must be passed to QEMU running as a child process.

```
#!/usr/bin/python
fd = os.open("/dev/usvhost-1", os.O_RDWR)
subprocess.call
("qemu-system-x86_64 ... -netdev tap,id=vhostnet0,vhost=on,vhostfd="
  + fd +"...", shell=True)
```

Note: This process is automated in the QEMU wrapper script discussed in Section 24.7.3.

Mapping the Virtual Machine's Memory

For the DPDK vhost-net sample code to be run correctly, QEMU must allocate the VM's memory on hugetlbfs. This is done by specifying mem-prealloc and mem-path when executing

QEMU. The vhost-net sample code accesses the virtio-net device's virtual rings and packet buffers by finding and mapping the VM's physical memory on hugetlbfs. In this case, the path passed to the guest should be that of the 1 GB page hugetlbfs:

```
qemu-system-x86_64 ... -mem-prealloc -mem-path /dev/hugepages ...
```

Note: This process is automated in the QEMU wrapper script discussed in Section 24.7.3. The following two sections only applies to vhost cuse. For vhost-user, please make corresponding changes to qemu-wrapper script and guest XML file.

QEMU Wrapper Script

The QEMU wrapper script automatically detects and calls QEMU with the necessary parameters required to integrate with the vhost sample code. It performs the following actions:

- Automatically detects the location of the hugetlbfs and inserts this into the command line parameters.
- Automatically open file descriptors for each virtio-net device and inserts this into the command line parameters.
- Disables offloads on each virtio-net device.
- Calls Qemu passing both the command line parameters passed to the script itself and those it has auto-detected.

The QEMU wrapper script will automatically configure calls to QEMU:

```
qemu-wrap.py -machine pc-i440fx-1.4,accel=kvm,usb=off \
-cpu SandyBridge -smp 4,sockets=4,cores=1,threads=1 \
-netdev tap,id=hostnet1,vhost=on \
-device virtio-net-pci,netdev=hostnet1,id=net1 \
-hda <disk img> -m 4096
```

which will become the following call to QEMU:

```
qemu-system-x86_64 -machine pc-i440fx-1.4,accel=kvm,usb=off \
-cpu SandyBridge -smp 4,sockets=4,cores=1,threads=1 \
-netdev tap,id=hostnet1,vhost=on,vhostfd=<open fd> \
-device virtio-net-pci,netdev=hostnet1,id=net1, \
csum=off,gso=off,guest_tso4=off,guest_tso6=off,guest_ecn=off \
-hda <disk img> -m 4096 -mem-path /dev/hugepages -mem-prealloc
```

Libvirt Integration

The QEMU wrapper script (qemu-wrap.py) “wraps” libvirt calls to QEMU, such that QEMU is called with the correct parameters described above. To call the QEMU wrapper automatically from libvirt, the following configuration changes must be made:

- Place the QEMU wrapper script in libvirt's binary search PATH ($PATH). A good location is in the directory that contains the QEMU binary.
- Ensure that the script has the same owner/group and file permissions as the QEMU binary.
- Update the VM xml file using virsh edit <vm name>:

  - Set the VM to use the launch script
  - Set the emulator path contained in the #<emulator><emulator/> tags For example, replace <emulator>/usr/bin/qemu-kvm<emulator/> with <emulator>/usr/bin/qemu-wrap.py<emulator/>
  - Set the VM's virtio-net device's to use vhost-net offload:

    ```
    <interface type="network">
    <model type="virtio"/>
    <driver name="vhost"/>
    <interface/>
    ```

  - Enable libvirt to access the DPDK Vhost sample code's character device file by adding it to controllers cgroup for libvirtd using the following steps:

    ```
    cgroup_controllers = [ ... "devices", ... ] clear_emulator_capabilities = 0
    user = "root" group = "root"
    cgroup_device_acl = [
        "/dev/null", "/dev/full", "/dev/zero",
        "/dev/random", "/dev/urandom",
        "/dev/ptmx", "/dev/kvm", "/dev/kqemu",
        "/dev/rtc", "/dev/hpet", "/dev/net/tun",
        "/dev/<devbase-name>-<index>",
    ]
    ```

- Disable SELinux or set to permissive mode.
- Mount cgroup device controller:

  ```
  mkdir /dev/cgroup
  mount -t cgroup none /dev/cgroup -o devices
  ```

- Restart the libvirtd system process

  For example, on Fedora* "systemctl restart libvirtd.service"

- Edit the configuration parameters section of the script:
  - Configure the "emul_path" variable to point to the QEMU emulator.

    ```
    emul_path = "/usr/local/bin/qemu-system-x86_64"
    ```

  - Configure the "us_vhost_path" variable to point to the DPDK vhost-net sample code's character devices name. DPDK vhost-net sample code's character device will be in the format "/dev/<basename>".

    ```
    us_vhost_path = "/dev/usvhost"
    ```

Common Issues

- QEMU failing to allocate memory on hugetlbfs, with an error like the following:

  ```
  file_ram_alloc: can't mmap RAM pages: Cannot allocate memory
  ```

  When running QEMU the above error indicates that it has failed to allocate memory for the Virtual Machine on the hugetlbfs. This is typically due to insufficient hugepages being free to support the allocation request. The number of free hugepages can be checked as follows:

  ```
  cat /sys/kernel/mm/hugepages/hugepages-<pagesize>/nr_hugepages
  ```

  The command above indicates how many hugepages are free to support QEMU's allocation request.

- User space VHOST when the guest has 2MB sized huge pages:

  The guest may have 2MB or 1GB sized huge pages. The user space VHOST should work properly in both cases.

- User space VHOST will not work with QEMU without the `-mem-prealloc` option:

  The current implementation works properly only when the guest memory is pre-allocated, so it is required to use a QEMU version (e.g. 1.6) which supports `-mem-prealloc`. The `-mem-prealloc` option must be specified explicitly in the QEMU command line.

- User space VHOST will not work with a QEMU version without shared memory mapping:

  As shared memory mapping is mandatory for user space VHOST to work properly with the guest, user space VHOST needs access to the shared memory from the guest to receive and transmit packets. It is important to make sure the QEMU version supports shared memory mapping.

- In an Ubuntu environment, QEMU fails to start a new guest normally with user space VHOST due to not being able to allocate huge pages for the new guest:

  The solution for this issue is to add `-boot c` into the QEMU command line to make sure the huge pages are allocated properly and then the guest should start normally.

  Use `cat /proc/meminfo` to check if there is any changes in the value of `HugePages_Total` and `HugePages_Free` after the guest startup.

- Log message: `eventfd_link: module verification failed: signature and/or required key missing - tainting kernel`:

  This log message may be ignored. The message occurs due to the kernel module `eventfd_link`, which is not a standard Linux module but which is necessary for the user space VHOST current implementation (CUSE-based) to communicate with the guest.

7.29.8 Running DPDK in the Virtual Machine

For the DPDK vhost-net sample code to switch packets into the VM, the sample code must first learn the MAC address of the VM's virtio-net device. The sample code detects the address from packets being transmitted from the VM, similar to a learning switch.

This behavior requires no special action or configuration with the Linux* virtio-net driver in the VM as the Linux* Kernel will automatically transmit packets during device initialization. However, DPDK-based applications must be modified to automatically transmit packets during initialization to facilitate the DPDK vhost- net sample code's MAC learning.

The DPDK testpmd application can be configured to automatically transmit packets during initialization and to act as an L2 forwarding switch.

Testpmd MAC Forwarding

At high packet rates, a minor packet loss may be observed. To resolve this issue, a "wait and retry" mode is implemented in the testpmd and vhost sample code. In the "wait and retry" mode if the virtqueue is found to be full, then testpmd waits for a period of time before retrying to enqueue packets.

The “wait and retry” algorithm is implemented in DPDK testpmd as a forwarding method call “mac_retry”. The following sequence diagram describes the algorithm in detail.

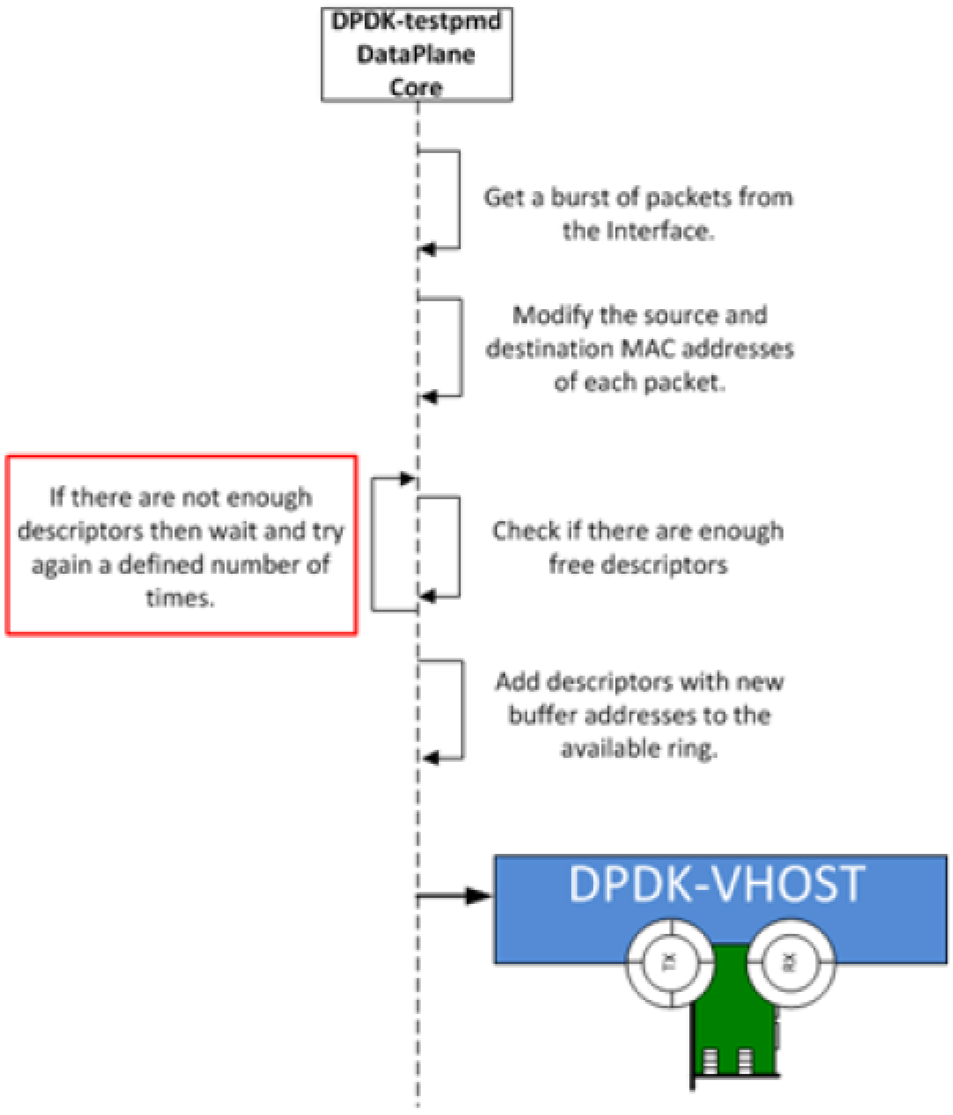

DPDK-testpmd DataPlane Core
Get a burst of packets from the Interface.
Modify the source and destination MAC addresses of each packet.
If there are not enough descriptors then wait and try again a defined number of times.
Check if there are enough free descriptors
Add descriptors with new buffer addresses to the available ring.
DPDK-VHOST
TX
RX

Fig. 7.24: Packet Flow on TX in DPDK-testpmd

Running Testpmd

The testpmd application is automatically built when DPDK is installed. Run the testpmd application as follows:

```
cd ${RTE_SDK}/x86_64-native-linuxapp-gcc/app
./testpmd -c 0x3 -n 4 --socket-mem 512 \
-- --burst=64 --i --disable-hw-vlan-filter
```

The destination MAC address for packets transmitted on each port can be set at the command line:

```
./testpmd -c 0x3 -n 4 --socket-mem 512 \
-- --burst=64 --i --disable-hw-vlan-filter \
--eth-peer=0,aa:bb:cc:dd:ee:ff --eth-peer=1,ff:ee:dd:cc:bb:aa
```

- Packets received on port 1 will be forwarded on port 0 to MAC address aa:bb:cc:dd:ee:ff

- Packets received on port 0 will be forwarded on port 1 to MAC address

  ff:ee:dd:cc:bb:aa

The testpmd application can then be configured to act as an L2 forwarding application:

```
testpmd> set fwd mac_retry
```

The testpmd can then be configured to start processing packets, transmitting packets first so the DPDK vhost sample code on the host can learn the MAC address:

```
testpmd> start tx_first
```

Note: Please note "set fwd mac_retry" is used in place of "set fwd mac_fwd" to ensure the retry feature is activated.

7.29.9 Passing Traffic to the Virtual Machine Device

For a virtio-net device to receive traffic, the traffic's Layer 2 header must include both the virtio-net device's MAC address and VLAN tag. The DPDK sample code behaves in a similar manner to a learning switch in that it learns the MAC address of the virtio-net devices from the first transmitted packet. On learning the MAC address, the DPDK vhost sample code prints a message with the MAC address and VLAN tag virtio-net device. For example:

```
DATA: (0) MAC_ADDRESS cc:bb:bb:bb:bb:bb and VLAN_TAG 1000 registered
```

The above message indicates that device 0 has been registered with MAC address cc:bb:bb:bb:bb:bb and VLAN tag 1000. Any packets received on the NIC with these values is placed on the devices receive queue. When a virtio-net device transmits packets, the VLAN tag is added to the packet by the DPDK vhost sample code.

7.30 Netmap Compatibility Sample Application

7.30.1 Introduction

The Netmap compatibility library provides a minimal set of APIs to give programs written against the Netmap APIs the ability to be run, with minimal changes to their source code, using the DPDK to perform the actual packet I/O.

Since Netmap applications use regular system calls, like `open()`, `ioctl()` and `mmap()` to communicate with the Netmap kernel module performing the packet I/O, the `compat_netmap` library provides a set of similar APIs to use in place of those system calls, effectively turning a Netmap application into a DPDK application.

The provided library is currently minimal and doesn't support all the features that Netmap supports, but is enough to run simple applications, such as the bridge example detailed below.

Knowledge of Netmap is required to understand the rest of this section. Please refer to the Netmap distribution for details about Netmap.

7.30.2 Available APIs

The library provides the following drop-in replacements for system calls usually used in Netmap applications:

- `rte_netmap_close()`
- `rte_netmap_ioctl()`
- `rte_netmap_open()`
- `rte_netmap_mmap()`
- `rte_netmap_poll()`

They use the same signature as their libc counterparts, and can be used as drop-in replacements in most cases.

7.30.3 Caveats

Given the difference between the way Netmap and the DPDK approach packet I/O, there are caveats and limitations to be aware of when trying to use the `compat_netmap` library, the most important of these are listed below. These may change as the library is updated:

- Any system call that can potentially affect file descriptors cannot be used with a descriptor returned by the `rte_netmap_open()` function.

Note that:

- The `rte_netmap_mmap()` function merely returns the address of a DPDK memzone. The address, length, flags, offset, and other arguments are ignored.
- The `rte_netmap_poll()` function only supports infinite (negative) or zero time outs. It effectively turns calls to the `poll()` system call made in a Netmap application into polling of the DPDK ports, changing the semantics of the usual POSIX defined poll.
- Not all of Netmap's features are supported: host rings, slot flags and so on are not supported or are simply not relevant in the DPDK model.
- The Netmap manual page states that "*a device obtained through /dev/netmap also supports the ioctl supported by network devices*". This is not the case with this compatibility layer.
- The Netmap kernel module exposes a sysfs interface to change some internal parameters, such as the size of the shared memory region. This interface is not available when using this compatibility layer.

7.30.4 Porting Netmap Applications

Porting Netmap applications typically involves two major steps:

- Changing the system calls to use their `compat_netmap` library counterparts.
- Adding further DPDK initialization code.

Since the `compat_netmap` functions have the same signature as the usual libc calls, the change is trivial in most cases.

The usual DPDK initialization code involving `rte_eal_init()` and `rte_eal_pci_probe()` has to be added to the Netmap application in the same way it is used in all other DPDK sample applications. Please refer to the *DPDK Programmer's Guide* and example source code for details about initialization.

In addition of the regular DPDK initialization code, the ported application needs to call initialization functions for the `compat_netmap` library, namely `rte_netmap_init()` and `rte_netmap_init_port()`.

These two initialization functions take `compat_netmap` specific data structures as parameters: `struct rte_netmap_conf` and `struct rte_netmap_port_conf`. The structures' fields are Netmap related and are self-explanatory for developers familiar with Netmap. They are defined in `$RTE_SDK/examples/netmap_compat/lib/compat_netmap.h`.

The bridge application is an example largely based on the bridge example shipped with the Netmap distribution. It shows how a minimal Netmap application with minimal and straightforward source code changes can be run on top of the DPDK. Please refer to `$RTE_SDK/examples/netmap_compat/bridge/bridge.c` for an example of a ported application.

7.30.5 Compiling the "bridge" Sample Application

1. Go to the example directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/netmap_compat
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Getting Started Guide for Linux* for possible `RTE_TARGET` values.

3. Build the application:

   ```
   make
   ```

7.30.6 Running the "bridge" Sample Application

The application requires a single command line option:

```
./build/bridge [EAL options] -- -i INTERFACE_A [-i INTERFACE_B]
```

where,

- `-i INTERFACE`: Interface (DPDK port number) to use.

  If a single `-i` parameter is given, the interface will send back all the traffic it receives. If two `-i` parameters are given, the two interfaces form a bridge, where traffic received on one interface is replicated and sent to the other interface.

For example, to run the application in a linuxapp environment using port 0 and 2:

```
./build/bridge [EAL options] -- -i 0 -i 2
```

Refer to the *DPDK Getting Started Guide for Linux* for general information on running applications and the Environment Abstraction Layer (EAL) options.

Note that unlike a traditional bridge or the `l2fwd` sample application, no MAC address changes are done on the frames. Do not forget to take this into account when configuring a traffic generators and testing this sample application.

7.31 Internet Protocol (IP) Pipeline Application

7.31.1 Application overview

The *Internet Protocol (IP) Pipeline* application is intended to be a vehicle for rapid development of packet processing applications running on multi-core CPUs.

The application provides a library of reusable functional blocks called pipelines. These pipelines can be seen as prefabricated blocks that can be instantiated and inter-connected through packet queues to create complete applications (super-pipelines).

Pipelines are created and inter-connected through the application configuration file. By using different configuration files, different applications are effectively created, therefore this application can be seen as an application generator. The configuration of each pipeline can be updated at run-time through the application Command Line Interface (CLI).

Main application components are:

A Library of reusable pipelines

- Each pipeline represents a functional block, e.g. flow classification, firewall, routing, master, etc.
- Each pipeline type can be instantiated several times in the same application, which each instance configured separately and mapped to a single CPU core. Each CPU core can run one or several pipeline instances, which can be of same or different type.
- Pipeline instances are inter-connected through packet queues (for packet processing) and message queues (for run-time configuration).
- Pipelines are implemented using DPDK Packet Framework.
- More pipeline types can always be built and added to the existing pipeline types.

The Configuration file

- The configuration file defines the application structure. By using different configuration files, different applications are created.
- All the application resources are created and configured through the application configuration file: pipeline instances, buffer pools, links (i.e. network interfaces), hardware device RX/TX queues, software queues, traffic manager devices, EAL startup arguments, etc.
- The configuration file syntax is “define by reference”, meaning that resources are defined as they are referenced. First time a resource name is detected, it is registered with default parameters. Optionally, the resource parameters can be further refined through a configuration file section dedicated to that resource.
- Command Line Interface (CLI)

Global CLI commands: link configuration, etc.

- Common pipeline CLI commands: ping (keep-alive), statistics, etc.

- Pipeline type specific CLI commands: used to configure instances of specific pipeline type. These commands are registered with the application when the pipeline type is registered. For example, the commands for routing pipeline instances include: route add, route delete, route list, etc.
- CLI commands can be grouped into scripts that can be invoked at initialization and at runtime.

7.31.2 Design goals

Rapid development

This application enables rapid development through quick connectivity of standard components called pipelines. These components are built using DPDK Packet Framework and encapsulate packet processing features at different levels: ports, tables, actions, pipelines and complete applications.

Pipeline instances are instantiated, configured and inter-connected through low complexity configuration files loaded during application initialization. Each pipeline instance is mapped to a single CPU core, with each CPU core able to run one or multiple pipeline instances of same or different types. By loading a different configuration file, a different application is effectively started.

Flexibility

Each packet processing application is typically represented as a chain of functional stages which is often called the functional pipeline of the application. These stages are mapped to CPU cores to create chains of CPU cores (pipeline model), clusters of CPU cores (run-to-completion model) or chains of clusters of CPU cores (hybrid model).

This application allows all the above programming models. By applying changes to the configuration file, the application provides the flexibility to reshuffle its building blocks in different ways until the configuration providing the best performance is identified.

Move pipelines around

The mapping of pipeline instances to CPU cores can be reshuffled through the configuration file. One or several pipeline instances can be mapped to the same CPU core.

Fig. 7.25: Example of moving pipeline instances across different CPU cores

Move tables around

There is some degree of flexibility for moving tables from one pipeline instance to another. Based on the configuration arguments passed to each pipeline instance in the configuration file, specific tables can be enabled or disabled. This way, a specific table can be "moved" from pipeline instance A to pipeline instance B by simply disabling its associated functionality for pipeline instance A while enabling it for pipeline instance B.

Due to requirement to have simple syntax for the configuration file, moving tables across different pipeline instances is not as flexible as the mapping of pipeline instances to CPU cores, or mapping actions to pipeline tables. Complete flexibility in moving tables from one pipeline to another could be achieved through a complex pipeline description language that would detail the structural elements of the pipeline (ports, tables and actions) and their connectivity, resulting in complex syntax for the configuration file, which is not acceptable. Good configuration file readability through simple syntax is preferred.

Example: the IP routing pipeline can run the routing function only (with ARP function run by a different pipeline instance), or it can run both the routing and ARP functions as part of the same pipeline instance.

Fig. 7.26: Example of moving tables across different pipeline instances

Move actions around

When it makes sense, packet processing actions can be moved from one pipeline instance to another. Based on the configuration arguments passed to each pipeline instance in the configuration file, specific actions can be enabled or disabled. This way, a specific action can be "moved" from pipeline instance A to pipeline instance B by simply disabling its associated functionality for pipeline instance A while enabling it for pipeline instance B.

Example: The flow actions of accounting, traffic metering, application identification, NAT, etc can be run as part of the flow classification pipeline instance or split across several flow actions pipeline instances, depending on the number of flow instances and their compute requirements.

Fig. 7.27: Example of moving actions across different tables and pipeline instances

Performance

Performance of the application is the highest priority requirement. Flexibility is not provided at the expense of performance.

The purpose of flexibility is to provide an incremental development methodology that allows monitoring the performance evolution:

- Apply incremental changes in the configuration (e.g. mapping on pipeline instances to CPU cores) in order to identify the configuration providing the best performance for a given application;
- Add more processing incrementally (e.g. by enabling more actions for specific pipeline instances) until the application is feature complete while checking the performance impact at each step.

Debug capabilities

The application provides a significant set of debug capabilities:

- Command Line Interface (CLI) support for statistics polling: pipeline instance ping (keep-alive checks), pipeline instance statistics per input port/output port/table, link statistics, etc;
- Logging: Turn on/off application log messages based on priority level;

7.31.3 Running the application

The application startup command line is:

```
ip_pipeline [-f CONFIG_FILE] [-s SCRIPT_FILE] -p PORT_MASK [-l LOG_LEVEL]
```

The application startup arguments are:

`-f CONFIG_FILE`

- Optional: Yes
- Default: `./config/ip_pipeline.cfg`
- Argument: Path to the configuration file to be loaded by the application. Please refer to the *Configuration file syntax* for details on how to write the configuration file.

`-s SCRIPT_FILE`

- Optional: Yes
- Default: Not present
- Argument: Path to the CLI script file to be run by the master pipeline at application startup. No CLI script file will be run at startup of this argument is not present.

`-p PORT_MASK`

- Optional: No
- Default: N/A
- Argument: Hexadecimal mask of NIC port IDs to be used by the application. First port enabled in this mask will be referenced as LINK0 as part of the application configuration file, next port as LINK1, etc.

`-l LOG_LEVEL`

- Optional: Yes
- Default: 1 (High priority)
- Argument: Log level to determine which application messages are to be printed to standard output. Available log levels are: 0 (None), 1 (High priority), 2 (Low priority). Only application messages whose priority is higher than or equal to the application log level will be printed.

7.31.4 Application stages

Configuration

During this stage, the application configuration file is parsed and its content is loaded into the application data structures. In case of any configuration file parse error, an error message is displayed and the application is terminated. Please refer to the *Configuration file syntax* for a description of the application configuration file format.

Configuration checking

In the absence of any parse errors, the loaded content of application data structures is checked for overall consistency. In case of any configuration check error, an error message is displayed and the application is terminated.

Initialization

During this stage, the application resources are initialized and the handles to access them are saved into the application data structures. In case of any initialization error, an error message is displayed and the application is terminated.

The typical resources to be initialized are: pipeline instances, buffer pools, links (i.e. network interfaces), hardware device RX/TX queues, software queues, traffic management devices, etc.

Run-time

Each CPU core runs the pipeline instances assigned to it in time sharing mode and in round robin order:

1. *Packet processing task*: The pipeline run-time code is typically a packet *processing* task built on top of DPDK Packet Framework rte_pipeline library, which reads bursts of packets from the pipeline input ports, performs table lookups and executes the identified actions for all tables in the pipeline, with packet eventually written to pipeline output ports or dropped.
2. *Message handling task*: Each CPU core will also periodically execute the *message handling* code of each of the pipelines mapped to it. The pipeline message handling code is processing the messages that are pending in the pipeline input message queues, which are typically sent by the master CPU core for the on-the-fly pipeline configuration: check that pipeline is still alive (ping), add/delete entries in the pipeline tables, get statistics, etc. The frequency of executing the message handling code is usually much smaller than the frequency of executing the packet processing work.

Please refer to the *PIPELINE section* for more details about the application pipeline module encapsulation.

7.31.5 Configuration file syntax

Syntax overview

The syntax of the configuration file is designed to be simple, which favors readability. The configuration file is parsed using the DPDK library librte_cfgfile, which supports simple INI file format for configuration files.

As result, the configuration file is split into several sections, with each section containing one or more entries. The scope of each entry is its section, and each entry specifies a variable that is assigned a specific value. Any text after the ; character is considered a comment and is therefore ignored.

The following are application specific: number of sections, name of each section, number of entries of each section, name of the variables used for each section entry, the value format (e.g. signed/unsigned integer, string, etc) and range of each section entry variable.

Generic example of configuration file section:

```
[<section_name>]

<variable_name_1> = <value_1>

...

<variable_name_N> = <value_N>
```

Application resources present in the configuration file

Table 7.3: Application resource names in the configuration file

Resource type	Format	Examples
Pipeline	PIPELINE<ID>	PIPELINE0, PIPELINE1
Mempool	MEMPOOL<ID>	MEMPOOL0, MEMPOOL1
Link (network interface)	LINK<ID>	LINK0, LINK1
Link RX queue	RXQ<LINK_ID>.<QUEUE_ID>	RXQ0.0, RXQ1.5
Link TX queue	TXQ<LINK_ID>.<QUEUE_ID>	TXQ0.0, TXQ1.5
Software queue	SWQ<ID>	SWQ0, SWQ1
Traffic Manager	TM<LINK_ID>	TM0, TM1
Source	SOURCE<ID>	SOURCE0, SOURCE1
Sink	SINK<ID>	SINK0, SINK1
Message queue	MSGQ<ID> MSGQ-REQ-PIPELINE<ID> MSGQ-RSP-PIPELINE<ID> MSGQ-REQ-CORE-<CORE_ID> MSGQ-RSP-CORE-<CORE_ID>	MSGQ0, MSGQ1, MSGQ-REQ-PIPELINE2, MSGQ-RSP-PIPELINE2, MSGQ-REQ-CORE-s0c1, MSGQ-RSP-CORE-s0c1

LINK instances are created implicitly based on the PORT_MASK application startup argument. LINK0 is the first port enabled in the PORT_MASK, port 1 is the next one, etc. The LINK ID is different than the DPDK PMD-level NIC port ID, which is the actual position in the bitmask mentioned above. For example, if bit 5 is the first bit set in the bitmask, then LINK0 is having the PMD ID of 5. This mechanism creates a contiguous LINK ID space and isolates the configuration file against changes in the board PCIe slots where NICs are plugged in.

RXQ, TXQ and TM instances have the LINK ID as part of their name. For example, RXQ2.1, TXQ2.1 and TM2 are all associated with LINK2.

Rules to parse the configuration file

The main rules used to parse the configuration file are:

1. Application resource name determines the type of resource based on the name prefix.

   Example: all software queues need to start with SWQ prefix, so SWQ0 and SWQ5 are valid software queue names.

2. An application resource is defined by creating a configuration file section with its name. The configuration file section allows fine tuning on any of the resource parameters. Some resource parameters are mandatory, in which case it is required to have them specified as part of the section, while some others are optional, in which case they get assigned their default value when not present.

   Example: section `SWQ0` defines a software queue named SWQ0, whose parameters are detailed as part of this section.

3. An application resource can also be defined by referencing it. Referencing a resource takes place by simply using its name as part of the value assigned to a variable in any configuration file section. In this case, the resource is registered with all its parameters having their default values. Optionally, a section with the resource name can be added to the configuration file to fine tune some or all of the resource parameters.

   Example: in section `PIPELINE3`, variable `pktq_in` includes `SWQ5` as part of its list, which results in defining a software queue named `SWQ5`; when there is no `SWQ5` section present in the configuration file, `SWQ5` gets registered with default parameters.

PIPELINE section

Table 7.4: Configuration file PIPELINE section (1/2)

Section	Description	Optional	Range	Default value
type	Pipeline type. Defines the functionality to be executed.	NO	See "List of pipeline types"	N/A
core	CPU core to run the current pipeline.	YES	See "CPU Core notation"	CPU socket 0, core 0, hyper-thread 0
pktq_in	Packet queues to serve as input ports for the current pipeline instance. The acceptable packet queue types are: `RXQ`, `SWQ`, `TM` and `SOURCE`. First device in this list is used as pipeline input port 0, second as pipeline input port 1, etc.	YES	List of input packet queue IDs	Empty list
pktq_out	Packet queues to serve as output ports for the current pipeline instance. The acceptable packet queue types are: `TXQ`, `SWQ`, `TM` and `SINK`. First device in this list is used as pipeline output port 0, second as pipeline output port 1, etc.	YES	List of output packet queue IDs.	Empty list

Table 7.5: Configuration file PIPELINE section (2/2)

Section	Description	Optional	Range	Default value
msgq_in	Input message queues. These queues contain request messages that need to be handled by the current pipeline instance. The type and format of request messages is defined by the pipeline type. For each pipeline instance, there is an input message queue defined implicitly, whose name is: `MSGQ-REQ-<PIPELINE_ID>`. This message queue should not be mentioned as part of msgq_in list.	YES	List of message queue IDs	Empty list
msgq_out	Output message queues. These queues are used by the current pipeline instance to write response messages as result of request messages being handled. The type and format of response messages is defined by the pipeline type. For each pipeline instance, there is an output message queue defined implicitly, whose name is: `MSGQ-RSP-<PIPELINE_ID>`. This message queue should not be mentioned as part of msgq_out list.	YES	List of message queue IDs	Empty list
timer_period	Time period, measured in milliseconds, for handling the input message queues.	YES	milliseconds	1 ms
<any other>	Arguments to be passed to the current pipeline instance. Format of the arguments, their type, whether each argument is optional or mandatory and its default value (when optional) are defined by the pipeline type. The value of the arguments is applicable to the current pipeline instance only.	Depends on pipeline type	Depends on pipeline type	Depends on pipeline type

CPU core notation

The CPU Core notation is:

```
<CPU core> ::= [s|S<CPU socket ID>][c|C]<CPU core ID>[h|H]
```

For example:

```
CPU socket 0, core 0, hyper-thread 0: 0, c0, s0c0

CPU socket 0, core 0, hyper-thread 1: 0h, c0h, s0c0h

CPU socket 3, core 9, hyper-thread 1: s3c9h
```

MEMPOOL section

Table 7.6: Configuration file MEMPOOL section

Section	Description	Optional	Type	Default value
buffer_size	Buffer size (in bytes) for the current buffer pool.	YES	uint32_t	2048 + sizeof(struct rte_mbuf) + HEADROOM
pool_size	Number of buffers in the current buffer pool.	YES	uint32_t	32K
cache_size	Per CPU thread cache size (in number of buffers) for the current buffer pool.	YES	uint32_t	256
cpu	CPU socket ID where to allocate memory for the current buffer pool.	YES	uint32_t	0

LINK section

Table 7.7: Configuration file LINK section

Section entry	Description	Optional	Type	Default value
arp_q	NIC RX queue where ARP packets should be filtered.	YES	0 .. 127	0 (default queue)
tcp_syn_local_q	NIC RX queue where TCP packets with SYN flag should be filtered.	YES	0 .. 127	0 (default queue)
ip_local_q	NIC RX queue where IP packets with local destination should be filtered. When TCP, UDP and SCTP local queues are defined, they take higher priority than this queue.	YES	0 .. 127	0 (default queue)
tcp_local_q	NIC RX queue where TCP packets with local destination should be filtered.	YES	0 .. 127	0 (default queue)
udp_local_q	NIC RX queue where TCP packets with local destination should be filtered.	YES	0 .. 127	0 (default queue)
sctp_local_q	NIC RX queue where TCP packets with local destination should be filtered.	YES	0 .. 127	0 (default queue)
promisc	Indicates whether current link should be started in promiscuous mode.	YES	YES/NO	YES

RXQ section

Table 7.8: Configuration file RXQ section

Section	Description	Optional	Type	Default value
mempool	Mempool to use for buffer allocation for current NIC RX queue. The mempool ID has to be associated with a valid instance defined in the mempool entry of the global section.	YES	uint32_t	MEMPOOL0
Size	NIC RX queue size (number of descriptors)	YES	uint32_t	128
burst	Read burst size (number of descriptors)	YES	uint32_t	32

TXQ section

Table 7.9: Configuration file TXQ section

Section	Description	Optional	Type	Default value
size	NIC TX queue size (number of descriptors)	YES	uint32_t power of 2 > 0	512
burst	Write burst size (number of descriptors)	YES	uint32_t power of 2 0 < burst < size	32
dropless	When dropless is set to NO, packets can be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is non- blocking. When dropless is set to YES, packets cannot be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is blocking, as the write operation is retried until enough free slots become available and all the packets are successfully written to the queue.	YES	YES/NO	NO
n_retries	Number of retries. Valid only when dropless is set to YES. When set to 0, it indicates unlimited number of retries.	YES	uint32_t	0

SWQ section

Table 7.10: Configuration file SWQ section

Section	Description	Optional	Type	Default value
size	Queue size (number of packets)	YES	uint32_t power of 2	256
burst_read	Read burst size (number of packets)	YES	uint32_t power of 2 0 < burst < size	32
burst_write	Write burst size (number of packets)	YES	uint32_t power of 2 0 < burst < size	32
dropless	When dropless is set to NO, packets can be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is non- blocking. When dropless is set to YES, packets cannot be dropped if not enough free slots are currently available in the queue, so the write operation to the queue is blocking, as the write operation is retried until enough free slots become available and all the packets are successfully written to the queue.	YES	YES/NO	NO
n_retries	Number of retries. Valid only when dropless is set to YES. When set to 0, it indicates unlimited number of retries.	YES	uint32_t	0
cpu	CPU socket ID where to allocate memory for this SWQ.	YES	uint32_t	0

TM section

Table 7.11: Configuration file TM section

Section	Description	Optional	Type	Default value
Cfg	File name to parse for the TM configuration to be applied. The syntax of this file is described in the examples/qos_sched DPDK application documentation.	YES	string	tm_profile
burst_read	Read burst size (number of packets)	YES	uint32_t	64
burst_write	Write burst size (number of packets)	YES	uint32_t	32

SOURCE section

Table 7.12: Configuration file SOURCE section

Section	Description	Optional	Type	Default value
Mempool	Mempool to use for buffer allocation.	YES	uint32_t	MEMPOOL0
Burst	Read burst size (number of packets)		uint32_t	32

SINK section

Currently, there are no parameters to be passed to a sink device, so SINK section is not allowed.

MSGQ section

Table 7.13: Configuration file MSGQ section

Section	Description	Optional	Type	Default value
size	Queue size (number of packets)	YES	uint32_t != 0 power of 2	64
cpu	CPU socket ID where to allocate memory for the current queue.	YES	uint32_t	0

EAL section

The application generates the EAL parameters rather than reading them from the command line.

The CPU core mask parameter is generated based on the core entry of all PIPELINE sections. All the other EAL parameters can be set from this section of the application configuration file.

7.31.6 Library of pipeline types

Pipeline module

A pipeline is a self-contained module that implements a packet processing function and is typically implemented on top of the DPDK Packet Framework *librte_pipeline* library. The application provides a run-time mechanism to register different pipeline types.

Depending on the required configuration, each registered pipeline type (pipeline class) is instantiated one or several times, with each pipeline instance (pipeline object) assigned to one of the available CPU cores. Each CPU core can run one or more pipeline instances, which might be of same or different types. For more information of the CPU core threading model, please refer to the *Run-time* section.

Pipeline type

Each pipeline type is made up of a back-end and a front-end. The back-end represents the packet processing engine of the pipeline, typically implemented using the DPDK Packet Framework libraries, which reads packets from the input packet queues, handles them and eventually writes them to the output packet queues or drops them. The front-end represents the run-time configuration interface of the pipeline, which is exposed as CLI commands. The front-end communicates with the back-end through message queues.

Table 7.14: Pipeline back-end

Field name	Field type	Description
f_init	Function pointer	Function to initialize the back-end of the current pipeline instance. Typical work implemented by this function for the current pipeline instance: Memory allocation; Parse the pipeline type specific arguments; Initialize the pipeline input ports, output ports and tables, interconnect input ports to tables; Set the message handlers.
f_free	Function pointer	Function to free the resources allocated by the back-end of the current pipeline instance.
f_run	Function pointer	Set to NULL for pipelines implemented using the DPDK library librte_pipeline (typical case), and to non-NULL otherwise. This mechanism is made available to support quick integration of legacy code. This function is expected to provide the packet processing related code to be called as part of the CPU thread dispatch loop, so this function is not allowed to contain an infinite loop.
f_timer	Function pointer	Function to read the pipeline input message queues, handle the request messages, create response messages and write the response queues. The format of request and response messages is defined by each pipeline type, with the exception of some requests which are mandatory for all pipelines (e.g. ping, statistics).
f_track	Function pointer	See section Tracking pipeline output port to physical link

Table 7.15: Pipeline front-end

Field name	Field type	Description
f_init	Function pointer	Function to initialize the front-end of the current pipeline instance.
f_free	Function pointer	Function to free the resources allocated by the front-end of the current pipeline instance.
cmds	Array of CLI commands	Array of CLI commands to be registered to the application CLI for the current pipeline type. Even though the CLI is executed by a different pipeline (typically, this is the master pipeline), from modularity perspective is more efficient to keep the message client side (part of the front-end) together with the message server side (part of the back-end).

Tracking pipeline output port to physical link

Each pipeline instance is a standalone block that does not have visibility into the other pipeline instances or the application-level pipeline inter-connectivity. In some cases, it is useful for a pipeline instance to get application level information related to pipeline connectivity, such as to identify the output link (e.g. physical NIC port) where one of its output ports connected, either directly or indirectly by traversing other pipeline instances.

Tracking can be successful or unsuccessful. Typically, tracking for a specific pipeline instance is successful when each one of its input ports can be mapped to a single output port, meaning that all packets read from the current input port can only go out on a single output port. Depending on the pipeline type, some exceptions may be allowed: a small portion of the packets, considered exception packets, are sent out on an output port that is pre-configured for this purpose.

For pass-through pipeline type, the tracking is always successful. For pipeline types as flow classification, firewall or routing, the tracking is only successful when the number of output ports for the current pipeline instance is 1.

This feature is used by the IP routing pipeline for adding/removing implicit routes every time a link is brought up/down.

Table copies

Fast table copy: pipeline table used by pipeline for the packet processing task, updated through messages, table data structures are optimized for lookup operation.

Slow table copy: used by the configuration layer, typically updated through CLI commands, kept in sync with the fast copy (its update triggers the fast copy update). Required for executing advanced table queries without impacting the packet processing task, therefore the slow copy is typically organized using different criteria than the fast copy.

Examples:

- Flow classification: Search through current set of flows (e.g. list all flows with a specific source IP address);
- Firewall: List rules in descending order of priority;
- Routing table: List routes sorted by prefix depth and their type (local, remote, default);
- ARP: List entries sorted per output interface.

Packet meta-data

Packet meta-data field offsets provided as argument to pipeline instances are essentially defining the data structure for the packet meta-data used by the current application use-case. It is very useful to put it in the configuration file as a comment in order to facilitate the readability of the configuration file.

The reason to use field offsets for defining the data structure for the packet meta-data is due to the C language limitation of not being able to define data structures at run-time. Feature to consider: have the configuration file parser automatically generate and print the data structure defining the packet meta-data for the current application use-case.

Packet meta-data typically contains:

1. Pure meta-data: intermediate data per packet that is computed internally, passed between different tables of the same pipeline instance (e.g. lookup key for the ARP table is obtained from the routing table), or between different pipeline instances (e.g. flow ID, traffic metering color, etc);
2. Packet fields: typically, packet header fields that are read directly from the packet, or read from the packet and saved (duplicated) as a working copy at a different location within the packet meta-data (e.g. Diffserv 5-tuple, IP destination address, etc).

Several strategies are used to design the packet meta-data, as described in the next subsections.

Store packet meta-data in a different cache line as the packet headers This approach is able to support protocols with variable header length, like MPLS, where the offset of IP header from the start of the packet (and, implicitly, the offset of the IP header in the packet buffer) is not fixed. Since the pipelines typically require the specification of a fixed offset to the packet fields (e.g. Diffserv 5-tuple, used by the flow classification pipeline, or the IP destination address, used by the IP routing pipeline), the workaround is to have the packet RX pipeline copy these fields at fixed offsets within the packet meta-data.

As this approach duplicates some of the packet fields, it requires accessing more cache lines per packet for filling in selected packet meta-data fields (on RX), as well as flushing selected packet meta-data fields into the packet (on TX).

Example:

```
; struct app_pkt_metadata {
;   uint32_t ip_da;
;      uint32_t hash;
;      uint32_t flow_id;
;      uint32_t color;
; } __attribute__((__packed__));
;

[PIPELINE1]
; Packet meta-data offsets
ip_da_offset = 0;   Used by: routing
hash_offset = 4;    Used by: RX, flow classification
flow_id_offset = 8; Used by: flow classification, flow actions
color_offset = 12;  Used by: flow actions, routing
```

Overlay the packet meta-data in the same cache line with the packet headers This approach is minimizing the number of cache line accessed per packet by storing the packet metadata in the same cache line with the packet headers. To enable this strategy, either some headroom is reserved for meta-data at the beginning of the packet headers cache line (e.g. if 16 bytes are needed for meta-data, then the packet headroom can be set to 128+16 bytes, so that NIC writes the first byte of the packet at offset 16 from the start of the first packet cache line), or meta-data is reusing the space of some packet headers that are discarded from the packet (e.g. input Ethernet header).

Example:

```
; struct app_pkt_metadata {
;     uint8_t headroom[RTE_PKTMBUF_HEADROOM]; /* 128 bytes (default) */
;     union {
```

```
;           struct {
;               struct ether_hdr ether; /* 14 bytes */
;               struct qinq_hdr qinq; /* 8 bytes */
;           };
;           struct {
;               uint32_t hash;
;               uint32_t flow_id;
;               uint32_t color;
;           };
;       };
;       struct ipv4_hdr ip; /* 20 bytes */
; } __attribute__((__packed__));
;
[PIPELINE2]
; Packet meta-data offsets
qinq_offset = 142;    Used by: RX, flow classification
ip_da_offset = 166;   Used by: routing
hash_offset = 128;    Used by: RX, flow classification
flow_id_offset = 132; Used by: flow classification, flow actions
color_offset = 136;   Used by: flow actions, routing
```

List of pipeline types

Table 7.16: List of pipeline types provided with the application

Name	Table(s)	Actions	Messages
Pass-through Note: depending on port type, can be used for RX, TX, IP fragmentation, IP reassembly or Traffic Management	Passthrough	1. Pkt metadata build 2. Flow hash 3. Pkt checks 4. Load balancing	1. Ping 2. Stats
Flow classification	Exact match • Key = byte array (source: pkt metadata) • Data = action dependent	1. Flow ID 2. Flow stats 3. Metering 4. Network Address 5. Translation (NAT)	1. Ping 2. Stats 3. Flow stats 4. Action stats 5. Flow add/ update/ delete 6. Default flow add/ update/ delete 7. Action update
Flow actions	Array • Key = Flow ID (source: pkt metadata) • Data = action dependent	1. Flow stats 2. Metering 3. Network Address 4. Translation (NAT)	1. Ping 2. Stats 3. Action stats 4. Action update
Firewall	ACL • Key = n-tuple (source: pkt headers) • Data = none	1. Allow/Drop	1. Ping 2. Stats 3. Rule add/ update/ delete 4. Default rule add/ update/ delete
IP routing	LPM (IPv4 or IPv6, depending on pipeline type) • Key = IP destination (source: pkt metadata) • Data = Dependent on actions and next hop type Hash table (for ARP, only when ARP is enabled) • Key = (Port ID, next hop IP address) (source: pkt meta-data) • Data: MAC address	1. TTL decrement and 2. IPv4 checksum 3. update 4. Header 5. encapsulation 6. (based on next hop 7. type)	1. Ping 2. Stats 3. Route add/ update/ delete 4. Default route add/ update/ delete 5. ARP entry add/ update/ delete 6. Default ARP entry add/ update/ delete

7.31.7 Command Line Interface (CLI)

Global CLI commands

Table 7.17: Global CLI commands

Command	Description	Syntax
run	Run CLI commands script file.	run <file> <file> = path to file with CLI commands to execute
quit	Gracefully terminate the application.	quit

CLI commands for link configuration

Table 7.18: List of run-time configuration commands for link configuration

Command	Description	Syntax
link config	Link configuration	link <link ID> config <IP address> <depth>
link up	Link up	link <link ID> up
link down	Link down	link <link ID> down
link ls	Link list	link ls

CLI commands common for all pipeline types

Table 7.19: CLI commands mandatory for all pipelines

Command	Description	Syntax
ping	Check whether specific pipeline instance is alive. The master pipeline sends a ping request message to given pipeline instance and waits for a response message back. Timeout message is displayed when the response message is not received before the timer expires.	p <pipeline ID> ping
stats	Display statistics for specific pipeline input port, output port or table.	p <pipeline ID> stats port in <port in ID> p <pipeline ID> stats port out <port out ID> p <pipeline ID> stats table <table ID>
input port enable	Enable given input port for specific pipeline instance.	p <pipeline ID> port in <port ID> enable
input port disable	Disable given input port for specific pipeline instance.	p <pipeline ID> port in <port ID> disable

Pipeline type specific CLI commands

The pipeline specific CLI commands are part of the pipeline type front-end.

7.32 Test Pipeline Application

The Test Pipeline application illustrates the use of the DPDK Packet Framework tool suite. Its purpose is to demonstrate the performance of single-table DPDK pipelines.

7.32.1 Overview

The application uses three CPU cores:

- Core A ("RX core") receives traffic from the NIC ports and feeds core B with traffic through SW queues.
- Core B ("Pipeline core") implements a single-table DPDK pipeline whose type is selectable through specific command line parameter. Core B receives traffic from core A through software queues, processes it according to the actions configured in the table entries that are hit by the input packets and feeds it to core C through another set of software queues.
- Core C ("TX core") receives traffic from core B through software queues and sends it to the NIC ports for transmission.

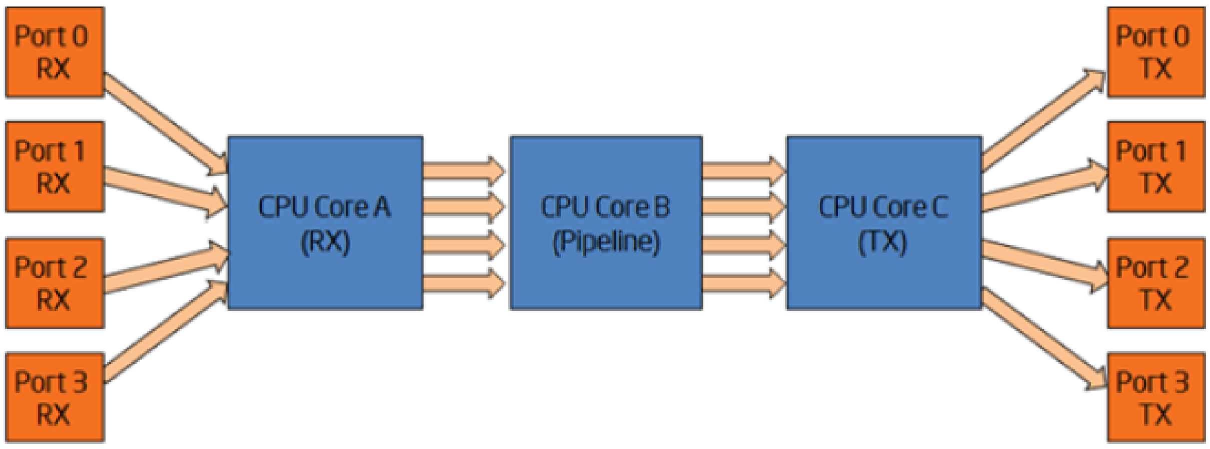


Fig. 7.28: Test Pipeline Application

7.32.2 Compiling the Application

1. Go to the app/test directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/app/test/test-pipeline
```

2. Set the target (a default target is used if not specified):

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

3. Build the application:

```
make
```

7.32.3 Running the Application

Application Command Line

The application execution command line is:

```
./test-pipeline [EAL options] -- -p PORTMASK --TABLE_TYPE
```

The -c EAL CPU core mask option has to contain exactly 3 CPU cores. The first CPU core in the core mask is assigned for core A, the second for core B and the third for core C.

The PORTMASK parameter must contain 2 or 4 ports.

Table Types and Behavior

Table 7.20 describes the table types used and how they are populated.

The hash tables are pre-populated with 16 million keys. For hash tables, the following parameters can be selected:

- **Configurable key size implementation or fixed (specialized) key size implementation (e.g. hash-8-ext or hash-spec-8-ext).** The key size specialized implementations are expected to provide better performance for 8-byte and 16-byte key sizes, while the key-size-non-specialized implementation is expected to provide better performance for larger key sizes;
- **Key size (e.g. hash-spec-8-ext or hash-spec-16-ext).** The available options are 8, 16 and 32 bytes;
- **Table type (e.g. hash-spec-16-ext or hash-spec-16-lru).** The available options are ext (extendable bucket) or lru (least recently used).

Table 7.20: Table Types

#	TABLE_TYPE	Description of Core B Table	Pre-added Table Entries
1	none	Core B is not implementing a DPDK pipeline. Core B is implementing a pass-through from its input set of software queues to its output set of software queues.	N/A
2	stub	Stub table. Core B is implementing the same pass-through functionality as described for the "none" option by using the DPDK Packet Framework by using one stub table for each input NIC port.	N/A
3	hash-[spec]-8-lru	LRU hash table with 8-byte key size and 16 million entries.	16 million entries are successfully added to the hash table with the following key format: [4-byte index, 4 bytes of 0] The action configured for all table entries is "Sendto output port", with the output port index uniformly distributed for the range of output ports. The default table rule (used in the case of a lookup miss) is to drop the packet. At run time, core A is creating the following lookup key and storing it into the packet meta data for core B to use for table lookup: [destination IPv4 address, 4 bytes of 0]
4	hash-[spec]-8-ext	Extendable bucket hash table with 8-byte key size and 16 million entries.	Same as hash-[spec]-8-lru table entries, above.
5	hash-[spec]-16-lru	LRU hash table with 16-byte key size and 16 million entries.	16 million entries are successfully added to the hash table with the following key format: [4-byte index, 12

Input Traffic

Regardless of the table type used for the core B pipeline, the same input traffic can be used to hit all table entries with uniform distribution, which results in uniform distribution of packets sent out on the set of output NIC ports. The profile for input traffic is TCP/IPv4 packets with:

- destination IP address as A.B.C.D with A fixed to 0 and B, C,D random
- source IP address fixed to 0.0.0.0
- destination TCP port fixed to 0
- source TCP port fixed to 0

7.33 Distributor Sample Application

The distributor sample application is a simple example of packet distribution to cores using the Data Plane Development Kit (DPDK).

7.33.1 Overview

The distributor application performs the distribution of packets that are received on an RX_PORT to different cores. When processed by the cores, the destination port of a packet is the port from the enabled port mask adjacent to the one on which the packet was received, that is, if the first four ports are enabled (port mask 0xf), ports 0 and 1 RX/TX into each other, and ports 2 and 3 RX/TX into each other.

This application can be used to benchmark performance using the traffic generator as shown in the figure below.

Fig. 7.29: Performance Benchmarking Setup (Basic Environment)

7.33.2 Compiling the Application

1. Go to the sample application directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/distributor
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

7.33.3 Running the Application

1. The application has a number of command line options:

   ```
   ./build/distributor_app [EAL options] -- -p PORTMASK
   ```

where,

- -p PORTMASK: Hexadecimal bitmask of ports to configure

2. To run the application in linuxapp environment with 10 lcores, 4 ports, issue the command:

```
$ ./build/distributor_app -c 0x4003fe -n 4 -- -p f
```

3. Refer to the DPDK Getting Started Guide for general information on running applications and the Environment Abstraction Layer (EAL) options.

7.33.4 Explanation

The distributor application consists of three types of threads: a receive thread (lcore_rx()), a set of worker threads(lcore_worker()) and a transmit thread(lcore_tx()). How these threads work together is shown in Fig. 7.30 below. The main() function launches threads of these three types. Each thread has a while loop which will be doing processing and which is terminated only upon SIGINT or ctrl+C. The receive and transmit threads communicate using a software ring (rte_ring structure).

The receive thread receives the packets using rte_eth_rx_burst() and gives them to the distributor (using rte_distributor_process() API) which will be called in context of the receive thread itself. The distributor distributes the packets to workers threads based on the tagging of the packet - indicated by the hash field in the mbuf. For IP traffic, this field is automatically filled by the NIC with the “usr” hash value for the packet, which works as a per-flow tag.

More than one worker thread can exist as part of the application, and these worker threads do simple packet processing by requesting packets from the distributor, doing a simple XOR operation on the input port mbuf field (to indicate the output port which will be used later for packet transmission) and then finally returning the packets back to the distributor in the RX thread.

Meanwhile, the receive thread will call the distributor api rte_distributor_returned_pkts() to get the packets processed, and will enqueue them to a ring for transfer to the TX thread for transmission on the output port. The transmit thread will dequeue the packets from the ring and transmit them on the output port specified in packet mbuf.

Users who wish to terminate the running of the application have to press ctrl+C (or send SIGINT to the app). Upon this signal, a signal handler provided in the application will terminate all running threads gracefully and print final statistics to the user.

Fig. 7.30: Distributor Sample Application Layout

7.33.5 Debug Logging Support

Debug logging is provided as part of the application; the user needs to uncomment the line “#define DEBUG” defined in start of the application in main.c to enable debug logs.

7.33.6 Statistics

Upon SIGINT (or) ctrl+C, the print_stats() function displays the count of packets processed at the different stages in the application.

7.33.7 Application Initialization

Command line parsing is done in the same way as it is done in the L2 Forwarding Sample Application. See Section 9.4.1, “Command Line Arguments”.

Mbuf pool initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See Section 9.4.2, “Mbuf Pool Initialization”.

Driver Initialization is done in same way as it is done in the L2 Forwarding Sample Application. See Section 9.4.3, “Driver Initialization”.

RX queue initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See Section 9.4.4, “RX Queue Initialization”.

TX queue initialization is done in the same way as it is done in the L2 Forwarding Sample Application. See Section 9.4.5, “TX Queue Initialization”.

7.34 VM Power Management Application

7.34.1 Introduction

Applications running in Virtual Environments have an abstract view of the underlying hardware on the Host, in particular applications cannot see the binding of virtual to physical hardware. When looking at CPU resourcing, the pinning of Virtual CPUs(vCPUs) to Host Physical CPUs(pCPUS) is not apparent to an application and this pinning may change over time. Furthermore, Operating Systems on virtual machines do not have the ability to govern their own power policy; the Machine Specific Registers (MSRs) for enabling P-State transitions are not exposed to Operating Systems running on Virtual Machines(VMs).

The Virtual Machine Power Management solution shows an example of how a DPDK application can indicate its processing requirements using VM local only information(vCPU/lcore) to a Host based Monitor which is responsible for accepting requests for frequency changes for a vCPU, translating the vCPU to a pCPU via libvirt and affecting the change in frequency.

The solution is comprised of two high-level components:

1. Example Host Application

   Using a Command Line Interface(CLI) for VM->Host communication channel management allows adding channels to the Monitor, setting and querying the vCPU to pCPU pinning, inspecting and manually changing the frequency for each CPU. The CLI runs on a single lcore while the thread responsible for managing VM requests runs on a second lcore.

   VM requests arriving on a channel for frequency changes are passed to the librte_power ACPI cpufreq sysfs based library. The Host Application relies on both qemu-kvm and libvirt to function.

2. librte_power for Virtual Machines

   Using an alternate implementation for the librte_power API, requests for frequency changes are forwarded to the host monitor rather than the APCI cpufreq sysfs interface used on the host.

   The l3fwd-power application will use this implementation when deployed on a VM (see Chapter 11 “L3 Forwarding with Power Management Application”).

Fig. 7.31: Highlevel Solution

7.34.2 Overview

VM Power Management employs qemu-kvm to provide communications channels between the host and VMs in the form of Virtio-Serial which appears as a paravirtualized serial device on a VM and can be configured to use various backends on the host. For this example each Virtio-Serial endpoint on the host is configured as AF_UNIX file socket, supporting poll/select and epoll for event notification. In this example each channel endpoint on the host is monitored via epoll for EPOLLIN events. Each channel is specified as qemu-kvm arguments or as libvirt XML for each VM, where each VM can have a number of channels up to a maximum of 64 per VM, in this example each DPDK lcore on a VM has exclusive access to a channel.

To enable frequency changes from within a VM, a request via the librte_power interface is forwarded via Virtio-Serial to the host, each request contains the vCPU and power command(scale up/down/min/max). The API for host and guest librte_power is consistent across environments, with the selection of VM or Host Implementation determined at automatically at runtime based on the environment.

Upon receiving a request, the host translates the vCPU to a pCPU via the libvirt API before forwarding to the host librte_power.

Fig. 7.32: VM request to scale frequency

Performance Considerations

While Haswell Microarchitecture allows for independent power control for each core, earlier Microarchtectures do not offer such fine grained control. When deployed on pre-Haswell platforms greater care must be taken in selecting which cores are assigned to a VM, for instance a core will not scale down until its sibling is similarly scaled.

7.34.3 Configuration

BIOS

Enhanced Intel SpeedStep® Technology must be enabled in the platform BIOS if the power management feature of DPDK is to be used. Otherwise, the sys file folder /sys/devices/system/cpu/cpu0/cpufreq will not exist, and the CPU frequency-based power management cannot be used. Consult the relevant BIOS documentation to determine how these settings can be accessed.

Host Operating System

The Host OS must also have the *apci_cpufreq* module installed, in some cases the *intel_pstate* driver may be the default Power Management environment. To enable *acpi_cpufreq* and disable *intel_pstate*, add the following to the grub Linux command line:

```
intel_pstate=disable
```

Upon rebooting, load the *acpi_cpufreq* module:

```
modprobe acpi_cpufreq
```

Hypervisor Channel Configuration

Virtio-Serial channels are configured via libvirt XML:

```
<name>{vm_name}</name>
<controller type='virtio-serial' index='0'>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
</controller>
<channel type='unix'>
  <source mode='bind' path='/tmp/powermonitor/{vm_name}.{channel_num}'/>
  <target type='virtio' name='virtio.serial.port.poweragent.{vm_channel_num}/>
  <address type='virtio-serial' controller='0' bus='0' port='{N}'/>
</channel>
```

Where a single controller of type *virtio-serial* is created and up to 32 channels can be associated with a single controller and multiple controllers can be specified. The convention is to use the name of the VM in the host path *{vm_name}* and to increment *{channel_num}* for each channel, likewise the port value *{N}* must be incremented for each channel.

Each channel on the host will appear in *path*, the directory */tmp/powermonitor/* must first be created and given qemu permissions

```
mkdir /tmp/powermonitor/
chown qemu:qemu /tmp/powermonitor
```

Note that files and directories within /tmp are generally removed upon rebooting the host and the above steps may need to be carried out after each reboot.

The serial device as it appears on a VM is configured with the *target* element attribute *name* and must be in the form of *virtio.serial.port.poweragent.{vm_channel_num}*, where *vm_channel_num* is typically the lcore channel to be used in DPDK VM applications.

Each channel on a VM will be present at */dev/virtio-ports/virtio.serial.port.poweragent.{vm_channel_num}*

7.34.4 Compiling and Running the Host Application

Compiling

1. export RTE_SDK=/path/to/rte_sdk
2. cd ${RTE_SDK}/examples/vm_power_manager
3. make

Running

The application does not have any specific command line options other than *EAL*:

```
./build/vm_power_mgr [EAL options]
```

The application requires exactly two cores to run, one core is dedicated to the CLI, while the other is dedicated to the channel endpoint monitor, for example to run on cores 0 & 1 on a system with 4 memory channels:

```
./build/vm_power_mgr -c 0x3 -n 4
```

After successful initialization the user is presented with VM Power Manager CLI:

```
vm_power>
```

Virtual Machines can now be added to the VM Power Manager:

```
vm_power> add_vm {vm_name}
```

When a {vm_name} is specified with the *add_vm* command a lookup is performed with libvirt to ensure that the VM exists, {vm_name} is used as an unique identifier to associate channels with a particular VM and for executing operations on a VM within the CLI. VMs do not have to be running in order to add them.

A number of commands can be issued via the CLI in relation to VMs:

> Remove a Virtual Machine identified by {vm_name} from the VM Power Manager.
>
> ```
> rm_vm {vm_name}
> ```
>
> Add communication channels for the specified VM, the virtio channels must be enabled in the VM configuration(qemu/libvirt) and the associated VM must be active. {list} is a comma-separated list of channel numbers to add, using the keyword 'all' will attempt to add all channels for the VM:
>
> ```
> add_channels {vm_name} {list}|all
> ```
>
> Enable or disable the communication channels in {list}(comma-separated) for the specified VM, alternatively list can be replaced with keyword 'all'. Disabled channels will still receive packets on the host, however the commands they specify will be ignored. Set status to 'enabled' to begin processing requests again:
>
> ```
> set_channel_status {vm_name} {list}|all enabled|disabled
> ```
>
> Print to the CLI the information on the specified VM, the information lists the number of vCPUS, the pinning to pCPU(s) as a bit mask, along with any communication channels associated with each VM, along with the status of each channel:
>
> ```
> show_vm {vm_name}
> ```
>
> Set the binding of Virtual CPU on VM with name {vm_name} to the Physical CPU mask:
>
> ```
> set_pcpu_mask {vm_name} {vcpu} {pcpu}
> ```
>
> Set the binding of Virtual CPU on VM to the Physical CPU:
>
> ```
> set_pcpu {vm_name} {vcpu} {pcpu}
> ```

Manual control and inspection can also be carried in relation CPU frequency scaling:

> Get the current frequency for each core specified in the mask:
>
> ```
> show_cpu_freq_mask {mask}
> ```
>
> Set the current frequency for the cores specified in {core_mask} by scaling each up/down/min/max:
>
> ```
> set_cpu_freq {core_mask} up|down|min|max
> ```
>
> Get the current frequency for the specified core:
>
> ```
> show_cpu_freq {core_num}
> ```
>
> Set the current frequency for the specified core by scaling up/down/min/max:

```
    set_cpu_freq {core_num} up|down|min|max
```

7.34.5 Compiling and Running the Guest Applications

For compiling and running l3fwd-power, see Chapter 11 “L3 Forwarding with Power Management Application”.

A guest CLI is also provided for validating the setup.

For both l3fwd-power and guest CLI, the channels for the VM must be monitored by the host application using the *add_channels* command on the host.

Compiling

1. export RTE_SDK=/path/to/rte_sdk
2. cd ${RTE_SDK}/examples/vm_power_manager/guest_cli
3. make

Running

The application does not have any specific command line options other than *EAL*:

```
./build/vm_power_mgr [EAL options]
```

The application for example purposes uses a channel for each lcore enabled, for example to run on cores 0,1,2,3 on a system with 4 memory channels:

```
./build/guest_vm_power_mgr -c 0xf -n 4
```

After successful initialization the user is presented with VM Power Manager Guest CLI:

```
vm_power(guest)>
```

To change the frequency of a lcore, use the set_cpu_freq command. Where {core_num} is the lcore and channel to change frequency by scaling up/down/min/max.

```
set_cpu_freq {core_num} up|down|min|max
```

7.35 TEP termination Sample Application

The TEP (Tunnel End point) termination sample application simulates a VXLAN Tunnel Endpoint (VTEP) termination in DPDK, which is used to demonstrate the offload and filtering capabilities of Intel® XL710 10/40 Gigabit Ethernet Controller for VXLAN packet. This sample uses the basic virtio devices management mechanism from vhost example, and also uses the us-vHost interface and tunnel filtering mechanism to direct a specified traffic to a specific VM. In addition, this sample is also designed to show how tunneling protocols can be handled.

7.35.1 Background

With virtualization, overlay networks allow a network structure to be built or imposed across physical nodes which is abstracted away from the actual underlining physical network connections. This allows network isolation, QOS, etc to be provided on a per client basis.

Fig. 7.33: Overlay Networking.

In a typical setup, the network overlay tunnel is terminated at the Virtual/Tunnel End Point (VEP/TEP). The TEP is normally located at the physical host level ideally in the software switch. Due to processing constraints and the inevitable bottleneck that the switch becomes the ability to offload overlay support features becomes an important requirement. Intel® XL710 10/40 G Ethernet network card provides hardware filtering and offload capabilities to support overlay networks implementations such as MAC in UDP and MAC in GRE.

7.35.2 Sample Code Overview

The DPDK TEP termination sample code demonstrates the offload and filtering capabilities of Intel® XL710 10/40 Gigabit Ethernet Controller for VXLAN packet.

The sample code is based on vhost library. The vhost library is developed for user space Ethernet switch to easily integrate with vhost functionality.

The sample will support the followings:

- Tunneling packet recognition.
- The port of UDP tunneling is configurable
- Directing incoming traffic to the correct queue based on the tunnel filter type. The supported filter type are listed below.
  - Inner MAC and VLAN and tenant ID
  - Inner MAC and tenant ID, and Outer MAC
  - Inner MAC and tenant ID

  The tenant ID will be assigned from a static internal table based on the us-vhost device ID. Each device will receive a unique device ID. The inner MAC will be learned by the first packet transmitted from a device.
- Decapsulation of RX VXLAN traffic. This is a software only operation.
- Encapsulation of TX VXLAN traffic. This is a software only operation.
- Inner IP and inner L4 checksum offload.
- TSO offload support for tunneling packet.

The following figure shows the framework of the TEP termination sample application based on vhost-cuse.

Fig. 7.34: TEP termination Framework Overview

7.35.3 Supported Distributions

The example in this section have been validated with the following distributions:

- Fedora* 18
- Fedora* 19

- Fedora* 20

7.35.4 Prerequisites

Refer to the guide in section 27.4 in the vhost sample.

7.35.5 Compiling the Sample Code

1. Compile vhost lib:

   To enable vhost, turn on vhost library in the configure file config/common_linuxapp.

   ```
   CONFIG_RTE_LIBRTE_VHOST=n
   ```

   vhost user is turned on by default in the configure file config/common_linuxapp. To enable vhost cuse, disable vhost user.

   ```
      CONFIG_RTE_LIBRTE_VHOST_USER=y

   After vhost is enabled and the implementation is selected, build the vhost library.
   ```

2. Go to the examples directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/tep_termination
   ```

3. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

4. Build the application:

   ```
   cd ${RTE_SDK}
   make config ${RTE_TARGET}
   make install ${RTE_TARGET}
   cd ${RTE_SDK}/examples/tep_termination
   make
   ```

5. Go to the eventfd_link directory(vhost cuse required):

   ```
   cd ${RTE_SDK}/lib/librte_vhost/eventfd_link
   ```

6. Build the eventfd_link kernel module(vhost cuse required):

   ```
   make
   ```

7.35.6 Running the Sample Code

1. Install the cuse kernel module(vhost cuse required):

   ```
   modprobe cuse
   ```

2. Go to the eventfd_link directory(vhost cuse required):

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/lib/librte_vhost/eventfd_link
   ```

3. Install the eventfd_link module(vhost cuse required):

   ```
   insmod ./eventfd_link.ko
   ```

4. Go to the examples directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/tep_termination
```

5. Run the tep_termination sample code:

```
user@target:~$ ./build/app/tep_termination -c f -n 4 --huge-dir /mnt/huge --
                -p 0x1 --dev-basename tep-termination --nb-devices 4
                --udp-port 4789 --filter-type 1
```

Note: Please note the huge-dir parameter instructs the DPDK to allocate its memory from the 2 MB page hugetlbfs.

Parameters

The same parameters with the vhost sample.

Refer to the guide in section 27.6.1 in the vhost sample for the meanings of 'Basename', 'Stats', 'RX Retry', 'RX Retry Number' and 'RX Retry Delay Time'.

Number of Devices.

The nb-devices option specifies the number of virtIO device. The default value is 2.

```
user@target:~$ ./build/app/tep_termination -c f -n 4 --huge-dir /mnt/huge --
                --nb-devices 2
```

Tunneling UDP port.

The udp-port option is used to specify the destination UDP number for UDP tunneling packet. The default value is 4789.

```
user@target:~$ ./build/app/tep_termination -c f -n 4 --huge-dir /mnt/huge --
                --nb-devices 2 --udp-port 4789
```

Filter Type.

The filter-type option is used to specify which filter type is used to filter UDP tunneling packet to a specified queue. The default value is 1, which means the filter type of inner MAC and tenant ID is used.

```
user@target:~$ ./build/app/tep_termination -c f -n 4 --huge-dir /mnt/huge --
            --nb-devices 2 --udp-port 4789 --filter-type 1
```

TX Checksum.

The tx-checksum option is used to enable or disable the inner header checksum offload. The default value is 0, which means the checksum offload is disabled.

```
user@target:~$ ./build/app/tep_termination -c f -n 4 --huge-dir /mnt/huge --
            --nb-devices 2 --tx-checksum
```

TCP segment size.

The tso-segsz option specifies the TCP segment size for TSO offload for tunneling packet. The default value is 0, which means TSO offload is disabled.

```
user@target:~$ ./build/app/tep_termination -c f -n 4 --huge-dir /mnt/huge --
            --tx-checksum --tso-segsz 800
```

Decapsulation option.

The decap option is used to enable or disable decapsulation operation for received VXLAN packet. The default value is 1.

```
user@target:~$ ./build/app/tep_termination -c f -n 4 --huge-dir /mnt/huge --
            --nb-devices 4 --udp-port 4789 --decap 1
```

Encapsulation option.

The encap option is used to enable or disable encapsulation operation for transmitted packet. The default value is 1.

```
user@target:~$ ./build/app/tep_termination -c f -n 4 --huge-dir /mnt/huge --
            --nb-devices 4 --udp-port 4789 --encap 1
```

7.35.7 Running the Virtual Machine (QEMU)

Refer to the guide in section 27.7 in the vhost sample.

7.35.8 Running DPDK in the Virtual Machine

Refer to the guide in section 27.8 in the vHost sample.

7.35.9 Passing Traffic to the Virtual Machine Device

For a virtio-net device to receive traffic, the traffic's Layer 2 header must include both the virtio-net device's MAC address. The DPDK sample code behaves in a similar manner to a learning switch in that it learns the MAC address of the virtio-net devices from the first transmitted packet. On learning the MAC address, the DPDK vhost sample code prints a message with the MAC address and tenant ID virtio-net device. For example:

```
DATA: (0) MAC_ADDRESS cc:bb:bb:bb:bb:bb and VNI 1000 registered
```

The above message indicates that device 0 has been registered with MAC address cc:bb:bb:bb:bb:bb and VNI 1000. Any packets received on the NIC with these values are placed on the devices receive queue.

7.36 dpdk_proc_info Application

The dpdk_proc_info application is a Data Plane Development Kit (DPDK) application that runs as a DPDK secondary process and is capable of retrieving port statistics, resetting port statistics and printing DPDK memory information. This application extends the original functionality that was supported by dump_cfg.

7.36.1 Running the Application

The application has a number of command line options:

```
./$(RTE_TARGET)/app/dpdk_proc_info -- -m | [-p PORTMASK] [--stats | --xstats |
--stats-reset | --xstats-reset]
```

Parameters

-p PORTMASK: Hexadecimal bitmask of ports to configure.

–stats The stats parameter controls the printing of generic port statistics. If no port mask is specified stats are printed for all DPDK ports.

–xstats The stats parameter controls the printing of extended port statistics. If no port mask is specified xstats are printed for all DPDK ports.

–stats-reset The stats-reset parameter controls the resetting of generic port statistics. If no port mask is specified, the generic stats are reset for all DPDK ports.

–xstats-reset The xstats-reset parameter controls the resetting of extended port statistics. If no port mask is specified xstats are reset for all DPDK ports.

-m: Print DPDK memory information.

7.37 PTP Client Sample Application

The PTP (Precision Time Protocol) client sample application is a simple example of using the DPDK IEEE1588 API to communicate with a PTP master clock to synchronize the time on the NIC and, optionally, on the Linux system.

Note, PTP is a time syncing protocol and cannot be used within DPDK as a time-stamping mechanism. See the following for an explanation of the protocol: Precision Time Protocol.

7.37.1 Limitations

The PTP sample application is intended as a simple reference implementation of a PTP client using the DPDK IEEE1588 API. In order to keep the application simple the following assumptions are made:

- The first discovered master is the master for the session.
- Only L2 PTP packets are supported.
- Only the PTP v2 protocol is supported.
- Only the slave clock is implemented.

7.37.2 How the Application Works

Fig. 7.35: PTP Synchronization Protocol

The PTP synchronization in the sample application works as follows:

- Master sends *Sync* message - the slave saves it as T2.
- Master sends *Follow Up* message and sends time of T1.
- Slave sends *Delay Request* frame to PTP Master and stores T3.
- Master sends *Delay Response* T4 time which is time of received T3.

The adjustment for slave can be represented as:

> adj = -[(T2-T1)-(T4 - T3)]/2

If the command line parameter `-T 1` is used the application also synchronizes the PTP PHC clock with the Linux kernel clock.

7.37.3 Compiling the Application

To compile the application, export the path to the DPDK source tree and edit the `config/common_linuxapp` configuration file to enable IEEE1588:

```
export RTE_SDK=/path/to/rte_sdk

# Edit  common_linuxapp and set the following options:
CONFIG_RTE_LIBRTE_IEEE1588=y
```

Set the target, for example:

```
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

See the *DPDK Getting Started* Guide for possible `RTE_TARGET` values.

Build the application as follows:

```
# Recompile DPDK.
make install T=$RTE_TARGET

# Compile the application.
cd ${RTE_SDK}/examples/ptpclient
make
```

7.37.4 Running the Application

To run the example in a `linuxapp` environment:

```
./build/ptpclient -c 2 -n 4 -- -p 0x1 -T 0
```

Refer to *DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

- `-p portmask`: Hexadecimal portmask.
- `-T 0`: Update only the PTP slave clock.
- `-T 1`: Update the PTP slave clock and synchronize the Linux Kernel to the PTP clock.

7.37.5 Code Explanation

The following sections provide an explanation of the main components of the code.

All DPDK library functions used in the sample code are prefixed with `rte_` and are explained in detail in the *DPDK API Documentation*.

The Main Function

The `main()` function performs the initialization and calls the execution threads for each lcore.

The first task is to initialize the Environment Abstraction Layer (EAL). The `argc` and `argv` arguments are provided to the `rte_eal_init()` function. The value returned is the number of parsed arguments:

```
int ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

And than we parse application specific arguments

```
argc -= ret;
argv += ret;

ret = ptp_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Error with PTP initialization\n");
```

The `main()` also allocates a mempool to hold the mbufs (Message Buffers) used by the application:

```
mbuf_pool = rte_mempool_create("MBUF_POOL",
                               NUM_MBUFS * nb_ports,
                               MBUF_SIZE,
                               MBUF_CACHE_SIZE,
                               sizeof(struct rte_pktmbuf_pool_private),
                               rte_pktmbuf_pool_init, NULL,
                               rte_pktmbuf_init,      NULL,
                               rte_socket_id(),
                               0);
```

Mbufs are the packet buffer structure used by DPDK. They are explained in detail in the “Mbuf Library” section of the *DPDK Programmer’s Guide*.

The `main()` function also initializes all the ports using the user defined `port_init()` function with portmask provided by user:

```
for (portid = 0; portid < nb_ports; portid++)
    if ((ptp_enabled_port_mask & (1 << portid)) != 0) {

        if (port_init(portid, mbuf_pool) == 0) {
            ptp_enabled_ports[ptp_enabled_port_nb] = portid;
            ptp_enabled_port_nb++;
        } else {
            rte_exit(EXIT_FAILURE, "Cannot init port %"PRIu8 "\n",
                    portid);
        }
    }
```

Once the initialization is complete, the application is ready to launch a function on an lcore. In this example `lcore_main()` is called on a single lcore.

```
lcore_main();
```

The `lcore_main()` function is explained below.

The Lcores Main

As we saw above the `main()` function calls an application function on the available lcores.

The main work of the application is done within the loop:

```
for (portid = 0; portid < ptp_enabled_port_nb; portid++) {

    portid = ptp_enabled_ports[portid];
    nb_rx = rte_eth_rx_burst(portid, 0, &m, 1);

    if (likely(nb_rx == 0))
        continue;

    if (m->ol_flags & PKT_RX_IEEE1588_PTP)
        parse_ptp_frames(portid, m);

    rte_pktmbuf_free(m);
}
```

Packets are received one by one on the RX ports and, if required, PTP response packets are transmitted on the TX ports.

If the offload flags in the mbuf indicate that the packet is a PTP packet then the packet is parsed to determine which type:

```
if (m->ol_flags & PKT_RX_IEEE1588_PTP)
     parse_ptp_frames(portid, m);
```

All packets are freed explicitly using `rte_pktmbuf_free()`.

The forwarding loop can be interrupted and the application closed using `Ctrl-C`.

PTP parsing

The `parse_ptp_frames()` function processes PTP packets, implementing slave PTP IEEE1588 L2 functionality.

```
void
parse_ptp_frames(uint8_t portid, struct rte_mbuf *m) {
    struct ptp_header *ptp_hdr;
    struct ether_hdr *eth_hdr;
    uint16_t eth_type;

    eth_hdr = rte_pktmbuf_mtod(m, struct ether_hdr *);
    eth_type = rte_be_to_cpu_16(eth_hdr->ether_type);

    if (eth_type == PTP_PROTOCOL) {
        ptp_data.m = m;
        ptp_data.portid = portid;
        ptp_hdr = (struct ptp_header *)(rte_pktmbuf_mtod(m, char *)
                    + sizeof(struct ether_hdr));

        switch (ptp_hdr->msgtype) {
        case SYNC:
            parse_sync(&ptp_data);
            break;
        case FOLLOW_UP:
            parse_fup(&ptp_data);
            break;
        case DELAY_RESP:
            parse_drsp(&ptp_data);
            print_clock_info(&ptp_data);
            break;
        default:
            break;
            }
        }
    }
```

There are 3 types of packets on the RX path which we must parse to create a minimal implementation of the PTP slave client:

- SYNC packet.
- FOLLOW UP packet
- DELAY RESPONSE packet.

When we parse the *FOLLOW UP* packet we also create and send a *DELAY_REQUEST* packet. Also when we parse the *DELAY RESPONSE* packet, and all conditions are met we adjust the PTP slave clock.

7.38 Performance Thread Sample Application

The performance thread sample application is a derivative of the standard L3 forwarding application that demonstrates different threading models.

7.38.1 Overview

For a general description of the L3 forwarding applications capabilities please refer to the documentation of the standard application in L3 Forwarding Sample Application.

The performance thread sample application differs from the standard L3 forwarding example in that it divides the TX and RX processing between different threads, and makes it possible to assign individual threads to different cores.

Three threading models are considered:

1. When there is one EAL thread per physical core.
2. When there are multiple EAL threads per physical core.
3. When there are multiple lightweight threads per EAL thread.

Since DPDK release 2.0 it is possible to launch applications using the `--lcores` EAL parameter, specifying cpu-sets for a physical core. With the performance thread sample application its is now also possible to assign individual RX and TX functions to different cores.

As an alternative to dividing the L3 forwarding work between different EAL threads the performance thread sample introduces the possibility to run the application threads as lightweight threads (L-threads) within one or more EAL threads.

In order to facilitate this threading model the example includes a primitive cooperative scheduler (L-thread) subsystem. More details of the L-thread subsystem can be found in *The L-thread subsystem*.

Note: Whilst theoretically possible it is not anticipated that multiple L-thread schedulers would be run on the same physical core, this mode of operation should not be expected to yield useful performance and is considered invalid.

7.38.2 Compiling the Application

The application is located in the sample application folder in the `performance-thread` folder.

1. Go to the example applications folder

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/performance-thread/l3fwd-thread
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the *DPDK Linux Getting Started Guide* for possible RTE_TARGET values.

3. Build the application:

   make

7.38.3 Running the Application

The application has a number of command line options:

```
./build/l3fwd-thread [EAL options] --
    -p PORTMASK [-P]
    --rx(port,queue,lcore,thread)[,(port,queue,lcore,thread)]
    --tx(lcore,thread)[,(lcore,thread)]
    [--enable-jumbo] [--max-pkt-len PKTLEN]]  [--no-numa]
    [--hash-entry-num] [--ipv6] [--no-lthreads] [--stat-lcore lcore]
```

Where:

- `-p PORTMASK`: Hexadecimal bitmask of ports to configure.
- `-P`: optional, sets all ports to promiscuous mode so that packets are accepted regardless of the packet's Ethernet MAC destination address. Without this option, only packets with the Ethernet MAC destination address set to the Ethernet address of the port are accepted.
- `--rx (port,queue,lcore,thread)[,(port,queue,lcore,thread)]`: the list of NIC RX ports and queues handled by the RX lcores and threads. The parameters are explained below.
- `--tx (lcore,thread)[,(lcore,thread)]`: the list of TX threads identifying the lcore the thread runs on, and the id of RX thread with which it is associated. The parameters are explained below.
- `--enable-jumbo`: optional, enables jumbo frames.
- `--max-pkt-len`: optional, maximum packet length in decimal (64-9600).
- `--no-numa`: optional, disables numa awareness.
- `--hash-entry-num`: optional, specifies the hash entry number in hex to be setup.
- `--ipv6`: optional, set it if running ipv6 packets.
- `--no-lthreads`: optional, disables l-thread model and uses EAL threading model. See below.
- `--stat-lcore`: optional, run CPU load stats collector on the specified lcore.

The parameters of the `--rx` and `--tx` options are:

- `--rx` parameters

port	RX port
queue	RX queue that will be read on the specified RX port
lcore	Core to use for the thread
thread	Thread id (continuously from 0 to N)

- `--tx` parameters

lcore	Core to use for L3 route match and transmit
thread	Id of RX thread to be associated with this TX thread

The `l3fwd-thread` application allows you to start packet processing in two threading models: L-Threads (default) and EAL Threads (when the `--no-lthreads` parameter is used). For consistency all parameters are used in the same way for both models.

Running with L-threads

When the L-thread model is used (default option), lcore and thread parameters in `--rx/--tx` are used to affinitize threads to the selected scheduler.

For example, the following places every l-thread on different lcores:

```
l3fwd-thread -c ff -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)"
```

The following places RX l-threads on lcore 0 and TX l-threads on lcore 1 and 2 and so on:

```
l3fwd-thread -c ff -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,0,1)" \
             --tx="(1,0)(2,1)"
```

Running with EAL threads

When the `--no-lthreads` parameter is used, the L-threading model is turned off and EAL threads are used for all processing. EAL threads are enumerated in the same way as L-threads, but the `--lcores` EAL parameter is used to affinitize threads to the selected cpu-set (scheduler). Thus it is possible to place every RX and TX thread on different lcores.

For example, the following places every EAL thread on different lcores:

```
l3fwd-thread -c ff -n 2 -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

To affinitize two or more EAL threads to one cpu-set, the EAL `--lcores` parameter is used.

The following places RX EAL threads on lcore 0 and TX EAL threads on lcore 1 and 2 and so on:

```
l3fwd-thread -c ff -n 2 --lcores="(0,1)@0,(2,3)@1" -- -P -p 3 \
             --rx="(0,0,0,0)(1,0,1,1)" \
             --tx="(2,0)(3,1)" \
             --no-lthreads
```

Examples

For selected scenarios the command line configuration of the application for L-threads and its corresponding EAL threads command line can be realized as follows:

1. Start every thread on different scheduler (1:1):

   ```
   l3fwd-thread -c ff -n 2 -- -P -p 3 \
                --rx="(0,0,0,0)(1,0,1,1)" \
                --tx="(2,0)(3,1)"
   ```

   EAL thread equivalent:

   ```
   l3fwd-thread -c ff -n 2 -- -P -p 3 \
                --rx="(0,0,0,0)(1,0,1,1)" \
                --tx="(2,0)(3,1)" \
                --no-lthreads
   ```

2. Start all threads on one core (N:1).

   Start 4 L-threads on lcore 0:

   ```
   l3fwd-thread -c ff -n 2 -- -P -p 3 \
                --rx="(0,0,0,0)(1,0,0,1)" \
                --tx="(0,0)(0,1)"
   ```

   Start 4 EAL threads on cpu-set 0:

   ```
   l3fwd-thread -c ff -n 2 --lcores="(0-3)@0" -- -P -p 3 \
                --rx="(0,0,0,0)(1,0,0,1)" \
                --tx="(2,0)(3,1)" \
                --no-lthreads
   ```

3. Start threads on different cores (N:M).

   Start 2 L-threads for RX on lcore 0, and 2 L-threads for TX on lcore 1:

   ```
   l3fwd-thread -c ff -n 2 -- -P -p 3 \
                --rx="(0,0,0,0)(1,0,0,1)" \
                --tx="(1,0)(1,1)"
   ```

   Start 2 EAL threads for RX on cpu-set 0, and 2 EAL threads for TX on cpu-set 1:

   ```
   l3fwd-thread -c ff -n 2 --lcores="(0-1)@0,(2-3)@1" -- -P -p 3 \
                --rx="(0,0,0,0)(1,0,1,1)" \
                --tx="(2,0)(3,1)" \
                --no-lthreads
   ```

7.38.4 Explanation

To a great extent the sample application differs little from the standard L3 forwarding application, and readers are advised to familiarize themselves with the material covered in the L3 Forwarding Sample Application documentation before proceeding.

The following explanation is focused on the way threading is handled in the performance thread example.

Mode of operation with EAL threads

The performance thread sample application has split the RX and TX functionality into two different threads, and the RX and TX threads are interconnected via software rings. With respect to these rings the RX threads are producers and the TX threads are consumers.

On initialization the TX and RX threads are started according to the command line parameters.

The RX threads poll the network interface queues and post received packets to a TX thread via a corresponding software ring.

The TX threads poll software rings, perform the L3 forwarding hash/LPM match, and assemble packet bursts before performing burst transmit on the network interface.

As with the standard L3 forward application, burst draining of residual packets is performed periodically with the period calculated from elapsed time using the timestamps counter.

The diagram below illustrates a case with two RX threads and three TX threads.

Mode of operation with L-threads

Like the EAL thread configuration the application has split the RX and TX functionality into different threads, and the pairs of RX and TX threads are interconnected via software rings.

On initialization an L-thread scheduler is started on every EAL thread. On all but the master EAL thread only a a dummy L-thread is initially started. The L-thread started on the master EAL thread then spawns other L-threads on different L-thread schedulers according the the command line parameters.

The RX threads poll the network interface queues and post received packets to a TX thread via the corresponding software ring.

The ring interface is augmented by means of an L-thread condition variable that enables the TX thread to be suspended when the TX ring is empty. The RX thread signals the condition whenever it posts to the TX ring, causing the TX thread to be resumed.

Additionally the TX L-thread spawns a worker L-thread to take care of polling the software rings, whilst it handles burst draining of the transmit buffer.

The worker threads poll the software rings, perform L3 route lookup and assemble packet bursts. If the TX ring is empty the worker thread suspends itself by waiting on the condition variable associated with the ring.

Burst draining of residual packets, less than the burst size, is performed by the TX thread which sleeps (using an L-thread sleep function) and resumes periodically to flush the TX buffer.

This design means that L-threads that have no work, can yield the CPU to other L-threads and avoid having to constantly poll the software rings.

The diagram below illustrates a case with two RX threads and three TX functions (each comprising a thread that processes forwarding and a thread that periodically drains the output buffer of residual packets).

CPU load statistics

It is possible to display statistics showing estimated CPU load on each core. The statistics indicate the percentage of CPU time spent: processing received packets (forwarding), polling queues/rings (waiting for work), and doing any other processing (context switch and other overhead).

When enabled statistics are gathered by having the application threads set and clear flags when they enter and exit pertinent code sections. The flags are then sampled in real time by a statistics collector thread running on another core. This thread displays the data in real time on the console.

This feature is enabled by designating a statistics collector core, using the `--stat-lcore` parameter.

7.38.5 The L-thread subsystem

The L-thread subsystem resides in the examples/performance-thread/common directory and is built and linked automatically when building the `l3fwd-thread` example.

The subsystem provides a simple cooperative scheduler to enable arbitrary functions to run as cooperative threads within a single EAL thread. The subsystem provides a pthread like API that is intended to assist in reuse of legacy code written for POSIX pthreads.

The following sections provide some detail on the features, constraints, performance and porting considerations when using L-threads.

Comparison between L-threads and POSIX pthreads

The fundamental difference between the L-thread and pthread models is the way in which threads are scheduled. The simplest way to think about this is to consider the case of a processor with a single CPU. To run multiple threads on a single CPU, the scheduler must frequently switch between the threads, in order that each thread is able to make timely progress. This is the basis of any multitasking operating system.

This section explores the differences between the pthread model and the L-thread model as implemented in the provided L-thread subsystem. If needed a theoretical discussion of preemptive vs cooperative multi-threading can be found in any good text on operating system design.

Scheduling and context switching

The POSIX pthread library provides an application programming interface to create and synchronize threads. Scheduling policy is determined by the host OS, and may be configurable. The OS may use sophisticated rules to determine which thread should be run next, threads may suspend themselves or make other threads ready, and the scheduler may employ a time slice giving each thread a maximum time quantum after which it will be preempted in favor of another thread that is ready to run. To complicate matters further threads may be assigned different scheduling priorities.

By contrast the L-thread subsystem is considerably simpler. Logically the L-thread scheduler performs the same multiplexing function for L-threads within a single pthread as the OS scheduler does for pthreads within an application process. The L-thread scheduler is simply the main loop of a pthread, and in so far as the host OS is concerned it is a regular pthread just like any other. The host OS is oblivious about the existence of and not at all involved in the scheduling of L-threads.

The other and most significant difference between the two models is that L-threads are scheduled cooperatively. L-threads cannot not preempt each other, nor can the L-thread scheduler preempt a running L-thread (i.e. there is no time slicing). The consequence is that programs implemented with L-threads must possess frequent rescheduling points, meaning that they must explicitly and of their own volition return to the scheduler at frequent intervals, in order to allow other L-threads an opportunity to proceed.

In both models switching between threads requires that the current CPU context is saved and a new context (belonging to the next thread ready to run) is restored. With pthreads this context switching is handled transparently and the set of CPU registers that must be preserved between context switches is as per an interrupt handler.

An L-thread context switch is achieved by the thread itself making a function call to the L-thread scheduler. Thus it is only necessary to preserve the callee registers. The caller is responsible to save and restore any other registers it is using before a function call, and restore them on return, and this is handled by the compiler. For `X86_64` on both Linux and BSD the System V calling convention is used, this defines registers RSP, RBP, and R12-R15 as callee-save registers (for more detailed discussion a good reference is X86 Calling Conventions).

Taking advantage of this, and due to the absence of preemption, an L-thread context switch is achieved with less than 20 load/store instructions.

The scheduling policy for L-threads is fixed, there is no prioritization of L-threads, all L-threads are equal and scheduling is based on a FIFO ready queue.

An L-thread is a struct containing the CPU context of the thread (saved on context switch) and other useful items. The ready queue contains pointers to threads that are ready to run. The L-thread scheduler is a simple loop that polls the ready queue, reads from it the next thread ready to run, which it resumes by saving the current context (the current position in the scheduler loop) and restoring the context of the next thread from its thread struct. Thus an L-thread is always resumed at the last place it yielded.

A well behaved L-thread will call the context switch regularly (at least once in its main loop) thus returning to the scheduler's own main loop. Yielding inserts the current thread at the back of the ready queue, and the process of servicing the ready queue is repeated, thus the system runs by flipping back and forth the between L-threads and scheduler loop.

In the case of pthreads, the preemptive scheduling, time slicing, and support for thread prioritization means that progress is normally possible for any thread that is ready to run. This comes at the price of a relatively heavier context switch and scheduling overhead.

With L-threads the progress of any particular thread is determined by the frequency of rescheduling opportunities in the other L-threads. This means that an errant L-thread monopolizing the CPU might cause scheduling of other threads to be stalled. Due to the lower cost of context switching, however, voluntary rescheduling to ensure progress of other threads, if managed sensibly, is not a prohibitive overhead, and overall performance can exceed that of an application using pthreads.

Mutual exclusion

With pthreads preemption means that threads that share data must observe some form of mutual exclusion protocol.

The fact that L-threads cannot preempt each other means that in many cases mutual exclusion devices can be completely avoided.

Locking to protect shared data can be a significant bottleneck in multi-threaded applications so a carefully designed cooperatively scheduled program can enjoy significant performance advantages.

So far we have considered only the simplistic case of a single core CPU, when multiple CPUs are considered things are somewhat more complex.

First of all it is inevitable that there must be multiple L-thread schedulers, one running on each EAL thread. So long as these schedulers remain isolated from each other the above assertions about the potential advantages of cooperative scheduling hold true.

A configuration with isolated cooperative schedulers is less flexible than the pthread model where threads can be affinitized to run on any CPU. With isolated schedulers scaling of applications to utilize fewer or more CPUs according to system demand is very difficult to achieve.

The L-thread subsystem makes it possible for L-threads to migrate between schedulers running on different CPUs. Needless to say if the migration means that threads that share data end up running on different CPUs then this will introduce the need for some kind of mutual exclusion system.

Of course `rte_ring` software rings can always be used to interconnect threads running on different cores, however to protect other kinds of shared data structures, lock free constructs or else explicit locking will be required. This is a consideration for the application design.

In support of this extended functionality, the L-thread subsystem implements thread safe mutexes and condition variables.

The cost of affinitizing and of condition variable signaling is significantly lower than the equivalent pthread operations, and so applications using these features will see a performance benefit.

Thread local storage

As with applications written for pthreads an application written for L-threads can take advantage of thread local storage, in this case local to an L-thread. An application may save and retrieve a single pointer to application data in the L-thread struct.

For legacy and backward compatibility reasons two alternative methods are also offered, the first is modelled directly on the pthread get/set specific APIs, the second approach is modelled on the `RTE_PER_LCORE` macros, whereby `PER_LTHREAD` macros are introduced, in both cases the storage is local to the L-thread.

Constraints and performance implications when using L-threads

API compatibility

The L-thread subsystem provides a set of functions that are logically equivalent to the corresponding functions offered by the POSIX pthread library, however not all pthread functions have a corresponding L-thread equivalent, and not all features available to pthreads are implemented for L-threads.

The pthread library offers considerable flexibility via programmable attributes that can be associated with threads, mutexes, and condition variables.

By contrast the L-thread subsystem has fixed functionality, the scheduler policy cannot be varied, and L-threads cannot be prioritized. There are no variable attributes associated with any L-thread objects. L-threads, mutexes and conditional variables, all have fixed functionality. (Note: reserved parameters are included in the APIs to facilitate possible future support for attributes).

The table below lists the pthread and equivalent L-thread APIs with notes on differences and/or constraints. Where there is no L-thread entry in the table, then the L-thread subsystem provides no equivalent function.

Table 7.21: Pthread and equivalent L-thread APIs.

Pthread function	L-thread function	Notes
pthread_barrier_destroy		
pthread_barrier_init		
pthread_barrier_wait		
pthread_cond_broadcast	lthread_cond_broadcast	See note 1
pthread_cond_destroy	lthread_cond_destroy	
pthread_cond_init	lthread_cond_init	
pthread_cond_signal	lthread_cond_signal	See note 1
pthread_cond_timedwait		
pthread_cond_wait	lthread_cond_wait	See note 5
pthread_create	lthread_create	See notes 2, 3
pthread_detach	lthread_detach	See note 4
pthread_equal		
pthread_exit	lthread_exit	
pthread_getspecific	lthread_getspecific	
pthread_getcpuclockid		
pthread_join	lthread_join	
pthread_key_create	lthread_key_create	
pthread_key_delete	lthread_key_delete	
pthread_mutex_destroy	lthread_mutex_destroy	
pthread_mutex_init	lthread_mutex_init	
pthread_mutex_lock	lthread_mutex_lock	See note 6
pthread_mutex_trylock	lthread_mutex_trylock	See note 6
pthread_mutex_timedlock		
pthread_mutex_unlock	lthread_mutex_unlock	
pthread_once		
pthread_rwlock_destroy		
pthread_rwlock_init		
pthread_rwlock_rdlock		
pthread_rwlock_timedrdlock		
pthread_rwlock_timedwrlock		
pthread_rwlock_tryrdlock		
pthread_rwlock_trywrlock		
pthread_rwlock_unlock		
pthread_rwlock_wrlock		
pthread_self	lthread_current	
pthread_setspecific	lthread_setspecific	
pthread_spin_init		See note 10
pthread_spin_destroy		See note 10
pthread_spin_lock		See note 10

Continued on next page

Table 7.21 – continued from previous page

Pthread function	L-thread function	Notes
pthread_spin_trylock		See note 10
pthread_spin_unlock		See note 10
pthread_cancel	lthread_cancel	
pthread_setcancelstate		
pthread_setcanceltype		
pthread_testcancel		
pthread_getschedparam		
pthread_setschedparam		
pthread_yield	lthread_yield	See note 7
pthread_setaffinity_np	lthread_set_affinity	See notes 2, 3, 8
	lthread_sleep	See note 9
	lthread_sleep_clks	See note 9

Note 1:

Neither lthread signal nor broadcast may be called concurrently by L-threads running on different schedulers, although multiple L-threads running in the same scheduler may freely perform signal or broadcast operations. L-threads running on the same or different schedulers may always safely wait on a condition variable.

Note 2:

Pthread attributes may be used to affinitize a pthread with a cpu-set. The L-thread subsystem does not support a cpu-set. An L-thread may be affinitized only with a single CPU at any time.

Note 3:

If an L-thread is intended to run on a different NUMA node than the node that creates the thread then, when calling `lthread_create()` it is advantageous to specify the destination core as a parameter of `lthread_create()`. See *Memory allocation and NUMA awareness* for details.

Note 4:

An L-thread can only detach itself, and cannot detach other L-threads.

Note 5:

A wait operation on a pthread condition variable is always associated with and protected by a mutex which must be owned by the thread at the time it invokes `pthread_wait()`. By contrast L-thread condition variables are thread safe (for waiters) and do not use an associated mutex. Multiple L-threads (including L-threads running on other schedulers) can safely wait on a L-thread condition variable. As a consequence the performance of an L-thread condition variables is typically an order of magnitude faster than its pthread counterpart.

Note 6:

Recursive locking is not supported with L-threads, attempts to take a lock recursively will be detected and rejected.

Note 7:

`lthread_yield()` will save the current context, insert the current thread to the back of the ready queue, and resume the next ready thread. Yielding increases ready queue backlog, see *Ready queue backlog* for more details about the implications of this.

N.B. The context switch time as measured from immediately before the call to `lthread_yield()` to the point at which the next ready thread is resumed, can be an order of magnitude faster that the same measurement for pthread_yield.

Note 8:

`lthread_set_affinity()` is similar to a yield apart from the fact that the yielding thread is inserted into a peer ready queue of another scheduler. The peer ready queue is actually a separate thread safe queue, which means that threads appearing in the peer ready queue can jump any backlog in the local ready queue on the destination scheduler.

The context switch time as measured from the time just before the call to `lthread_set_affinity()` to just after the same thread is resumed on the new scheduler can be orders of magnitude faster than the same measurement for `pthread_setaffinity_np()`.

Note 9:

Although there is no `pthread_sleep()` function, `lthread_sleep()` and `lthread_sleep_clks()` can be used wherever `sleep()`, `usleep()` or `nanosleep()` might ordinarily be used. The L-thread sleep functions suspend the current thread, start an `rte_timer` and resume the thread when the timer matures. The `rte_timer_manage()` entry point is called on every pass of the scheduler loop. This means that the worst case jitter on timer expiry is determined by the longest period between context switches of any running L-threads.

In a synthetic test with many threads sleeping and resuming then the measured jitter is typically orders of magnitude lower than the same measurement made for `nanosleep()`.

Note 10:

Spin locks are not provided because they are problematical in a cooperative environment, see *Locks and spinlocks* for a more detailed discussion on how to avoid spin locks.

Thread local storage

Of the three L-thread local storage options the simplest and most efficient is storing a single application data pointer in the L-thread struct.

The `PER_LTHREAD` macros involve a run time computation to obtain the address of the variable being saved/retrieved and also require that the accesses are de-referenced via a pointer. This means that code that has used `RTE_PER_LCORE` macros being ported to L-threads might need some slight adjustment (see *Thread local storage* for hints about porting code that makes use of thread local storage).

The get/set specific APIs are consistent with their pthread counterparts both in use and in performance.

Memory allocation and NUMA awareness

All memory allocation is from DPDK huge pages, and is NUMA aware. Each scheduler maintains its own caches of objects: lthreads, their stacks, TLS, mutexes and condition variables. These caches are implemented as unbounded lock free MPSC queues. When objects are created they are always allocated from the caches on the local core (current EAL thread).

If an L-thread has been affinitized to a different scheduler, then it can always safely free resources to the caches from which they originated (because the caches are MPSC queues).

If the L-thread has been affinitized to a different NUMA node then the memory resources associated with it may incur longer access latency.

The commonly used pattern of setting affinity on entry to a thread after it has started, means that memory allocation for both the stack and TLS will have been made from caches on the NUMA node on which the threads creator is running. This has the side effect that access latency will be sub-optimal after affinitizing.

This side effect can be mitigated to some extent (although not completely) by specifying the destination CPU as a parameter of `lthread_create()` this causes the L-thread's stack and TLS to be allocated when it is first scheduled on the destination scheduler, if the destination is a on another NUMA node it results in a more optimal memory allocation.

Note that the lthread struct itself remains allocated from memory on the creating node, this is unavoidable because an L-thread is known everywhere by the address of this struct.

Object cache sizing

The per lcore object caches pre-allocate objects in bulk whenever a request to allocate an object finds a cache empty. By default 100 objects are pre-allocated, this is defined by `LTHREAD_PREALLOC` in the public API header file lthread_api.h. This means that the caches constantly grow to meet system demand.

In the present implementation there is no mechanism to reduce the cache sizes if system demand reduces. Thus the caches will remain at their maximum extent indefinitely.

A consequence of the bulk pre-allocation of objects is that every 100 (default value) additional new object create operations results in a call to `rte_malloc()`. For creation of objects such as L-threads, which trigger the allocation of even more objects (i.e. their stacks and TLS) then this can cause outliers in scheduling performance.

If this is a problem the simplest mitigation strategy is to dimension the system, by setting the bulk object pre-allocation size to some large number that you do not expect to be exceeded. This means the caches will be populated once only, the very first time a thread is created.

Ready queue backlog

One of the more subtle performance considerations is managing the ready queue backlog. The fewer threads that are waiting in the ready queue then the faster any particular thread will get serviced.

In a naive L-thread application with N L-threads simply looping and yielding, this backlog will always be equal to the number of L-threads, thus the cost of a yield to a particular L-thread will be N times the context switch time.

This side effect can be mitigated by arranging for threads to be suspended and wait to be resumed, rather than polling for work by constantly yielding. Blocking on a mutex or condition variable or even more obviously having a thread sleep if it has a low frequency workload are all mechanisms by which a thread can be excluded from the ready queue until it really does need to be run. This can have a significant positive impact on performance.

Initialization, shutdown and dependencies

The L-thread subsystem depends on DPDK for huge page allocation and depends on the `rte_timer subsystem`. The DPDK EAL initialization and `rte_timer_subsystem_init()` **MUST** be completed before the L-thread sub system can be used.

Thereafter initialization of the L-thread subsystem is largely transparent to the application. Constructor functions ensure that global variables are properly initialized. Other than global variables each scheduler is initialized independently the first time that an L-thread is created by a particular EAL thread.

If the schedulers are to be run as isolated and independent schedulers, with no intention that L-threads running on different schedulers will migrate between schedulers or synchronize with L-threads running on other schedulers, then initialization consists simply of creating an L-thread, and then running the L-thread scheduler.

If there will be interaction between L-threads running on different schedulers, then it is important that the starting of schedulers on different EAL threads is synchronized.

To achieve this an additional initialization step is necessary, this is simply to set the number of schedulers by calling the API function `lthread_num_schedulers_set(n)`, where `n` is the number of EAL threads that will run L-thread schedulers. Setting the number of schedulers to a number greater than 0 will cause all schedulers to wait until the others have started before beginning to schedule L-threads.

The L-thread scheduler is started by calling the function `lthread_run()` and should be called from the EAL thread and thus become the main loop of the EAL thread.

The function `lthread_run()`, will not return until all threads running on the scheduler have exited, and the scheduler has been explicitly stopped by calling `lthread_scheduler_shutdown(lcore)` or `lthread_scheduler_shutdown_all()`.

All these function do is tell the scheduler that it can exit when there are no longer any running L-threads, neither function forces any running L-thread to terminate. Any desired application shutdown behavior must be designed and built into the application to ensure that L-threads complete in a timely manner.

Important Note: It is assumed when the scheduler exits that the application is terminating for good, the scheduler does not free resources before exiting and running the scheduler a subsequent time will result in undefined behavior.

Porting legacy code to run on L-threads

Legacy code originally written for a pthread environment may be ported to L-threads if the considerations about differences in scheduling policy, and constraints discussed in the previous sections can be accommodated.

This section looks in more detail at some of the issues that may have to be resolved when porting code.

pthread API compatibility

The first step is to establish exactly which pthread APIs the legacy application uses, and to understand the requirements of those APIs. If there are corresponding L-lthread APIs, and where the default pthread functionality is used by the application then, notwithstanding the other issues discussed here, it should be feasible to run the application with L-threads. If the legacy code modifies the default behavior using attributes then if may be necessary to make some adjustments to eliminate those requirements.

Blocking system API calls

It is important to understand what other system services the application may be using, bearing in mind that in a cooperatively scheduled environment a thread cannot block without stalling the scheduler and with it all other cooperative threads. Any kind of blocking system call, for example file or socket IO, is a potential problem, a good tool to analyze the application for this purpose is the `strace` utility.

There are many strategies to resolve these kind of issues, each with it merits. Possible solutions include:

- Adopting a polled mode of the system API concerned (if available).
- Arranging for another core to perform the function and synchronizing with that core via constructs that will not block the L-thread.
- Affinitizing the thread to another scheduler devoted (as a matter of policy) to handling threads wishing to make blocking calls, and then back again when finished.

Locks and spinlocks

Locks and spinlocks are another source of blocking behavior that for the same reasons as system calls will need to be addressed.

If the application design ensures that the contending L-threads will always run on the same scheduler then it its probably safe to remove locks and spin locks completely.

The only exception to the above rule is if for some reason the code performs any kind of context switch whilst holding the lock (e.g. yield, sleep, or block on a different lock, or on a condition variable). This will need to determined before deciding to eliminate a lock.

If a lock cannot be eliminated then an L-thread mutex can be substituted for either kind of lock.

An L-thread blocking on an L-thread mutex will be suspended and will cause another ready L-thread to be resumed, thus not blocking the scheduler. When default behavior is required, it can be used as a direct replacement for a pthread mutex lock.

Spin locks are typically used when lock contention is likely to be rare and where the period during which the lock may be held is relatively short. When the contending L-threads are running on the same scheduler then an L-thread blocking on a spin lock will enter an infinite loop stopping the scheduler completely (see *Infinite loops* below).

If the application design ensures that contending L-threads will always run on different schedulers then it might be reasonable to leave a short spin lock that rarely experiences contention in place.

If after all considerations it appears that a spin lock can neither be eliminated completely, replaced with an L-thread mutex, or left in place as is, then an alternative is to loop on a flag, with a call to `lthread_yield()` inside the loop (n.b. if the contending L-threads might ever run on different schedulers the flag will need to be manipulated atomically).

Spinning and yielding is the least preferred solution since it introduces ready queue backlog (see also *Ready queue backlog*).

Sleeps and delays

Yet another kind of blocking behavior (albeit momentary) are delay functions like `sleep()`, `usleep()`, `nanosleep()` etc. All will have the consequence of stalling the L-thread scheduler and unless the delay is very short (e.g. a very short nanosleep) calls to these functions will need to be eliminated.

The simplest mitigation strategy is to use the L-thread sleep API functions, of which two variants exist, `lthread_sleep()` and `lthread_sleep_clks()`. These functions start an rte_timer against the L-thread, suspend the L-thread and cause another ready L-thread to be resumed. The suspended L-thread is resumed when the rte_timer matures.

Infinite loops

Some applications have threads with loops that contain no inherent rescheduling opportunity, and rely solely on the OS time slicing to share the CPU. In a cooperative environment this will stop everything dead. These kind of loops are not hard to identify, in a debug session you will find the debugger is always stopping in the same loop.

The simplest solution to this kind of problem is to insert an explicit `lthread_yield()` or `lthread_sleep()` into the loop. Another solution might be to include the function performed by the loop into the execution path of some other loop that does in fact yield, if this is possible.

Thread local storage

If the application uses thread local storage, the use case should be studied carefully.

In a legacy pthread application either or both the `__thread` prefix, or the pthread set/get specific APIs may have been used to define storage local to a pthread.

In some applications it may be a reasonable assumption that the data could or in fact most likely should be placed in L-thread local storage.

If the application (like many DPDK applications) has assumed a certain relationship between a pthread and the CPU to which it is affinitized, there is a risk that thread local storage may have been used to save some data items that are correctly logically associated with the CPU, and others items which relate to application context for the thread. Only a good understanding of the application will reveal such cases.

If the application requires an that an L-thread is to be able to move between schedulers then care should be taken to separate these kinds of data, into per lcore, and per L-thread storage. In this way a migrating thread will bring with it the local data it needs, and pick up the new logical core specific values from pthread local storage at its new home.

Pthread shim

A convenient way to get something working with legacy code can be to use a shim that adapts pthread API calls to the corresponding L-thread ones. This approach will not mitigate any of the porting considerations mentioned in the previous sections, but it will reduce the amount of code churn that would otherwise been involved. It is a reasonable approach to evaluate L-threads, before investing effort in porting to the native L-thread APIs.

Overview

The L-thread subsystem includes an example pthread shim. This is a partial implementation but does contain the API stubs needed to get basic applications running. There is a simple "hello world" application that demonstrates the use of the pthread shim.

A subtlety of working with a shim is that the application will still need to make use of the genuine pthread library functions, at the very least in order to create the EAL threads in which the L-thread schedulers will run. This is the case with DPDK initialization, and exit.

To deal with the initialization and shutdown scenarios, the shim is capable of switching on or off its adaptor functionality, an application can control this behavior by the calling the function `pt_override_set()`. The default state is disabled.

The pthread shim uses the dynamic linker loader and saves the loaded addresses of the genuine pthread API functions in an internal table, when the shim functionality is enabled it performs the adaptor function, when disabled it invokes the genuine pthread function.

The function `pthread_exit()` has additional special handling. The standard system header file pthread.h declares `pthread_exit()` with `__attribute__((noreturn))` this is an optimization that is possible because the pthread is terminating and this enables the compiler to omit the normal handling of stack and protection of registers since the function is not expected to return, and in fact the thread is being destroyed. These optimizations are applied in both the callee and the caller of the `pthread_exit()` function.

In our cooperative scheduling environment this behavior is inadmissible. The pthread is the L-thread scheduler thread, and, although an L-thread is terminating, there must be a return to the scheduler in order that the system can continue to run. Further, returning from a function with attribute `noreturn` is invalid and may result in undefined behavior.

The solution is to redefine the `pthread_exit` function with a macro, causing it to be mapped to a stub function in the shim that does not have the `noreturn` attribute. This macro is defined in the file `pthread_shim.h`. The stub function is otherwise no different than any of the other stub functions in the shim, and will switch between the real `pthread_exit()` function or the `lthread_exit()` function as required. The only difference is that the mapping to the stub by macro substitution.

A consequence of this is that the file `pthread_shim.h` must be included in legacy code wishing to make use of the shim. It also means that dynamic linkage of a pre-compiled binary that did not include pthread_shim.h is not be supported.

Given the requirements for porting legacy code outlined in *Porting legacy code to run on L-threads* most applications will require at least some minimal adjustment and recompilation to run on L-threads so pre-compiled binaries are unlikely to be met in practice.

In summary the shim approach adds some overhead but can be a useful tool to help establish the feasibility of a code reuse project. It is also a fairly straightforward task to extend the shim if necessary.

Note: Bearing in mind the preceding discussions about the impact of making blocking calls then switching the shim in and out on the fly to invoke any pthread API this might block is something that should typically be avoided.

Building and running the pthread shim

The shim example application is located in the sample application in the performance-thread folder

To build and run the pthread shim example

1. Go to the example applications folder

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd ${RTE_SDK}/examples/performance-thread/pthread_shim
   ```

2. Set the target (a default target is used if not specified). For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

   See the DPDK Getting Started Guide for possible RTE_TARGET values.

3. Build the application:

   ```
   make
   ```

4. To run the pthread_shim example

   ```
   lthread-pthread-shim -c core_mask -n number_of_channels
   ```

L-thread Diagnostics

When debugging you must take account of the fact that the L-threads are run in a single pthread. The current scheduler is defined by `RTE_PER_LCORE(this_sched)`, and the current lthread is stored at `RTE_PER_LCORE(this_sched)->current_lthread`. Thus on a breakpoint in a GDB session the current lthread can be obtained by displaying the pthread local variable `per_lcore_this_sched->current_lthread`.

Another useful diagnostic feature is the possibility to trace significant events in the life of an L-thread, this feature is enabled by changing the value of LTHREAD_DIAG from 0 to 1 in the file `lthread_diag_api.h`.

Tracing of events can be individually masked, and the mask may be programmed at run time. An unmasked event results in a callback that provides information about the event. The default callback simply prints trace information. The default mask is 0 (all events off) the mask can be modified by calling the function `lthread_diagniostic_set_mask()`.

It is possible register a user callback function to implement more sophisticated diagnostic functions. Object creation events (lthread, mutex, and condition variable) accept, and store in the created object, a user supplied reference value returned by the callback function.

The lthread reference value is passed back in all subsequent event callbacks, the mutex and APIs are provided to retrieve the reference value from mutexes and condition variables. This enables a user to monitor, count, or filter for specific events, on specific objects, for example to monitor for a specific thread signaling a specific condition variable, or to monitor on all timer events, the possibilities and combinations are endless.

The callback function can be set by calling the function `lthread_diagnostic_enable()` supplying a callback function pointer and an event mask.

Setting `LTHREAD_DIAG` also enables counting of statistics about cache and queue usage, and these statistics can be displayed by calling the function `lthread_diag_stats_display()`. This function also performs a consistency check on the caches and queues. The function should only be called from the master EAL thread after all slave threads have stopped and returned to the C main program, otherwise the consistency check will fail.

Figures

CHAPTER 8

Testpmd Application User Guide

8.1 Introduction

This document is a user guide for the `testpmd` example application that is shipped as part of the Data Plane Development Kit.

The `testpmd` application can be used to test the DPDK in a packet forwarding mode and also to access NIC hardware features such as Flow Director. It also serves as a example of how to build a more fully-featured application using the DPDK SDK.

The guide shows how to build and run the testpmd application and how to configure the application from the command line and the run-time environment.

8.2 Compiling the Application

The `testpmd` application is compiled as part of the main compilation of the DPDK libraries and tools. Refer to the DPDK Getting Started Guides for details. The basic compilation steps are:

1. Set the required environmental variables and go to the source directory:

   ```
   export RTE_SDK=/path/to/rte_sdk
   cd $RTE_SDK
   ```

2. Set the compilation target. For example:

   ```
   export RTE_TARGET=x86_64-native-linuxapp-gcc
   ```

3. Build the application:

   ```
   make install T=$RTE_TARGET
   ```

   The compiled application will be located at:

   ```
   $RTE_SDK/$RTE_TARGET/build/app/testpmd
   ```

8.3 Running the Application

8.3.1 EAL Command-line Options

The following are the EAL command-line options that can be used in conjunction with the testpmd, or any other DPDK application. See the DPDK Getting Started Guides for more information on these options.

- `-c COREMASK`

  Set the hexadecimal bitmask of the cores to run on.

- `-l CORELIST`

  List of cores to run on

  The argument format is `<c1>[-c2][,c3[-c4],...]` where `c1`, `c2`, etc are core indexes between 0 and 128.

- `--lcores COREMAP`

  Map lcore set to physical cpu set

  The argument format is:

  ```
  <lcores[@cpus]>[<,lcores[@cpus]>...]
  ```

  Lcore and CPU lists are grouped by `(` and `)` Within the group. The `-` character is used as a range separator and `,` is used as a single number separator. The grouping `()` can be omitted for single element group. The `@` can be omitted if cpus and lcores have the same value.

- `--master-lcore ID`

  Core ID that is used as master.

- `-n NUM`

  Set the number of memory channels to use.

- `-b, --pci-blacklist domain:bus:devid.func`

  Blacklist a PCI devise to prevent EAL from using it. Multiple -b options are allowed.

- `-d LIB.so`

  Load an external driver. Multiple -d options are allowed.

- `-w, --pci-whitelist domain:bus:devid:func`

  Add a PCI device in white list.

- `-m MB`

  Memory to allocate. See also `--socket-mem`.

- `-r NUM`

  Set the number of memory ranks (auto-detected by default).

- `-v`

  Display the version information on startup.

- `--xen-dom0`

  Support application running on Xen Domain0 without hugetlbfs.

- `--syslog`

  Set the syslog facility.

- `--socket-mem`

  Set the memory to allocate on specific sockets (use comma separated values).

- `--huge-dir`

  Specify the directory where the hugetlbfs is mounted.

- `--proc-type`

  Set the type of the current process.

- `--file-prefix`

  Prefix for hugepage filenames.

- `-vmware-tsc-map`

  Use VMware TSC map instead of native RDTSC.

- `--vdev`

  Add a virtual device using the format:

  ```
  <driver><id>[,key=val, ...]
  ```

  For example:

  ```
  --vdev 'eth_pcap0,rx_pcap=input.pcap,tx_pcap=output.pcap'
  ```

- `--base-virtaddr`

  Specify base virtual address.

- `--create-uio-dev`

  Create `/dev/uioX` (usually done by hotplug).

- `--no-shconf`

  No shared config (mmap-ed files).

- `--no-pci`

  Disable pci.

- `--no-hpet`

  Disable hpet.

- `--no-huge`

  Use malloc instead of hugetlbfs.

8.3.2 Testpmd Command-line Options

The following are the command-line options for the testpmd applications. They must be separated from the EAL options, shown in the previous section, with a `--` separator:

```
sudo ./testpmd -c 0xF -n 4 -- -i --portmask=0x1 --nb-cores=2
```

The commandline options are:

- `-i, --interactive`

  Run testpmd in interactive mode. In this mode, the testpmd starts with a prompt that can be used to start and stop forwarding, configure the application and display stats on the current packet processing session. See *Testpmd Runtime Functions* for more details.

  In non-interactive mode, the application starts with the configuration specified on the command-line and immediately enters forwarding mode.

- `-h, --help`

  Display a help message and quit.

- `-a, --auto-start`

  Start forwarding on initialization.

- `--nb-cores=N`

  Set the number of forwarding cores, where 1 <= N <= "number of cores" or `CONFIG_RTE_MAX_LCORE` from the configuration file. The default value is 1.

- `--nb-ports=N`

  Set the number of forwarding ports, where 1 <= N <= "number of ports" on the board or `CONFIG_RTE_MAX_ETHPORTS` from the configuration file. The default value is the number of ports on the board.

- `--coremask=0xXX`

  Set the hexadecimal bitmask of the cores running the packet forwarding test. The master lcore is reserved for command line parsing only and cannot be masked on for packet forwarding.

- `--portmask=0xXX`

  Set the hexadecimal bitmask of the ports used by the packet forwarding test.

- `--numa`

  Enable NUMA-aware allocation of RX/TX rings and of RX memory buffers (mbufs).

- `--port-numa-config=(port,socket)[,(port,socket)]`

  Specify the socket on which the memory pool to be used by the port will be allocated.

- `--ring-numa-config=(port,flag,socket)[,(port,flag,socket)]`

  Specify the socket on which the TX/RX rings for the port will be allocated. Where flag is 1 for RX, 2 for TX, and 3 for RX and TX.

- `--socket-num=N`

  Set the socket from which all memory is allocated in NUMA mode, where 0 <= N < number of sockets on the board.

- `--mbuf-size=N`

  Set the data size of the mbufs used to N bytes, where N < 65536. The default value is 2048.

- `--total-num-mbufs=N`

  Set the number of mbufs to be allocated in the mbuf pools, where N > 1024.

- `--max-pkt-len=N`

  Set the maximum packet size to N bytes, where N >= 64. The default value is 1518.

- `--eth-peers-configfile=name`

  Use a configuration file containing the Ethernet addresses of the peer ports. The configuration file should contain the Ethernet addresses on separate lines:

  ```
  XX:XX:XX:XX:XX:01
  XX:XX:XX:XX:XX:02
  ...
  ```

- `--eth-peer=N,XX:XX:XX:XX:XX:XX`

  Set the MAC address `XX:XX:XX:XX:XX:XX` of the peer port N, where 0 <= N < `CONFIG_RTE_MAX_ETHPORTS` from the configuration file.

- `--pkt-filter-mode=mode`

  Set Flow Director mode where mode is either `none` (the default), `signature` or `perfect`. See *flow_director_filter* for more details.

- `--pkt-filter-report-hash=mode`

  Set Flow Director hash match reporting mode where mode is `none`, `match` (the default) or `always`.

- `--pkt-filter-size=N`

  Set Flow Director allocated memory size, where N is 64K, 128K or 256K. Sizes are in kilobytes. The default is 64.

- `--pkt-filter-flexbytes-offset=N`

  Set the flexbytes offset. The offset is defined in words (not bytes) counted from the first byte of the destination Ethernet MAC address, where N is 0 <= N <= 32. The default value is 0x6.

- `--pkt-filter-drop-queue=N`

  Set the drop-queue. In perfect filter mode, when a rule is added with queue = -1, the packet will be enqueued into the RX drop-queue. If the drop-queue does not exist, the packet is dropped. The default value is N=127.

- `--crc-strip`

  Enable hardware CRC stripping.

- `--enable-rx-cksum`

  Enable hardware RX checksum offload.

- `--disable-hw-vlan`

  Disable hardware VLAN.

- `--disable-hw-vlan-filter`

  Disable hardware VLAN filter.

- `--disable-hw-vlan-strip`

  Disable hardware VLAN strip.

- `--disable-hw-vlan-extend`

  Disable hardware VLAN extend.

- `--enable-drop-en`

  Enable per-queue packet drop for packets with no descriptors.

- `--disable-rss`

  Disable RSS (Receive Side Scaling).

- `--port-topology=mode`

  Set port topology, where mode is `paired` (the default) or `chained`.

  In `paired` mode, the forwarding is between pairs of ports, for example: (0,1), (2,3), (4,5).

  In `chained` mode, the forwarding is to the next available port in the port mask, for example: (0,1), (1,2), (2,0).

  The ordering of the ports can be changed using the portlist testpmd runtime function.

- `--forward-mode=mode`

  Set the forwarding mode where `mode` is one of the following:

  ```
  io (the default)
  mac
  mac_retry
  mac_swap
  flowgen
  rxonly
  txonly
  csum
  icmpecho
  ieee1588
  ```

- `--rss-ip`

  Set RSS functions for IPv4/IPv6 only.

- `--rss-udp`

  Set RSS functions for IPv4/IPv6 and UDP.

- `--rxq=N`

  Set the number of RX queues per port to N, where 1 <= N <= 65535. The default value is 1.

- `--rxd=N`

  Set the number of descriptors in the RX rings to N, where N > 0. The default value is 128.

- `--txq=N`

  Set the number of TX queues per port to N, where 1 <= N <= 65535. The default value is 1.

- `--txd=N`

  Set the number of descriptors in the TX rings to N, where N > 0. The default value is 512.

- `--burst=N`

  Set the number of packets per burst to N, where 1 <= N <= 512. The default value is 16.

- `--mbcache=N`

  Set the cache of mbuf memory pools to N, where 0 <= N <= 512. The default value is 16.

- `--rxpt=N`

  Set the prefetch threshold register of RX rings to N, where N >= 0. The default value is 8.

- `--rxht=N`

  Set the host threshold register of RX rings to N, where N >= 0. The default value is 8.

- `--rxfreet=N`

  Set the free threshold of RX descriptors to N, where 0 <= N < value of –rxd. The default value is 0.

- `--rxwt=N`

  Set the write-back threshold register of RX rings to N, where N >= 0. The default value is 4.

- `--txpt=N`

  Set the prefetch threshold register of TX rings to N, where N >= 0. The default value is 36.

- `--txht=N`

  Set the host threshold register of TX rings to N, where N >= 0. The default value is 0.

- `--txwt=N`

  Set the write-back threshold register of TX rings to N, where N >= 0. The default value is 0.

- `--txfreet=N`

  Set the transmit free threshold of TX rings to N, where 0 <= N <= value of `--txd`. The default value is 0.

- `--txrst=N`

  Set the transmit RS bit threshold of TX rings to N, where 0 <= N <= value of `--txd`. The default value is 0.

- `--txqflags=0xXXXXXXXX`

  Set the hexadecimal bitmask of TX queue flags, where 0 <= N <= 0x7FFFFFFF. The default value is 0.

Note: When using hardware offload functions such as vlan or checksum add `txqflags=0` to force the full-featured TX code path. In some PMDs this may already be the default.

- `--rx-queue-stats-mapping=(port,queue,mapping)[,(port,queue,mapping)]`

  Set the RX queues statistics counters mapping 0 <= mapping <= 15.
- `--tx-queue-stats-mapping=(port,queue,mapping)[,(port,queue,mapping)]`

  Set the TX queues statistics counters mapping 0 <= mapping <= 15.
- `--no-flush-rx`

  Don't flush the RX streams before starting forwarding. Used mainly with the PCAP PMD.
- `--txpkts=X[,Y]`

  Set TX segment sizes.
- `--disable-link-check`

  Disable check on link status when starting/stopping ports.

8.4 Testpmd Runtime Functions

Where the testpmd application is started in interactive mode, (`-i|--interactive`), it displays a prompt that can be used to start and stop forwarding, configure the application, display statistics, set the Flow Director and other tasks:

```
testpmd>
```

The testpmd prompt has some, limited, readline support. Common bash command-line functions such as `Ctrl+a` and `Ctrl+e` to go to the start and end of the prompt line are supported as well as access to the command history via the up-arrow.

There is also support for tab completion. If you type a partial command and hit `<TAB>` you get a list of the available completions:

```
testpmd> show port <TAB>

    info [Mul-choice STRING]: show|clear port info|stats|fdir|stat_qmap|dcb_tc X
    info [Mul-choice STRING]: show|clear port info|stats|fdir|stat_qmap|dcb_tc all
    stats [Mul-choice STRING]: show|clear port info|stats|fdir|stat_qmap|dcb_tc X
    stats [Mul-choice STRING]: show|clear port info|stats|fdir|stat_qmap|dcb_tc all
    ...
```

Note: Some examples in this document are too long to fit on one line are are shown wrapped at "\" for display purposes:

```
testpmd> set flow_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
         (pause_time) (send_xon) (port_id)
```

In the real `testpmd>` prompt these commands should be on a single line.

8.4.1 Help Functions

The testpmd has on-line help for the functions that are available at runtime. These are divided into sections and can be accessed using help, help section or help all:

```
testpmd> help

    help control    : Start and stop forwarding.
    help display    : Displaying port, stats and config information.
    help config     : Configuration information.
    help ports      : Configuring ports.
    help registers  : Reading and setting port registers.
    help filters    : Filters configuration help.
    help all        : All of the above sections.
```

8.4.2 Control Functions

start

Start packet forwarding with current configuration:

```
testpmd> start
```

start tx_first

Start packet forwarding with current configuration after sending one burst of packets:

```
testpmd> start tx_first
```

stop

Stop packet forwarding, and display accumulated statistics:

```
testpmd> stop
```

quit

Quit to prompt:

```
testpmd> quit
```

8.4.3 Display Functions

The functions in the following sections are used to display information about the testpmd configuration or the NIC status.

show port

Display information for a given port or all ports:

```
testpmd> show port (info|stats|fdir|stat_qmap|dcb_tc) (port_id|all)
```

The available information categories are:

- `info`: General port information such as MAC address.
- `stats`: RX/TX statistics.
- `fdir`: Flow Director information and statistics.

- `stat_qmap`: Queue statistics mapping.
- `dcb_tc`: DCB information such as TC mapping.

For example:

```
testpmd> show port info 0

********************* Infos for port 0 *********************

MAC address: XX:XX:XX:XX:XX:XX
Connect to socket: 0
memory allocation on the socket: 0
Link status: up
Link speed: 40000 Mbps
Link duplex: full-duplex
Promiscuous mode: enabled
Allmulticast mode: disabled
Maximum number of MAC addresses: 64
Maximum number of MAC addresses of hash filtering: 0
VLAN offload:
    strip on
    filter on
    qinq(extend) off
Redirection table size: 512
Supported flow types:
  ipv4-frag
  ipv4-tcp
  ipv4-udp
  ipv4-sctp
  ipv4-other
  ipv6-frag
  ipv6-tcp
  ipv6-udp
  ipv6-sctp
  ipv6-other
  l2_payload
```

show port rss reta

Display the rss redirection table entry indicated by masks on port X:

```
testpmd> show port (port_id) rss reta (size) (mask0, mask1...)
```

size is used to indicate the hardware supported reta size

show port rss-hash

Display the RSS hash functions and RSS hash key of a port:

```
testpmd> show port (port_id) rss-hash ipv4|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp|ipv4-other|ipv
```

clear port

Clear the port statistics for a given port or for all ports:

```
testpmd> clear port (info|stats|fdir|stat_qmap) (port_id|all)
```

For example:

```
testpmd> clear port stats all
```

show (rxq|txq)

Display information for a given port's RX/TX queue:

```
testpmd> show (rxq|txq) info (port_id) (queue_id)
```

show config

Displays the configuration of the application. The configuration comes from the command-line, the runtime or the application defaults:

```
testpmd> show config (rxtx|cores|fwd|txpkts)
```

The available information categories are:

- `rxtx`: RX/TX configuration items.
- `cores`: List of forwarding cores.
- `fwd`: Packet forwarding configuration.
- `txpkts`: Packets to TX configuration.

For example:

```
testpmd> show config rxtx

io packet forwarding - CRC stripping disabled - packets/burst=16
nb forwarding cores=2 - nb forwarding ports=1
RX queues=1 - RX desc=128 - RX free threshold=0
RX threshold registers: pthresh=8 hthresh=8 wthresh=4
TX queues=1 - TX desc=512 - TX free threshold=0
TX threshold registers: pthresh=36 hthresh=0 wthresh=0
TX RS bit threshold=0 - TXQ flags=0x0
```

set fwd

Set the packet forwarding mode:

```
testpmd> set fwd (io|mac|mac_retry|macswap|flowgen| \
                  rxonly|txonly|csum|icmpecho)
```

The available information categories are:

- `io`: Forwards packets "as-is" in I/O mode. This is the fastest possible forwarding operation as it does not access packets data. This is the default mode.
- `mac`: Changes the source and the destination Ethernet addresses of packets before forwarding them.
- `mac_retry`: Same as "mac" forwarding mode, but includes retries if the destination queue is full.
- `macswap`: MAC swap forwarding mode. Swaps the source and the destination Ethernet addresses of packets before forwarding them.
- `flowgen`: Multi-flow generation mode. Originates a number of flows (with varying destination IP addresses), and terminate receive traffic.
- `rxonly`: Receives packets but doesn't transmit them.

- `txonly`: Generates and transmits packets without receiving any.
- `csum`: Changes the checksum field with hardware or software methods depending on the offload flags on the packet.
- `icmpecho`: Receives a burst of packets, lookup for IMCP echo requests and, if any, send back ICMP echo replies.
- `ieee1588`: Demonstrate L2 IEEE1588 V2 PTP timestamping for RX and TX. Requires `CONFIG_RTE_LIBRTE_IEEE1588=y`.

Note: TX timestamping is only available in the “Full Featured” TX path. To force `testpmd` into this mode set `--txqflags=0`.

Example:

```
testpmd> set fwd rxonly

Set rxonly packet forwarding mode
```

read rxd

Display an RX descriptor for a port RX queue:

```
testpmd> read rxd (port_id) (queue_id) (rxd_id)
```

For example:

```
testpmd> read rxd 0 0 4
     0x0000000B - 0x001D0180 / 0x0000000B - 0x001D0180
```

read txd

Display a TX descriptor for a port TX queue:

```
testpmd> read txd (port_id) (queue_id) (txd_id)
```

For example:

```
testpmd> read txd 0 0 4
     0x00000001 - 0x24C3C440 / 0x000F0000 - 0x2330003C
```

8.4.4 Configuration Functions

The testpmd application can be configured from the runtime as well as from the command-line.

This section details the available configuration functions that are available.

Note: Configuration changes only become active when forwarding is started/restarted.

set default

Reset forwarding to the default configuration:

```
testpmd> set default
```

set verbose

Set the debug verbosity level:

```
testpmd> set verbose (level)
```

Currently the only available levels are 0 (silent except for error) and 1 (fully verbose).

set nbport

Set the number of ports used by the application:

set nbport (num)

This is equivalent to the `--nb-ports` command-line option.

set nbcore

Set the number of cores used by the application:

```
testpmd> set nbcore (num)
```

This is equivalent to the `--nb-cores` command-line option.

Note: The number of cores used must not be greater than number of ports used multiplied by the number of queues per port.

set coremask

Set the forwarding cores hexadecimal mask:

```
testpmd> set coremask (mask)
```

This is equivalent to the `--coremask` command-line option.

Note: The master lcore is reserved for command line parsing only and cannot be masked on for packet forwarding.

set portmask

Set the forwarding ports hexadecimal mask:

```
testpmd> set portmask (mask)
```

This is equivalent to the `--portmask` command-line option.

set burst

Set number of packets per burst:

```
testpmd> set burst (num)
```

This is equivalent to the `--burst command-line` option.

In `mac_retry` forwarding mode, the transmit delay time and number of retries can also be set:

```
testpmd> set burst tx delay (micrseconds) retry (num)
```

set txpkts

Set the length of each segment of the TX-ONLY packets:

```
testpmd> set txpkts (x[,y]*)
```

Where x[,y]* represents a CSV list of values, without white space.

set txsplit

Set the split policy for the TX packets, applicable for TX-ONLY and CSUM forwarding modes:

```
testpmd> set txsplit (off|on|rand)
```

Where:

- `off` disable packet copy & split for CSUM mode.
- `on` split outgoing packet into multiple segments. Size of each segment and number of segments per packet is determined by `set txpkts` command (see above).
- `rand` same as 'on', but number of segments per each packet is a random value between 1 and total number of segments.

set corelist

Set the list of forwarding cores:

```
testpmd> set corelist (x[,y]*)
```

For example, to change the forwarding cores:

```
testpmd> set corelist 3,1
testpmd> show config fwd

io packet forwarding - ports=2 - cores=2 - streams=2 - NUMA support disabled
Logical Core 3 (socket 0) forwards packets on 1 streams:
RX P=0/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer=02:00:00:00:00:01
Logical Core 1 (socket 0) forwards packets on 1 streams:
RX P=1/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00
```

Note: The cores are used in the same order as specified on the command line.

set portlist

Set the list of forwarding ports:

```
testpmd> set portlist (x[,y]*)
```

For example, to change the port forwarding:

```
testpmd> set portlist 0,2,1,3
testpmd> show config fwd

io packet forwarding - ports=4 - cores=1 - streams=4
Logical Core 3 (socket 0) forwards packets on 4 streams:
RX P=0/Q=0 (socket 0) -> TX P=2/Q=0 (socket 0) peer=02:00:00:00:00:01
RX P=2/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00
RX P=1/Q=0 (socket 0) -> TX P=3/Q=0 (socket 0) peer=02:00:00:00:00:03
RX P=3/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer=02:00:00:00:00:02
```

vlan set strip

Set the VLAN strip on a port:

```
testpmd> vlan set strip (on|off) (port_id)
```

vlan set stripq

Set the VLAN strip for a queue on a port:

```
testpmd> vlan set stripq (on|off) (port_id,queue_id)
```

vlan set filter

Set the VLAN filter on a port:

```
testpmd> vlan set filter (on|off) (port_id)
```

vlan set qinq

Set the VLAN QinQ (extended queue in queue) on for a port:

```
testpmd> vlan set qinq (on|off) (port_id)
```

vlan set tpid

Set the outer VLAN TPID for packet filtering on a port:

```
testpmd> vlan set tpid (value) (port_id)
```

Note: TPID value must be a 16-bit number (value <= 65536).

rx_vlan add

Add a VLAN ID, or all identifiers, to the set of VLAN identifiers filtered by port ID:

```
testpmd> rx_vlan add (vlan_id|all) (port_id)
```

Note: VLAN filter must be set on that port. VLAN ID < 4096. Depending on the NIC used, number of vlan_ids may be limited to the maximum entries in VFTA table. This is important if enabling all vlan_ids.

rx_vlan rm

Remove a VLAN ID, or all identifiers, from the set of VLAN identifiers filtered by port ID:

```
testpmd> rx_vlan rm (vlan_id|all) (port_id)
```

rx_vlan add (for VF)

Add a VLAN ID, to the set of VLAN identifiers filtered for VF(s) for port ID:

```
testpmd> rx_vlan add (vlan_id) port (port_id) vf (vf_mask)
```

rx_vlan rm (for VF)

Remove a VLAN ID, from the set of VLAN identifiers filtered for VF(s) for port ID:

```
testpmd> rx_vlan rm (vlan_id) port (port_id) vf (vf_mask)
```

rx_vlan set tpid

Set the outer VLAN TPID for packet filtering on a port:

```
testpmd> rx_vlan set tpid (value) (port_id)
```

tunnel_filter add

Add a tunnel filter on a port:

```
testpmd> tunnel_filter add (port_id) (outer_mac) (inner_mac) (ip_addr) \
         (inner_vlan) (tunnel_type) (filter_type) (tenant_id) (queue_id)
```

tunnel_filter remove

Remove a tunnel filter on a port:

```
testpmd> tunnel_filter rm (port_id) (outer_mac) (inner_mac) (ip_addr) \
         (inner_vlan) (tunnel_type) (filter_type) (tenant_id) (queue_id)
```

rx_vxlan_port add

Add an UDP port for VXLAN packet filter on a port:

```
testpmd> rx_vxlan_port add (udp_port) (port_id)
```

rx_vxlan_port remove

Remove an UDP port for VXLAN packet filter on a port:

```
testpmd> rx_vxlan_port rm (udp_port) (port_id)
```

tx_vlan set

Set hardware insertion of VLAN IDs in packets sent on a port:

```
testpmd> tx_vlan set (port_id) vlan_id[, vlan_id_outer]
```

For example, set a single VLAN ID (5) insertion on port 0:

```
tx_vlan set 0 5
```

Or, set double VLAN ID (inner: 2, outer: 3) insertion on port 1:

```
tx_vlan set 1 2 3
```

tx_vlan set pvid

Set port based hardware insertion of VLAN ID in packets sent on a port:

```
testpmd> tx_vlan set pvid (port_id) (vlan_id) (on|off)
```

tx_vlan reset

Disable hardware insertion of a VLAN header in packets sent on a port:

```
testpmd> tx_vlan reset (port_id)
```

csum set

Select hardware or software calculation of the checksum when transmitting a packet using the `csum` forwarding engine:

```
testpmd> csum set (ip|udp|tcp|sctp|outer-ip) (hw|sw) (port_id)
```

Where:

- `ip|udp|tcp|sctp` always relate to the inner layer.
- `outer-ip` relates to the outer IP layer (only for IPv4) in the case where the packet is recognized as a tunnel packet by the forwarding engine (vxlan, gre and ipip are supported). See also the `csum parse-tunnel` command.

Note: Check the NIC Datasheet for hardware limits.

csum parse-tunnel

Define how tunneled packets should be handled by the csum forward engine:

```
testpmd> csum parse-tunnel (on|off) (tx_port_id)
```

If enabled, the csum forward engine will try to recognize supported tunnel headers (vxlan, gre, ipip).

If disabled, treat tunnel packets as non-tunneled packets (a inner header is handled as a packet payload).

Note: The port argument is the TX port like in the `csum set` command.

Example:

Consider a packet in packet like the following:

```
eth_out/ipv4_out/udp_out/vxlan/eth_in/ipv4_in/tcp_in
```

- If parse-tunnel is enabled, the `ip|udp|tcp|sctp` parameters of `csum set` command relate to the inner headers (here `ipv4_in` and `tcp_in`), and the `outer-ip parameter` relates to the outer headers (here `ipv4_out`).
- **If parse-tunnel is disabled, the `ip|udp|tcp|sctp` parameters of `csum set`** command relate to the outer headers, here `ipv4_out` and `udp_out`.

csum show

Display tx checksum offload configuration:

```
testpmd> csum show (port_id)
```

tso set

Enable TCP Segmentation Offload (TSO) in the `csum` forwarding engine:

```
testpmd> tso set (segsize) (port_id)
```

Note: Check the NIC datasheet for hardware limits.

tso show

Display the status of TCP Segmentation Offload:

```
testpmd> tso show (port_id)
```

mac_addr add

Add an alternative MAC address to a port:

```
testpmd> mac_addr add (port_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr remove

Remove a MAC address from a port:

```
testpmd> mac_addr remove (port_id) (XX:XX:XX:XX:XX:XX)
```

mac_addr add(for VF)

Add an alternative MAC address for a VF to a port:

```
testpmd> mac_add add port (port_id) vf (vf_id) (XX:XX:XX:XX:XX:XX)
```

set port-uta

Set the unicast hash filter(s) on/off for a port:

```
testpmd> set port (port_id) uta (XX:XX:XX:XX:XX:XX|all) (on|off)
```

set promisc

Set the promiscuous mode on for a port or for all ports. In promiscuous mode packets are not dropped if they aren't for the specified MAC address:

```
testpmd> set promisc (port_id|all) (on|off)
```

set allmulti

Set the allmulti mode for a port or for all ports:

```
testpmd> set allmulti (port_id|all) (on|off)
```

Same as the ifconfig (8) option. Controls how multicast packets are handled.

set flow_ctrl rx

Set the link flow control parameter on a port:

```
testpmd> set flow_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
          (pause_time) (send_xon) mac_ctrl_frame_fwd (on|off) \
          autoneg (on|off) (port_id)
```

Where:

- `high_water` (integer): High threshold value to trigger XOFF.
- `low_water` (integer): Low threshold value to trigger XON.
- `pause_time` (integer): Pause quota in the Pause frame.
- `send_xon` (0/1): Send XON frame.
- `mac_ctrl_frame_fwd`: Enable receiving MAC control frames.
- `autoneg`: Change the auto-negotiation para mete.

set pfc_ctrl rx

Set the priority flow control parameter on a port:

```
testpmd> set pfc_ctrl rx (on|off) tx (on|off) (high_water) (low_water) \
          (pause_time) (priority) (port_id)
```

Where:

- `high_water` (integer): High threshold value.
- `low_water` (integer): Low threshold value.
- `pause_time` (integer): Pause quota in the Pause frame.
- `priority` (0-7): VLAN User Priority.

set stat_qmap

Set statistics mapping (qmapping 0..15) for RX/TX queue on port:

```
testpmd> set stat_qmap (tx|rx) (port_id) (queue_id) (qmapping)
```

For example, to set rx queue 2 on port 0 to mapping 5:

```
testpmd>set stat_qmap rx 0 2 5
```

set port - rx/tx (for VF)

Set VF receive/transmit from a port:

```
testpmd> set port (port_id) vf (vf_id) (rx|tx) (on|off)
```

set port - mac address filter (for VF)

Add/Remove unicast or multicast MAC addr filter for a VF:

```
testpmd> set port (port_id) vf (vf_id) (mac_addr) \
          (exact-mac|exact-mac-vlan|hashmac|hashmac-vlan) (on|off)
```

set port - rx mode(for VF)

Set the VF receive mode of a port:

```
testpmd> set port (port_id) vf (vf_id) \
         rxmode (AUPE|ROPE|BAM|MPE) (on|off)
```

The available receive modes are:

- `AUPE`: Accepts untagged VLAN.
- `ROPE`: Accepts unicast hash.
- `BAM`: Accepts broadcast packets.
- `MPE`: Accepts all multicast packets.

set port - tx_rate (for Queue)

Set TX rate limitation for a queue on a port:

```
testpmd> set port (port_id) queue (queue_id) rate (rate_value)
```

set port - tx_rate (for VF)

Set TX rate limitation for queues in VF on a port:

```
testpmd> set port (port_id) vf (vf_id) rate (rate_value) queue_mask (queue_mask)
```

set port - mirror rule

Set pool or vlan type mirror rule for a port:

```
testpmd> set port (port_id) mirror-rule (rule_id) \
          (pool-mirror-up|pool-mirror-down|vlan-mirror) \
          (poolmask|vlanid[,vlanid]*) dst-pool (pool_id) (on|off)
```

Set link mirror rule for a port:

```
testpmd> set port (port_id) mirror-rule (rule_id) \
         (uplink-mirror|downlink-mirror) dst-pool (pool_id) (on|off)
```

For example to enable mirror traffic with vlan 0,1 to pool 0:

```
set port 0 mirror-rule 0 vlan-mirror 0,1 dst-pool 0 on
```

reset port - mirror rule

Reset a mirror rule for a port:

```
testpmd> reset port (port_id) mirror-rule (rule_id)
```

set flush_rx

Set the flush on RX streams before forwarding. The default is flush `on`. Mainly used with PCAP drivers to turn off the default behavior of flushing the first 512 packets on RX streams:

```
testpmd> set flush_rx off
```

set bypass mode

Set the bypass mode for the lowest port on bypass enabled NIC:

```
testpmd> set bypass mode (normal|bypass|isolate) (port_id)
```

set bypass event

Set the event required to initiate specified bypass mode for the lowest port on a bypass enabled:

```
testpmd> set bypass event (timeout|os_on|os_off|power_on|power_off) \
         mode (normal|bypass|isolate) (port_id)
```

Where:

- `timeout`: Enable bypass after watchdog timeout.
- `os_on`: Enable bypass when OS/board is powered on.
- `os_off`: Enable bypass when OS/board is powered off.

- `power_on`: Enable bypass when power supply is turned on.
- `power_off`: Enable bypass when power supply is turned off.

set bypass timeout

Set the bypass watchdog timeout to `n` seconds where 0 = instant:

```
testpmd> set bypass timeout (0|1.5|2|3|4|8|16|32)
```

show bypass config

Show the bypass configuration for a bypass enabled NIC using the lowest port on the NIC:

```
testpmd> show bypass config (port_id)
```

set link up

Set link up for a port:

```
testpmd> set link-up port (port id)
```

set link down

Set link down for a port:

```
testpmd> set link-down port (port id)
```

8.4.5 Port Functions

The following sections show functions for configuring ports.

Note: Port configuration changes only become active when forwarding is started/restarted.

port attach

Attach a port specified by pci address or virtual device args.

To attach a new pci device, the device should be recognized by kernel first. Then it should be moved under DPDK management. Finally the port can be attached to testpmd.

For example, to move a pci device using ixgbe under DPDK management:

```
# Check the status of the available devices.
./tools/dpdk_nic_bind.py --status

Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:0a:00.0 '82599ES 10-Gigabit' if=eth2 drv=ixgbe unused=
# Bind the device to igb_uio.
sudo ./tools/dpdk_nic_bind.py -b igb_uio 0000:0a:00.0


# Recheck the status of the devices.
./tools/dpdk_nic_bind.py --status
Network devices using DPDK-compatible driver
============================================
0000:0a:00.0 '82599ES 10-Gigabit' drv=igb_uio unused=
```

To attach a port created by virtual device, above steps are not needed.

port attach (identifier)

For example, to attach a port whose pci address is 0000:0a:00.0.

```
testpmd> port attach 0000:0a:00.0
Attaching a new port...
EAL: PCI device 0000:0a:00.0 on NUMA socket -1
EAL:   probe driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory mapped at 0x7f83bfa00000
EAL:   PCI memory mapped at 0x7f83bfa80000
PMD: eth_ixgbe_dev_init(): MAC: 2, PHY: 18, SFP+: 5
PMD: eth_ixgbe_dev_init(): port 0 vendorID=0x8086 deviceID=0x10fb
Port 0 is attached. Now total ports is 1
Done
```

For example, to attach a port created by pcap PMD.

```
testpmd> port attach eth_pcap0
Attaching a new port...
PMD: Initializing pmd_pcap for eth_pcap0
PMD: Creating pcap-backed ethdev on numa socket 0
Port 0 is attached. Now total ports is 1
Done
```

In this case, identifier is `eth_pcap0`. This identifier format is the same as `--vdev` format of DPDK applications.

For example, to re-attach a bonded port which has been previously detached, the mode and slave parameters must be given.

```
testpmd> port attach eth_bond_0,mode=0,slave=1
Attaching a new port...
EAL: Initializing pmd_bond for eth_bond_0
EAL: Create bonded device eth_bond_0 on port 0 in mode 0 on socket 0.
Port 0 is attached. Now total ports is 1
Done
```

port detach

Detach a specific port.

Before detaching a port, the port should be closed:

```
testpmd> port detach (port_id)
```

For example, to detach a pci device port 0.

```
testpmd> port close 0
Closing ports...
Done
```

```
testpmd> port detach 0
Detaching a port...
EAL: PCI device 0000:0a:00.0 on NUMA socket -1
EAL:   remove driver: 8086:10fb rte_ixgbe_pmd
EAL:   PCI memory unmapped at 0x7f83bfa00000
EAL:   PCI memory unmapped at 0x7f83bfa80000
Done
```

For example, to detach a virtual device port 0.

```
testpmd> port close 0
Closing ports...
Done

testpmd> port detach 0
Detaching a port...
PMD: Closing pcap ethdev on numa socket 0
Port 'eth_pcap0' is detached. Now total ports is 0
Done
```

To remove a pci device completely from the system, first detach the port from testpmd. Then the device should be moved under kernel management. Finally the device can be removed using kernel pci hotplug functionality.

For example, to move a pci device under kernel management:

```
sudo ./tools/dpdk_nic_bind.py -b ixgbe 0000:0a:00.0

./tools/dpdk_nic_bind.py --status

Network devices using DPDK-compatible driver
============================================
<none>

Network devices using kernel driver
===================================
0000:0a:00.0 '82599ES 10-Gigabit' if=eth2 drv=ixgbe unused=igb_uio
```

To remove a port created by a virtual device, above steps are not needed.

port start

Start all ports or a specific port:

```
testpmd> port start (port_id|all)
```

port stop

Stop all ports or a specific port:

```
testpmd> port stop (port_id|all)
```

port close

Close all ports or a specific port:

```
testpmd> port close (port_id|all)
```

port start/stop queue

Start/stop a rx/tx queue on a specific port:

```
testpmd> port (port_id) (rxq|txq) (queue_id) (start|stop)
```

Only take effect when port is started.

port config - speed

Set the speed and duplex mode for all ports or a specific port:

```
testpmd> port config (port_id|all) speed (10|100|1000|10000|auto) \
         duplex (half|full|auto)
```

port config - queues/descriptors

Set number of queues/descriptors for rxq, txq, rxd and txd:

```
testpmd> port config all (rxq|txq|rxd|txd) (value)
```

This is equivalent to the `--rxq`, `--txq`, `--rxd` and `--txd` command-line options.

port config - max-pkt-len

Set the maximum packet length:

```
testpmd> port config all max-pkt-len (value)
```

This is equivalent to the `--max-pkt-len` command-line option.

port config - CRC Strip

Set hardware CRC stripping on or off for all ports:

```
testpmd> port config all crc-strip (on|off)
```

CRC stripping is off by default.

The `on` option is equivalent to the `--crc-strip` command-line option.

port config - RX Checksum

Set hardware RX checksum offload to on or off for all ports:

```
testpmd> port config all rx-cksum (on|off)
```

Checksum offload is off by default.

The `on` option is equivalent to the `--enable-rx-cksum` command-line option.

port config - VLAN

Set hardware VLAN on or off for all ports:

```
testpmd> port config all hw-vlan (on|off)
```

Hardware VLAN is on by default.

The `off` option is equivalent to the `--disable-hw-vlan` command-line option.

port config - VLAN filter

Set hardware VLAN filter on or off for all ports:

```
testpmd> port config all hw-vlan-filter (on|off)
```

Hardware VLAN filter is on by default.

The `off` option is equivalent to the `--disable-hw-vlan-filter` command-line option.

port config - VLAN strip

Set hardware VLAN strip on or off for all ports:

```
testpmd> port config all hw-vlan-strip (on|off)
```

Hardware VLAN strip is on by default.

The `off` option is equivalent to the `--disable-hw-vlan-strip` command-line option.

port config - VLAN extend

Set hardware VLAN extend on or off for all ports:

```
testpmd> port config all hw-vlan-extend (on|off)
```

Hardware VLAN extend is off by default.

The `off` option is equivalent to the `--disable-hw-vlan-extend` command-line option.

port config - Drop Packets

Set packet drop for packets with no descriptors on or off for all ports:

```
testpmd> port config all drop-en (on|off)
```

Packet dropping for packets with no descriptors is off by default.

The `on` option is equivalent to the `--enable-drop-en` command-line option.

port config - RSS

Set the RSS (Receive Side Scaling) mode on or off:

```
testpmd> port config all rss (all|ip|tcp|udp|sctp|ether|none)
```

RSS is on by default.

The `none` option is equivalent to the `--disable-rss` command-line option.

port config - RSS Reta

Set the RSS (Receive Side Scaling) redirection table:

```
testpmd> port config all rss reta (hash,queue)[,(hash,queue)]
```

port config - DCB

Set the DCB mode for an individual port:

```
testpmd> port config (port_id) dcb vt (on|off) (traffic_class) pfc (on|off)
```

The traffic class should be 4 or 8.

port config - Burst

Set the number of packets per burst:

```
testpmd> port config all burst (value)
```

This is equivalent to the `--burst` command-line option.

port config - Threshold

Set thresholds for TX/RX queues:

```
testpmd> port config all (threshold) (value)
```

Where the threshold type can be:

- `txpt:` Set the prefetch threshold register of the TX rings, 0 <= value <= 255.
- `txht:` Set the host threshold register of the TX rings, 0 <= value <= 255.
- `txwt:` Set the write-back threshold register of the TX rings, 0 <= value <= 255.
- `rxpt:` Set the prefetch threshold register of the RX rings, 0 <= value <= 255.
- `rxht:` Set the host threshold register of the RX rings, 0 <= value <= 255.
- `rxwt:` Set the write-back threshold register of the RX rings, 0 <= value <= 255.
- `txfreet:` Set the transmit free threshold of the TX rings, 0 <= value <= txd.
- `rxfreet:` Set the transmit free threshold of the RX rings, 0 <= value <= rxd.
- `txrst:` Set the transmit RS bit threshold of TX rings, 0 <= value <= txd.

These threshold options are also available from the command-line.

8.4.6 Link Bonding Functions

The Link Bonding functions make it possible to dynamically create and manage link bonding devices from within testpmd interactive prompt.

create bonded device

Create a new bonding device:

```
testpmd> create bonded device (mode) (socket)
```

For example, to create a bonded device in mode 1 on socket 0:

```
testpmd> create bonded 1 0
created new bonded device (port X)
```

add bonding slave

Adds Ethernet device to a Link Bonding device:

```
testpmd> add bonding slave (slave id) (port id)
```

For example, to add Ethernet device (port 6) to a Link Bonding device (port 10):

```
testpmd> add bonding slave 6 10
```

remove bonding slave

Removes an Ethernet slave device from a Link Bonding device:

```
testpmd> remove bonding slave (slave id) (port id)
```

For example, to remove Ethernet slave device (port 6) to a Link Bonding device (port 10):

```
testpmd> remove bonding slave 6 10
```

set bonding mode

Set the Link Bonding mode of a Link Bonding device:

```
testpmd> set bonding mode (value) (port id)
```

For example, to set the bonding mode of a Link Bonding device (port 10) to broadcast (mode 3):

```
testpmd> set bonding mode 3 10
```

set bonding primary

Set an Ethernet slave device as the primary device on a Link Bonding device:

```
testpmd> set bonding primary (slave id) (port id)
```

For example, to set the Ethernet slave device (port 6) as the primary port of a Link Bonding device (port 10):

```
testpmd> set bonding primary 6 10
```

set bonding mac

Set the MAC address of a Link Bonding device:

```
testpmd> set bonding mac (port id) (mac)
```

For example, to set the MAC address of a Link Bonding device (port 10) to 00:00:00:00:00:01:

```
testpmd> set bonding mac 10 00:00:00:00:00:01
```

set bonding xmit_balance_policy

Set the transmission policy for a Link Bonding device when it is in Balance XOR mode:

```
testpmd> set bonding xmit_balance_policy (port_id) (l2|l23|l34)
```

For example, set a Link Bonding device (port 10) to use a balance policy of layer 3+4 (IP addresses & UDP ports):

```
testpmd> set bonding xmit_balance_policy 10 l34
```

set bonding mon_period

Set the link status monitoring polling period in milliseconds for a bonding device.

This adds support for PMD slave devices which do not support link status interrupts. When the mon_period is set to a value greater than 0 then all PMD's which do not support link status ISR will be queried every polling interval to check if their link status has changed:

```
testpmd> set bonding mon_period (port_id) (value)
```

For example, to set the link status monitoring polling period of bonded device (port 5) to 150ms:

```
testpmd> set bonding mon_period 5 150
```

show bonding config

Show the current configuration of a Link Bonding device:

```
testpmd> show bonding config (port id)
```

For example, to show the configuration a Link Bonding device (port 9) with 3 slave devices (1, 3, 4) in balance mode with a transmission policy of layer 2+3:

```
testpmd> show bonding config 9
     Bonding mode: 2
     Balance Xmit Policy: BALANCE_XMIT_POLICY_LAYER23
     Slaves (3): [1 3 4]
     Active Slaves (3): [1 3 4]
     Primary: [3]
```

8.4.7 Register Functions

The Register Functions can be used to read from and write to registers on the network card referenced by a port number. This is mainly useful for debugging purposes. Reference should be made to the appropriate datasheet for the network card for details on the register addresses and fields that can be accessed.

read reg

Display the value of a port register:

```
testpmd> read reg (port_id) (address)
```

For example, to examine the Flow Director control register (FDIRCTL, 0x0000EE000) on an Intel 82599 10 GbE Controller:

```
testpmd> read reg 0 0xEE00
port 0 PCI register at offset 0xEE00: 0x4A060029 (1241907241)
```

read regfield

Display a port register bit field:

```
testpmd> read regfield (port_id) (address) (bit_x) (bit_y)
```

For example, reading the lowest two bits from the register in the example above:

```
testpmd> read regfield 0 0xEE00 0 1
port 0 PCI register at offset 0xEE00: bits[0, 1]=0x1 (1)
```

read regbit

Display a single port register bit:

```
testpmd> read regbit (port_id) (address) (bit_x)
```

For example, reading the lowest bit from the register in the example above:

```
testpmd> read regbit 0 0xEE00 0
port 0 PCI register at offset 0xEE00: bit 0=1
```

write reg

Set the value of a port register:

```
testpmd> write reg (port_id) (address) (value)
```

For example, to clear a register:

```
testpmd> write reg 0 0xEE00 0x0
port 0 PCI register at offset 0xEE00: 0x00000000 (0)
```

write regfield

Set bit field of a port register:

```
testpmd> write regfield (port_id) (address) (bit_x) (bit_y) (value)
```

For example, writing to the register cleared in the example above:

```
testpmd> write regfield 0 0xEE00 0 1 2
port 0 PCI register at offset 0xEE00: 0x00000002 (2)
```

write regbit

Set single bit value of a port register:

```
testpmd> write regbit (port_id) (address) (bit_x) (value)
```

For example, to set the high bit in the register from the example above:

```
testpmd> write regbit 0 0xEE00 31 1
port 0 PCI register at offset 0xEE00: 0x8000000A (2147483658)
```

8.4.8 Filter Functions

This section details the available filter functions that are available.

ethertype_filter

Add or delete a L2 Ethertype filter, which identify packets by their L2 Ethertype mainly assign them to a receive queue:

```
ethertype_filter (port_id) (add|del) (mac_addr|mac_ignr) (mac_address) \
                 ethertype (ether_type) (drop|fwd) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the Ethertype filter assigned on.
- `mac_addr`: Compare destination mac address.
- `mac_ignr`: Ignore destination mac address match.
- `mac_address`: Destination mac address to match.
- `ether_type`: The EtherType value want to match, for example 0x0806 for ARP packet. 0x0800 (IPv4) and 0x86DD (IPv6) are invalid.
- `queue_id`: The receive queue associated with this EtherType filter. It is meaningless when deleting or dropping.

Example, to add/remove an ethertype filter rule:

```
testpmd> ethertype_filter 0 add mac_ignr 00:11:22:33:44:55 \
                          ethertype 0x0806 fwd queue 3

testpmd> ethertype_filter 0 del mac_ignr 00:11:22:33:44:55 \
                          ethertype 0x0806 fwd queue 3
```

2tuple_filter

Add or delete a 2-tuple filter, which identifies packets by specific protocol and destination TCP/UDP port and forwards packets into one of the receive queues:

```
2tuple_filter (port_id) (add|del) dst_port (dst_port_value) \
              protocol (protocol_value) mask (mask_value) \
              tcp_flags (tcp_flags_value) priority (prio_value) \
              queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the 2-tuple filter assigned on.

- `dst_port_value`: Destination port in L4.
- `protocol_value`: IP L4 protocol.
- `mask_value`: Participates in the match or not by bit for field above, 1b means participate.
- `tcp_flags_value`: TCP control bits. The non-zero value is invalid, when the pro_value is not set to 0x06 (TCP).
- `prio_value`: Priority of this filter.
- `queue_id`: The receive queue associated with this 2-tuple filter.

Example, to add/remove an 2tuple filter rule:

```
testpmd> 2tuple_filter 0 add dst_port 32 protocol 0x06 mask 0x03 \
                       tcp_flags 0x02 priority 3 queue 3

testpmd> 2tuple_filter 0 del dst_port 32 protocol 0x06 mask 0x03 \
                       tcp_flags 0x02 priority 3 queue 3
```

5tuple_filter

Add or delete a 5-tuple filter, which consists of a 5-tuple (protocol, source and destination IP addresses, source and destination TCP/UDP/SCTP port) and routes packets into one of the receive queues:

```
5tuple_filter (port_id) (add|del) dst_ip (dst_address) src_ip \
              (src_address) dst_port (dst_port_value) \
              src_port (src_port_value) protocol (protocol_value) \
              mask (mask_value) tcp_flags (tcp_flags_value) \
              priority (prio_value) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the 5-tuple filter assigned on.
- `dst_address`: Destination IP address.
- `src_address`: Source IP address.
- `dst_port_value`: TCP/UDP destination port.
- `src_port_value`: TCP/UDP source port.
- `protocol_value`: L4 protocol.
- `mask_value`: Participates in the match or not by bit for field above, 1b means participate
- `tcp_flags_value`: TCP control bits. The non-zero value is invalid, when the protocol_value is not set to 0x06 (TCP).
- `prio_value`: The priority of this filter.
- `queue_id`: The receive queue associated with this 5-tuple filter.

Example, to add/remove an 5tuple filter rule:

```
testpmd> 5tuple_filter 0 add dst_ip 2.2.2.5 src_ip 2.2.2.4 \
         dst_port 64 src_port 32 protocol 0x06 mask 0x1F \
         flags 0x0 priority 3 queue 3

testpmd> 5tuple_filter 0 del dst_ip 2.2.2.5 src_ip 2.2.2.4 \
         dst_port 64 src_port 32 protocol 0x06 mask 0x1F \
         flags 0x0 priority 3 queue 3
```

syn_filter

Using the SYN filter, TCP packets whose *SYN* flag is set can be forwarded to a separate queue:

```
syn_filter (port_id) (add|del) priority (high|low) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the SYN filter assigned on.
- `high`: This SYN filter has higher priority than other filters.
- `low`: This SYN filter has lower priority than other filters.
- `queue_id`: The receive queue associated with this SYN filter

Example:

```
testpmd> syn_filter 0 add priority high queue 3
```

flex_filter

With flex filter, packets can be recognized by any arbitrary pattern within the first 128 bytes of the packet and routed into one of the receive queues:

```
flex_filter (port_id) (add|del) len (len_value) bytes (bytes_value) \
            mask (mask_value) priority (prio_value) queue (queue_id)
```

The available information parameters are:

- `port_id`: The port which the Flex filter is assigned on.
- `len_value`: Filter length in bytes, no greater than 128.
- `bytes_value`: A string in hexadecimal, means the value the flex filter needs to match.
- `mask_value`: A string in hexadecimal, bit 1 means corresponding byte participates in the match.
- `prio_value`: The priority of this filter.
- `queue_id`: The receive queue associated with this Flex filter.

Example:

```
testpmd> flex_filter 0 add len 16 bytes 0x00000000000000000000000008060000 \
                       mask 000C priority 3 queue 3

testpmd> flex_filter 0 del len 16 bytes 0x00000000000000000000000008060000 \
                       mask 000C priority 3 queue 3
```

flow_director_filter

The Flow Director works in receive mode to identify specific flows or sets of flows and route them to specific queues.

Four types of filtering are supported which are referred to as Perfect Match, Signature, Perfect-mac-vlan and Perfect-tunnel filters, the match mode is set by the `--pkt-filter-mode` command-line parameter:

- Perfect match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for IP flow.
- Signature filters. The hardware checks a match between a hash-based signature of the masked fields of the received packet.
- Perfect-mac-vlan match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for MAC VLAN flow.
- Perfect-tunnel match filters. The hardware checks a match between the masked fields of the received packets and the programmed filters. The masked fields are for tunnel flow.

The Flow Director filters can match the different fields for different type of packet: flow type, specific input set per flow type and the flexible payload.

The Flow Director can also mask out parts of all of these fields so that filters are only applied to certain fields or parts of the fields.

Different NICs may have different capabilities, command show port fdir (port_id) can be used to acquire the information.

# Commands to add flow director filters of different flow types:

```
flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-other|ipv4-frag|ipv6-other|ipv6-frag)
                     src (src_ip_address) dst (dst_ip_address) \
                     vlan (vlan_value) flexbytes (flexbytes_value) \
                     (drop|fwd) pf|vf(vf_id) queue (queue_id) \
                     fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-tcp|ipv4-udp|ipv6-tcp|ipv6-udp) \
                     src (src_ip_address) (src_port) \
                     dst (dst_ip_address) (dst_port) \
                     vlan (vlan_value) flexbytes (flexbytes_value) \
                     (drop|fwd) queue pf|vf(vf_id) (queue_id) \
                     fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) \
                     flow (ipv4-sctp|ipv6-sctp) \
                     src (src_ip_address) (src_port) \
                     dst (dst_ip_address) (dst_port)
                     tag (verification_tag) vlan (vlan_value) \
                     flexbytes (flexbytes_value) (drop|fwd) \
                     pf|vf(vf_id) queue (queue_id) fd_id (fd_id_value)

flow_director_filter (port_id) mode IP (add|del|update) flow l2_payload \
                     ether (ethertype) flexbytes (flexbytes_value) \
                     (drop|fwd) pf|vf(vf_id) queue (queue_id)
                     fd_id (fd_id_value)

flow_director_filter (port_id) mode MAC-VLAN (add|del|update) \
                     mac (mac_address) vlan (vlan_value) \
                     flexbytes (flexbytes_value) (drop|fwd) \
                     queue (queue_id) fd_id (fd_id_value)

flow_director_filter (port_id) mode Tunnel (add|del|update) \
                     mac (mac_address) vlan (vlan_value) \
                     tunnel (NVGRE|VxLAN) tunnel-id (tunnel_id_value) \
                     flexbytes (flexbytes_value) (drop|fwd) \
                     queue (queue_id) fd_id (fd_id_value)
```

For example, to add an ipv4-udp flow type filter:

```
testpmd> flow_director_filter 0 add flow ipv4-udp src 2.2.2.3 32 \
         dst 2.2.2.5 33 vlan 0x1 flexbytes (0x88,0x48) fwd pf queue 1 fd_id 1
```

For example, add an ipv4-other flow type filter:

```
testpmd> flow_director_filter 0 add flow ipv4-other src 2.2.2.3 \
          dst 2.2.2.5 vlan 0x1 flexbytes (0x88,0x48) fwd pf queue 1 fd_id 1
```

flush_flow_director

Flush all flow director filters on a device:

```
testpmd> flush_flow_director (port_id)
```

Example, to flush all flow director filter on port 0:

```
testpmd> flush_flow_director 0
```

flow_director_mask

Set flow director's input masks:

```
flow_director_mask (port_id) mode IP vlan (vlan_value) \
                   src_mask (ipv4_src) (ipv6_src) (src_port) \
                   dst_mask (ipv4_dst) (ipv6_dst) (dst_port)

flow_director_mask (port_id) mode MAC-VLAN vlan (vlan_value) \
                   mac (mac_value)

flow_director_mask (port_id) mode Tunnel vlan (vlan_value) \
                   mac (mac_value) tunnel-type (tunnel_type_value) \
                   tunnel-id (tunnel_id_value)
```

Example, to set flow director mask on port 0:

```
testpmd> flow_director_mask 0 vlan 0xefff \
         src_mask 255.255.255.255 \
             FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF 0xFFFF \
         dst_mask 255.255.255.255 \
             FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF 0xFFFF
```

flow_director_flex_mask

set masks of flow director's flexible payload based on certain flow type:

```
testpmd> flow_director_flex_mask (port_id) \
         flow (none|ipv4-other|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
               ipv6-other|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp| \
               l2_payload|all) (mask)
```

Example, to set flow director's flex mask for all flow type on port 0:

```
testpmd> flow_director_flex_mask 0 flow all \
         (0xff,0xff,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

flow_director_flex_payload

Configure flexible payload selection:

```
flow_director_flex_payload (port_id) (raw|l2|l3|l4) (config)
```

For example, to select the first 16 bytes from the offset 4 (bytes) of packet's payload as flexible payload:

```
testpmd> flow_director_flex_payload 0 l4 \
         (4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19)
```

get_sym_hash_ena_per_port

Get symmetric hash enable configuration per port:

```
get_sym_hash_ena_per_port (port_id)
```

For example, to get symmetric hash enable configuration of port 1:

```
testpmd> get_sym_hash_ena_per_port 1
```

set_sym_hash_ena_per_port

Set symmetric hash enable configuration per port to enable or disable:

```
set_sym_hash_ena_per_port (port_id) (enable|disable)
```

For example, to set symmetric hash enable configuration of port 1 to enable:

```
testpmd> set_sym_hash_ena_per_port 1 enable
```

get_hash_global_config

Get the global configurations of hash filters:

```
get_hash_global_config (port_id)
```

For example, to get the global configurations of hash filters of port 1:

```
testpmd> get_hash_global_config 1
```

set_hash_global_config

Set the global configurations of hash filters:

```
set_hash_global_config (port_id) (toeplitz|simple_xor|default) \
(ipv4|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp|ipv4-other|ipv6|ipv6-frag| \
ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other|l2_payload) \
(enable|disable)
```

For example, to enable simple_xor for flow type of ipv6 on port 2:

```
testpmd> set_hash_global_config 2 simple_xor ipv6 enable
```

set_hash_input_set

Set the input set for hash:

```
set_hash_input_set (port_id) (ipv4|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
ipv4-other|ipv6|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other| \
l2_payload) (ovlan|ivlan|src-ipv4|dst-ipv4|src-ipv6|dst-ipv6|ipv4-tos| \
ipv4-proto|ipv6-tc|ipv6-next-header|udp-src-port|udp-dst-port| \
tcp-src-port|tcp-dst-port|sctp-src-port|sctp-dst-port|sctp-veri-tag| \
udp-key|gre-key|fld-1st|fld-2nd|fld-3rd|fld-4th|fld-5th|fld-6th|fld-7th| \
fld-8th|none) (select|add)
```

For example, to add source IP to hash input set for flow type of ipv4 on port 0:

```
testpmd> set_hash_input_set 0 ipv4 src-ipv4 add
```

set_fdir_input_set

Set the input set for Fdir:

```
set_fdir_input_set (port_id) (ipv4|ipv4-frag|ipv4-tcp|ipv4-udp|ipv4-sctp| \
ipv4-other|ipv6|ipv6-frag|ipv6-tcp|ipv6-udp|ipv6-sctp|ipv6-other|l2_payload)
(src-ipv4|dst-ipv4|src-ipv6|dst-ipv6|udp-src-port|udp-dst-port| \
tcp-src-port|tcp-dst-port|sctp-src-port|sctp-dst-port|sctp-veri-tag| \
fld-1st|fld-2nd|fld-3rd|fld-4th|fld-5th|fld-6th|fld-7th|fld-8th|none) \
(select|add)
```

For example to add source IP to FD input set for flow type of ipv4 on port 0:

```
testpmd> set_fdir_input_set 0 ipv4 src-ipv4 add
```

global_config

Set different GRE key length for input set:

```
global_config (port_id) gre-key-len (number in bytes)
```

For example to set GRE key length for input set to 4 bytes on port 0:

```
testpmd> global_config 0 gre-key-len 4
```

CHAPTER 9

DPDK FAQ

This document contains some Frequently Asked Questions that arise when working with DPDK.

9.1 What does "EAL: map_all_hugepages(): open failed: Permission denied Cannot init memory" mean?

This is most likely due to the test application not being run with sudo to promote the user to a superuser. Alternatively, applications can also be run as regular user. For more information, please refer to *DPDK Getting Started Guide*.

9.2 If I want to change the number of TLB Hugepages allocated, how do I remove the original pages allocated?

The number of pages allocated can be seen by executing the following command:

```
grep Huge /proc/meminfo
```

Once all the pages are mmapped by an application, they stay that way. If you start a test application with less than the maximum, then you have free pages. When you stop and restart the test application, it looks to see if the pages are available in the `/dev/huge` directory and mmaps them. If you look in the directory, you will see `n` number of 2M pages files. If you specified 1024, you will see 1024 page files. These are then placed in memory segments to get contiguous memory.

If you need to change the number of pages, it is easier to first remove the pages. The tools/setup.sh script provides an option to do this. See the "Quick Start Setup Script" section in the *DPDK Getting Started Guide* for more information.

9.3 If I execute "l2fwd -c f -m 64 -n 3 – -p 3", I get the following output, indicating that there are no socket 0 hugepages to allocate the mbuf and ring structures to?

I have set up a total of 1024 Hugepages (that is, allocated 512 2M pages to each NUMA node).

The -m command line parameter does not guarantee that huge pages will be reserved on specific sockets. Therefore, allocated huge pages may not be on socket 0. To request memory to be reserved on a specific socket, please use the –socket-mem command-line parameter instead of -m.

9.4 I am running a 32-bit DPDK application on a NUMA system, and sometimes the application initializes fine but cannot allocate memory. Why is that happening?

32-bit applications have limitations in terms of how much virtual memory is available, hence the number of hugepages they are able to allocate is also limited (1 GB per page size). If your system has a lot (>1 GB per page size) of hugepage memory, not all of it will be allocated. Due to hugepages typically being allocated on a local NUMA node, the hugepages allocation the application gets during the initialization depends on which NUMA node it is running on (the EAL does not affinitize cores until much later in the initialization process). Sometimes, the Linux OS runs the DPDK application on a core that is located on a different NUMA node from DPDK master core and therefore all the hugepages are allocated on the wrong socket.

To avoid this scenario, either lower the amount of hugepage memory available to 1 GB per page size (or less), or run the application with taskset affinitizing the application to a would-be master core.

For example, if your EAL coremask is 0xff0, the master core will usually be the first core in the coremask (0x10); this is what you have to supply to taskset:

```
taskset 0x10 ./l2fwd -c 0xff0 -n 2
```

In this way, the hugepages have a greater chance of being allocated to the correct socket. Additionally, a `--socket-mem` option could be used to ensure the availability of memory for each socket, so that if hugepages were allocated on the wrong socket, the application simply will not start.

9.5 On application startup, there is a lot of EAL information printed. Is there any way to reduce this?

Yes, each EAL has a configuration file that is located in the /config directory. Within each configuration file, you will find CONFIG_RTE_LOG_LEVEL=8. You can change this to a lower value, such as 6 to reduce this printout of debug information. The following is a list of LOG levels that can be found in the rte_log.h file. You must remove, then rebuild, the EAL directory for the change to become effective as the configuration file creates the rte_config.h file in the EAL directory.

```
#define RTE_LOG_EMERG 1U    /* System is unusable. */
#define RTE_LOG_ALERT 2U    /* Action must be taken immediately. */
#define RTE_LOG_CRIT 3U     /* Critical conditions. */
#define RTE_LOG_ERR 4U      /* Error conditions. */
#define RTE_LOG_WARNING 5U  /* Warning conditions. */
#define RTE_LOG_NOTICE 6U   /* Normal but significant condition. */
#define RTE_LOG_INFO 7U     /* Informational. */
#define RTE_LOG_DEBUG 8U    /* Debug-level messages. */
```

9.6 How can I tune my network application to achieve lower latency?

Traditionally, there is a trade-off between throughput and latency. An application can be tuned to achieve a high throughput, but the end-to-end latency of an average packet typically increases as a result. Similarly, the application can be tuned to have, on average, a low end-to-end latency at the cost of lower throughput.

To achieve higher throughput, the DPDK attempts to aggregate the cost of processing each packet individually by processing packets in bursts. Using the testpmd application as an example, the "burst" size can be set on the command line to a value of 16 (also the default value). This allows the application to request 16 packets at a time from the PMD. The testpmd application then immediately attempts to transmit all the packets that were received, in this case, all 16 packets. The packets are not transmitted until the tail pointer is updated on the corresponding TX queue of the network port. This behavior is desirable when tuning for high throughput because the cost of tail pointer updates to both the RX and TX queues can be spread across 16 packets, effectively hiding the relatively slow MMIO cost of writing to the PCIe* device.

However, this is not very desirable when tuning for low latency, because the first packet that was received must also wait for the other 15 packets to be received. It cannot be transmitted until the other 15 packets have also been processed because the NIC will not know to transmit the packets until the TX tail pointer has been updated, which is not done until all 16 packets have been processed for transmission.

To consistently achieve low latency even under heavy system load, the application developer should avoid processing packets in bunches. The testpmd application can be configured from the command line to use a burst value of 1. This allows a single packet to be processed at a time, providing lower latency, but with the added cost of lower throughput.

9.7 Without NUMA enabled, my network throughput is low, why?

I have a dual Intel® Xeon® E5645 processors 2.40 GHz with four Intel® 82599 10 Gigabit Ethernet NICs. Using eight logical cores on each processor with RSS set to distribute network load from two 10 GbE interfaces to the cores on each processor.

Without NUMA enabled, memory is allocated from both sockets, since memory is interleaved. Therefore, each 64B chunk is interleaved across both memory domains.

The first 64B chunk is mapped to node 0, the second 64B chunk is mapped to node 1, the third to node 0, the fourth to node 1. If you allocated 256B, you would get memory that looks like this:

```
256B buffer
Offset 0x00 - Node 0
Offset 0x40 - Node 1
Offset 0x80 - Node 0
Offset 0xc0 - Node 1
```

Therefore, packet buffers and descriptor rings are allocated from both memory domains, thus incurring QPI bandwidth accessing the other memory and much higher latency. For best performance with NUMA disabled, only one socket should be populated.

9.8 I am getting errors about not being able to open files. Why?

As the DPDK operates, it opens a lot of files, which can result in reaching the open files limits, which is set using the ulimit command or in the limits.conf file. This is especially true when using a large number (>512) of 2 MB huge pages. Please increase the open file limit if your application is not able to open files. This can be done either by issuing a ulimit command or editing the limits.conf file. Please consult Linux manpages for usage information.

9.9 VF driver for IXGBE devices cannot be initialized

Some versions of Linux IXGBE driver do not assign a random MAC address to VF devices at initialization. In this case, this has to be done manually on the VM host, using the following command:

```
ip link set <interface> vf <VF function> mac <MAC address>
```

where <interface> being the interface providing the virtual functions for example, eth0, <VF function> being the virtual function number, for example 0, and <MAC address> being the desired MAC address.

9.10 Is it safe to add an entry to the hash table while running?

Currently the table implementation is not a thread safe implementation and assumes that locking between threads and processes is handled by the user's application. This is likely to be supported in future releases.

9.11 What is the purpose of setting iommu=pt?

DPDK uses a 1:1 mapping and does not support IOMMU. IOMMU allows for simpler VM physical address translation. The second role of IOMMU is to allow protection from unwanted memory access by an unsafe device that has DMA privileges. Unfortunately, the protection comes with an extremely high performance cost for high speed NICs.

Setting `iommu=pt` disables IOMMU support for the hypervisor.

9.12 When trying to send packets from an application to itself, meaning smac==dmac, using Intel(R) 82599 VF packets are lost.

Check on register `LLE(PFVMTXSSW[n])`, which allows an individual pool to send traffic and have it looped back to itself.

9.13 Can I split packet RX to use DPDK and have an application's higher order functions continue using Linux pthread?

The DPDK's lcore threads are Linux pthreads bound onto specific cores. Configure the DPDK to do work on the same cores and run the application's other work on other cores using the DPDK's "coremask" setting to specify which cores it should launch itself on.

9.14 Is it possible to exchange data between DPDK processes and regular userspace processes via some shared memory or IPC mechanism?

Yes - DPDK processes are regular Linux/BSD processes, and can use all OS provided IPC mechanisms.

9.15 Can the multiple queues in Intel(R) I350 be used with DPDK?

I350 has RSS support and 8 queue pairs can be used in RSS mode. It should work with multi-queue DPDK applications using RSS.

9.16 How can hugepage-backed memory be shared among multiple processes?

See the Primary and Secondary examples in the *multi-process sample application*.

CHAPTER 10

DPDK Release Notes

10.1 Description of Release

This document contains the release notes for Data Plane Development Kit (DPDK) release version 2.2.0 and previous releases.

It lists new features, fixed bugs, API and ABI changes and known issues.

For instructions on compiling and running the release, see the *DPDK Getting Started Guide*.

10.2 DPDK Release 2.2

10.2.1 New Features

- **Introduce ARMv7 and ARMv8 architectures.**
  - It is now possible to build DPDK for the ARMv7 and ARMv8 platforms.
  - ARMv7 can be tested with virtual PMD drivers.
  - ARMv8 can be tested with virtual and physical PMD drivers.
- **Enabled freeing of ring.**

  A new function `rte_ring_free()` has been added to allow the user to free a ring if it was created with `rte_ring_create()`.
- **Added keepalive support to EAL and example application.**
- **Added experimental cryptodev API**

  The cryptographic processing of packets is provided as a preview with two drivers for:
  - Intel QuickAssist devices
  - Intel AES-NI multi-buffer library

  Due to its experimental state, the API may change without prior notice.
- **Added ethdev APIs for additional IEEE1588 support.**

  Added functions to read, write and adjust system time in the NIC. Added client slave sample application to demonstrate the IEEE1588 functionality.

- **Extended Statistics.**

  Defined an extended statistics naming scheme to store metadata in the name string of each statistic. Refer to the Extended Statistics section of the Programmers Guide for more details.

  Implemented the extended statistics API for the following PMDs:

  - `igb`
  - `igbvf`
  - `i40e`
  - `i40evf`
  - `fm10k`
  - `virtio`

- **Added API in ethdev to retrieve RX/TX queue information.**
  - Added the ability for the upper layer to query RX/TX queue information.
  - Added new fields in `rte_eth_dev_info` to represent information about RX/TX descriptors min/max/align numbers, per queue, for the device.

- **Added RSS dynamic configuration to bonding.**

- **Updated the e1000 base driver.**

  The e1000 base driver was updated with several features including the following:

  - Added new i218 devices
  - Allowed both ULP and EEE in Sx state
  - Initialized 88E1543 (Marvell 1543) PHY
  - Added flags to set EEE advertisement modes
  - Supported inverted format ETrackId
  - Added bit to disable packetbuffer read
  - Added defaults for i210 RX/TX PBSIZE
  - Check more errors for ESB2 init and reset
  - Check more NVM read errors
  - Return code after setting receive address register
  - Removed all NAHUM6LP_HW tags

- **Added e1000 RX interrupt support.**

- **Added igb TSO support for both PF and VF.**

- **Added RSS enhancements to Intel x550 NIC.**
  - Added support for 512 entry RSS redirection table.
  - Added support for per VF RSS redirection table.

- **Added Flow director enhancements on Intel x550 NIC.**

  - Added 2 new flow director modes on x550. One is MAC VLAN mode, the other is tunnel mode.

- **Updated the i40e base driver.**

  The i40e base driver was updated with several changes including the following:

  - Added promiscuous on VLAN support
  - Added a workaround to drop all flow control frames
  - Added VF capabilities to virtual channel interface
  - Added TX Scheduling related AQ commands
  - Added additional PCTYPES supported for FortPark RSS
  - Added parsing for CEE DCBX TLVs
  - Added FortPark specific registers
  - Added AQ functions to handle RSS Key and LUT programming
  - Increased PF reset max loop limit

- **Added i40e vector RX/TX.**
- **Added i40e RX interrupt support.**
- **Added i40e flow control support.**
- **Added DCB support to i40e PF driver.**
- **Added RSS/FD input set granularity on Intel X710/XL710.**
- **Added different GRE key length for input set on Intel X710/XL710.**
- **Added flow director support in i40e VF.**
- **Added i40e support of early X722 series.**

  Added early X722 support, for evaluation only, as the hardware is alpha.

- **Added fm10k vector RX/TX.**
- **Added fm10k TSO support for both PF and VF.**
- **Added fm10k VMDQ support.**
- **New NIC Boulder Rapid support.**

  Added support for the Boulder Rapid variant of Intel's fm10k NIC family.

- **Enhanced support for the Chelsio CXGBE driver.**

  - Added support for Jumbo Frames.
  - Optimized forwarding performance for Chelsio T5 40GbE cards.

- **Improved enic TX packet rate.**

  Reduced frequency of TX tail pointer updates to the NIC.

- **Added support for link status interrupts in mlx4.**
- **Added partial support (TX only) for secondary processes in mlx4.**

- **Added support for Mellanox ConnectX-4 adapters (mlx5).**

  The mlx5 poll-mode driver implements support for Mellanox ConnectX-4 EN and Mellanox ConnectX-4 Lx EN families of 10/25/40/50/100 Gb/s adapters.

  Like mlx4, this PMD is only available for Linux and is disabled by default due to external dependencies (libibverbs and libmlx5).

- **Added driver for Netronome nfp-6xxx card.**

  Support for using Netronome nfp-6xxx with PCI VFs.

- **Added virtual szedata2 driver for COMBO cards.**

  Added virtual PMD for COMBO-100G and COMBO-80G cards. PMD is disabled in default configuration.

- **Enhanced support for virtio driver.**
  - Virtio ring layout optimization (fixed avail ring)
  - Vector RX
  - Simple TX
- **Added vhost-user multiple queue support.**
- **Added port hotplug support to vmxnet3.**
- **Added port hotplug support to xenvirt.**
- **Added ethtool shim and sample application.**
- **Added experimental performance thread example application.**

  The new sample application demonstrates L3 forwarding with different threading models: pthreads, cgroups, or lightweight threads. The example includes a simple cooperative scheduler.

  Due to its experimental state this application may change without notice. The application is supported only for Linux x86_64.

- **Enhancements to the IP pipeline application.**

  The following features have been added to the `ip_pipeline` application;
  - Added Multiple Producers/Multiple Consumers (MPSC) and fragmentation/reassembly support to software rings.
  - Added a dynamic pipeline reconfiguration feature that allows binding a pipeline to other threads at runtime using CLI commands.
  - Added enable/disable of `promisc` mode from `ip_pipeline` configuration file.
  - Added check on RX queues and TX queues of each link whether they are used correctly in the `ip_pipeline` configuration file.
  - Added flow id parameters to the flow-classification table entries.
  - Added more functions to the routing pipeline: ARP table enable/disable, Q-in-Q and MPLS encapsulation, add color (traffic-class for QoS) to the MPLS tag.
  - Added flow-actions pipeline for traffic metering/marking (for e.g. Two Rate Three Color Marker (trTCM)), policer etc.

  - Modified the pass-through pipeline's actions-handler to implement a generic approach to extract fields from the packet's header and copy them to packet metadata.

10.2.2 Resolved Issues

EAL

- **eal/linux: Fixed epoll timeout.**

  Fixed issue where the `rte_epoll_wait()` function didn't return when the underlying call to `epoll_wait()` timed out.

Drivers

- **e1000/base: Synchronize PHY interface on non-ME systems.**

  On power up, the MAC - PHY interface needs to be set to PCIe, even if the cable is disconnected. In ME systems, the ME handles this on exit from the Sx (Sticky mode) state. In non-ME, the driver handles it. Added a check for non-ME system to the driver code that handles it.

- **e1000/base: Increased timeout of reset check.**

  Previously, in `check_reset_block` RSPCIPHY was polled for 100 ms before determining that the ME veto was set. This was not enough and it was increased to 300 ms.

- **e1000/base: Disabled IPv6 extension header parsing on 82575.**

  Disabled IPv6 options as per hardware limitation.

- **e1000/base: Prevent ULP flow if cable connected.**

  Enabling ULP on link down when the cable is connected caused an infinite loop of link up/down indications in the NDIS driver. The driver now enables ULP only when the cable is disconnected.

- **e1000/base: Support different EEARBC for i210.**

  EEARBC has changed on i210. It means EEARBC has a different address on i210 than on other NICs. So, add a new entity named EEARBC_I210 to the register list and make sure the right one is being used on i210.

- **e1000/base: Fix K1 configuration.**

  Added fix for the following updates to the K1 configurations: TX idle period for entering K1 should be 128 ns. Minimum TX idle period in K1 should be 256 ns.

- **e1000/base: Fix link detect flow.**

  Fix link detect flow in case where auto-negotiate is not enabled, by calling `e1000_setup_copper_link_generic` instead of `e1000_phy_setup_autoneg`.

- **e1000/base: Fix link check for i354 M88E1112 PHY.**

  The `e1000_check_for_link_media_swap()` function is supposed to check PHY page 0 for copper and PHY page 1 for "other" (fiber) links. The driver switched back from page 1 to page 0 too soon, before `e1000_check_for_link_82575()` is executed and was never finding the link on the fiber (other).

If the link is copper, as the M88E1112 page address is set to 1, it should be set back to 0 before checking this link.

- **e1000/base: Fix beacon duration for i217.**

  Fix for I217 Packet Loss issue - The Management Engine sets the FEXTNVM4 Beacon Duration incorrectly. This fix ensures that the correct value will always be set. Correct value for this field is 8 usec.

- **e1000/base: Fix TIPG for non 10 half duplex mode.**

  TIPG value is increased when setting speed to 10 half duplex to prevent packet loss. However, it was never decreased again when speed changed. This caused performance issues in the NDIS driver. Fix this to restore TIPG to default value on non 10 half duplex.

- **e1000/base: Fix reset of DH89XXCC SGMII.**

  For DH89XXCC_SGMII, a write flush leaves registers of this device trashed (0xFFFFFFFF). Add check for this device.

  Also, after both Port SW Reset and Device Reset case, the platform should wait at least 3ms before reading any registers. Remove this condition since waiting is conditionally executed only for Device Reset.

- **e1000/base: Fix redundant PHY power down for i210.**

  Bit 11 of PHYREG 0 is used to power down PHY. The use of PHYREG 16 is no longer necessary.

- **e1000/base: fix jumbo frame CRC failures.**

  Change the value of register 776.20[11:2] for jumbo mode from 0x1A to 0x1F. This is to enlarge the gap between read and write pointers in the TX FIFO.

- **e1000/base: Fix link flap on 82579.**

  Several customers have reported a link flap issue on 82579. The symptoms are random and intermittent link losses when 82579 is connected to specific switches. the Issue was root caused as an inter-operability problem between the NIC and at least some Broadcom PHYs in the Energy Efficient Ethernet wake mechanism.

  To fix the issue, we are disabling the Phase Locked Loop shutdown in 100M Low Power Idle. This solution will cause an increase of power in 100M EEE link. It may cost an additional 28mW in this specific mode.

- **igb: Fixed IEEE1588 frame identification in I210.**

  Fixed issue where the flag `PKT_RX_IEEE1588_PTP` was not being set in the Intel I210 NIC, as the EtherType in RX descriptor is in bits 8:10 of Packet Type and not in the default bits 0:2.

- **igb: Fixed VF start with PF stopped.**

  VF needs the PF interrupt support initialized even if not started.

- **igb: Fixed VF MAC address when using with DPDK PF.**

  Assign a random MAC address in VF when not assigned by PF.

- **igb: Removed CRC bytes from byte counter statistics.**

- **ixgbe: Fixed issue with X550 DCB.**

  Fixed a DCB issue with x550 where for 8 TCs (Traffic Classes), if a packet with user priority 6 or 7 was injected to the NIC, then the NIC would only put 3 packets into the queue. There was also a similar issue for 4 TCs.

- **ixgbe: Removed burst size restriction of vector RX.**

  Fixed issue where a burst size less than 32 didn't receive anything.

- **ixgbe: Fixed VF start with PF stopped.**

  VF needs the PF interrupt support initialized even if not started.

- **ixgbe: Fixed TX hang when RS distance exceeds HW limit.**

  Fixed an issue where the TX queue can hang when a lot of highly fragmented packets have to be sent. As part of that fix, `tx_rs_thresh` for ixgbe PMD is not allowed to be greater then to 32 to comply with HW restrictions.

- **ixgbe: Fixed rx error statistic counter.**

  Fixed an issue that the rx error counter of ixgbe was not accurate. The mac short packet discard count (mspdc) was added to the counter. Mac local faults and mac remote faults are removed as they do not count packets but errors, and jabber errors were removed as they are already accounted for by the CRC error counter. Finally the XEC (l3 / l4 checksum error) counter was removed due to errata, see commit 256ff05a9cae for details.

- **ixgbe: Removed CRC bytes from byte counter statistics.**

- **i40e: Fixed base driver allocation when not using first numa node.**

  Fixed i40e issue that occurred when a DPDK application didn't initialize ports if memory wasn't available on socket 0.

- **i40e: Fixed maximum of 64 queues per port.**

  Fixed an issue in i40e where it would not support more than 64 queues per port, even though the hardware actually supports it. The real number of queues may vary, as long as the total number of queues used in PF, VFs, VMDq and FD does not exceeds the hardware maximum.

- **i40e: Fixed statistics of packets.**

  Added discarding packets on VSI to the stats and rectify the old statistics.

- **i40e: Fixed issue of not freeing memzone.**

  Fixed an issue of not freeing a memzone in the call to free the memory for adminq DMA.

- **i40e: Removed CRC bytes from byte counter statistics.**

- **mlx: Fixed driver loading.**

  The mlx drivers were unable to load when built as a shared library, due to a missing symbol in the mempool library.

- **mlx4: Performance improvements.**

  Fixed bugs in TX and RX flows that improves mlx4 performance.

- **mlx4: Fixed TX loss after initialization.**

- **mlx4: Fixed scattered TX with too many segments.**

- **mlx4: Fixed memory registration for indirect mbuf data.**
- **vhost: Fixed Qemu shutdown.**

  Fixed issue with libvirt `virsh destroy` not killing the VM.
- **virtio: Fixed crash after changing link state.**

  Fixed IO permission in the interrupt handler.
- **virtio: Fixed crash when releasing queue.**

  Fixed issue when releasing null control queue.

Libraries

- **hash: Fixed memory allocation of Cuckoo Hash key table.**

  Fixed issue where an incorrect Cuckoo Hash key table size could be calculated limiting the size to 4GB.
- **hash: Fixed incorrect lookup if key is all zero.**

  Fixed issue in hash library that occurred if an all zero key was not added to the table and the key was looked up, resulting in an incorrect hit.
- **hash: Fixed thread scaling by reducing contention.**

  Fixed issue in the hash library where, using multiple cores with hardware transactional memory support, thread scaling did not work, due to the global ring that is shared by all cores.

Examples

- **l3fwd: Fixed crash with IPv6.**
- **vhost_xen: Fixed compile error.**

Other

- This release drops compatibility with Linux kernel 2.6.33. The minimum kernel requirement is now 2.6.34.

10.2.3 Known Issues

- Some drivers do not fill in the packet type when receiving. As the l3fwd example application requires this info, the i40e vector driver must be disabled to benefit of the packet type with i40e.
- Some (possibly all) VF drivers (e.g. i40evf) do not handle any PF reset events/requests in the VF driver. This means that the VF driver may not work after a PF reset in the host side. The workaround is to avoid triggering any PF reset events/requests on the host side.
- 100G link report support is missing.

- **Mellanox PMDs (mlx4 & mlx5):**
  - PMDs do not support CONFIG_RTE_BUILD_COMBINE_LIBS and CONFIG_RTE_BUILD_SHARED_LIB simultaneously.
  - There is performance degradation for small packets when the PMD is compiled with `SGE_WR_N = 4` compared to the performance when `SGE_WR_N = 1`. If scattered packets are not used it is recommended to compile the PMD with `SGE_WR_N = 1`.
  - When a Multicast or Broadcast packet is sent to the SR-IOV mlx4 VF, it is returned back to the port.
  - PMDs report "bad" L4 checksum when IP packet is received.
  - mlx5 PMD reports "bad" checksum although the packet has "good" checksum. Will be fixed in upcoming MLNX_OFED release.

10.2.4 API Changes

- The deprecated flow director API is removed. It was replaced by `rte_eth_dev_filter_ctrl()`.
- The `dcb_queue` is renamed to `dcb_tc` in following dcb configuration structures: `rte_eth_dcb_rx_conf`, `rte_eth_dcb_tx_conf`, `rte_eth_vmdq_dcb_conf`, `rte_eth_vmdq_dcb_tx_conf`.
- The `rte_eth_rx_queue_count()` function now returns "int" instead of "uint32_t" to allow the use of negative values as error codes on return.
- The function `rte_eal_pci_close_one()` is removed. It was replaced by `rte_eal_pci_detach()`.
- The deprecated ACL API `ipv4vlan` is removed.
- The deprecated hash function `rte_jhash2()` is removed. It was replaced by `rte_jhash_32b()`.
- The deprecated KNI functions are removed: `rte_kni_create()`, `rte_kni_get_port_id()` and `rte_kni_info_get()`.
- The deprecated ring PMD functions are removed: `rte_eth_ring_pair_create()` and `rte_eth_ring_pair_attach()`.
- The devargs union field `virtual` is renamed to `virt` for C++ compatibility.

10.2.5 ABI Changes

- The EAL and ethdev structures `rte_intr_handle` and `rte_eth_conf` were changed to support RX interrupt. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.
- The ethdev flow director entries for SCTP were changed. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.
- The ethdev flow director structure `rte_eth_fdir_flow_ext` structure was changed. New fields were added to support flow director filtering in VF.

- The size of the ethdev structure `rte_eth_hash_filter_info` is changed by adding a new element `rte_eth_input_set_conf` in a union.
- New fields `rx_desc_lim` and `tx_desc_lim` are added into `rte_eth_dev_info` structure.
- For debug builds, the functions `rte_eth_rx_burst()`, `rte_eth_tx_burst()` `rte_eth_rx_descriptor_done()` and `rte_eth_rx_queue_count()` will no longer be separate functions in the DPDK libraries. Instead, they will only be present in the `rte_ethdev.h` header file.
- The maximum number of queues per port `CONFIG_RTE_MAX_QUEUES_PER_PORT` is increased to 1024.
- The mbuf structure was changed to support the unified packet type. This was already included in 2.1 under the `CONFIG_RTE_NEXT_ABI` #define.
- The dummy malloc library is removed. The content was moved into EAL in 2.1.
- The LPM structure is changed. The deprecated field `mem_location` is removed.
- librte_table LPM: A new parameter to hold the table name will be added to the LPM table parameter structure.
- librte_table hash: The key mask parameter is added to the hash table parameter structure for 8-byte key and 16-byte key extendable bucket and LRU tables.
- librte_port: Macros to access the packet meta-data stored within the packet buffer has been adjusted to cover the packet mbuf structure.
- librte_cfgfile: Allow longer names and values by increasing the constants `CFG_NAME_LEN` and `CFG_VALUE_LEN` to 64 and 256 respectively.
- vhost: a new field enabled is added to the `vhost_virtqueue` structure.
- vhost: a new field `virt_qp_nb` is added to `virtio_net` structure, and the `virtqueue` field is moved to the end of virtio_net structure.
- vhost: a new operation `vring_state_changed` is added to `virtio_net_device_ops` structure.
- vhost: a few spaces are reserved both at `vhost_virtqueue` and `virtio_net` structure for future extension.

10.2.6 Shared Library Versions

The libraries prepended with a plus sign were incremented in this version.

```
+ libethdev.so.2
+ librte_acl.so.2
+ librte_cfgfile.so.2
  librte_cmdline.so.1
  librte_distributor.so.1
+ librte_eal.so.2
+ librte_hash.so.2
  librte_ip_frag.so.1
  librte_ivshmem.so.1
  librte_jobstats.so.1
+ librte_kni.so.2
  librte_kvargs.so.1
+ librte_lpm.so.2
+ librte_mbuf.so.2
  librte_mempool.so.1
  librte_meter.so.1
+ librte_pipeline.so.2
  librte_pmd_bond.so.1
+ librte_pmd_ring.so.2
+ librte_port.so.2
  librte_power.so.1
  librte_reorder.so.1
  librte_ring.so.1
  librte_sched.so.1
+ librte_table.so.2
  librte_timer.so.1
+ librte_vhost.so.2
```

10.3 DPDK Release 2.1

10.3.1 New Features

- **Enabled cloning of indirect mbufs.**

  This feature removes a limitation of `rte_pktmbuf_attach()` which generated the warning: "mbuf we're attaching to must be direct".

  Now, when attaching to an indirect mbuf it is possible to:

  - Copy all relevant fields (address, length, offload, ...) as before.
  - Get the pointer to the mbuf that embeds the data buffer (direct mbuf), and increase the reference counter.

    When detaching the mbuf, we can now retrieve this direct mbuf as the pointer is determined from the buffer address.

- **Extended packet type support.**

  In previous releases mbuf packet types were indicated by 6 bits in the `ol_flags`. This was not enough for some supported NICs. For example i40e hardware can recognize more than 150 packet types. Not being able to identify these additional packet types limits access to hardware offload capabilities

  So an extended "unified" packet type was added to support all possible PMDs. The 16 bit packet_type in the mbuf structure was changed to 32 bits and used for this purpose.

  To avoid breaking ABI compatibility, the code changes for this feature are enclosed in a `RTE_NEXT_ABI` ifdef. This is enabled by default but can be turned off for ABI compatibility with DPDK R2.0.

- **Reworked memzone to be allocated by malloc and also support freeing.**

  In the memory hierarchy, memsegs are groups of physically contiguous hugepages, memzones are slices of memsegs, and malloc slices memzones into smaller memory chunks.

  This feature modifies `malloc()` so it partitions memsegs instead of memzones. Now memzones allocate their memory from the malloc heap.

  Backward compatibility with API and ABI are maintained.

This allow memzones, and any other structure based on memzones, for example mempools, to be freed. Currently only the API from freeing memzones is supported.

- **Interrupt mode PMD.**

  This feature introduces a low-latency one-shot RX interrupt into DPDK. It also adds a polling and interrupt mode switch control example.

  DPDK userspace interrupt notification and handling mechanism is based on UIO/VFIO with the following limitations:

  – Per queue RX interrupt events are only allowed in VFIO which supports multiple MSI-X vectors.

  – In UIO, the RX interrupt shares the same vector with other interrupts. When the RX interrupt and LSC interrupt are both enabled, only the former is available.

  – RX interrupt is only implemented for the linuxapp target.

  – The feature is only currently enabled for tow PMDs: ixgbe and igb.

- **Packet Framework enhancements.**

  Several enhancements were made to the Packet Framework:

  – A new configuration file syntax has been introduced for IP pipeline applications. Parsing of the configuration file is changed.

  – Implementation of the IP pipeline application is modified to make it more structured and user friendly.

  – Implementation of the command line interface (CLI) for each pipeline type has been moved to the separate compilation unit. Syntax of pipeline CLI commands has been changed.

  – Initialization of IP pipeline is modified to match the new parameters structure.

  – New implementation of pass-through pipeline, firewall pipeline, routing pipeline, and flow classification has been added.

  – Master pipeline with CLI interface has been added.

  – Added extended documentation of the IP Pipeline.

- **Added API for IEEE1588 timestamping.**

  This feature adds an ethdev API to enable, disable and read IEEE1588/802.1AS PTP timestamps from devices that support it. The following functions were added:

  – `rte_eth_timesync_enable()`

  – `rte_eth_timesync_disable()`

  – `rte_eth_timesync_read_rx_timestamp()`

  – `rte_eth_timesync_read_tx_timestamp()`

  The “ieee1588” forwarding mode in testpmd was also refactored to demonstrate the new API.

- **Added multicast address filtering.**

  Added multicast address filtering via a new ethdev function `set_mc_addr_list()`.

This overcomes a limitation in previous releases where the receipt of multicast packets on a given port could only be enabled by invoking the `rte_eth_allmulticast_enable()` function. This method did not work for VFs in SR-IOV architectures when the host PF driver does not allow these operation on VFs. In such cases, joined multicast addresses had to be added individually to the set of multicast addresses that are filtered by the [VF] port.

- **Added Flow Director extensions.**

  Several Flow Director extensions were added such as:

  - Support for RSS and Flow Director hashes in vector RX.
  - Added Flow Director for L2 payload.

- **Added RSS hash key size query per port.**

  This feature supports querying the RSS hash key size of each port. A new field `hash_key_size` has been added in the `rte_eth_dev_info` struct for storing hash key size in bytes.

- **Added userspace ethtool support.**

  Added userspace ethtool support to provide a familiar interface for applications that manage devices via kernel-space `ethtool_op` and `net_device_op`.

  The initial implementation focuses on operations that can be implemented through existing `netdev` APIs. More operations will be supported in later releases.

- **Updated the ixgbe base driver.**

  The ixgbe base driver was updated with several changes including the following:

  - Added a new 82599 device id.
  - Added new X550 PHY ids.
  - Added SFP+ dual-speed support.
  - Added wait helper for X550 IOSF accesses.
  - Added X550em features.
  - Added X557 PHY LEDs support.
  - Commands for flow director.
  - Issue firmware command when resetting X550em.

  See the git log for full details of the ixgbe/base changes.

- **Added additional hotplug support.**

  Port hotplug support was added to the following PMDs:

  - e1000/igb.
  - ixgbe.
  - i40e.
  - fm10k.
  - ring.

  - bonding.
  - virtio.

  Port hotplug support was added to BSD.

- **Added ixgbe LRO support.**

  Added LRO support for x540 and 82599 devices.

- **Added extended statistics for ixgbe.**

  Implemented `xstats_get()` and `xstats_reset()` in dev_ops for ixgbe to expose detailed error statistics to DPDK applications.

  These will be implemented for other PMDs in later releases.

- **Added proc_info application.**

  Created a new `proc_info` application, by refactoring the existing `dump_cfg` application, to demonstrate the usage of retrieving statistics, and the new extended statistics (see above), for DPDK interfaces.

- **Updated the i40e base driver.**

  The i40e base driver was updated with several changes including the following:

  - Support for building both PF and VF driver together.
  - Support for CEE DCBX on recent firmware versions.
  - Replacement of `i40e_debug_read_register()`.
  - Rework of `i40e_hmc_get_object_va`.
  - Update of shadow RAM read/write functions.
  - Enhancement of polling NVM semaphore.
  - Enhancements on adminq init and sending asq command.
  - Update of get/set LED functions.
  - Addition of AOC phy types to case statement in get_media_type.
  - Support for iSCSI capability.
  - Setting of FLAG_RD when sending driver version to FW.

  See the git log for full details of the i40e/base changes.

- **Added support for port mirroring in i40e.**

  Enabled mirror functionality in the i40e driver.

- **Added support for i40e double VLAN, QinQ, stripping and insertion.**

  Added support to the i40e driver for offloading double VLAN (QinQ) tags to the mbuf header, and inserting double vlan tags by hardware to the packets to be transmitted. Added a new field `vlan_tci_outer` in the `rte_mbuf` struct, and new flags in `ol_flags` to support this feature.

- **Added fm10k promiscuous mode support.**

  Added support for promiscuous/allmulticast enable and disable in the fm10k PF function. VF is not supported yet.

- **Added fm10k jumbo frame support.**

  Added support for jumbo frame less than 15K in both VF and PF functions in the fm10k pmd.

- **Added fm10k mac vlan filtering support.**

  Added support for the fm10k MAC filter, only available in PF. Updated the VLAN filter to add/delete one static entry in the MAC table for each combination of VLAN and MAC address.

- **Added support for the Broadcom bnx2x driver.**

  Added support for the Broadcom NetXtreme II bnx2x driver. It is supported only on Linux 64-bit and disabled by default.

- **Added support for the Chelsio CXGBE driver.**

  Added support for the CXGBE Poll Mode Driver for the Chelsio Terminator 5 series of 10G/40G adapters.

- **Enhanced support for Mellanox ConnectX-3 driver (mlx4).**

  – Support Mellanox OFED 3.0.

  – Improved performance for both RX and TX operations.

  – Better link status information.

  – Outer L3/L4 checksum offload support.

  – Inner L3/L4 checksum offload support for VXLAN.

- **Enabled VMXNET3 vlan filtering.**

  Added support for the VLAN filter functionality of the VMXNET3 interface.

- **Added support for vhost live migration.**

  Added support to allow live migration of vhost. Without this feature, qemu will report the following error: “migrate: Migration disabled: vhost lacks VHOST_F_LOG_ALL feature”.

- **Added support for pcap jumbo frames.**

  Extended the PCAP PMD to support jumbo frames for RX and TX.

- **Added support for the TILE-Gx architecture.**

  Added support for the EZchip TILE-Gx family of SoCs.

- **Added hardware memory transactions/lock elision for x86.**

  Added the use of hardware memory transactions (HTM) on fast-path for rwlock and spinlock (a.k.a. lock elision). The methods are implemented for x86 using Restricted Transactional Memory instructions (Intel(r) Transactional Synchronization Extensions). The implementation fall-backs to the normal rwlock if HTM is not available or memory transactions fail. This is not a replacement for all rwlock usages since not all critical sections protected by locks are friendly to HTM. For example, an attempt to perform a HW I/O operation inside a hardware memory transaction always aborts the transaction since the CPU is not able to roll-back should the transaction fail. Therefore, hardware transactional locks are not advised to be used around `rte_eth_rx_burst()` and `rte_eth_tx_burst()` calls.

- **Updated Jenkins Hash function**

  Updated the version of the Jenkins Hash (jhash) function used in DPDK from the 1996 version to the 2006 version. This gives up to 35% better performance, compared to the original one.

  Note, the hashes generated by the updated version differ from the hashes generated by the previous version.

- **Added software implementation of the Toeplitz RSS hash**

  Added a software implementation of the Toeplitz hash function used by RSS. It can be used either for packet distribution on a single queue NIC or for simulating RSS computation on a specific NIC (for example after GRE header de-encapsulation).

- **Replaced the existing hash library with a Cuckoo hash implementation.**

  Replaced the existing hash library with another approach, using the Cuckoo Hash method to resolve collisions (open addressing). This method pushes items from a full bucket when a new entry must be added to it, storing the evicted entry in an alternative location, using a secondary hash function.

  This gives the user the ability to store more entries when a bucket is full, in comparison with the previous implementation.

  The API has not been changed, although new fields have been added in the `rte_hash` structure, which has been changed to internal use only.

  The main change when creating a new table is that the number of entries per bucket is now fixed, so its parameter is ignored now (it is still there to maintain the same parameters structure).

  Also, the maximum burst size in lookup_burst function hash been increased to 64, to improve performance.

- **Optimized KNI RX burst size computation.**

  Optimized KNI RX burst size computation by avoiding checking how many entries are in `kni->rx_q` prior to actually pulling them from the fifo.

- **Added KNI multicast.**

  Enabled adding multicast addresses to KNI interfaces by adding an empty callback for `set_rx_mode` (typically used for setting up hardware) so that the ioctl succeeds. This is the same thing as the Linux tap interface does.

- **Added cmdline polling mode.**

  Added the ability to process console input in the same thread as packet processing by using the `poll()` function.

- **Added VXLAN Tunnel End point sample application.**

  Added a Tunnel End point (TEP) sample application that simulates a VXLAN Tunnel Endpoint (VTEP) termination in DPDK. It is used to demonstrate the offload and filtering capabilities of Intel XL710 10/40 GbE NICsfor VXLAN packets.

- **Enabled combining of the "-m" and "–no-huge" EAL options.**

  Added option to allow combining of the `-m` and `--no-huge` EAL command line options.

This allows user application to run as non-root but with higher memory allocations, and removes a constraint on `--no-huge` mode being limited to 64M.

10.3.2 Resolved Issues

- **acl: Fix ambiguity between test rules.**

  Some test rules had equal priority for the same category. That could cause an ambiguity in building the trie and test results.

- **acl: Fix invalid rule wildness calculation for bitmask field type.**
- **acl: Fix matching rule.**
- **acl: Fix unneeded trie splitting for subset of rules.**

  When rebuilding a trie for limited rule-set, don't try to split the rule-set even further.

- **app/testpmd: Fix crash when port id out of bound.**

  Fixed issues in testpmd where using a port greater than 32 would cause a seg fault.

  Fixes: edab33b1c01d ("app/testpmd: support port hotplug")

- **app/testpmd: Fix reply to a multicast ICMP request.**

  Set the IP source and destination addresses in the IP header of the ICMP reply.

- **app/testpmd: fix MAC address in ARP reply.**

  Fixed issue where in the `icmpecho` forwarding mode, ARP replies from testpmd contain invalid zero-filled MAC addresses.

  Fixes: 31db4d38de72 ("net: change arp header struct declaration")

- **app/testpmd: fix default flow control values.**

  Fixes: 422a20a4e62d ("app/testpmd: fix uninitialized flow control variables")

- **bonding: Fix crash when stopping inactive slave.**
- **bonding: Fix device initialization error handling.**
- **bonding: Fix initial link status of slave.**

  On Fortville NIC, link status change interrupt callback was not executed when slave in bonding was (re-)started.

- **bonding: Fix socket id for LACP slave.**

  Fixes: 46fb43683679 ("bond: add mode 4")

- **bonding: Fix device initialization error handling.**
- **cmdline: Fix small memory leak.**

  A function in `cmdline.c` had a return that did not free the buf properly.

- **config: Enable same drivers options for Linux and BSD.**

  Enabled vector ixgbe and i40e bulk alloc for BSD as it is already done for Linux.

  Fixes: 304caba12643 ("config: fix bsd options") Fixes: 0ff3324da2eb ("ixgbe: rework vector pmd following mbuf changes")

- **devargs: Fix crash on failure.**

  This problem occurred when passing an invalid PCI id to the blacklist API in devargs.
- **e1000/i40e: Fix descriptor done flag with odd address.**
- **e1000/igb: fix ieee1588 timestamping initialization.**

  Fixed issue with e1000 ieee1588 timestamp initialization. On initialization the IEEE1588 functions read the system time to set their timestamp. However, on some 1G NICs, for example, i350, system time is disabled by default and the IEEE1588 timestamp was always 0.
- **eal/bsd: Fix inappropriate header guards.**
- **eal/bsd: Fix virtio on FreeBSD.**

  Closing the `/dev/io` fd caused a SIGBUS in inb/outb instructions as the process lost the IOPL privileges once the fd is closed.

  Fixes: 8a312224bcde ("eal/bsd: fix fd leak")
- **eal/linux: Fix comments on vfio MSI.**
- **eal/linux: Fix irq handling with igb_uio.**

  Fixed an issue where the the introduction of `uio_pci_generic` broke interrupt handling with igb_uio.

  Fixes: c112df6875a5 ("eal/linux: toggle interrupt for uio_pci_generic")
- **eal/linux: Fix numa node detection.**
- **eal/linux: Fix socket value for undetermined numa node.**

  Sets zero as the default value of pci device numa_node if the socket could not be determined. This provides the same default value as FreeBSD which has no NUMA support, and makes the return value of `rte_eth_dev_socket_id()` be consistent with the API description.
- **eal/ppc: Fix cpu cycle count for little endian.**

  On IBM POWER8 PPC64 little endian architecture, the definition of tsc union will be different. This fix enables the right output from `rte_rdtsc()`.
- **ethdev: Fix check of threshold for TX freeing.**

  Fixed issue where the parameter to `tx_free_thresh` was not consistent between the drivers.
- **ethdev: Fix crash if malloc of user callback fails.**

  If `rte_zmalloc()` failed in `rte_eth_dev_callback_register` then the NULL pointer would be dereferenced.
- **ethdev: Fix illegal port access.**

  To obtain a detachable flag, `pci_drv` is accessed in `rte_eth_dev_is_detachable()`. However `pci_drv` is only valid if port is enabled. Fixed by checking `rte_eth_dev_is_valid_port()` first.
- **ethdev: Make tables const.**
- **ethdev: Rename and extend the mirror type.**

- **examples/distributor: Fix debug macro.**

  The macro to turn on additional debug output when the app was compiled with `-DDEBUG` was broken.

  Fixes: 07db4a975094 ("examples/distributor: new sample app")
- **examples/kni: Fix crash on exit.**
- **examples/vhost: Fix build with debug enabled.**

  Fixes: 72ec8d77ac68 ("examples/vhost: rework duplicated code")
- **fm10k: Fix RETA table initialization.**

  The fm10k driver has 128 RETA entries in 32 registers, but it only initialized the first 32 when doing multiple RX queue configurations. This fix initializes all 128 entries.
- **fm10k: Fix RX buffer size.**
- **fm10k: Fix TX multi-segment frame.**
- **fm10k: Fix TX queue cleaning after start error.**
- **fm10k: Fix Tx queue cleaning after start error.**
- **fm10k: Fix default mac/vlan in switch.**
- **fm10k: Fix interrupt fault handling.**
- **fm10k: Fix jumbo frame issue.**
- **fm10k: Fix mac/vlan filtering.**
- **fm10k: Fix maximum VF number.**
- **fm10k: Fix maximum queue number for VF.**

  Both PF and VF shared code in function `fm10k_stats_get()`. The function worked with PF, but had problems with VF since it has less queues than PF.

  Fixes: a6061d9e7075 ("fm10k: register PF driver")
- **fm10k: Fix queue disabling.**
- **fm10k: Fix switch synchronization.**
- **i40e/base: Fix error handling of NVM state update.**
- **i40e/base: Fix hardware port number for pass-through.**
- **i40e/base: Rework virtual address retrieval for lan queue.**
- **i40e/base: Update LED blinking.**
- **i40e/base: Workaround for PHY type with firmware < 4.4.**
- **i40e: Disable setting of PHY configuration.**
- **i40e: Fix SCTP flow director.**
- **i40e: Fix check of descriptor done flag.**

  Fixes: 4861cde46116 ("i40e: new poll mode driver") Fixes: 05999aab4ca6 ("i40e: add or delete flow director")
- **i40e: Fix condition to get VMDQ info.**

- **i40e: Fix registers access from big endian CPU.**
- **i40evf: Clear command when error occurs.**
- **i40evf: Fix RSS with less RX queues than TX queues.**
- **i40evf: Fix crash when setup TX queues.**
- **i40evf: Fix jumbo frame support.**
- **i40evf: Fix offload capability flags.**

  Added checksum offload capability flags which have already been supported for a long time.
- **ivshmem: Fix crash in corner case.**

  Fixed issues where depending on the configured segments it was possible to hit a segmentation fault as a result of decrementing an unsigned index with value 0.

  Fixes: 40b966a211ab ("ivshmem: library changes for mmaping using ivshmem")
- **ixgbe/base: Fix SFP probing.**
- **ixgbe/base: Fix TX pending clearing.**
- **ixgbe/base: Fix X550 CS4227 address.**
- **ixgbe/base: Fix X550 PCIe master disabling.**
- **ixgbe/base: Fix X550 check.**
- **ixgbe/base: Fix X550 init early return.**
- **ixgbe/base: Fix X550 link speed.**
- **ixgbe/base: Fix X550em CS4227 speed mode.**
- **ixgbe/base: Fix X550em SFP+ link stability.**
- **ixgbe/base: Fix X550em UniPHY link configuration.**
- **ixgbe/base: Fix X550em flow control for KR backplane.**
- **ixgbe/base: Fix X550em flow control to be KR only.**
- **ixgbe/base: Fix X550em link setup without SFP.**
- **ixgbe/base: Fix X550em mux after MAC reset.**

  Fixes: d2e72774e58c ("ixgbe/base: support X550")
- **ixgbe/base: Fix bus type overwrite.**
- **ixgbe/base: Fix init handling of X550em link down.**
- **ixgbe/base: Fix lan id before first i2c access.**
- **ixgbe/base: Fix mac type checks.**
- **ixgbe/base: Fix tunneled UDP and TCP frames in flow director.**
- **ixgbe: Check mbuf refcnt when clearing a ring.**

  The function to clear the TX ring when a port was being closed, e.g. on exit in testpmd, was not checking the mbuf refcnt before freeing it. Since the function in the vector driver to clear the ring after TX does not setting the pointer to NULL post-free, this caused crashes if mbuf debugging was turned on.

- **ixgbe: Fix RX with buffer address not word aligned.**

  Niantic HW expects the Header Buffer Address in the RXD must be word aligned.

- **ixgbe: Fix RX with buffer address not word aligned.**
- **ixgbe: Fix Rx queue reset.**

  Fix to reset vector related RX queue fields to their initial values.

  Fixes: c95584dc2b18 ("ixgbe: new vectorized functions for Rx/Tx")

- **ixgbe: Fix TSO in IPv6.**

  When TSO was used with IPv6, the generated frames were incorrect. The L4 frame was OK, but the length field of IPv6 header was not populated correctly.

- **ixgbe: Fix X550 flow director check.**
- **ixgbe: Fix check for split packets.**

  The check for split packets to be reassembled in the vector ixgbe PMD was incorrectly only checking the first 16 elements of the array instead of all 32.

  Fixes: cf4b4708a88a ("ixgbe: improve slow-path perf with vector scattered Rx")

- **ixgbe: Fix data access on big endian cpu.**
- **ixgbe: Fix flow director flexbytes offset.**

  Fixes: d54a9888267c ("ixgbe: support flexpayload configuration of flow director")

- **ixgbe: Fix number of segments with vector scattered Rx.**

  Fixes: cf4b4708a88a (ixgbe: improve slow-path perf with vector scattered Rx)

- **ixgbe: Fix offload config option name.**

  The RX_OLFLAGS option was renamed from DISABLE to ENABLE in the driver code and Linux config. It is now renamed also in the BSD config and documentation.

  Fixes: 359f106a69a9 ("ixgbe: prefer enabling olflags rather than not disabling")

- **ixgbe: Fix release queue mbufs.**

  The calculations of what mbufs were valid in the RX and TX queues were incorrect when freeing the mbufs for the vector PMD. This led to crashes due to invalid reference counts when mbuf debugging was turned on, and possibly other more subtle problems (such as mbufs being freed when in use) in other cases.

  Fixes: c95584dc2b18 ("ixgbe: new vectorized functions for Rx/Tx")

- **ixgbe: Move PMD specific fields out of base driver.**

  Move `rx_bulk_alloc_allowed` and `rx_vec_allowed` from `ixgbe_hw` to `ixgbe_adapter`.

  Fixes: 01fa1d6215fa ("ixgbe: unify Rx setup")

- **ixgbe: Rename TX queue release function.**

- **ixgbevf: Fix RX function selection.**

  The logic to select ixgbe the VF RX function is different than the PF.

- **ixgbevf: Fix link status for PF up/down events.**
- **kni: Fix RX loop limit.**

  Loop processing packets dequeued from rx_q was using the number of packets requested, not how many it actually received.

- **kni: Fix ioctl in containers, like Docker.**
- **kni: Fix multicast ioctl handling.**
- **log: Fix crash after log_history dump.**
- **lpm: Fix big endian support.**
- **lpm: Fix depth small entry add.**
- **mbuf: Fix cloning with private mbuf data.**

  Added a new `priv_size` field in mbuf structure that should be initialized at mbuf pool creation. This field contains the size of the application private data in mbufs.

  Introduced new static inline functions `rte_mbuf_from_indirect()` and `rte_mbuf_to_baddr()` to replace the existing macros, which take the private size into account when attaching and detaching mbufs.

- **mbuf: Fix data room size calculation in pool init.**

  Deduct the mbuf data room size from `mempool->elt_size` and `priv_size`, instead of using an hardcoded value that is not related to the real buffer size.

  To use `rte_pktmbuf_pool_init()`, the user can either:

  – Give a NULL parameter to rte_pktmbuf_pool_init(): in this case, the private size is assumed to be 0, and the room size is `mp->elt_size - sizeof(struct rte_mbuf)`.

  – Give the `rte_pktmbuf_pool_private` filled with appropriate data_room_size and priv_size values.

- **mbuf: Fix init when private size is not zero.**

  Allow the user to use the default `rte_pktmbuf_init()` function even if the mbuf private size is not 0.

- **mempool: Add structure for object headers.**

  Each object stored in mempools are prefixed by a header, allowing for instance to retrieve the mempool pointer from the object. When debug is enabled, a cookie is also added in this header that helps to detect corruptions and double-frees.

  Introduced a structure that materializes the content of this header, and will simplify future patches adding things in this header.

- **mempool: Fix pages computation to determine number of objects.**
- **mempool: Fix returned value after counting objects.**

  Fixes: 148f963fb532 (“xen: core library changes”)

- **mlx4: Avoid requesting TX completion events to improve performance.**

  Instead of requesting a completion event for each TX burst, request it on a fixed schedule once every MLX4_PMD_TX_PER_COMP_REQ (currently 64) packets to improve performance.

- **mlx4: Fix compilation as a shared library and on 32 bit platforms.**

- **mlx4: Fix possible crash on scattered mbuf allocation failure.**

  Fixes issue where failing to allocate a segment, `mlx4_rx_burst_sp()` could call `rte_pktmbuf_free()` on an incomplete scattered mbuf whose next pointer in the last segment is not set.

- **mlx4: Fix support for multiple vlan filters.**

  This fixes the "Multiple RX VLAN filters can be configured, but only the first one works" bug.

- **pcap: Fix storage of name and type in queues.**

  pcap_rx_queue/pcap_tx_queue should store it's own copy of name/type values, not the pointer to temporary allocated space.

- **pci: Fix memory leaks and needless increment of map address.**

- **pci: Fix uio mapping differences between linux and bsd.**

- **port: Fix unaligned access to metadata.**

  Fix RTE_MBUF_METADATA macros to allow for unaligned accesses to meta-data fields.

- **ring: Fix return of new port id on creation.**

- **timer: Fix race condition.**

  Eliminate problematic race condition in `rte_timer_manage()` that can lead to corruption of per-lcore pending-lists (implemented as skip-lists).

- **vfio: Fix overflow of BAR region offset and size.**

  Fixes: 90a1633b2347 ("eal/Linux: allow to map BARs with MSI-X tables")

- **vhost: Fix enqueue/dequeue to handle chained vring descriptors.**

- **vhost: Fix race for connection fd.**

- **vhost: Fix virtio freeze due to missed interrupt.**

- **virtio: Fix crash if CQ is not negotiated.**

  Fix NULL dereference if virtio control queue is not negotiated.

- **virtio: Fix ring size negotiation.**

  Negotiate the virtio ring size. The host may allow for very large rings but application may only want a smaller ring. Conversely, if the number of descriptors requested exceeds the virtio host queue size, then just silently use the smaller host size.

  This fixes issues with virtio in non-QEMU environments. For example Google Compute Engine allows up to 16K elements in ring.

- **vmxnet3: Fix link state handling.**

10.3.3 Known Issues

- When running the `vmdq` sample or `vhost` sample applications with the Intel(R) XL710 (i40e) NIC, the configuration option `CONFIG_RTE_MAX_QUEUES_PER_PORT` should be increased from 256 to 1024.
- VM power manager may not work on systems with more than 64 cores.

10.3.4 API Changes

- The order that user supplied RX and TX callbacks are called in has been changed to the order that they were added (fifo) in line with end-user expectations. The previous calling order was the reverse of this (lifo) and was counter intuitive for users. The actual API is unchanged.

10.3.5 ABI Changes

- The `rte_hash` structure has been changed to internal use only.

10.4 DPDK Release 2.0

10.4.1 New Features

- Poll-mode driver support for an early release of the PCIE host interface of the Intel(R) Ethernet Switch FM10000.
  - Basic Rx/Tx functions for PF/VF
  - Interrupt handling support for PF/VF
  - Per queue start/stop functions for PF/VF
  - Support Mailbox handling between PF/VF and PF/Switch Manager
  - Receive Side Scaling (RSS) for PF/VF
  - Scatter receive function for PF/VF
  - Reta update/query for PF/VF
  - VLAN filter set for PF
  - Link status query for PF/VF

Note: The software is intended to run on pre-release hardware and may contain unknown or unresolved defects or issues related to functionality and performance. The poll mode driver is also pre-release and will be updated to a released version post hardware and base driver release. Should the official hardware release be made between DPDK releases an updated poll-mode driver will be made available.

- Link Bonding
  - Support for adaptive load balancing (mode 6) to the link bonding library.

  - Support for registration of link status change callbacks with link bonding devices.
  - Support for slaves devices which do not support link status change interrupts in the link bonding library via a link status polling mechanism.
- PCI Hotplug with NULL PMD sample application
- ABI versioning
- x32 ABI
- Non-EAL Thread Support
- Multi-pthread Support
- Re-order Library
- ACL for AVX2
- Architecture Independent CRC Hash
- uio_pci_generic Support
- KNI Optimizations
- Vhost-user support
- Virtio (link, vlan, mac, port IO, perf)
- IXGBE-VF RSS
- RX/TX Callbacks
- Unified Flow Types
- Indirect Attached MBUF Flag
- Use default port configuration in TestPMD
- Tunnel offloading in TestPMD
- Poll Mode Driver - 40 GbE Controllers (librte_pmd_i40e)
  - Support for Flow Director
  - Support for ethertype filter
  - Support RSS in VF
  - Support configuring redirection table with different size from 1GbE and 10 GbE
  - 128/512 entries of 40GbE PF
  - 64 entries of 40GbE VF
  - Support configuring hash functions
  - Support for VXLAN packet on Intel® 40GbE Controllers
- Poll Mode Driver for Mellanox ConnectX-3 EN adapters (mlx4)

Note: This PMD is only available for Linux and is disabled by default due to external dependencies (libibverbs and libmlx4). Please refer to the NIC drivers guide for more information.

- Packet Distributor Sample Application

- Job Stats library and Sample Application.
- Enhanced Jenkins hash (jhash) library

Note: The hash values returned by the new jhash library are different from the ones returned by the previous library.

10.5 DPDK Release 1.8

10.5.1 New Features

- Link Bonding
  - Support for 802.3ad link aggregation (mode 4) and transmit load balancing (mode 5) to the link bonding library.
  - Support for registration of link status change callbacks with link bonding devices.
  - Support for slaves devices which do not support link status change interrupts in the link bonding library via a link status polling mechanism.
- Poll Mode Driver - 40 GbE Controllers (librte_pmd_i40e)
  - Support for Flow Director
  - Support for ethertype filter
  - Support RSS in VF
  - Support configuring redirection table with different size from 1GbE and 10 GbE
  - 128/512 entries of 40GbE PF
  - 64 entries of 40GbE VF
  - Support configuring hash functions
  - Support for VXLAN packet on Intel 40GbE Controllers
- Packet Distributor Sample Application

10.6 Supported Operating Systems

The following Linux distributions were successfully used to compiler or run DPDK.

- FreeBSD 10
- Fedora release 20
- Ubuntu 14.04 LTS
- Wind River Linux 6
- Red Hat Enterprise Linux 6.5
- SUSE Enterprise Linux 11 SP3

These distributions may need additional packages that are not installed by default, or a specific kernel. Refer to the *Linux guide* and *FreeBSD guide* for details.

10.7 Known Issues and Limitations in Legacy Releases

This section describes known issues with the DPDK software that aren't covered in the version specific release notes sections.

10.7.1 Unit Test for Link Bonding may fail at test_tlb_tx_burst()

Description: Unit tests will fail in `test_tlb_tx_burst()` function with error for uneven distribution of packets.

Implication: Unit test link_bonding_autotest will fail.

Resolution/Workaround: There is no workaround available.

Affected Environment/Platform: Fedora 20.

Driver/Module: Link Bonding.

10.7.2 Pause Frame Forwarding does not work properly on igb

Description: For igb devices rte_eth_flow_ctrl_set does not work as expected. Pause frames are always forwarded on igb, regardless of the `RFCE`, `MPMCF` and `DPF` registers.

Implication: Pause frames will never be rejected by the host on 1G NICs and they will always be forwarded.

Resolution/Workaround: There is no workaround available.

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

10.7.3 In packets provided by the PMD, some flags are missing

Description: In packets provided by the PMD, some flags are missing. The application does not have access to information provided by the hardware (packet is broadcast, packet is multicast, packet is IPv4 and so on).

Implication: The `ol_flags` field in the `rte_mbuf` structure is not correct and should not be used.

Resolution/Workaround: The application has to parse the Ethernet header itself to get the information, which is slower.

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

10.7.4 The rte_malloc library is not fully implemented

Description: The `rte_malloc` library is not fully implemented.

Implication: All debugging features of rte_malloc library described in architecture documentation are not yet implemented.

Resolution/Workaround: No workaround available.

Affected Environment/Platform: All.

Driver/Module: `rte_malloc`.

10.7.5 HPET reading is slow

Description: Reading the HPET chip is slow.

Implication: An application that calls `rte_get_hpet_cycles()` or `rte_timer_manage()` runs slower.

Resolution/Workaround: The application should not call these functions too often in the main loop. An alternative is to use the TSC register through `rte_rdtsc()` which is faster, but specific to an lcore and is a cycle reference, not a time reference.

Affected Environment/Platform: All.

Driver/Module: Environment Abstraction Layer (EAL).

10.7.6 HPET timers do not work on the Osage customer reference platform

Description: HPET timers do not work on the Osage customer reference platform which includes an Intel® Xeon® processor 5500 series processor) using the released BIOS from Intel.

Implication: On Osage boards, the implementation of the `rte_delay_us()` function must be changed to not use the HPET timer.

Resolution/Workaround: This can be addressed by building the system with the `CONFIG_RTE_LIBEAL_USE_HPET=n` configuration option or by using the `--no-hpet` EAL option.

Affected Environment/Platform: The Osage customer reference platform. Other vendor platforms with Intel® Xeon® processor 5500 series processors should work correctly, provided the BIOS supports HPET.

Driver/Module: `lib/librte_eal/common/include/rte_cycles.h`

10.7.7 Not all variants of supported NIC types have been used in testing

Description: The supported network interface cards can come in a number of variants with different device ID's. Not all of these variants have been tested with the Intel® DPDK.

The NIC device identifiers used during testing:

- Intel® Ethernet Controller XL710 for 40GbE QSFP+ [8086:1584]
- Intel® Ethernet Controller XL710 for 40GbE QSFP+ [8086:1583]

- Intel® Ethernet Controller X710 for 10GbE SFP+ [8086:1572]
- Intel® 82576 Gigabit Ethernet Controller [8086:10c9]
- Intel® 82576 Quad Copper Gigabit Ethernet Controller [8086:10e8]
- Intel® 82580 Dual Copper Gigabit Ethernet Controller [8086:150e]
- Intel® I350 Quad Copper Gigabit Ethernet Controller [8086:1521]
- Intel® 82599 Dual Fibre 10 Gigabit Ethernet Controller [8086:10fb]
- Intel® Ethernet Server Adapter X520-T2 [8086: 151c]
- Intel® Ethernet Controller X540-T2 [8086:1528]
- Intel® 82574L Gigabit Network Connection [8086:10d3]
- Emulated Intel® 82540EM Gigabit Ethernet Controller [8086:100e]
- Emulated Intel® 82545EM Gigabit Ethernet Controller [8086:100f]
- Intel® Ethernet Server Adapter X520-4 [8086:154a]
- Intel® Ethernet Controller I210 [8086:1533]

Implication: Risk of issues with untested variants.

Resolution/Workaround: Use tested NIC variants. For those supported Ethernet controllers, additional device IDs may be added to the software if required.

Affected Environment/Platform: All.

Driver/Module: Poll-mode drivers

10.7.8 Multi-process sample app requires exact memory mapping

Description: The multi-process example application assumes that it is possible to map the hugepage memory to the same virtual addresses in client and server applications. Occasionally, very rarely with 64-bit, this does not occur and a client application will fail on startup. The Linux “address-space layout randomization” security feature can sometimes cause this to occur.

Implication: A multi-process client application fails to initialize.

Resolution/Workaround: See the “Multi-process Limitations” section in the Intel® DPDK Programmer’s Guide for more information.

Affected Environment/Platform: All.

Driver/Module: Multi-process example application

10.7.9 Packets are not sent by the 1 GbE/10 GbE SR-IOV driver when the source MAC is not the MAC assigned to the VF NIC

Description: The 1 GbE/10 GbE SR-IOV driver can only send packets when the Ethernet header’s source MAC address is the same as that of the VF NIC. The reason for this is that the Linux `ixgbe` driver module in the host OS has its anti-spoofing feature enabled.

Implication: Packets sent using the 1 GbE/10 GbE SR-IOV driver must have the source MAC address correctly set to that of the VF NIC. Packets with other source address values are dropped by the NIC if the application attempts to transmit them.

Resolution/Workaround: Configure the Ethernet source address in each packet to match that of the VF NIC.

Affected Environment/Platform: All.

Driver/Module: 1 GbE/10 GbE VF Poll Mode Driver (PMD).

10.7.10 SR-IOV drivers do not fully implement the rte_ethdev API

Description: The SR-IOV drivers only supports the following rte_ethdev API functions:

- rte_eth_dev_configure()
- rte_eth_tx_queue_setup()
- rte_eth_rx_queue_setup()
- rte_eth_dev_info_get()
- rte_eth_dev_start()
- rte_eth_tx_burst()
- rte_eth_rx_burst()
- rte_eth_dev_stop()
- rte_eth_stats_get()
- rte_eth_stats_reset()
- rte_eth_link_get()
- rte_eth_link_get_no_wait()

Implication: Calling an unsupported function will result in an application error.

Resolution/Workaround: Do not use other rte_ethdev API functions in applications that use the SR-IOV drivers.

Affected Environment/Platform: All.

Driver/Module: VF Poll Mode Driver (PMD).

10.7.11 PMD does not work with –no-huge EAL command line parameter

Description: Currently, the DPDK does not store any information about memory allocated by `malloc()` (for example, NUMA node, physical address), hence PMD drivers do not work when the ``--no-huge` command line parameter is supplied to EAL.

Implication: Sending and receiving data with PMD will not work.

Resolution/Workaround: Use huge page memory or use VFIO to map devices.

Affected Environment/Platform: Systems running the DPDK on Linux

Driver/Module: Poll Mode Driver (PMD).

10.7.12 Some hardware off-load functions are not supported by the VF Driver

Description: Currently, configuration of the following items is not supported by the VF driver:

- IP/UDP/TCP checksum offload
- Jumbo Frame Receipt
- HW Strip CRC

Implication: Any configuration for these items in the VF register will be ignored. The behavior is dependent on the current PF setting.

Resolution/Workaround: For the PF (Physical Function) status on which the VF driver depends, there is an option item under PMD in the config file. For others, the VF will keep the same behavior as PF setting.

Affected Environment/Platform: All.

Driver/Module: VF (SR-IOV) Poll Mode Driver (PMD).

10.7.13 Kernel crash on IGB port unbinding

Description: Kernel crash may occur when unbinding 1G ports from the igb_uio driver, on 2.6.3x kernels such as shipped with Fedora 14.

Implication: Kernel crash occurs.

Resolution/Workaround: Use newer kernels or do not unbind ports.

Affected Environment/Platform: 2.6.3x kernels such as shipped with Fedora 14

Driver/Module: IGB Poll Mode Driver (PMD).

10.7.14 Twinpond and Ironpond NICs do not report link status correctly

Description: Twin Pond/Iron Pond NICs do not bring the physical link down when shutting down the port.

Implication: The link is reported as up even after issuing `shutdown` command unless the cable is physically disconnected.

Resolution/Workaround: None.

Affected Environment/Platform: Twin Pond and Iron Pond NICs

Driver/Module: Poll Mode Driver (PMD).

10.7.15 Discrepancies between statistics reported by different NICs

Description: Gigabit Ethernet devices from Intel include CRC bytes when calculating packet reception statistics regardless of hardware CRC stripping state, while 10-Gigabit Ethernet devices from Intel do so only when hardware CRC stripping is disabled.

Implication: There may be a discrepancy in how different NICs display packet reception statistics.

Resolution/Workaround: None

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

10.7.16 Error reported opening files on DPDK initialization

Description: On DPDK application startup, errors may be reported when opening files as part of the initialization process. This occurs if a large number, for example, 500 or more, or if hugepages are used, due to the per-process limit on the number of open files.

Implication: The DPDK application may fail to run.

Resolution/Workaround: If using 2 MB hugepages, consider switching to a fewer number of 1 GB pages. Alternatively, use the `ulimit` command to increase the number of files which can be opened by a process.

Affected Environment/Platform: All.

Driver/Module: Environment Abstraction Layer (EAL).

10.7.17 Intel® QuickAssist Technology sample application does not work on a 32-bit OS on Shumway

Description: The Intel® Communications Chipset 89xx Series device does not fully support NUMA on a 32-bit OS. Consequently, the sample application cannot work properly on Shumway, since it requires NUMA on both nodes.

Implication: The sample application cannot work in 32-bit mode with emulated NUMA, on multi-socket boards.

Resolution/Workaround: There is no workaround available.

Affected Environment/Platform: Shumway

Driver/Module: All.

10.7.18 Differences in how different Intel NICs handle maximum packet length for jumbo frame

Description: 10 Gigabit Ethernet devices from Intel do not take VLAN tags into account when calculating packet size while Gigabit Ethernet devices do so for jumbo frames.

Implication: When receiving packets with VLAN tags, the actual maximum size of useful payload that Intel Gigabit Ethernet devices are able to receive is 4 bytes (or 8 bytes in the case of packets with extended VLAN tags) less than that of Intel 10 Gigabit Ethernet devices.

Resolution/Workaround: Increase the configured maximum packet size when using Intel Gigabit Ethernet devices.

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

10.7.19 Binding PCI devices to igb_uio fails on Linux kernel 3.9 when more than one device is used

Description: A known bug in the uio driver included in Linux kernel version 3.9 prevents more than one PCI device to be bound to the igb_uio driver.

Implication: The Poll Mode Driver (PMD) will crash on initialization.

Resolution/Workaround: Use earlier or later kernel versions, or apply the following patch.

Affected Environment/Platform: Linux systems with kernel version 3.9

Driver/Module: igb_uio module

10.7.20 GCC might generate Intel® AVX instructions for processors without Intel® AVX support

Description: When compiling Intel® DPDK (and any DPDK app), gcc may generate Intel® AVX instructions, even when the processor does not support Intel® AVX.

Implication: Any DPDK app might crash while starting up.

Resolution/Workaround: Either compile using icc or set `EXTRA_CFLAGS='-O3'` prior to compilation.

Affected Environment/Platform: Platforms which processor does not support Intel® AVX.

Driver/Module: Environment Abstraction Layer (EAL).

10.7.21 Ethertype filter could receive other packets (non-assigned) in Niantic

Description: On Intel® Ethernet Controller 82599EB When Ethertype filter (priority enable) was set, unmatched packets also could be received on the assigned queue, such as ARP packets without 802.1q tags or with the user priority not equal to set value. Launch the testpmd by disabling RSS and with multiply queues, then add the ethertype filter like the following and then start forwarding:

```
add_ethertype_filter 0 ethertype 0x0806 priority enable 3 queue 2 index 1
```

When sending ARP packets without 802.1q tag and with user priority as non-3 by tester, all the ARP packets can be received on the assigned queue.

Implication: The user priority comparing in Ethertype filter cannot work probably. It is a NIC's issue due to the following: "In fact, ETQF.UP is not functional, and the information will be added in errata of 82599 and X540."

Resolution/Workaround: None

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

10.7.22 Cannot set link speed on Intel® 40G Ethernet controller

Description: On Intel® 40G Ethernet Controller you cannot set the link to specific speed.

Implication: The link speed cannot be changed forcibly, though it can be configured by application.

Resolution/Workaround: None

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

10.7.23 Stopping the port does not down the link on Intel® 40G Ethernet controller

Description: On Intel® 40G Ethernet Controller stopping the port does not really down the port link.

Implication: The port link will be still up after stopping the port.

Resolution/Workaround: None

Affected Environment/Platform: All.

Driver/Module: Poll Mode Driver (PMD).

10.7.24 Devices bound to igb_uio with VT-d enabled do not work on Linux kernel 3.15-3.17

Description: When VT-d is enabled (`iommu=pt intel_iommu=on`), devices are 1:1 mapped. In the Linux kernel unbinding devices from drivers removes that mapping which result in IOMMU errors. Introduced in Linux kernel 3.15 commit, solved in Linux kernel 3.18 commit.

Implication: Devices will not be allowed to access memory, resulting in following kernel errors:

```
dmar: DRHD: handling fault status reg 2
dmar: DMAR:[DMA Read] Request device [02:00.0] fault addr a0c58000
DMAR:[fault reason 02] Present bit in context entry is clear
```

Resolution/Workaround: Use earlier or later kernel versions, or avoid driver binding on boot by blacklisting the driver modules. I.e., in the case of `ixgbe`, we can pass the kernel command line option: `modprobe.blacklist=ixgbe`. This way we do not need to unbind the device to bind it to igb_uio.

Affected Environment/Platform: Linux systems with kernel versions 3.15 to 3.17.

Driver/Module: `igb_uio` module.

10.7.25 VM power manager may not work on systems with more than 64 cores

Description: When using VM power manager on a system with more than 64 cores, VM(s) should not use cores 64 or higher.

Implication: VM power manager should not be used with VM(s) that are using cores 64 or above.

Resolution/Workaround: Do not use cores 64 or above.

Affected Environment/Platform: Platforms with more than 64 cores.

Driver/Module: VM power manager application.

10.8 ABI and API Deprecation

See the guidelines document for details of the ABI policy. API and ABI deprecation notices are to be posted here.

10.8.1 Deprecation Notices

- The following fields have been deprecated in rte_eth_stats: ibadcrc, ibadlen, imcasts, fdirmatch, fdirmiss, tx_pause_xon, rx_pause_xon, tx_pause_xoff, rx_pause_xoff
- The ethdev structures rte_eth_link, rte_eth_dev_info and rte_eth_conf must be updated to support 100G link and to have a cleaner link speed API.
- ABI changes is planned for the reta field in struct rte_eth_rss_reta_entry64 which handles at most 256 queues (8 bits) while newer NICs support larger tables (512 queues). It should be integrated in release 2.3.
- ABI changes are planned for struct rte_eth_fdir_flow in order to support extend flow director's input set. The release 2.2 does not contain these ABI changes, but release 2.3 will, and no backwards compatibility is planned.
- ABI changes are planned for rte_eth_ipv4_flow and rte_eth_ipv6_flow to include more fields to be matched against. The release 2.2 does not contain these ABI changes, but release 2.3 will.
- ABI changes are planned for adding four new flow types. This impacts RTE_ETH_FLOW_MAX. The release 2.2 does not contain these ABI changes, but release 2.3 will.
- ABI changes are planned for rte_eth_tunnel_filter_conf. Change the fields of outer_mac and inner_mac from pointer to struct in order to keep the code's readability. The release 2.2 does not contain these ABI changes, but release 2.3 will, and no backwards compatibility is planned.
- The scheduler statistics structure will change to allow keeping track of RED actions.
- librte_pipeline: The prototype for the pipeline input port, output port and table action handlers will be updated: the pipeline parameter will be added, the packets mask parameter will be either removed (for input port action handler) or made input-only.
- ABI changes are planned in cmdline buffer size to allow the use of long commands (such as RETA update in testpmd). This should impact CMDLINE_PARSE_RESULT_BUFSIZE, STR_TOKEN_SIZE and RDLINE_BUF_SIZE. It should be integrated in release 2.3.

CHAPTER 11

Contributor's Guidelines

11.1 DPDK Coding Style

11.1.1 Description

This document specifies the preferred style for source files in the DPDK source tree. It is based on the Linux Kernel coding guidelines and the FreeBSD 7.2 Kernel Developer's Manual (see man style(9)), but was heavily modified for the needs of the DPDK.

11.1.2 General Guidelines

The rules and guidelines given in this document cannot cover every situation, so the following general guidelines should be used as a fallback:

- The code style should be consistent within each individual file.
- In the case of creating new files, the style should be consistent within each file in a given directory or module.
- The primary reason for coding standards is to increase code readability and comprehensibility, therefore always use whatever option will make the code easiest to read.

Line length is recommended to be not more than 80 characters, including comments. [Tab stop size should be assumed to be 8-characters wide].

Note: The above is recommendation, and not a hard limit. However, it is expected that the recommendations should be followed in all but the rarest situations.

11.1.3 C Comment Style

Usual Comments

These comments should be used in normal cases. To document a public API, a doxygen-like format must be used: refer to *Doxygen Guidelines*.

```
/*
 * VERY important single-line comments look like this.
 */
```

```
/* Most single-line comments look like this. */

/*
 * Multi-line comments look like this.  Make them real sentences. Fill
 * them so they look like real paragraphs.
 */
```

License Header

Each file should begin with a special comment containing the appropriate copyright and license for the file. Generally this is the BSD License, except for code for Linux Kernel modules. After any copyright header, a blank line should be left before any other contents, e.g. include statements in a C file.

11.1.4 C Preprocessor Directives

Header Includes

In DPDK sources, the include files should be ordered as following:

1. libc includes (system includes first)
2. DPDK EAL includes
3. DPDK misc libraries includes
4. application-specific includes

Include files from the local application directory are included using quotes, while includes from other paths are included using angle brackets: “<>”.

Example:

```
#include <stdio.h>
#include <stdlib.h>

#include <rte_eal.h>

#include <rte_ring.h>
#include <rte_mempool.h>

#include "application.h"
```

Header File Guards

Headers should be protected against multiple inclusion with the usual:

```
#ifndef _FILE_H_
#define _FILE_H_

/* Code */

#endif /* _FILE_H_ */
```

Macros

Do not `#define` or declare names except with the standard DPDK prefix: `RTE_`. This is to ensure there are no collisions with definitions in the application itself.

The names of "unsafe" macros (ones that have side effects), and the names of macros for manifest constants, are all in uppercase.

The expansions of expression-like macros are either a single token or have outer parentheses. If a macro is an inline expansion of a function, the function name is all in lowercase and the macro has the same name all in uppercase. If the macro encapsulates a compound statement, enclose it in a do-while loop, so that it can be used safely in if statements. Any final statement-terminating semicolon should be supplied by the macro invocation rather than the macro, to make parsing easier for pretty-printers and editors.

For example:

```
#define MACRO(x, y) do {                                        \
        variable = (x) + (y);                                   \
        (y) += 2;                                               \
} while(0)
```

Note: Wherever possible, enums and inline functions should be preferred to macros, since they provide additional degrees of type-safety and can allow compilers to emit extra warnings about unsafe code.

Conditional Compilation

- When code is conditionally compiled using `#ifdef` or `#if`, a comment may be added following the matching `#endif` or `#else` to permit the reader to easily discern where conditionally compiled code regions end.
- This comment should be used only for (subjectively) long regions, regions greater than 20 lines, or where a series of nested `#ifdef`'s may be confusing to the reader. Exceptions may be made for cases where code is conditionally not compiled for the purposes of lint(1), or other tools, even though the uncompiled region may be small.
- The comment should be separated from the `#endif` or `#else` by a single space.
- For short conditionally compiled regions, a closing comment should not be used.
- The comment for `#endif` should match the expression used in the corresponding `#if` or `#ifdef`.
- The comment for `#else` and `#elif` should match the inverse of the expression(s) used in the preceding `#if` and/or `#elif` statements.
- In the comments, the subexpression `defined(FOO)` is abbreviated as "FOO". For the purposes of comments, `#ifndef FOO` is treated as `#if !defined(FOO)`.

```
#ifdef KTRACE
#include <sys/ktrace.h>
#endif

#ifdef COMPAT_43
/* A large region here, or other conditional code. */
#else /* !COMPAT_43 */
```

```
/* Or here. */
#endif /* COMPAT_43 */

#ifndef COMPAT_43
/* Yet another large region here, or other conditional code. */
#else /* COMPAT_43 */
/* Or here. */
#endif /* !COMPAT_43 */
```

Note: Conditional compilation should be used only when absolutely necessary, as it increases the number of target binaries that need to be built and tested.

11.1.5 C Types

Integers

For fixed/minimum-size integer values, the project uses the form uintXX_t (from stdint.h) instead of older BSD-style integer identifiers of the form u_intXX_t.

Enumerations

- Enumeration values are all uppercase.

```
enum enumtype { ONE, TWO } et;
```

- Enum types should be used in preference to macros #defining a set of (sequential) values.
- Enum types should be prefixed with `rte_` and the elements by a suitable prefix [generally starting `RTE_<enum>_` - where <enum> is a shortname for the enum type] to avoid namespace collisions.

Bitfields

The developer should group bitfields that are included in the same integer, as follows:

```
struct grehdr {
  uint16_t rec:3,
      srr:1,
      seq:1,
      key:1,
      routing:1,
      csum:1,
      version:3,
      reserved:4,
      ack:1;
/* ... */
}
```

Variable Declarations

In declarations, do not put any whitespace between asterisks and adjacent tokens, except for tokens that are identifiers related to types. (These identifiers are the names of basic types, type qualifiers, and typedef-names other than the one being declared.) Separate these identifiers from asterisks using a single space.

For example:

```
int *x;         /* no space after asterisk */
int * const x;  /* space after asterisk when using a type qualifier */
```

- All externally-visible variables should have an `rte_` prefix in the name to avoid namespace collisions.
- Do not use uppercase letters - either in the form of ALL_UPPERCASE, or CamelCase - in variable names. Lower-case letters and underscores only.

Structure Declarations

- In general, when declaring variables in new structures, declare them sorted by use, then by size (largest to smallest), and then in alphabetical order. Sorting by use means that commonly used variables are used together and that the structure layout makes logical sense. Ordering by size then ensures that as little padding is added to the structure as possible.
- For existing structures, additions to structures should be added to the end so for backward compatibility reasons.
- Each structure element gets its own line.
- Try to make the structure readable by aligning the member names using spaces as shown below.
- Names following extremely long types, which therefore cannot be easily aligned with the rest, should be separated by a single space.

```
struct foo {
        struct foo      *next;          /* List of active foo. */
        struct mumble   amumble;        /* Comment for mumble. */
        int             bar;            /* Try to align the comments. */
        struct verylongtypename *baz;   /* Won't fit with other members */
};
```

- Major structures should be declared at the top of the file in which they are used, or in separate header files if they are used in multiple source files.
- Use of the structures should be by separate variable declarations and those declarations must be extern if they are declared in a header file.
- Externally visible structure definitions should have the structure name prefixed by `rte_` to avoid namespace collisions.

Queues

Use queue(3) macros rather than rolling your own lists, whenever possible. Thus, the previous example would be better written:

```
#include <sys/queue.h>

struct foo {
        LIST_ENTRY(foo) link;      /* Use queue macros for foo lists. */
        struct mumble   amumble;   /* Comment for mumble. */
        int             bar;        /* Try to align the comments. */
        struct verylongtypename *baz;    /* Won't fit with other members */
};
LIST_HEAD(, foo) foohead;           /* Head of global foo list. */
```

DPDK also provides an optimized way to store elements in lockless rings. This should be used in all data-path code, when there are several consumer and/or producers to avoid locking for concurrent access.

Typedefs

Avoid using typedefs for structure types.

For example, use:

```
struct my_struct_type {
/* ... */
};

struct my_struct_type my_var;
```

rather than:

```
typedef struct my_struct_type {
/* ... */
} my_struct_type;

my_struct_type my_var
```

Typedefs are problematic because they do not properly hide their underlying type; for example, you need to know if the typedef is the structure itself, as shown above, or a pointer to the structure. In addition, they must be declared exactly once, whereas an incomplete structure type can be mentioned as many times as necessary. Typedefs are difficult to use in stand-alone header files. The header that defines the typedef must be included before the header that uses it, or by the header that uses it (which causes namespace pollution), or there must be a back-door mechanism for obtaining the typedef.

Note that #defines used instead of typedefs also are problematic (since they do not propagate the pointer type correctly due to direct text replacement). For example, `#define pint int *` does not work as expected, while `typedef int *pint` does work. As stated when discussing macros, typedefs should be preferred to macros in cases like this.

When convention requires a typedef; make its name match the struct tag. Avoid typedefs ending in `_t`, except as specified in Standard C or by POSIX.

Note: It is recommended to use typedefs to define function pointer types, for reasons of code readability. This is especially true when the function type is used as a parameter to another function.

For example:

```
/**
 * Definition of a remote launch function.
 */
typedef int (lcore_function_t)(void *);

/* launch a function of lcore_function_t type */
int rte_eal_remote_launch(lcore_function_t *f, void *arg, unsigned slave_id);
```

11.1.6 C Indentation

General

- Indentation is a hard tab, that is, a tab character, not a sequence of spaces,

Note: Global whitespace rule in DPDK, use tabs for indentation, spaces for alignment.

- Do not put any spaces before a tab for indentation.
- If you have to wrap a long statement, put the operator at the end of the line, and indent again.
- For control statements (if, while, etc.), continuation it is recommended that the next line be indented by two tabs, rather than one, to prevent confusion as to whether the second line of the control statement forms part of the statement body or not. Alternatively, the line continuation may use additional spaces to line up to an appropriately point on the preceding line, for example, to align to an opening brace.

Note: As with all style guidelines, code should match style already in use in an existing file.

```
while (really_long_variable_name_1 == really_long_variable_name_2 &&
    var3 == var4){  /* confusing to read as */
    x = y + z;      /* control stmt body lines up with second line of */
    a = b + c;      /* control statement itself if single indent used */
}

if (really_long_variable_name_1 == really_long_variable_name_2 &&
        var3 == var4){  /* two tabs used */
    x = y + z;          /* statement body no longer lines up */
    a = b + c;
}

z = a + really + long + statement + that + needs +
        two + lines + gets + indented + on + the +
        second + and + subsequent + lines;
```

- Do not add whitespace at the end of a line.
- Do not add whitespace or a blank line at the end of a file.

Control Statements and Loops

- Include a space after keywords (if, while, for, return, switch).
- Do not use braces (`{` and `}`) for control statements with zero or just a single statement, unless that statement is more than a single line in which case the braces are permitted.

```
for (p = buf; *p != '\0'; ++p)
        ;       /* nothing */
for (;;)
        stmt;
for (;;) {
        z = a + really + long + statement + that + needs +
                two + lines + gets + indented + on + the +
                second + and + subsequent + lines;
}
for (;;) {
        if (cond)
                stmt;
}
if (val != NULL)
        val = realloc(val, newsize);
```

- Parts of a for loop may be left empty.

```
for (; cnt < 15; cnt++) {
        stmt1;
        stmt2;
}
```

- Closing and opening braces go on the same line as the else keyword.
- Braces that are not necessary should be left out.

```
if (test)
        stmt;
else if (bar) {
        stmt;
        stmt;
} else
        stmt;
```

Function Calls

- Do not use spaces after function names.
- Commas should have a space after them.
- No spaces after `(` or `[` or preceding the `]` or `)` characters.

```
error = function(a1, a2);
if (error != 0)
        exit(error);
```

Operators

- Unary operators do not require spaces, binary operators do.
- Do not use parentheses unless they are required for precedence or unless the statement is confusing without them. However, remember that other people may be more easily confused than you.

Exit

Exits should be 0 on success, or 1 on failure.

```
        exit(0);        /*
                         * Avoid obvious comments such as
                         * "Exit 0 on success."
                         */
}
```

Local Variables

- Variables should be declared at the start of a block of code rather than in the middle. The exception to this is when the variable is `const` in which case the declaration must be at the point of first use/assignment.
- When declaring variables in functions, multiple variables per line are OK. However, if multiple declarations would cause the line to exceed a reasonable line length, begin a new set of declarations on the next line rather than using a line continuation.
- Be careful to not obfuscate the code by initializing variables in the declarations, only the last variable on a line should be initialized. If multiple variables are to be initialized when defined, put one per line.
- Do not use function calls in initializers, except for `const` variables.

```
int i = 0, j = 0, k = 0;  /* bad, too many initializer */

char a = 0;        /* OK, one variable per line with initializer */
char b = 0;

float x, y = 0.0;  /* OK, only last variable has initializer */
```

Casts and sizeof

- Casts and sizeof statements are not followed by a space.
- Always write sizeof statements with parenthesis. The redundant parenthesis rules do not apply to sizeof(var) instances.

11.1.7 C Function Definition, Declaration and Use

Prototypes

- It is recommended (and generally required by the compiler) that all non-static functions are prototyped somewhere.
- Functions local to one source module should be declared static, and should not be prototyped unless absolutely necessary.
- Functions used from other parts of code (external API) must be prototyped in the relevant include file.
- Function prototypes should be listed in a logical order, preferably alphabetical unless there is a compelling reason to use a different ordering.
- Functions that are used locally in more than one module go into a separate header file, for example, "extern.h".
- Do not use the `__P` macro.
- Functions that are part of an external API should be documented using Doxygen-like comments above declarations. See *Doxygen Guidelines* for details.
- Functions that are part of the external API must have an `rte_` prefix on the function name.

- Do not use uppercase letters - either in the form of ALL_UPPERCASE, or CamelCase - in function names. Lower-case letters and underscores only.
- When prototyping functions, associate names with parameter types, for example:

```
void function1(int fd); /* good */
void function2(int);    /* bad */
```

- Short function prototypes should be contained on a single line. Longer prototypes, e.g. those with many parameters, can be split across multiple lines. The second and subsequent lines should be further indented as for line statement continuations as described in the previous section.

```
static char *function1(int _arg, const char *_arg2,
       struct foo *_arg3,
       struct bar *_arg4,
       struct baz *_arg5);
static void usage(void);
```

Note: Unlike function definitions, the function prototypes do not need to place the function return type on a separate line.

Definitions

- The function type should be on a line by itself preceding the function.
- The opening brace of the function body should be on a line by itself.

```
static char *
function(int a1, int a2, float fl, int a4)
{
```

- Do not declare functions inside other functions. ANSI C states that such declarations have file scope regardless of the nesting of the declaration. Hiding file declarations in what appears to be a local scope is undesirable and will elicit complaints from a good compiler.
- Old-style (K&R) function declaration should not be used, use ANSI function declarations instead as shown below.
- Long argument lists should be wrapped as described above in the function prototypes section.

```
/*
 * All major routines should have a comment briefly describing what
 * they do. The comment before the "main" routine should describe
 * what the program does.
 */
int
main(int argc, char *argv[])
{
        char *ep;
        long num;
        int ch;
```

11.1.8 C Statement Style and Conventions

NULL Pointers

- NULL is the preferred null pointer constant. Use NULL instead of `(type *)0` or `(type *)NULL`, except where the compiler does not know the destination type e.g. for variadic args to a function.
- Test pointers against NULL, for example, use:

```
if (p == NULL) /* Good, compare pointer to NULL */

if (!p) /* Bad, using ! on pointer */
```

- Do not use ! for tests unless it is a boolean, for example, use:

```
if (*p == '\0') /* check character against (char)0 */
```

Return Value

- Functions which create objects, or allocate memory, should return pointer types, and NULL on error. The error type should be indicated may setting the variable `rte_errno` appropriately.
- Functions which work on bursts of packets, such as RX-like or TX-like functions, should return the number of packets handled.
- Other functions returning int should generally behave like system calls: returning 0 on success and -1 on error, setting `rte_errno` to indicate the specific type of error.
- Where already standard in a given library, the alternative error approach may be used where the negative value is not -1 but is instead `-errno` if relevant, for example, `-EINVAL`. Note, however, to allow consistency across functions returning integer or pointer types, the previous approach is preferred for any new libraries.
- For functions where no error is possible, the function type should be `void` not `int`.
- Routines returning `void *` should not have their return values cast to any pointer type. (Typecasting can prevent the compiler from warning about missing prototypes as any implicit definition of a function returns int, which, unlike `void *`, needs a typecast to assign to a pointer variable.)

Note: The above rule about not typecasting `void *` applies to malloc, as well as to DPDK functions.

- Values in return statements should not be enclosed in parentheses.

Logging and Errors

In the DPDK environment, use the logging interface provided:

```
#define RTE_LOGTYPE_TESTAPP1 RTE_LOGTYPE_USER1
#define RTE_LOGTYPE_TESTAPP2 RTE_LOGTYPE_USER2

/* enable these logs type */
rte_set_log_type(RTE_LOGTYPE_TESTAPP1, 1);
rte_set_log_type(RTE_LOGTYPE_TESTAPP2, 1);

/* log in debug level */
rte_set_log_level(RTE_LOG_DEBUG);
RTE_LOG(DEBUG, TESTAPP1, "this is is a debug level message\n");
RTE_LOG(INFO, TESTAPP1, "this is is a info level message\n");
RTE_LOG(WARNING, TESTAPP1, "this is is a warning level message\n");

/* log in info level */
rte_set_log_level(RTE_LOG_INFO);
RTE_LOG(DEBUG, TESTAPP2, "debug level message (not displayed)\n");
```

Branch Prediction

- When a test is done in a critical zone (called often or in a data path) the code can use the `likely()` and `unlikely()` macros to indicate the expected, or preferred fast path. They are expanded as a compiler builtin and allow the developer to indicate if the branch is likely to be taken or not. Example:

```
#include <rte_branch_prediction.h>
if (likely(x > 1))
  do_stuff();
```

Note: The use of `likely()` and `unlikely()` should only be done in performance critical paths, and only when there is a clearly preferred path, or a measured performance increase gained from doing so. These macros should be avoided in non-performance-critical code.

Static Variables and Functions

- All functions and variables that are local to a file must be declared as `static` because it can often help the compiler to do some optimizations (such as, inlining the code).
- Functions that should be inlined should to be declared as `static inline` and can be defined in a .c or a .h file.

Note: Static functions defined in a header file must be declared as `static inline` in order to prevent compiler warnings about the function being unused.

Const Attribute

The `const` attribute should be used as often as possible when a variable is read-only.

Inline ASM in C code

The `asm` and `volatile` keywords do not have underscores. The AT&T syntax should be used. Input and output operands should be named to avoid confusion, as shown in the following example:

```
asm volatile("outb %[val], %[port]"
        : :
        [port] "dN" (port),
        [val] "a" (val));
```

Control Statements

- Forever loops are done with for statements, not while statements.
- Elements in a switch statement that cascade should have a FALLTHROUGH comment. For example:

```
switch (ch) {         /* Indent the switch. */
case 'a':             /* Don't indent the case. */
        aflag = 1;    /* Indent case body one tab. */
        /* FALLTHROUGH */
case 'b':
        bflag = 1;
        break;
case '?':
default:
        usage();
        /* NOTREACHED */
}
```

11.2 Design

11.2.1 Environment or Architecture-specific Sources

In DPDK and DPDK applications, some code is specific to an architecture (i686, x86_64) or to an executive environment (bsdapp or linuxapp) and so on. As far as is possible, all such instances of architecture or env-specific code should be provided via standard APIs in the EAL.

By convention, a file is common if it is not located in a directory indicating that it is specific. For instance, a file located in a subdir of “x86_64” directory is specific to this architecture. A file located in a subdir of “linuxapp” is specific to this execution environment.

Note: Code in DPDK libraries and applications should be generic. The correct location for architecture or executive environment specific code is in the EAL.

When absolutely necessary, there are several ways to handle specific code:

- Use a `#ifdef` with the CONFIG option in the C code. This can be done when the differences are small and they can be embedded in the same C file:

  ```
  #ifdef RTE_ARCH_I686
  toto();
  #else
  titi();
  #endif
  ```

- Use the CONFIG option in the Makefile. This is done when the differences are more significant. In this case, the code is split into two separate files that are architecture or environment specific. This should only apply inside the EAL library.

Note: As in the linux kernel, the `CONFIG_` prefix is not used in C code. This is only needed in Makefiles or shell scripts.

Per Architecture Sources

The following config options can be used:

- `CONFIG_RTE_ARCH` is a string that contains the name of the architecture.
- `CONFIG_RTE_ARCH_I686`, `CONFIG_RTE_ARCH_X86_64`, `CONFIG_RTE_ARCH_X86_64_32` or `CONFIG_RTE_ARCH_PPC_64` are defined only if we are building for those architectures.

Per Execution Environment Sources

The following config options can be used:

- `CONFIG_RTE_EXEC_ENV` is a string that contains the name of the executive environment.
- `CONFIG_RTE_EXEC_ENV_BSDAPP` or `CONFIG_RTE_EXEC_ENV_LINUXAPP` are defined only if we are building for this execution environment.

11.2.2 Library Statistics

Description

This document describes the guidelines for DPDK library-level statistics counter support. This includes guidelines for turning library statistics on and off and requirements for preventing ABI changes when implementing statistics.

Mechanism to allow the application to turn library statistics on and off

Each library that maintains statistics counters should provide a single build time flag that decides whether the statistics counter collection is enabled or not. This flag should be exposed as a variable within the DPDK configuration file. When this flag is set, all the counters supported by current library are collected for all the instances of every object type provided by the library. When this flag is cleared, none of the counters supported by the current library are collected for any instance of any object type provided by the library:

```
# DPDK file config/common_linuxapp, config/common_bsdapp, etc.
CONFIG_RTE_<LIBRARY_NAME>_STATS_COLLECT=y/n
```

The default value for this DPDK configuration file variable (either “yes” or “no”) is decided by each library.

Prevention of ABI changes due to library statistics support

The layout of data structures and prototype of functions that are part of the library API should not be affected by whether the collection of statistics counters is turned on or off for the current library. In practical terms, this means that space should always be allocated in the API data structures for statistics counters and the statistics related API functions are always built into the code, regardless of whether the statistics counter collection is turned on or off for the current library.

When the collection of statistics counters for the current library is turned off, the counters retrieved through the statistics related API functions should have a default value of zero.

Motivation to allow the application to turn library statistics on and off

It is highly recommended that each library provides statistics counters to allow an application to monitor the library-level run-time events. Typical counters are: number of packets received/dropped/transmitted, number of buffers allocated/freed, number of occurrences for specific events, etc.

However, the resources consumed for library-level statistics counter collection have to be spent out of the application budget and the counters collected by some libraries might not be relevant to the current application. In order to avoid any unwanted waste of resources and/or performance impacts, the application should decide at build time whether the collection of library-level statistics counters should be turned on or off for each library individually.

Library-level statistics counters can be relevant or not for specific applications:

- For Application A, counters maintained by Library X are always relevant and the application needs to use them to implement certain features, such as traffic accounting, logging, application-level statistics, etc. In this case, the application requires that collection of statistics counters for Library X is always turned on.
- For Application B, counters maintained by Library X are only useful during the application debug stage and are not relevant once debug phase is over. In this case, the application may decide to turn on the collection of Library X statistics counters during the debug phase and at a later stage turn them off.
- For Application C, counters maintained by Library X are not relevant at all. It might be that the application maintains its own set of statistics counters that monitor a different set of run-time events (e.g. number of connection requests, number of active users, etc). It might also be that the application uses multiple libraries (Library X, Library Y, etc) and it is interested in the statistics counters of Library Y, but not in those of Library X. In this case, the application may decide to turn the collection of statistics counters off for Library X and on for Library Y.

The statistics collection consumes a certain amount of CPU resources (cycles, cache bandwidth, memory bandwidth, etc) that depends on:

- Number of libraries used by the current application that have statistics counters collection turned on.
- Number of statistics counters maintained by each library per object type instance (e.g. per port, table, pipeline, thread, etc).
- Number of instances created for each object type supported by each library.

- Complexity of the statistics logic collection for each counter: when only some occurrences of a specific event are valid, additional logic is typically needed to decide whether the current occurrence of the event should be counted or not. For example, in the event of packet reception, when only TCP packets with destination port within a certain range should be recorded, conditional branches are usually required. When processing a burst of packets that have been validated for header integrity, counting the number of bits set in a bitmask might be needed.

11.3 Managing ABI updates

11.3.1 Description

This document details some methods for handling ABI management in the DPDK. Note this document is not exhaustive, in that C library versioning is flexible allowing multiple methods to achieve various goals, but it will provide the user with some introductory methods

11.3.2 General Guidelines

1. Whenever possible, ABI should be preserved
2. The libraries marked in experimental state may change without constraint.
3. The addition of symbols is generally not problematic
4. The modification of symbols can generally be managed with versioning
5. The removal of symbols generally is an ABI break and requires bumping of the LIBABIVER macro

11.3.3 What is an ABI

An ABI (Application Binary Interface) is the set of runtime interfaces exposed by a library. It is similar to an API (Application Programming Interface) but is the result of compilation. It is also effectively cloned when applications link to dynamic libraries. That is to say when an application is compiled to link against dynamic libraries, it is assumed that the ABI remains constant between the time the application is compiled/linked, and the time that it runs. Therefore, in the case of dynamic linking, it is critical that an ABI is preserved, or (when modified), done in such a way that the application is unable to behave improperly or in an unexpected fashion.

11.3.4 The DPDK ABI policy

ABI versions are set at the time of major release labeling, and the ABI may change multiple times, without warning, between the last release label and the HEAD label of the git tree.

ABI versions, once released, are available until such time as their deprecation has been noted in the Release Notes for at least one major release cycle. For example consider the case where the ABI for DPDK 2.0 has been shipped and then a decision is made to modify it during the development of DPDK 2.1. The decision will be recorded in the Release Notes for the DPDK 2.1 release and the modification will be made available in the DPDK 2.2 release.

ABI versions may be deprecated in whole or in part as needed by a given update.

Some ABI changes may be too significant to reasonably maintain multiple versions. In those cases ABI's may be updated without backward compatibility being provided. The requirements for doing so are:

1. At least 3 acknowledgments of the need to do so must be made on the dpdk.org mailing list.
2. The changes (including an alternative map file) must be gated with the `RTE_NEXT_ABI` option, and provided with a deprecation notice at the same time. It will become the default ABI in the next release.
3. A full deprecation cycle, as explained above, must be made to offer downstream consumers sufficient warning of the change.
4. At the beginning of the next release cycle, every `RTE_NEXT_ABI` conditions will be removed, the `LIBABIVER` variable in the makefile(s) where the ABI is changed will be incremented, and the map files will be updated.

Note that the above process for ABI deprecation should not be undertaken lightly. ABI stability is extremely important for downstream consumers of the DPDK, especially when distributed in shared object form. Every effort should be made to preserve the ABI whenever possible. The ABI should only be changed for significant reasons, such as performance enhancements. ABI breakage due to changes such as reorganizing public structure fields for aesthetic or readability purposes should be avoided.

11.3.5 Examples of Deprecation Notices

The following are some examples of ABI deprecation notices which would be added to the Release Notes:

- The Macro `#RTE_FOO` is deprecated and will be removed with version 2.0, to be replaced with the inline function `rte_foo()`.
- The function `rte_mbuf_grok()` has been updated to include a new parameter in version 2.0. Backwards compatibility will be maintained for this function until the release of version 2.1
- The members of `struct rte_foo` have been reorganized in release 2.0 for performance reasons. Existing binary applications will have backwards compatibility in release 2.0, while newly built binaries will need to reference the new structure variant `struct rte_foo2`. Compatibility will be removed in release 2.2, and all applications will require updating and rebuilding to the new structure at that time, which will be renamed to the original `struct rte_foo`.
- Significant ABI changes are planned for the `librte_dostuff` library. The upcoming release 2.0 will not contain these changes, but release 2.1 will, and no backwards compatibility is planned due to the extensive nature of these changes. Binaries using this library built prior to version 2.1 will require updating and recompilation.

11.3.6 Versioning Macros

When a symbol is exported from a library to provide an API, it also provides a calling convention (ABI) that is embodied in its name, return type and arguments. Occasionally that function may need to change to accommodate new functionality or behavior. When that occurs, it is desirable to allow for backward compatibility for a time with older binaries that are dynamically linked to the DPDK.

To support backward compatibility the `lib/librte_compat/rte_compat.h` header file provides macros to use when updating exported functions. These macros are used in conjunction with the `rte_<library>_version.map` file for a given library to allow multiple versions of a symbol to exist in a shared library so that older binaries need not be immediately recompiled.

The macros exported are:

- `VERSION_SYMBOL(b, e, n)`: Creates a symbol version table entry binding versioned symbol `b@DPDK_n` to the internal function `b_e`.
- `BIND_DEFAULT_SYMBOL(b, e, n)`: Creates a symbol version entry instructing the linker to bind references to symbol `b` to the internal symbol `b_e`.
- `MAP_STATIC_SYMBOL(f, p)`: Declare the prototype `f`, and map it to the fully qualified function `p`, so that if a symbol becomes versioned, it can still be mapped back to the public symbol name.

11.3.7 Examples of ABI Macro use

Updating a public API

Assume we have a function as follows

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param)
{
        ...
}
```

Assume that struct rte_acl_ctx is a private structure, and that a developer wishes to enhance the acl api so that a debugging flag can be enabled on a per-context basis. This requires an addition to the structure (which, being private, is safe), but it also requires modifying the code as follows

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param, int debug)
{
        ...
}
```

Note also that, being a public function, the header file prototype must also be changed, as must all the call sites, to reflect the new ABI footprint. We will maintain previous ABI versions that are accessible only to previously compiled binaries

The addition of a parameter to the function is ABI breaking as the function is public, and existing application may use it in its current form. However, the compatibility macros in DPDK allow a developer to use symbol versioning so that multiple functions can be mapped to the same public symbol based on when an application was linked to it. To see how this is done, we start with the requisite libraries version map file. Initially the version map file for the acl library looks like this

```
DPDK_2.0 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_create;
     rte_acl_dump;
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
     rte_acl_reset_rules;
     rte_acl_set_ctx_classify;

     local: *;
};
```

This file needs to be modified as follows

```
DPDK_2.0 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_create;
     rte_acl_dump;
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
     rte_acl_reset_rules;
     rte_acl_set_ctx_classify;

     local: *;
};

DPDK_2.1 {
     global:
     rte_acl_create;

} DPDK_2.0;
```

The addition of the new block tells the linker that a new version node is available (DPDK_2.1), which contains the symbol rte_acl_create, and inherits the symbols from the DPDK_2.0 node. This list is directly translated into a list of exported symbols when DPDK is compiled as a shared library

Next, we need to specify in the code which function map to the rte_acl_create symbol at which versions. First, at the site of the initial symbol definition, we need to update the function so that it is uniquely named, and not in conflict with the public symbol name

```
 struct rte_acl_ctx *
-rte_acl_create(const struct rte_acl_param *param)
+rte_acl_create_v20(const struct rte_acl_param *param)
{
        size_t sz;
        struct rte_acl_ctx *ctx;
        ...
```

Note that the base name of the symbol was kept intact, as this is conducive to the macros used for versioning symbols. That is our next step, mapping this new symbol name to the initial symbol name at version node 2.0. Immediately after the function, we add this line of code

```
VERSION_SYMBOL(rte_acl_create, _v20, 2.0);
```

Remembering to also add the rte_compat.h header to the requisite c file where these changes are being made. The above macro instructs the linker to create a new symbol `rte_acl_create@DPDK_2.0`, which matches the symbol created in older builds, but now points to the above newly named function. We have now mapped the original rte_acl_create symbol to the original function (but with a new name)

Next, we need to create the 2.1 version of the symbol. We create a new function name, with a different suffix, and implement it appropriately

```
struct rte_acl_ctx *
rte_acl_create_v21(const struct rte_acl_param *param, int debug);
{
     struct rte_acl_ctx *ctx = rte_acl_create_v20(param);

     ctx->debug = debug;

     return ctx;
}
```

This code serves as our new API call. Its the same as our old call, but adds the new parameter in place. Next we need to map this function to the symbol `rte_acl_create@DPDK_2.1`. To do this, we modify the public prototype of the call in the header file, adding the macro there to inform all including applications, that on re-link, the default rte_acl_create symbol should point to this function. Note that we could do this by simply naming the function above rte_acl_create, and the linker would chose the most recent version tag to apply in the version script, but we can also do this in the header file

```
struct rte_acl_ctx *
-rte_acl_create(const struct rte_acl_param *param);
+rte_acl_create(const struct rte_acl_param *param, int debug);
+BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 2.1);
```

The BIND_DEFAULT_SYMBOL macro explicitly tells applications that include this header, to link to the rte_acl_create_v21 function and apply the DPDK_2.1 version node to it. This method is more explicit and flexible than just re-implementing the exact symbol name, and allows for other features (such as linking to the old symbol version by default, when the new ABI is to be opt-in for a period.

One last thing we need to do. Note that we've taken what was a public symbol, and duplicated it into two uniquely and differently named symbols. We've then mapped each of those back to the public symbol `rte_acl_create` with different version tags. This only applies to dynamic linking, as static linking has no notion of versioning. That leaves this code in a position of no longer having a symbol simply named `rte_acl_create` and a static build will fail on that missing symbol.

To correct this, we can simply map a function of our choosing back to the public symbol in the static build with the `MAP_STATIC_SYMBOL` macro. Generally the assumption is that the most recent version of the symbol is the one you want to map. So, back in the C file where, immediately after `rte_acl_create_v21` is defined, we add this

```
struct rte_acl_create_v21(const struct rte_acl_param *param, int debug)
{
     ...
}
MAP_STATIC_SYMBOL(struct rte_acl_create(const struct rte_acl_param *param, int debug), rte_acl_
```

That tells the compiler that, when building a static library, any calls to the symbol `rte_acl_create` should be linked to `rte_acl_create_v21`

That's it, on the next shared library rebuild, there will be two versions of rte_acl_create, an old DPDK_2.0 version, used by previously built applications, and a new DPDK_2.1 version, used by future built applications.

Deprecating part of a public API

Lets assume that you've done the above update, and after a few releases have passed you decide you would like to retire the old version of the function. After having gone through the ABI deprecation announcement process, removal is easy. Start by removing the symbol from the requisite version map file:

```
  DPDK_2.0 {
       global:

       rte_acl_add_rules;
       rte_acl_build;
       rte_acl_classify;
       rte_acl_classify_alg;
       rte_acl_classify_scalar;
       rte_acl_dump;
-      rte_acl_create
       rte_acl_find_existing;
       rte_acl_free;
       rte_acl_ipv4vlan_add_rules;
       rte_acl_ipv4vlan_build;
       rte_acl_list_dump;
       rte_acl_reset;
       rte_acl_reset_rules;
       rte_acl_set_ctx_classify;

       local: *;
  };

  DPDK_2.1 {
       global:
       rte_acl_create;
  } DPDK_2.0;
```

Next remove the corresponding versioned export.

```
-VERSION_SYMBOL(rte_acl_create, _v20, 2.0);
```

Note that the internal function definition could also be removed, but its used in our example by the newer version _v21, so we leave it in place. This is a coding style choice.

Lastly, we need to bump the LIBABIVER number for this library in the Makefile to indicate to applications doing dynamic linking that this is a later, and possibly incompatible library version:

```
-LIBABIVER := 1
+LIBABIVER := 2
```

Deprecating an entire ABI version

While removing a symbol from and ABI may be useful, it is often more practical to remove an entire version node at once. If a version node completely specifies an API, then removing part of it, typically makes it incomplete. In those cases it is better to remove the entire node

To do this, start by modifying the version map file, such that all symbols from the node to be removed are merged into the next node in the map

In the case of our map above, it would transform to look as follows

```
  DPDK_2.1 {
       global:

       rte_acl_add_rules;
       rte_acl_build;
       rte_acl_classify;
       rte_acl_classify_alg;
       rte_acl_classify_scalar;
       rte_acl_dump;
       rte_acl_create
       rte_acl_find_existing;
       rte_acl_free;
       rte_acl_ipv4vlan_add_rules;
       rte_acl_ipv4vlan_build;
       rte_acl_list_dump;
       rte_acl_reset;
       rte_acl_reset_rules;
       rte_acl_set_ctx_classify;

       local: *;
};
```

Then any uses of BIND_DEFAULT_SYMBOL that pointed to the old node should be updated to point to the new version node in any header files for all affected symbols.

```
-BIND_DEFAULT_SYMBOL(rte_acl_create, _v20, 2.0);
+BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 2.1);
```

Lastly, any VERSION_SYMBOL macros that point to the old version node should be removed, taking care to keep, where need old code in place to support newer versions of the symbol.

11.3.8 Running the ABI Validator

The `scripts` directory in the DPDK source tree contains a utility program, `validate-abi.sh`, for validating the DPDK ABI based on the Linux ABI Compliance Checker.

This has a dependency on the `abi-compliance-checker` and `and abi-dumper` utilities which can be installed via a package manager. For example:

```
sudo yum install abi-compliance-checker
sudo yum install abi-dumper
```

The syntax of the `validate-abi.sh` utility is:

```
./scripts/validate-abi.sh <REV1> <REV2> <TARGET>
```

Where `REV1` and `REV2` are valid gitrevisions(7) https://www.kernel.org/pub/software/scm/git/docs/gitrevisions.h
on the local repo and target is the usual DPDK compilation target.

For example:

> # Check between the previous and latest commit: ./scripts/validate-abi.sh HEAD~1 HEAD x86_64-native-linuxapp-gcc
>
> # Check between two tags: ./scripts/validate-abi.sh v2.0.0 v2.1.0 x86_64-native-linuxapp-gcc
>
> # Check between git master and local topic-branch "vhost-hacking": ./scripts/validate-abi.sh master vhost-hacking x86_64-native-linuxapp-gcc

After the validation script completes (it can take a while since it need to compile both tags) it will create compatibility reports in the `./compat_report` directory. Listed incompatibilities can be found as follows:

```
grep -lr Incompatible compat_reports/
```

11.4 DPDK Documentation Guidelines

This document outlines the guidelines for writing the DPDK Guides and API documentation in RST and Doxygen format.

It also explains the structure of the DPDK documentation and shows how to build the Html and PDF versions of the documents.

11.4.1 Structure of the Documentation

The DPDK source code repository contains input files to build the API documentation and User Guides.

The main directories that contain files related to documentation are shown below:

```
lib
|-- librte_acl
|-- librte_cfgfile
|-- librte_cmdline
|-- librte_compat
|-- librte_eal
|   |-- ...
...
doc
|-- api
+-- guides
    |-- freebsd_gsg
    |-- linux_gsg
    |-- prog_guide
    |-- sample_app_ug
    |-- guidelines
    |-- testpmd_app_ug
    |-- rel_notes
    |-- nics
    |-- xen
    |-- ...
```

The API documentation is built from Doxygen comments in the header files. These files are mainly in the `lib/librte_*` directories although some of the Poll Mode Drivers in `drivers/net` are also documented with Doxygen.

The configuration files that are used to control the Doxygen output are in the `doc/api` directory.

The user guides such as *The Programmers Guide* and the *FreeBSD* and *Linux Getting Started* Guides are generated from RST markup text files using the Sphinx Documentation Generator.

These files are included in the `doc/guides/` directory. The output is controlled by the `doc/guides/conf.py` file.

11.4.2 Role of the Documentation

The following items outline the roles of the different parts of the documentation and when they need to be updated or added to by the developer.

- **Release Notes**

  The Release Notes document which features have been added in the current and previous releases of DPDK and highlight any known issues. The Releases Notes also contain notifications of features that will change ABI compatibility in the next major release.

  Developers should include updates to the Release Notes with patch sets that relate to any of the following sections:

  - New Features
  - Resolved Issues (see below)
  - Known Issues
  - API Changes
  - ABI Changes
  - Shared Library Versions

  Resolved Issues should only include issues from previous releases that have been resolved in the current release. Issues that are introduced and then fixed within a release cycle do not have to be included here.

  Refer to the Release Notes from the previous DPDK release for the correct format of each section.

- **API documentation**

  The API documentation explains how to use the public DPDK functions. The API index page shows the generated API documentation with related groups of functions.

  The API documentation should be updated via Doxygen comments when new functions are added.

- **Getting Started Guides**

  The Getting Started Guides show how to install and configure DPDK and how to run DPDK based applications on different OSes.

  A Getting Started Guide should be added when DPDK is ported to a new OS.

- **The Programmers Guide**

  The Programmers Guide explains how the API components of DPDK such as the EAL, Memzone, Rings and the Hash Library work. It also explains how some higher level functionality such as Packet Distributor, Packet Framework and KNI work. It also shows the build system and explains how to add applications.

  The Programmers Guide should be expanded when new functionality is added to DPDK.

- **App Guides**

  The app guides document the DPDK applications in the `app` directory such as `testpmd`.

  The app guides should be updated if functionality is changed or added.

- **Sample App Guides**

  The sample app guides document the DPDK example applications in the examples directory. Generally they demonstrate a major feature such as L2 or L3 Forwarding, Multi Process or Power Management. They explain the purpose of the sample application, how to run it and step through some of the code to explain the major functionality.

  A new sample application should be accompanied by a new sample app guide. The guide for the Skeleton Forwarding app is a good starting reference.

- **Network Interface Controller Drivers**

  The NIC Drivers document explains the features of the individual Poll Mode Drivers, such as software requirements, configuration and initialization.

  New documentation should be added for new Poll Mode Drivers.

- **Guidelines**

  The guideline documents record community process, expectations and design directions.

  They can be extended, amended or discussed by submitting a patch and getting community approval.

11.4.3 Building the Documentation

Dependencies

The following dependencies must be installed to build the documentation:

- Doxygen.
- Sphinx (also called python-sphinx).
- TexLive (at least TexLive-core and the extra Latex support).
- Inkscape.

Doxygen generates documentation from commented source code. It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install doxygen

# Red Hat/Fedora.
sudo yum     -y install doxygen
```

Sphinx is a Python documentation tool for converting RST files to Html or to PDF (via LaTeX). For full support with figure and table captioning the latest version of Sphinx can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install python-pip
sudo pip install --upgrade sphinx

# Red Hat/Fedora.
sudo yum     -y install python-pip
sudo pip install --upgrade sphinx
```

For further information on getting started with Sphinx see the Sphinx Tutorial.

Note: To get full support for Figure and Table numbering it is best to install Sphinx 1.3.1 or later.

Inkscape is a vector based graphics program which is used to create SVG images and also to convert SVG images to PDF images. It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install inkscape

# Red Hat/Fedora.
sudo yum     -y install inkscape
```

TexLive is an installation package for Tex/LaTeX. It is used to generate the PDF versions of the documentation. The main required packages can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install texlive-latex-extra

# Red Hat/Fedora, selective install.
sudo yum     -y install texlive-collection-latexextra
```

Build commands

The documentation is built using the standard DPDK build system. Some examples are shown below:

- Generate all the documentation targets:

  ```
  make doc
  ```

- Generate the Doxygen API documentation in Html:

  ```
  make doc-api-html
  ```

- Generate the guides documentation in Html:

  ```
  make doc-guides-html
  ```

- Generate the guides documentation in Pdf:

  ```
  make doc-guides-pdf
  ```

The output of these commands is generated in the `build` directory:

```
build/doc
      |-- html
      |   |-- api
      |   +-- guides
```

```
    |
    +-- pdf
        +-- guides
```

Note: Make sure to fix any Sphinx or Doxygen warnings when adding or updating documentation.

The documentation output files can be removed as follows:

```
make doc-clean
```

11.4.4 Document Guidelines

Here are some guidelines in relation to the style of the documentation:

- Document the obvious as well as the obscure since it won't always be obvious to the reader. For example an instruction like "Set up 64 2MB Hugepages" is better when followed by a sample commandline or a link to the appropriate section of the documentation.
- Use American English spellings throughout. This can be checked using the `aspell` utility:

  ```
  aspell --lang=en_US --check doc/guides/sample_app_ug/mydoc.rst
  ```

11.4.5 RST Guidelines

The RST (reStructuredText) format is a plain text markup format that can be converted to Html, PDF or other formats. It is most closely associated with Python but it can be used to document any language. It is used in DPDK to document everything apart from the API.

The Sphinx documentation contains a very useful RST Primer which is a good place to learn the minimal set of syntax required to format a document.

The official reStructuredText website contains the specification for the RST format and also examples of how to use it. However, for most developers the RST Primer is a better resource.

The most common guidelines for writing RST text are detailed in the Documenting Python guidelines. The additional guidelines below reiterate or expand upon those guidelines.

Line Length

- The recommended style for the DPDK documentation is to put sentences on separate lines. This allows for easier reviewing of patches. Multiple sentences which are not separated by a blank line are joined automatically into paragraphs, for example:

  ```
  Here is an example sentence.
  Long sentences over the limit shown below can be wrapped onto
  a new line.
  These three sentences will be joined into the same paragraph.

  This is a new paragraph, since it is separated from the
  previous paragraph by a blank line.
  ```

  This would be rendered as follows:

> *Here is an example sentence. Long sentences over the limit shown below can be wrapped onto a new line. These three sentences will be joined into the same paragraph.*
>
> *This is a new paragraph, since it is separated from the previous paragraph by a blank line.*

- Long sentences should be wrapped at 120 characters +/- 10 characters. They should be wrapped at words.
- Lines in literal blocks must by less than 80 characters since they aren't wrapped by the document formatters and can exceed the page width in PDF documents.

Whitespace

- Standard RST indentation is 3 spaces. Code can be indented 4 spaces, especially if it is copied from source files.
- No tabs. Convert tabs in embedded code to 4 or 8 spaces.
- No trailing whitespace.
- Add 2 blank lines before each section header.
- Add 1 blank line after each section header.
- Add 1 blank line between each line of a list.

Section Headers

- Section headers should use the use the following underline formats:

```
Level 1 Heading
===============


Level 2 Heading
---------------


Level 3 Heading
~~~~~~~~~~~~~~~


Level 4 Heading
^^^^^^^^^^^^^^^
```

- Level 4 headings should be used sparingly.
- The underlines should match the length of the text.
- In general, the heading should be less than 80 characters, for conciseness.
- As noted above:
  - Add 2 blank lines before each section header.
  - Add 1 blank line after each section header.

Lists

- Bullet lists should be formatted with a leading `*` as follows:

```
* Item one.

* Item two is a long line that is wrapped and then indented to match
  the start of the previous line.

* One space character between the bullet and the text is preferred.
```

- Numbered lists can be formatted with a leading number but the preference is to use `#.` which will give automatic numbering. This is more convenient when adding or removing items:

```
#. Item one.

#. Item two is a long line that is wrapped and then indented
   to match the start of the e first line.

#. Item two is a long line that is wrapped and then indented to match
   the start of the previous line.
```

- Definition lists can be written with or without a bullet:

```
* Item one.

  Some text about item one.

* Item two.

  Some text about item two.
```

- All lists, and sub-lists, must be separated from the preceding text by a blank line. This is a syntax requirement.
- All list items should be separated by a blank line for readability.

Code and Literal block sections

- Inline text that is required to be rendered with a fixed width font should be enclosed in backquotes like this: “text“, so that it appears like this: `text`.
- Fixed width, literal blocks of texts should be indented at least 3 spaces and prefixed with `::` like this:

```
Here is some fixed width text::

   0x0001 0x0001 0x00FF 0x00FF
```

- It is also possible to specify an encoding for a literal block using the `.. code-block::` directive so that syntax highlighting can be applied. Examples of supported highlighting are:

```
.. code-block:: console
.. code-block:: c
.. code-block:: python
.. code-block:: diff
.. code-block:: none
```

  That can be applied as follows:

```
.. code-block:: c

   #include<stdio.h>

   int main() {

      printf("Hello World\n");

      return 0;
   }
```

Which would be rendered as:

```c
#include<stdio.h>

int main() {

   printf("Hello World\n");

   return 0;
}
```

- The default encoding for a literal block using the simplified `::` directive is `none`.
- Lines in literal blocks must be less than 80 characters since they can exceed the page width when converted to PDF documentation. For long literal lines that exceed that limit try to wrap the text at sensible locations. For example a long command line could be documented like this and still work if copied directly from the docs:

```
build/app/testpmd -c7 -n3 --vdev=eth_pcap0,iface=eth0     \
                          --vdev=eth_pcap1,iface=eth1     \
                          -- -i --nb-cores=2 --nb-ports=2 \
                             --total-num-mbufs=2048
```

- Long lines that cannot be wrapped, such as application output, should be truncated to be less than 80 characters.

Images

- All images should be in SVG scalar graphics format. They should be true SVG XML files and should not include binary formats embedded in a SVG wrapper.
- The DPDK documentation contains some legacy images in PNG format. These will be converted to SVG in time.
- Inkscape is the recommended graphics editor for creating the images. Use some of the older images in `doc/guides/prog_guide/img/` as a template, for example `mbuf1.svg` or `ring-enqueue.svg`.
- The SVG images should include a copyright notice, as an XML comment.
- Images in the documentation should be formatted as follows:
  - The image should be preceded by a label in the format `.. _figure_XXXX:` with a leading underscore and where `XXXX` is a unique descriptive name.
  - Images should be included using the `.. figure::` directive and the file type should be set to `*` (not `.svg`). This allows the format of the image to be changed if required, without updating the documentation.
  - Images must have a caption as part of the `.. figure::` directive.

- Here is an example of the previous three guidelines:

  ```
  .. _figure_mempool:

  .. figure:: img/mempool.*

     A mempool in memory with its associated ring.
  ```

- Images can then be linked to using the `:numref:` directive:

  ```
  The mempool layout is shown in :numref:`figure_mempool`.
  ```

  This would be rendered as: *The mempool layout is shown in Fig 6.3*.

  Note: The `:numref:` directive requires Sphinx 1.3.1 or later. With earlier versions it will still be rendered as a link but won't have an automatically generated number.

- The caption of the image can be generated, with a link, using the `:ref:` directive:

  ```
  :ref:`figure_mempool`
  ```

  This would be rendered as: *A mempool in memory with its associated ring.*

Tables

- RST tables should be used sparingly. They are hard to format and to edit, they are often rendered incorrectly in PDF format, and the same information can usually be shown just as clearly with a definition or bullet list.
- Tables in the documentation should be formatted as follows:
  - The table should be preceded by a label in the format `.. _table_XXXX:` with a leading underscore and where `XXXX` is a unique descriptive name.
  - Tables should be included using the `.. table::` directive and must have a caption.
- Here is an example of the previous two guidelines:

  ```
  .. _table_qos_pipes:

  .. table:: Sample configuration for QOS pipes.

     +----------+----------+----------+
     | Header 1 | Header 2 | Header 3 |
     |          |          |          |
     +==========+==========+==========+
     | Text     | Text     | Text     |
     +----------+----------+----------+
     | ...      | ...      | ...      |
     +----------+----------+----------+
  ```

- Tables can be linked to using the `:numref:` and `:ref:` directives, as shown in the previous section for images. For example:

  ```
  The QOS configuration is shown in :numref:`table_qos_pipes`.
  ```

- Tables should not include merged cells since they are not supported by the PDF renderer.

Hyperlinks

- Links to external websites can be plain URLs. The following is rendered as http://dpdk.org:

  ```
  http://dpdk.org
  ```

- They can contain alternative text. The following is rendered as Check out DPDK:

  ```
  `Check out DPDK <http://dpdk.org>`_
  ```

- An internal link can be generated by placing labels in the document with the format `.. _label_name`.
- The following links to the top of this section: *Hyperlinks*:

  ```
  .. _links:

  Hyperlinks
  ~~~~~~~~~~

  * The following links to the top of this section: :ref:`links`:
  ```

Note: The label must have a leading underscore but the reference to it must omit it. This is a frequent cause of errors and warnings.

- The use of a label is preferred since it works across files and will still work if the header text changes.

11.4.6 Doxygen Guidelines

The DPDK API is documented using Doxygen comment annotations in the header files. Doxygen is a very powerful tool, it is extremely configurable and with a little effort can be used to create expressive documents. See the Doxygen website for full details on how to use it.

The following are some guidelines for use of Doxygen in the DPDK API documentation:

- New libraries that are documented with Doxygen should be added to the Doxygen configuration file: `doc/api/doxy-api.conf`. It is only required to add the directory that contains the files. It isn't necessary to explicitly name each file since the configuration matches all `rte_*.h` files in the directory.
- Use proper capitalization and punctuation in the Doxygen comments since they will become sentences in the documentation. This in particular applies to single line comments, which is the case the is most often forgotten.
- Use `@` style Doxygen commands instead of `\` style commands.
- Add a general description of each library at the head of the main header files:

  ```
  /**
   * @file
   * RTE Mempool.
   *
   * A memory pool is an allocator of fixed-size object. It is
   * identified by its name, and uses a ring to store free objects.
   * ...
   */
  ```

- Document the purpose of a function, the parameters used and the return value:

```
/**
 * Attach a new Ethernet device specified by arguments.
 *
 * @param devargs
 *  A pointer to a strings array describing the new device
 *  to be attached. The strings should be a pci address like
 *  `0000:01:00.0` or **virtual** device name like `eth_pcap0`.
 * @param port_id
 *  A pointer to a port identifier actually attached.
 *
 * @return
 *  0 on success and port_id is filled, negative on error.
 */
int rte_eth_dev_attach(const char *devargs, uint8_t *port_id);
```

- Doxygen supports Markdown style syntax such as bold, italics, fixed width text and lists. For example the second line in the `devargs` parameter in the previous example will be rendered as:

  > The strings should be a pci address like `0000:01:00.0` or **virtual** device name like `eth_pcap0`.

- Use `-` instead of `*` for lists within the Doxygen comment since the latter can get confused with the comment delimiter.
- Add an empty line between the function description, the `@params` and `@return` for readability.
- Place the `@params` description on separate line and indent it by 2 spaces. (It would be better to use no indentation since this is more common and also because checkpatch complains about leading whitespace in comments. However this is the convention used in the existing DPDK code.)
- Documented functions can be linked to simply by adding `()` to the function name:

```
/**
 * The functions exported by the application Ethernet API to setup
 * a device designated by its port identifier must be invoked in
 * the following order:
 *     - rte_eth_dev_configure()
 *     - rte_eth_tx_queue_setup()
 *     - rte_eth_rx_queue_setup()
 *     - rte_eth_dev_start()
 */
```

  In the API documentation the functions will be rendered as links, see the online section of the rte_ethdev.h docs that contains the above text.

- The `@see` keyword can be used to create a *see also* link to another file or library. This directive should be placed on one line at the bottom of the documentation section.

```
/**
 * ...
 *
 * Some text that references mempools.
 *
 * @see eal_memzone.c
 */
```

- Doxygen supports two types of comments for documenting variables, constants and members: prefix and postfix:

```
/** This is a prefix comment. */
#define RTE_FOO_ERROR  0x023.

#define RTE_BAR_ERROR  0x024. /**< This is a postfix comment. */
```

- Postfix comments are preferred for struct members and constants if they can be documented in the same way:

  ```
  struct rte_eth_stats {
      uint64_t ipackets; /**< Total number of received packets. */
      uint64_t opackets; /**< Total number of transmitted packets.*/
      uint64_t ibytes;   /**< Total number of received bytes. */
      uint64_t obytes;   /**< Total number of transmitted bytes. */
      uint64_t imissed;  /**< Total of RX missed packets. */
      uint64_t ibadcrc;  /**< Total of RX packets with CRC error. */
      uint64_t ibadlen;  /**< Total of RX packets with bad length. */
  }
  ```

  Note: postfix comments should be aligned with spaces not tabs in accordance with the *DPDK Coding Style*.

- If a single comment type can't be used, due to line length limitations then prefix comments should be preferred. For example this section of the code contains prefix comments, postfix comments on the same line and postfix comments on a separate line:

  ```
  /** Number of elements in the elt_pa array. */
  uint32_t    pg_num __rte_cache_aligned;
  uint32_t    pg_shift;     /**< LOG2 of the physical pages. */
  uintptr_t   pg_mask;      /**< Physical page mask value. */
  uintptr_t   elt_va_start;
  /**< Virtual address of the first mempool object. */
  uintptr_t   elt_va_end;
  /**< Virtual address of the <size + 1> mempool object. */
  phys_addr_t elt_pa[MEMPOOL_PG_NUM_DEFAULT];
  /**< Array of physical page addresses for the mempool buffer. */
  ```

  This doesn't have an effect on the rendered documentation but it is confusing for the developer reading the code. It this case it would be clearer to use prefix comments throughout:

  ```
  /** Number of elements in the elt_pa array. */
  uint32_t    pg_num __rte_cache_aligned;
  /** LOG2 of the physical pages. */
  uint32_t    pg_shift;
  /** Physical page mask value. */
  uintptr_t   pg_mask;
  /** Virtual address of the first mempool object. */
  uintptr_t   elt_va_start;
  /** Virtual address of the <size + 1> mempool object. */
  uintptr_t   elt_va_end;
  /** Array of physical page addresses for the mempool buffer. */
  phys_addr_t elt_pa[MEMPOOL_PG_NUM_DEFAULT];
  ```

- Check for Doxygen warnings in new code by checking the API documentation build:

  ```
  make doc-api-html >/dev/null
  ```

- Read the rendered section of the documentation that you have added for correctness, clarity and consistency with the surrounding text.

11.5 Contributing Code to DPDK

This document outlines the guidelines for submitting code to DPDK.

The DPDK development process is modelled (loosely) on the Linux Kernel development model so it is worth reading the Linux kernel guide on submitting patches: How to Get Your Change Into the Linux Kernel. The rationale for many of the DPDK guidelines is explained in greater detail in the kernel guidelines.

11.5.1 The DPDK Development Process

The DPDK development process has the following features:

- The code is hosted in a public git repository.
- There is a mailing list where developers submit patches.
- There are maintainers for hierarchical components.
- Patches are reviewed publicly on the mailing list.
- Successfully reviewed patches are merged to the master branch of the repository.

The mailing list for DPDK development is dev@dpkg.org. Contributors will need to register for the mailing list in order to submit patches. It is also worth registering for the DPDK Patchwork

The development process requires some familiarity with the `git` version control system. Refer to the Pro Git Book for further information.

11.5.2 Getting the Source Code

The source code can be cloned using either of the following:

```
git clone git://dpdk.org/dpdk

git clone http://dpdk.org/git/dpdk
```

11.5.3 Make your Changes

Make your planned changes in the cloned `dpdk` repo. Here are some guidelines and requirements:

- Follow the *DPDK Coding Style* guidelines.
- If you add new files or directories you should add your name to the `MAINTAINERS` file.
- New external functions should be added to the local `version.map` file. See the Guidelines for ABI policy and versioning. New external functions should also be added in alphabetical order.
- Important changes will require an addition to the release notes in `doc/guides/rel_notes/`. See the *Release Notes section of the Documentation Guidelines* for details.
- Test the compilation works with different targets, compilers and options, see *Checking Compilation*.

- Don't break compilation between commits with forward dependencies in a patchset. Each commit should compile on its own to allow for `git bisect` and continuous integration testing.
- Add tests to the the `app/test` unit test framework where possible.
- Add documentation, if relevant, in the form of Doxygen comments or a User Guide in RST format. See the *Documentation Guidelines*.

Once the changes have been made you should commit them to your local repo.

For small changes, that do not require specific explanations, it is better to keep things together in the same patch. Larger changes that require different explanations should be separated into logical patches in a patchset. A good way of thinking about whether a patch should be split is to consider whether the change could be applied without dependencies as a backport.

As a guide to how patches should be structured run `git log` on similar files.

11.5.4 Commit Messages: Subject Line

The first, summary, line of the git commit message becomes the subject line of the patch email. Here are some guidelines for the summary line:

- The summary line must capture the area and the impact of the change.
- The summary line should be around 50 characters.
- The summary line should be lowercase apart from acronyms.
- It should be prefixed with the component name (use git log to check existing components). For example:

  ```
  ixgbe: fix offload config option name

  config: increase max queues per port
  ```

- Use the imperative of the verb (like instructions to the code base).
- Don't add a period/full stop to the subject line or you will end up two in the patch name: `dpdk_description..patch`.

The actual email subject line should be prefixed by `[PATCH]` and the version, if greater than v1, for example: `PATCH v2`. The is generally added by `git send-email` or `git format-patch`, see below.

If you are submitting an RFC draft of a feature you can use `[RFC]` instead of `[PATCH]`. An RFC patch doesn't have to be complete. It is intended as a way of getting early feedback.

11.5.5 Commit Messages: Body

Here are some guidelines for the body of a commit message:

- The body of the message should describe the issue being fixed or the feature being added. It is important to provide enough information to allow a reviewer to understand the purpose of the patch.
- When the change is obvious the body can be blank, apart from the signoff.
- The commit message must end with a `Signed-off-by:` line which is added using:

```
git commit --signoff # or -s
```

The purpose of the signoff is explained in the Developer's Certificate of Origin section of the Linux kernel guidelines.

Note: All developers must ensure that they have read and understood the Developer's Certificate of Origin section of the documentation prior to applying the signoff and submitting a patch.

- The signoff must be a real name and not an alias or nickname. More than one signoff is allowed.
- The text of the commit message should be wrapped at 72 characters.
- When fixing a regression, it is a good idea to reference the id of the commit which introduced the bug. You can generate the required text using the following git alias:

  ```
  git config alias.fixline "log -1 --abbrev=12 --format='Fixes: %h (\"%s\")'"
  ```

  The `Fixes:` line can then be added to the commit message:

  ```
  doc: fix vhost sample parameter

  Update the docs to reflect removed dev-index.

  Fixes: 17b8320a3e11 ("vhost: remove index parameter")

  Signed-off-by: Alex Smith <alex.smith@example.com>
  ```

- When fixing an error or warning it is useful to add the error message and instructions on how to reproduce it.
- Use correct capitalization, punctuation and spelling.

In addition to the `Signed-off-by:` name the commit messages can also have one or more of the following:

- `Reported-by:` The reporter of the issue.
- `Tested-by:` The tester of the change.
- `Reviewed-by:` The reviewer of the change.
- `Suggested-by:` The person who suggested the change.
- `Acked-by:` When a previous version of the patch was acked and the ack is still relevant.

11.5.6 Creating Patches

It is possible to send patches directly from git but for new contributors it is recommended to generate the patches with `git format-patch` and then when everything looks okay, and the patches have been checked, to send them with `git send-email`.

Here are some examples of using `git format-patch` to generate patches:

```
# Generate a patch from the last commit.
git format-patch -1

# Generate a patch from the last 3 commits.
git format-patch -3
```

```
# Generate the patches in a directory.
git format-patch -3 -o ~/patch/

# Add a cover letter to explain a patchset.
git format-patch -3 -o ~/patch/ --cover-letter

# Add a prefix with a version number.
git format-patch -3 -o ~/patch/ -v 2
```

Cover letters are useful for explaining a patchset and help to generate a logical threading to the patches. Smaller notes can be put inline in the patch after the `---` separator, for example:

```
Subject: [PATCH] fm10k/base: add FM10420 device ids

Add the device ID for Boulder Rapids and Atwood Channel to enable
drivers to support those devices.

Signed-off-by: Alex Smith <alex.smith@example.com>
---

ADD NOTES HERE.

 drivers/net/fm10k/base/fm10k_api.c  | 6 ++++++
 drivers/net/fm10k/base/fm10k_type.h | 6 ++++++
 2 files changed, 12 insertions(+)
...
```

Version 2 and later of a patchset should also include a short log of the changes so the reviewer knows what has changed. This can be added to the cover letter or the annotations. For example:

```
---
v3:
* Fixed issued with version.map.

v2:
* Added i40e support.
* Renamed ethdev functions from rte_eth_ieee15888_*() to rte_eth_timesync_*()
  since 802.1AS can be supported through the same interfaces.
```

11.5.7 Checking the Patches

Patches should be checked for formatting and syntax issues using the `checkpatches.sh` script in the `scripts` directory of the DPDK repo. This uses the Linux kernel development tool `checkpatch.pl` which can be obtained by cloning, and periodically, updating the Linux kernel sources.

The path to the original Linux script must be set in the environment variable `DPDK_CHECKPATCH_PATH`. This, and any other configuration variables required by the development tools, are loaded from the following files, in order of preference:

```
.develconfig
~/.config/dpdk/devel.config
/etc/dpdk/devel.config.
```

Once the environment variable the script can be run as follows:

```
scripts/checkpatches.sh ~/patch/
```

The script usage is:

```
checkpatches.sh [-h] [-q] [-v] [patch1 [patch2] ...]]"
```

Where:

- `-h`: help, usage.
- `-q`: quiet. Don't output anything for files without issues.
- `-v`: verbose.
- `patchX`: path to one or more patches.

11.5.8 Checking Compilation

Compilation of patches and changes should be tested using the the `test-build.sh` script in the `scripts` directory of the DPDK repo:

```
scripts/test-build.sh x86_64-native-linuxapp-gcc+next+shared+combined
```

The script usage is:

```
test-build.sh [-h] [-jX] [-s] [config1 [config2] ...]]
```

Where:

- `-h`: help, usage.
- `-jX`: use X parallel jobs in "make".
- `-s`: short test with only first config and without examples/doc.
- `config`: default config name plus config switches delimited with a `+` sign.

Examples of configs are:

```
x86_64-native-linuxapp-gcc
x86_64-native-linuxapp-gcc+next+shared+combined
x86_64-native-linuxapp-gcc+shared+next
x86_64-native-linuxapp-clang+shared+combined
i686-native-linuxapp-gcc+combined
```

The builds can be modifies via the following environmental variables:

- `DPDK_BUILD_TEST_CONFIGS` (target1+option1+option2 target2)
- `DPDK_DEP_CFLAGS`
- `DPDK_DEP_LDFLAGS`
- `DPDK_DEP_MOFED` (y/[n])
- `DPDK_DEP_PCAP` (y/[n])
- `DPDK_NOTIFY` (notify-send)

These can be set from the command line or in the config files shown above in the *Checking the Patches*.

The recommended configurations and options to test compilation prior to submitting patches are:

```
x86_64-native-linuxapp-gcc+shared+next
x86_64-native-linuxapp-clang+shared+combined
i686-native-linuxapp-gcc+combined
```

```
export DPDK_DEP_ZLIB=y
export DPDK_DEP_PCAP=y
export DPDK_DEP_SSL=y
```

11.5.9 Sending Patches

Patches should be sent to the mailing list using `git send-email`. You can configure an external SMTP with something like the following:

```
[sendemail]
    smtpuser = name@domain.com
    smtpserver = smtp.domain.com
    smtpserverport = 465
    smtpencryption = ssl
```

See the Git send-email documentation for more details.

The patches should be sent to `dev@dpdk.org`. If the patches are a change to existing files then you should send them TO the maintainer(s) and CC `dev@dpdk.org`. The appropriate maintainer can be found in the `MAINTAINERS` file:

```
git send-email --to maintainer@some.org --cc dev@dpdk.org 000*.patch
```

New additions can be sent without a maintainer:

```
git send-email --to dev@dpdk.org 000*.patch
```

You can test the emails by sending it to yourself or with the `--dry-run` option.

If the patch is in relation to a previous email thread you can add it to the same thread using the Message ID:

```
git send-email --to dev@dpdk.org --in-reply-to <1234-foo@bar.com> 000*.patch
```

The Message ID can be found in the raw text of emails or at the top of each Patchwork patch, for example. Shallow threading (`--thread --no-chain-reply-to`) is preferred for a patch series.

Once submitted your patches will appear on the mailing list and in Patchwork.

Experienced committers may send patches directly with `git send-email` without the `git format-patch` step. The options `--annotate` and `confirm = always` are recommended for checking patches before sending.

11.5.10 The Review Process

The more work you put into the previous steps the easier it will be to get a patch accepted.

The general cycle for patch review and acceptance is:

1. Submit the patch.
2. Check the automatic test reports in the coming hours.
3. Wait for review comments. While you are waiting review some other patches.
4. Fix the review comments and submit a `v n+1` patchset:

   ```
   git format-patch -3 -v 2
   ```

5. Update Patchwork to mark your previous patches as “Superseded”.

6. If the patch is deemed suitable for merging by the relevant maintainer(s) or other developers they will `ack` the patch with an email that includes something like:

   ```
   Acked-by: Alex Smith <alex.smith@example.com>
   ```

   Note: When acking patches please remove as much of the text of the patch email as possible. It is generally best to delete everything after the `Signed-off-by:` line.

7. Having the patch `Reviewed-by:` and/or `Tested-by:` will also help the patch to be accepted.

8. If the patch isn't deemed suitable based on being out of scope or conflicting with existing functionality it may receive a `nack`. In this case you will need to make a more convincing technical argument in favor of your patches.

9. In addition a patch will not be accepted if it doesn't address comments from a previous version with fixes or valid arguments.

10. Acked patches will be merged in the current or next merge window.

11.6 Patch Cheatsheet

Fig. 11.1: Cheat sheet for submitting patches to dev@dpdk.org