Getting Started Guide for Linux

Release 2.1.0

August 17, 2015

CONTENTS

August 17, 2015

Contents

CHAPTER

ONE

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 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.

Note: These documents are available for download as a separate documentation package at the same location as the DPDK code package.

CHAPTER

TWO

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*.

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 *Chapter 5. Enabling Additional Functionality* for more information on the required changes.

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
- 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. This release has been tested using version 12.1.
- 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.

2.3 Running DPDK Applications

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

2.3.1 System Software

Required:

- Kernel version >= 2.6.33

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

  ```
  uname -r
  ```

For details of the patches needed to use the DPDK with earlier kernel versions, see the DPDK FAQ included in the *DPDK Release Notes*. Note also that Red hat* Linux* 6.2 and 6.3 uses a 2.6.32 kernel that already has all the necessary patches applied.

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

  The version can be checked using the ldd –version command. A sample output is shown below:

  ```
  # ldd --version

  ldd (GNU libc) 2.14.90
  Copyright (C) 2011 Free Software Foundation, Inc.
  This is free software; see the source for copying conditions. There is NO
  warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
  Written by Roland McGrath and Ulrich Drepper.
  ```

- 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 *Section 5.1 High Precision Event Timer (HPET) Functionality* for more details.

2.3.2 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 in Section 2.3). 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
```

2.3.3 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.

CHAPTER

THREE

COMPILING THE DPDK TARGET FROM SOURCE

Note: Parts of this process can also be done using the setup script described in Chapter 6 of this document.

3.1 Install the DPDK and Browse Sources

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

```
user@host:~$ unzip DPDK-<version>.zip
user@host:~$ cd DPDK-<version>
user@host:~/DPDK-<version>$ ls
app/   config/   drivers/   examples/   lib/   LICENSE.GPL   LICENSE.LGPL   Makefile   mk/   s
```

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

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 compile all 64-bit targets using gcc, use:

```
make install T=x86_64*gcc
```

To compile all 64-bit targets using both gcc and icc, use:

```
make install T=x86_64-*
```

Note: The wildcard operator (*) can be used to create multiple targets at the same time.

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.

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
```

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.

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).

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:

```
root@host:DPDK# ./tools/dpdk_nic_bind.py --status

Network devices using DPDK-compatible driver
============================================
0000:82:00.0 '82599EB 10-Gigabit SFI/SFP+ Network Connection' drv=uio_pci_generic unused=ixgbe
0000:82:00.1 '82599EB 10-Gigabit SFI/SFP+ Network Connection' drv=uio_pci_generic unused=ixgbe

Network devices using kernel driver
===================================
0000:04:00.0 'I350 Gigabit Network Connection' if=em0 drv=igb unused=uio_pci_generic *Active*
0000:04:00.1 'I350 Gigabit Network Connection' if=eth1 drv=igb unused=uio_pci_generic
0000:04:00.2 'I350 Gigabit Network Connection' if=eth2 drv=igb unused=uio_pci_generic
0000:04:00.3 'I350 Gigabit Network Connection' 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:

```
root@host:DPDK# ./tools/dpdk_nic_bind.py --bind=uio_pci_generic 04:00.1
```

or, alternatively,

```
root@host:DPDK# ./tools/dpdk_nic_bind.py --bind=uio_pci_generic eth1
```

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

```
root@host:DPDK# ./tools/dpdk_nic_bind.py --bind=ixgbe 82:00.0
```

CHAPTER

FOUR

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 in **Chapter 6** of this document.

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.

```
user@host:~/DPDK$ cd examples/helloworld/
user@host:~/DPDK/examples/helloworld$ export RTE_SDK=$HOME/DPDK
user@host:~/DPDK/examples/helloworld$ export RTE_TARGET=x86_64-native-linuxapp-gcc
user@host:~/DPDK/examples/helloworld$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map

user@host:~/DPDK/examples/helloworld$ 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.

```
user@host:~$ export RTE_SDK=/home/user/DPDK
user@host:~$ cp -r $(RTE_SDK)/examples/helloworld my_rte_app
user@host:~$ cd my_rte_app/
user@host:~$ export RTE_TARGET=x86_64-native-linuxapp-gcc
user@host:~/my_rte_app$ make
    CC main.o
    LD helloworld
    INSTALL-APP helloworld
    INSTALL-MAP helloworld.map
```

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 Section 3.5, 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
```

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 the -n options are 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):

```
user@target:~$ ./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.

4.2.1 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.

4.2.2 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.

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.

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.

CHAPTER

FIVE

ENABLING ADDITIONAL FUNCTIONALITY

5.1 High Precision Event Timer HPET) Functionality

5.1.1 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.

5.1.2 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.

5.1.3 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.

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.

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

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
```

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.

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.

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.

5.7.1 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_requ st_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

5.7.2 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.

5.7.3 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.

CHAPTER

SIX

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.

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.

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.

```
user@host:~/rte$ 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>>
```

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.

```
rte@rte-desktop:~/rte/examples$ cd helloworld/
rte@rte-desktop:~/rte/examples/helloworld$ make
CC main.o
LD helloworld
INSTALL-APP helloworld
INSTALL-MAP helloworld.map

rte@rte-desktop:~/rte/examples/helloworld$ 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
```