DPDK

DATA PLANE DEVELOPMENT KIT

Getting Started Guide for Linux

Release 20.08.0

Aug 08, 2020

CHAPTER

ONE

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. The document describes how to compile and run a DPDK application in a Linux application (linux) 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 (linux) 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.

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, BIOS setting changes may be needed. Consult the section on *Enabling Additional Functionality* for more information on the required changes.

Note: If UEFI secure boot is enabled, the Linux kernel may disallow the use of UIO on the system. Therefore, devices for use by DPDK should be bound to the `vfio-pci` kernel module rather than `igb_uio` or `uio_pci_generic`. For more details see *Binding and Unbinding Network Ports to/from the Kernel Modules*.

2.2 Compilation of the DPDK

Required Tools and Libraries:

Note: The setup commands and installed packages needed on various systems may be different. For details on Linux distributions and the versions tested, please consult the DPDK Release Notes.

- General development tools including `make`, and a supported C compiler such as `gcc` (version 4.9+) or `clang` (version 3.4+).
  - For RHEL/Fedora systems these can be installed using `dnf groupinstall "Development Tools"`
  - For Ubuntu/Debian systems these can be installed using `apt install build-essential`
- Python, recommended version 3.5+.

  - Python v3.5+ is needed to build DPDK using meson and ninja
  - Python 2.7+ or 3.2+, to use various helper scripts included in the DPDK package.
- Meson (version 0.47.1+) and ninja
  - `meson` & `ninja-build` packages in most Linux distributions
  - If the packaged version is below the minimum version, the latest versions can be installed from Python's "pip" repository: `pip3 install meson ninja`
- Library for handling NUMA (Non Uniform Memory Access).
  - `numactl-devel` in RHEL/Fedora;
  - `libnuma-dev` in Debian/Ubuntu;
- Linux kernel headers or sources required to build kernel modules.

Note: Please ensure that the latest patches are applied to third party libraries and software to avoid any known vulnerabilities.

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.

Additional Libraries

A number of DPDK components, such as libraries and poll-mode drivers (PMDs) have additional dependencies. For DPDK builds using meson, the presence or absence of these dependencies will be automatically detected enabling or disabling the relevant components appropriately.

For builds using make, these components are disabled in the default configuration and need to be enabled manually by changing the relevant setting to "y" in the build configuration file i.e. the `.config` file in the build folder.

In each case, the relevant library development package (`-devel` or `-dev`) is needed to build the DPDK components.

For libraries the additional dependencies include:

- libarchive: for some unit tests using tar to get their resources.
- libelf: to compile and use the bpf library.

For poll-mode drivers, the additional dependencies for each driver can be found in that driver's documentation in the relevant DPDK guide document, e.g. ../nics/index

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 >= 3.16

  The kernel version required is based on the oldest long term stable kernel available at kernel.org when the DPDK version is in development. Compatibility for recent distribution kernels will be kept, notably RHEL/CentOS 7.

  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:

  - 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.

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 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/admin-guide/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
```

CHAPTER

THREE

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.

3.1 Uncompress DPDK and Browse Sources

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

```
tar xJf dpdk-<version>.tar.xz
cd dpdk-<version>
```

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, buildtools, mk: Framework-related makefiles, scripts and configuration

3.2 Compiling and Installing DPDK System-wide

DPDK can be configured, built and installed on your system using the tools `meson` and `ninja`.

Note: The older makefile-based build system used in older DPDK releases is still present and its use is described in section *Installation of DPDK Target Environment using Make*.

3.2.1 DPDK Configuration

To configure a DPDK build use:

```
meson <options> build
```

where "build" is the desired output build directory, and "<options>" can be empty or one of a number of meson or DPDK-specific build options, described later in this section. The configuration process will finish with a summary of what DPDK libraries and drivers are to be built and installed, and for each

item disabled, a reason why that is the case. This information can be used, for example, to identify any missing required packages for a driver.

Once configured, to build and then install DPDK system-wide use:

```
cd build
ninja
ninja install
ldconfig
```

The last two commands above generally need to be run as root, with the *ninja install* step copying the built objects to their final system-wide locations, and the last step causing the dynamic loader *ld.so* to update its cache to take account of the new objects.

Note: On some linux distributions, such as Fedora or Redhat, paths in */usr/local* are not in the default paths for the loader. Therefore, on these distributions, */usr/local/lib* and */usr/local/lib64* should be added to a file in */etc/ld.so.conf.d/* before running *ldconfig*.

3.2.2 Adjusting Build Options

DPDK has a number of options that can be adjusted as part of the build configuration process. These options can be listed by running `meson configure` inside a configured build folder. Many of these options come from the "meson" tool itself and can be seen documented on the Meson Website.

For example, to change the build-type from the default, "debugoptimized", to a regular "debug" build, you can either:

- pass `-Dbuildtype=debug` or `--buildtype=debug` to meson when configuring the build folder initially
- run `meson configure -Dbuildtype=debug` inside the build folder after the initial meson run.

Other options are specific to the DPDK project but can be adjusted similarly. To set the "max_lcores" value to 256, for example, you can either:

- pass `-Dmax_lcores=256` to meson when configuring the build folder initially
- run `meson configure -Dmax_lcores=256` inside the build folder after the initial meson run.

Some of the DPDK sample applications in the *examples* directory can be automatically built as part of a meson build too. To do so, pass a comma-separated list of the examples to build to the *-Dexamples* meson option as below:

```
meson -Dexamples=l2fwd,l3fwd build
```

As with other meson options, this can also be set post-initial-config using *meson configure* in the build directory. There is also a special value "all" to request that all example applications whose dependencies are met on the current system are built. When *-Dexamples=all* is set as a meson option, meson will check each example application to see if it can be built, and add all which can be built to the list of tasks in the ninja build configuration file.

3.2.3 Building Applications Using Installed DPDK

When installed system-wide, DPDK provides a pkg-config file `libdpdk.pc` for applications to query as part of their build. It's recommended that the pkg-config file be used, rather than hard-coding the parameters (cflags/ldflags) for DPDK into the application build process.

An example of how to query and use the pkg-config file can be found in the `Makefile` of each of the example applications included with DPDK. A simplified example snippet is shown below, where the target binary name has been stored in the variable `$(APP)` and the sources for that build are stored in `$(SRCS-y)`.

```
PKGCONF = pkg-config

CFLAGS += -O3 $(shell $(PKGCONF) --cflags libdpdk)
LDFLAGS += $(shell $(PKGCONF) --libs libdpdk)

$(APP): $(SRCS-y) Makefile
        $(CC) $(CFLAGS) $(SRCS-y) -o $@ $(LDFLAGS)
```

Note: Unlike with the older make build system, the meson system is not designed to be used directly from a build directory. Instead it is recommended that it be installed either system-wide or to a known location in the user's home directory. The install location can be set using the *–prefix* meson option (default: */usr/local*).

---

an equivalent build recipe for a simple DPDK application using meson as a build system is shown below:

```
project('dpdk-app', 'c')

dpdk = dependency('libdpdk')
sources = files('main.c')
executable('dpdk-app', sources, dependencies: dpdk)
```

3.3 Installation of DPDK Target Environment using Make

Note: The building of DPDK using make will be deprecated in a future release. It is therefore recommended that DPDK installation is done using meson and ninja as described above.

Get a native target environment automatically:

```
make defconfig O=mybuild
```

Note: 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.

Or get a specific target environment:

```
make config T=x86_64-native-linux-gcc O=mybuild
```

The format of a DPDK target is "ARCH-MACHINE-EXECENV-TOOLCHAIN". Available targets can be found with:

```
make help
```

Customize the target configuration in the generated `.config` file. Example for enabling the pcap PMD:

```
sed -ri 's,(PMD_PCAP=).*,\1y,' mybuild/.config
```

Compile the target:

```
make -j4 O=mybuild
```

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.

Install the target in a separate directory:

```
make install O=mybuild DESTDIR=myinstall prefix=
```

The environment is ready to build a DPDK application:

```
RTE_SDK=$(pwd)/myinstall/share/dpdk RTE_TARGET=x86_64-native-linux-gcc make -C myapp
```

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

3.4 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 applications are built under the app directory, which may be used for testing. A kmod directory is also present that contains kernel modules which may be loaded if needed.

CHAPTER

FOUR

CROSS COMPILE DPDK FOR ARM64

This chapter describes how to cross compile DPDK for ARM64 from x86 build hosts.

Note: Whilst it is recommended to natively build DPDK on ARM64 (just like with x86), it is also possible to cross-build DPDK for ARM64. An ARM64 cross compile GNU toolchain is used for this.

4.1 Obtain the cross tool chain

The latest cross compile tool chain can be downloaded from: https://developer.arm.com/open-source/gnu-toolchain/gnu-a/downloads.

It is always recommended to check and get the latest compiler tool from the page and use it to generate better code. As of this writing 8.3-2019.03 is the newest, the following description is an example of this version.

```
wget https://developer.arm.com/-/media/Files/downloads/gnu-a/8.3-2019.03/binrel/gcc-arm-8.3-201
```

4.2 Unzip and add into the PATH

```
tar -xvf gcc-arm-8.3-2019.03-x86_64-aarch64-linux-gnu.tar.xz
export PATH=$PATH:<cross_install_dir>/gcc-arm-8.3-2019.03-x86_64-aarch64-linux-gnu/bin
```

Note: For the host requirements and other info, refer to the release note section: https://releases.linaro.org/components/toolchain/binaries/

4.3 Getting the prerequisite library

NUMA is required by most modern machines, not needed for non-NUMA architectures.

Note: For compiling the NUMA lib, run libtool –version to ensure the libtool version >= 2.2, otherwise the compilation will fail with errors.

```
git clone https://github.com/numactl/numactl.git
cd numactl
git checkout v2.0.13 -b v2.0.13
./autogen.sh
autoconf -i
./configure --host=aarch64-linux-gnu CC=aarch64-linux-gnu-gcc --prefix=<numa install dir>
make install
```

The numa header files and lib file is generated in the include and lib folder respectively under <numa install dir>.

4.4 Augment the cross toolchain with NUMA support

Note: This way is optional, an alternative is to use extra CFLAGS and LDFLAGS, depicted in *Cross Compiling DPDK using Meson* below.

Copy the NUMA header files and lib to the cross compiler's directories:

```
cp <numa_install_dir>/include/numa*.h <cross_install_dir>/gcc-arm-8.3-2019.03-x86_64-aarch64-li
cp <numa_install_dir>/lib/libnuma.a <cross_install_dir>/gcc-arm-8.3-2019.03-x86_64-aarch64-linu
cp <numa_install_dir>/lib/libnuma.so <cross_install_dir>/gcc-arm-8.3-2019.03-x86_64-aarch64-lir
```

4.5 Cross Compiling DPDK using Meson

Meson depends on pkgconfig to find the dependencies. The package `pkg-config-aarch64-linux-gnu` is required for aarch64. To install it in Ubuntu:

```
sudo apt-get install pkg-config-aarch64-linux-gnu
```

To cross-compile DPDK on a desired target machine we can use the following command:

```
meson cross-build --cross-file <target_machine_configuration>
ninja -C cross-build
```

For example if the target machine is arm64 we can use the following command:

```
meson arm64-build --cross-file config/arm/arm64_armv8_linux_gcc
ninja -C arm64-build
```

4.6 Configure and Cross Compile DPDK using Make

To configure a build, choose one of the target configurations, like arm64-dpaa-linux-gcc and arm64-thunderx-linux-gcc.

```
make config T=arm64-armv8a-linux-gcc
```

To cross-compile, without compiling the kernel modules, use the following command:

```
make -j CROSS=aarch64-linux-gnu- CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n
```

To cross-compile, including the kernel modules, the kernel source tree needs to be specified by setting RTE_KERNELDIR:

```
make -j CROSS=aarch64-linux-gnu- RTE_KERNELDIR=<kernel_src_rootdir> CROSS_COMPILE=aarch64-linux
```

To compile for non-NUMA targets, without compiling the kernel modules, use the following command:

```
make -j CROSS=aarch64-linux-gnu- CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n CONFIG_RTE_LIBR
```

Note: 1. EXTRA_CFLAGS and EXTRA_LDFLAGS should be added to include the NUMA headers and link the library respectively, if the above step *Augment the cross toolchain with NUMA support* was skipped therefore the toolchain was not augmented with NUMA support.

2. "-isystem <numa_install_dir>/include" should be add to EXTRA_CFLAGS, otherwise the numa.h file will get a lot of compiling errors of Werror=cast-qual, Werror=strict-prototypes and Werror=old-style-definition.

An example is given below:

```
make -j CROSS=aarch64-linux-gnu- CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n EXTRA_CFLAGS="-
```

CHAPTER

FIVE

LINUX DRIVERS

Different PMDs may require different kernel drivers in order to work properly. Depends on the PMD being used, a corresponding kernel driver should be load and bind to the network ports.

5.1 UIO

A small kernel module to set up the device, map device memory to user-space and register interrupts. 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
```

Note: `uio_pci_generic` module doesn't support the creation of virtual functions.

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: `igb_uio` module is disabled by default starting from `DPDK v20.02`. To build it, the config option `CONFIG_RTE_EAL_IGB_UIO` should be enabled. It is planned to move `igb_uio` module to a different git repository.

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`.

Note: If UEFI secure boot is enabled, the Linux kernel may disallow the use of UIO on the system. Therefore, devices for use by DPDK should be bound to the `vfio-pci` kernel module rather than `igb_uio` or `uio_pci_generic`. For more details see *Binding and Unbinding Network Ports to/from the Kernel Modules* below.

Note: If the devices used for DPDK are bound to the `uio_pci_generic` kernel module, please make sure that the IOMMU is disabled or passthrough. One can add `intel_iommu=off` or

`amd_iommu=off` or `intel_iommu=on iommu=pt` in GRUB command line on x86_64 systems, or add `iommu.passthrough=1` on aarch64 system.

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

5.2 VFIO

A more robust and secure driver in compare to the `UIO`, relying on IOMMU protection. To 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.

The `vfio-pci` module since Linux version 5.7 supports the creation of virtual functions. After the PF is bound to vfio-pci module, the user can create the VFs by sysfs interface, and these VFs are bound to vfio-pci module automatically.

When the PF is bound to vfio-pci, it has initial VF token generated by random. For security reason, this token is write only, the user can't read it from the kernel directly. To access the VF, the user needs to start the PF with token parameter to setup a VF token in UUID format, then the VF can be accessed with this new token.

Since the `vfio-pci` module uses the VF token as internal data to provide the collaboration between SR-IOV PF and VFs, so DPDK can use the same VF token for all PF devices which bound to one application. This VF token can be specified by the EAL parameter `--vfio-vf-token`.

```
1. Generate the VF token by uuid command
    14d63f20-8445-11ea-8900-1f9ce7d5650d

2. sudo modprobe vfio-pci enable_sriov=1

2. ./usertools/dpdk-devbind.py -b vfio-pci 0000:86:00.0

3. echo 2 > /sys/bus/pci/devices/0000:86:00.0/sriov_numvfs

4. Start the PF:
    ./x86_64-native-linux-gcc/app/testpmd -l 22-25 -n 4 -w 86:00.0 \
     --vfio-vf-token=14d63f20-8445-11ea-8900-1f9ce7d5650d --file-prefix=pf -- -i

5. Start the VF:
    ./x86_64-native-linux-gcc/app/testpmd -l 26-29 -n 4 -w 86:02.0 \
     --vfio-vf-token=14d63f20-8445-11ea-8900-1f9ce7d5650d --file-prefix=vf0 -- -i
```

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

Note: `vfio-pci` module doesn't support the creation of virtual functions before Linux version 5.7.

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 dpdk-setup.sh and located in the usertools directory).

Note: VFIO can be used without IOMMU. While this is just as unsafe as using UIO, it does make it possible for the user to keep the degree of device access and programming that VFIO has, in situations where IOMMU is not available.

5.3 Bifurcated Driver

PMDs which use the bifurcated driver co-exists with the device kernel driver. On such model the NIC is controlled by the kernel, while the data path is performed by the PMD directly on top of the device.

Such model has the following benefits:

- It is secure and robust, as the memory management and isolation is done by the kernel.
- It enables the user to use legacy linux tools such as `ethtool` or `ifconfig` while running DPDK application on the same network ports.
- It enables the DPDK application to filter only part of the traffic, while the rest will be directed and handled by the kernel driver. The flow bifurcation is performed by the NIC hardware. As an example, using flow_isolated_mode allows to choose strictly what is received in DPDK.

More about the bifurcated driver can be found in Mellanox Bifurcated DPDK PMD.

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

Note: PMDs Which use the bifurcated driver should not be unbind from their kernel drivers. this section is for PMDs which use the UIO or VFIO drivers.

As of release 1.4, DPDK applications no longer automatically unbind all supported network ports from the kernel driver in use. Instead, in case the PMD being used use the UIO or VFIO drivers, 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. For such PMDs, any network ports under Linux* control will be ignored and cannot be used by the application.

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-devbind.py is provided in the usertools 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-devbind.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-devbind.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:

```
./usertools/dpdk-devbind.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:

```
./usertools/dpdk-devbind.py --bind=uio_pci_generic 04:00.1
```

or, alternatively,

```
./usertools/dpdk-devbind.py --bind=uio_pci_generic eth1
```

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

```
./usertools/dpdk-devbind.py --bind=ixgbe 82:00.0
```

CHAPTER

SIX

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.

6.1 Compiling a Sample Application

Once an DPDK target environment directory has been created (such as `x86_64-native-linux-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-linux-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-linux-gcc

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

6.2 Running a Sample Application

Warning: Before running the application make sure:

- Hugepages setup is done.
- Any kernel driver being used is loaded.
- In case needed, ports being used by the application should be bound to the corresponding kernel driver.

refer to *Linux Drivers* for more details.

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 | -l CORELIST] [-n NUM] [-b <domain:bus:devid.func>] \
          [--socket-mem=MB,...] [-d LIB.so|DIR] [-m MB] [-r NUM] [-v] [--file-prefix] \
          [--proc-type <primary|secondary|auto>]
```

The EAL options are as follows:

- `-c COREMASK` or `-l CORELIST`: An hexadecimal bit mask of the cores to run on. Note that core numbering can change between platforms and should be determined beforehand. The corelist is a set of core numbers instead of a bitmap core mask.
- `-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. In dynamic memory mode, this memory will also be pinned (i.e. not released back to the system until application closes).
- `--socket-limit`: Limit maximum memory available for allocation on each socket. Does not support legacy memory mode.

- `-d`: Add a driver or driver directory to be loaded. The application should use this option to load the pmd drivers that are built as shared libraries.
- `-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.
- `mbuf-pool-ops-name`: Pool ops name for mbuf to use.
- `--file-prefix`: The prefix text used for hugepage filenames.
- `--proc-type`: The type of process instance.
- `--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).
- `--legacy-mem`: Run DPDK in legacy memory mode (disable memory reserve/unreserve at runtime, but provide more IOVA-contiguous memory).
- `--single-file-segments`: Store memory segments in fewer files (dynamic memory mode only - does not affect legacy memory mode).

The `-c` or `-l` 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 -l 0-3 -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.

6.2.1 Logical Core Use by Applications

The coremask (-c 0x0f) or corelist (-l 0-3) parameter is always mandatory for DPDK applications. Each bit of the mask corresponds to the equivalent logical core number as reported by Linux. The preferred corelist option is a cleaner method to define cores to be used. 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/corelist 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/corelist.

6.2.2 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.

6.3 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.

6.4 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

SEVEN

EAL PARAMETERS

This document contains a list of all EAL parameters. These parameters can be used by any DPDK application running on Linux.

7.1 Common EAL parameters

The following EAL parameters are common to all platforms supported by DPDK.

7.1.1 Lcore-related options

- `-c <core mask>`

  Set the hexadecimal bitmask of the cores to run on.

- `-l <core list>`

  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 <core map>`

  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.

Note: At a given instance only one core option `--lcores`, `-l` or `-c` can be used.

- `--master-lcore <core ID>`

  Core ID that is used as master.

- `-s <service core mask>`

  Hexadecimal bitmask of cores to be used as service cores.

7.1.2 Device-related options

- `-b,--pci-blacklist <[domain:]bus:devid.func>`

  Blacklist a PCI device to prevent EAL from using it. Multiple -b options are allowed.

Note: PCI blacklist cannot be used with `-w` option.

- `-w,--pci-whitelist <[domain:]bus:devid.func>`

  Add a PCI device in white list.

Note: PCI whitelist cannot be used with `-b` option.

- `--vdev <device arguments>`

  Add a virtual device using the format:

  ```
  <driver><id>[,key=val, ...]
  ```

  For example:

  ```
  --vdev 'net_pcap0,rx_pcap=input.pcap,tx_pcap=output.pcap'
  ```

- `-d <path to shared object or directory>`

  Load external drivers. An argument can be a single shared object file, or a directory containing multiple driver shared objects. Multiple -d options are allowed.

- `--no-pci`

  Disable PCI bus.

7.1.3 Multiprocessing-related options

- `--proc-type <primary|secondary|auto>`

  Set the type of the current process.

- `--base-virtaddr <address>`

  Attempt to use a different starting address for all memory maps of the primary DPDK process. This can be helpful if secondary processes cannot start due to conflicts in address map.

7.1.4 Memory-related options

- `-n <number of channels>`

  Set the number of memory channels to use.

- `-r <number of ranks>`

  Set the number of memory ranks (auto-detected by default).

- `-m <megabytes>`

  Amount of memory to preallocate at startup.

- `--in-memory`

  Do not create any shared data structures and run entirely in memory. Implies `--no-shconf` and (if applicable) `--huge-unlink`.

- `--iova-mode <pa|va>`

  Force IOVA mode to a specific value.

7.1.5 Debugging options

- `--no-shconf`

  No shared files created (implies no secondary process support).

- `--no-huge`

  Use anonymous memory instead of hugepages (implies no secondary process support).

- `--log-level <type:val>`

  Specify log level for a specific component. For example:

  ```
  --log-level lib.eal:debug
  ```

  Can be specified multiple times.

- `--trace=<regex-match>`

  Enable trace based on regular expression trace name. By default, the trace is disabled. User must specify this option to enable trace. For example:

  Global trace configuration for EAL only:

  ```
  --trace=eal
  ```

  Global trace configuration for ALL the components:

  ```
  --trace=.*
  ```

  Can be specified multiple times up to 32 times.

- `--trace-dir=<directory path>`

  Specify trace directory for trace output. For example:

  Configuring `/tmp/` as a trace output directory:

  ```
  --trace-dir=/tmp
  ```

  By default, trace output will created at `home` directory and parameter must be specified once only.

- `--trace-bufsz=<val>`

  Specify maximum size of allocated memory for trace output for each thread. Valid unit can be either `B` or `K` or `M` for `Bytes`, `KBytes` and `MBytes` respectively. For example:

  Configuring `2MB` as a maximum size for trace output file:

  ```
  --trace-bufsz=2M
  ```

  By default, size of trace output file is `1MB` and parameter must be specified once only.

- `--trace-mode=<o[verwrite] | d[iscard] >`

  Specify the mode of update of trace output file. Either update on a file can be wrapped or discarded when file size reaches its maximum limit. For example:

  To `discard` update on trace output file:

  ```
  --trace-mode=d or --trace-mode=discard
  ```

  Default mode is `overwrite` and parameter must be specified once only.

7.1.6 Other options

- `-h, --help`

  Display help message listing all EAL parameters.

- `-v`

  Display the version information on startup.

- `mbuf-pool-ops-name`:

  Pool ops name for mbuf to use.

- `--telemetry`:

  Enable telemetry (enabled by default).

- `--no-telemetry`:

  Disable telemetry.

7.2 Linux-specific EAL parameters

In addition to common EAL parameters, there are also Linux-specific EAL parameters.

7.2.1 Device-related options

- `--create-uio-dev`

  Create `/dev/uioX` files for devices bound to igb_uio kernel driver (usually done by the igb_uio driver itself).

- `--vmware-tsc-map`

  Use VMware TSC map instead of native RDTSC.

- `--no-hpet`

  Do not use the HPET timer.

- `--vfio-intr <legacy|msi|msix>`

  Use specified interrupt mode for devices bound to VFIO kernel driver.

- `--vfio-vf-token <uuid>`

  Use specified VF token for devices bound to VFIO kernel driver.

7.2.2 Multiprocessing-related options

- `--file-prefix <prefix name>`

  Use a different shared data file prefix for a DPDK process. This option allows running multiple independent DPDK primary/secondary processes under different prefixes.

7.2.3 Memory-related options

- `--legacy-mem`

  Use legacy DPDK memory allocation mode.

- `--socket-mem <amounts of memory per socket>`

  Preallocate specified amounts of memory per socket. The parameter is a comma-separated list of values. For example:

  ```
  --socket-mem 1024,2048
  ```

  This will allocate 1 gigabyte of memory on socket 0, and 2048 megabytes of memory on socket 1.

- `--socket-limit <amounts of memory per socket>`

  Place a per-socket upper limit on memory use (non-legacy memory mode only). 0 will disable the limit for a particular socket.

- `--single-file-segments`

  Create fewer files in hugetlbfs (non-legacy mode only).

- `--huge-dir <path to hugetlbfs directory>`

  Use specified hugetlbfs directory instead of autodetected ones.

- `--huge-unlink`

  Unlink hugepage files after creating them (implies no secondary process support).

- `--match-allocations`

  Free hugepages back to system exactly as they were originally allocated.

7.2.4 Other options

- `--syslog <syslog facility>`

  Set syslog facility. Valid syslog facilities are:

  ```
  auth
  cron
  daemon
  ftp
  kern
  lpr
  mail
  news
  syslog
  user
  uucp
  ```

```
local0
local1
local2
local3
local4
local5
local6
local7
```

CHAPTER

EIGHT

ENABLING ADDITIONAL FUNCTIONALITY

8.1 High Precision Event Timer (HPET) Functionality

8.1.1 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.

8.1.2 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.

8.1.3 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.

8.2 Running DPDK Applications Without Root Privileges

Note: The instructions below will allow running DPDK as non-root with older Linux kernel versions. However, since version 4.0, the kernel does not allow unprivileged processes to read the physical address information from the pagemaps file, making it impossible for those processes to use HW devices which require physical addresses

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.

8.3 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
```

8.4 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
```

8.5 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.

8.6 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`.

CHAPTER

NINE

QUICK START SETUP SCRIPT

The dpdk-setup.sh script, found in the usertools 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.

9.1 Script Organization

The dpdk-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.

9.2 Use Cases

The following are some example of how to use the dpdk-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 dpdk-setup.sh script should be run with root privileges.

```
source usertools/dpdk-setup.sh

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

RTE_SDK exported as /home/user/rte

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

Step 1: Select the DPDK environment to build

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

[1] i686-native-linux-gcc

[2] i686-native-linux-icc

[3] ppc_64-power8-linux-gcc

[4] x86_64-native-freebsd-clang

[5] x86_64-native-freebsd-gcc

[6] x86_64-native-linux-clang

[7] x86_64-native-linux-gcc

[8] x86_64-native-linux-icc

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

Step 2: Setup linux 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 linux 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-linux-gcc` DPDK library.

```
Option: 9

================== Installing x86_64-native-linux-gcc
```

```
Configuration done
== Build lib
...
Build complete
RTE_TARGET exported as x86_64-native-linux-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>>
```

9.3 Applications

Once the user has run the dpdk-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 -l 0-3 -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
```

CHAPTER

TEN

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.

10.1 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.

10.1.1 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.

10.1.2 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. Establish the steady state for the system, consider reviewing BIOS settings desired for best performance characteristic e.g. optimize for performance or energy efficiency.
2. Match the BIOS settings to the needs of the application you are testing.
3. Typically, **Performance** as the CPU Power and Performance policy is a reasonable starting point.
4. Consider using Turbo Boost to increase the frequency on cores.
5. Disable all virtualization options when you test the physical function of the NIC, and turn on VT-d if you wants to use VFIO.

10.1.3 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
```

10.2 Configurations before running DPDK

1. Reserve huge pages. See the earlier section on *Use of Hugepages in the Linux Environment* for more details.

```
# 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 the DPDK `cpu_layout` utility:

```
cd dpdk_folder

usertools/cpu_layout.py
```

   Or run `lscpu` to check 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. Check which kernel drivers needs to be loaded and whether there is a need to unbind the network ports from their kernel drivers. More details about DPDK setup and Linux kernel requirements see *Compiling the DPDK Target from Source* and *Linux Drivers*.