DPL Runtime Service

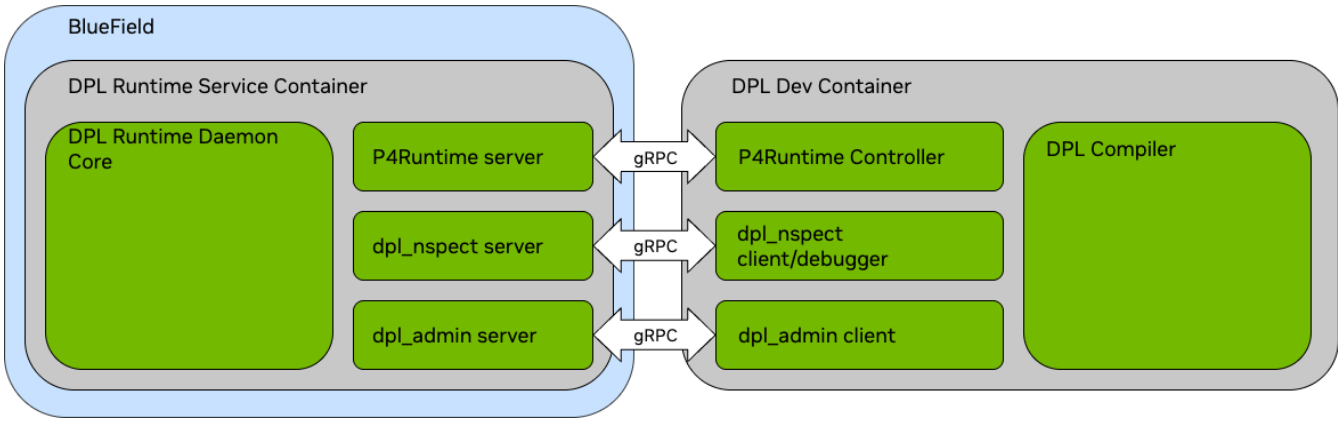
Table of contents

DPL Runtime Service

The DPL Runtime Service is an NVIDIA® BlueField® service that implements the backend functionality to manage and program the DPU datapath. It is broken down into 4 components:

- DPL Runtime Core. This component serves as the manager of the runtime system, accepting requests from the various servers and is responsible for managing resources and programming the hardware.
- dpl_nspect server. This gRPC based server receives requests from the dpl_nspect client application for debug information, and transmits debug packets to the DPL debugger.
- dpl_admin server. This gRPC based server receives administrative requests about the daemon's core state and can control its configuration dynamically.
- P4Runtime server. This gRPC based server binds the P4Runtime protobuf interface to the underlying hardware driver APIs. The P 4Runtime server listens on TCP port 9559, which is the port that has been allocated by IANA for the P4Runtime service. The server allows a P4 Controller to connect over gRPC so that it can set the `ForwardingPipelineConfig` , which installs and loads into hardware the compiled DPL program output and the associated P4Info metadata. Furthermore, the controller can query the target for the `ForwardingPipelineConfig` to retrieve the device config and the P4Info, as well as performing P4 table maintenance (which were defined in the DPL program source code).

High level system illustration



Requirements

Supported Platforms

The following NVIDIA® BlueField® DPUs and SuperNICs are supported with DPL:

NVIDIA SKU	**Legacy OPN**	**PSID**	**Description**
900-9D3B6-00CV-AA0	N/A	MT_0000000884	BlueField-3 B3220 P-Series FHHL DPU; 200GbE (default mode) / NDR200 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
900-9D3B6-00SV-AA0	N/A	MT_0000000965	BlueField-3 B3220 P-Series FHHL DPU; 200GbE (default mode) / NDR200 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Disabled
900-9D3B6-00CC-AA0	N/A	MT_0000001024	BlueField-3 B3210 P-Series FHHL DPU; 100GbE (default mode) / HDR100 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Enabled
900-9D3B6-00SC-AA0	N/A	MT_0000001025	BlueField-3 B3210 P-Series FHHL DPU; 100GbE (default mode) / HDR100 IB; Dual-port QSFP112; PCIe Gen5.0 x16 with x16 PCIe extension option; 16 Arm cores; 32GB on-board DDR; integrated BMC; Crypto Disabled

The following NVIDIA® DPUs are supported with DOCA on the host:

- Bluefield-3 devices

Hardware Prerequisites

For the the system requirements, see DPU's hardware user guide .

Software Prerequisites

This quick start guide assumes that an NVIDIA® BlueField® networking platform has been installed on a server. The minimum version requirements are described in the table below:

OS / Product	Version
ubuntu	22.04
bf-bundle	2.10.0-xx

Installation Procedures

Software Versions

The firmware version must be 32.44.1000 or higher. The firmware is included in the BFB bundle, which can be downloaded from the NVIDIA Dev Zone. Select BlueField/BF-FW-Bundle/BFB. Installation example:

```
bfb-install --bfb bf-fwbundle-2.10.0-28_25.0-ubuntu-22.04-prod.bfb --rshim rshim0
```

At a minimum DOCA-networking 2.10.0 should be installed on the host. For complete DOCA installation instructions, please see DOCA Installation Guide for Linux.

The deployment guide has more information about this here: Setup DPU Management access and update BlueField-Bundle

Firmware settings

The following firmware settings are needed

- FLEX_PARSER_PROFILE_ENABLE=4
- PROG_PARSE_GRAPH=true
- SRIOV_EN=1

(SRIOV_EN=1 is only needed if you intend to use VFs)

For details on how to query and adjust these settings, please see Using mlxconfig.

Some changes require a firmware reset, please see mlxfwreset.

How to deploy and run the DPL Runtime Service container

The DPL Runtime Service runs on the DPU of BlueField, which requires that BlueField is in DPU mode. for details, see this page: BlueField Modes of Operation

The DPL Runtime Service is deployed from NGC, for detailed instructions, please refer to this page: Container Deployment

Both SR-IOV Virtual Functions (VFs) and Scalable Function (SFs) are supported. **VFs must be created on the host server before setting up the DPL Runtime Service in the DPU**.

Fetching the configuration files from NGC will create a directory named `dpl_rt_service_<version>` . Inside, you'll find `scripts/dpl_dpu_setup.sh`.

Running this script on the DPU will allow the usage of SR-IOV Virtual-Function interfaces and create the directory structure of the configuration files. In addition, the script will call the necessary `mlxconfig` commands needed for the DPL Runtime Service .

DPL Runtime Service Configuration files

When the image is installed, containers can be created to run the DPL Runtime Service . Before that, however, you must create the configuration files that specify which network interfaces are usable by the DPL program and their IDs, along with other system parameters.

These configuration files are best created on the DPU file system to make them persistent and shared with the container file system. Containers by default have their own separate file system unless specific paths are mounted in the container upon creation. Mounting the configuration path in the container is the preferred way to make sure the configuration is accessible to the DPL Runtime Service . For more details, see this page: Service Configuration

Supported Port Forwarding

The following table lists out the possible ingress and egress ports for a given packet that is processed by a BlueField pipeline (DPU mode):

Ingress Port	Egress Port			
	Wire Port P0	PF0hpf	pf0vf_n	pf0vf_m
Wire port P0	Allowed	Allowed	Allowed	Allowed
pf0hpf	Allowed	Disabled	Allowed	Allowed
pf0vf_n	Allowed	Allowed	Disabled	Allowed
pf0vf_m	Allowed	Allowed	Allowed	Disabled

Info

Anything that is allowed with SR-IOV Virtual Functions (VFs) in the table above is also allowed with Scalable Functions (SFs) .

To set the system in multiport e-switch mode, use the following command:

```
mlxconfig -d <pci>  s LAG_RESOURCE_ALLOCATION=1
mlxfwreset -d <pci> -y -l 3 --sync 1 r
devlink dev param set <pci> name esw_multiport value 1 cmode
runtime
```

This mode will allow DPL to process packets on both wire ports P0 and P1. Note that the devlink command is not persistent across reboots.

DPL Dev Container

Please refer to the DPL Installation Guide documentation.

Implement your own P4Runtime Controller

The most powerful use-cases involve a controller that responds to live traffic and adapt the rules accordingly. This can be achieved by implementing your own P4Runtime client application that connects to the gRPC server of the running daemon. The controller provided in the DPL Dev Container is enough to load a DPL program but you can expect

the best performance if you create an optimized client that is running directly on the Arm subsystem inside the BlueField DPU.

The necessary .proto files can be found here: p4runtime/proto/p4 at v1.3.0 · p4lang/p4runtime · GitHub In addition, it's possible to extract the files from a running DPL Runtime Service container. See inside the container in the directory `/opt/mellanox/third_party/dpl_rt_service/p4runtime/`.

This user guide does not explain how to compile and build the .proto files into C++ code and link them into an executable. Please see the information online for these open-source projects. For example, additional information about that can be found here: C++ | gRPC

Troubleshooting

Kubelet logs can be viewed with command: `sudo journalctl -u kubelet --since -5m` Installed (pulled) images can be observed with command: `sudo crictl images` Created pods can be observed with command: `sudo crictl pods`

Running containers pods can be observed with command: `sudo crictl ps` The log of the DPL Runtime Service is available in `/var/log/doca/dpl_rt_service/dpl_rtd.log`

Alternatively, the DPL Runtime Service logs may be observed from your development environment by using the DPL Admin Control tool.

Things to check:

- Did you create VFs in the host before setting up the DPU?
- Ensure that the BlueField-3 DPU is in `DPU MODE` mode. Refer to BlueField Modes of Operation
- Did you create configuration files? In the correct place?
- Did you follow the naming convention for the configuration files?
- Does the device ID in the configuration file match the file name?
- Did you get all the interface names correct?
- Is your firmware version up to date? (See mlxup-mft)

- Make sure the link type is set to ETH in step 5 of the "Installing Software on Host" section in the DOCA Installation Guide for Linux.

Container Deployment

Preparing the BlueField DPU

Set BlueField to DPU Mode

BlueField must run in DPU mode to use the DPL Runtime Service . For details how to change modes, see here: BlueField Modes of Operation.

Determine Your BlueField Variant

Your BlueField may be Installed in a host server or it may be a standalone server.

If your BlueField is a standalone server, please ignore the parts that mention the host server or SR-IOV.

You may still use Scalable Functions (SFs) if your BlueField is a standalone server.

Setup DPU Management Access and Update BlueField-Bundle

These pages provide detailed information about DPU management access and software installation and updates:

- Host-side Interface Configuration - NVIDIA Docs
- BF-Bundle Installation and Upgrade
- NVIDIA DOCA Downloads | NVIDIA Developer

Systems with a Host Server typically use RShim (i.e. the `tmfifo_net0` interface).

Standalone systems will have to use the OOB interface option for management access.

Port Configuration

Creating SR-IOV Virtual Functions (Host Server)

The first step to use SR-IOV is to create Virtual Functions (VFs) on the host server.

VFs can be created using the following sequence:

```
sudo -s    # enter sudo shell
echo 4 > /sys/class/net/eth2/device/sriov_numvfs
exit  # exit sudo shell
```

Info

Entering sudo shell rather than just issuing a single `sudo` command is necessary because otherwise the `sudo` applies only to the echo command and not the hosting shell and the redirection fails with "Permission denied"

This example creates 4 VFs under Physical Function eth2. Please adjust according to your needs.

If a PF already has VFs and you'd like to change the number of VFs, please set it to 0 before applying the new value.

Scalable Functions (DPU)

For more information, see this: BlueField Scalable Function User Guide

If you create SFs, refer to their representors in the configuration file.

Install the DPL Runtime Service on the DPU

Pulling the Container Resources and Scripts from NGC

Start by downloading and installing the ngc-cli tools.

Fetch the configuration files from NGC, this will create a directory named `dpl_rt_service_<version>` .

e.g. `dpl_rt_service_v1.0.0-doca2.10.0`

Commands:

```
wget --content-disposition
https://api.ngc.nvidia.com/v2/resources/nvidia/ngc-
apps/ngc_cli/versions/3.58.0/files/ngccli_arm64.zip -O
ngccli_arm64.zip
unzip ngccli_arm64.zip
./ngc-cli/ngc registry resource download-version
"nvidia/doca/dpl_rt_service"
cd dpl_rt_service_v1.0.0-doca2.10.0
```

Running the Preparation Script

Inside the directory with the scripts and YAML files that you pulled with the ngc-cli tool, you'll find `scripts/dpl_dpu_setup.sh`.

Running this script on the DPU (requires sudo) will allow the usage of SR-IOV Virtual-Function interfaces and create the directory structure of the configuration files in directory `/etc/dpl_rt_service`. In addition, the script will set "hugepages" and call the necessary `mlxconfig` commands to use DPL Runtime Service.

Run the following sequence of commands from the working directory you pulled with the ngc-cli tool:

```
chmod +x ./scripts/dpl_dpu_setup.sh
sudo ./scripts/dpl_dpu_setup.sh
sudo systemctl restart kubelet.service
sudo systemctl restart containerd.service
```

Restarting the services is necessary for the "hugepages" change to apply to them.

> **Info**
>
> The following firmware settings are set by the setup script:
>
> - FLEX_PARSER_PROFILE_ENABLE=4
> - PROG_PARSE_GRAPH=true
> - SRIOV_EN=1

Edit the Configuration Files

Modify your configuration files as they are described here: Service Configuration

Important: you must create at least one device configuration under `/etc/dpl_rt_service/devices.d/` . It's advisable to start by making a copy of file `/etc/dpl_rt_service/devices.d/NAME.conf.template` .

e.g.

```
cp /etc/dpl_rt_service/devices.d/NAME.conf.template
/etc/dpl_rt_service/devices.d/1000.conf
```

Setting up the kubelet Pod

Now that everything is ready, copy the file `configs/dpl_rt_service.yaml` from the directory that you pulled with the ngc-cli into directory `/etc/kubelet.d` .

Please allow a few minutes for the image to be pulled and the pod to be started. you may check the progress with command `sudo journalctl -u kubelet --since -5m`, make sure to scroll down to see the latest log lines.

When the image is pulled, you will see it by using the command `sudo crictl images` .

When the pod is loaded, you will see it by using the command `sudo crictl pods` .

When the DPL Runtime Service is successfully running inside the pod, you will be able to find the log file in `/var/log/doca/dpl_rt_service/dpl_rtd.log`

Recap, Full Command Sequence

```
wget --content-disposition
https://api.ngc.nvidia.com/v2/resources/nvidia/ngc-
apps/ngc_cli/versions/3.58.0/files/ngccli_arm64.zip -O
ngccli_arm64.zip
unzip ngccli_arm64.zip
./ngc-cli/ngc registry resource download-version
"nvidia/doca/dpl_rt_service"
cd dpl_rt_service_v1.0.0v1
chmod +x ./scripts/dpl_dpu_setup.sh
sudo ./scripts/dpl_dpu_setup.sh
sudo systemctl restart kubelet.service
sudo systemctl restart containerd.service

sudo cp /etc/dpl_rt_service/devices.d/NAME.conf.template
/etc/dpl_rt_service/devices.d/1000.conf
## Modify the configuration file /etc/dpl_rt_service/devices.d/1000.conf
```

```
sudo cp configs/dpl_rt_service.yaml /etc/kubelet.d/
```

> **Note**
>
> The device ID and version numbers may be different in your case, please adapt as needed.

Service Configuration

In the current release of the DPL Runtime Service, there are three types of configuration files.

Each format is similar in nature to INI files but they allow repeated sections (e.g. `[section]`) and the comments are marked by `#` rather than `;`

Configuration of the general behavior - dpl_rt.conf

The DPL Runtime Service searches for this path:
`/etc/dpl_rt_service/dpl_rt.conf`

The contents typically look like this:

```
# Example of a possible DPL RT Service GENERAL configuration file

[LOGGING]
log_file_path=/var/log/doca/dpl_rt_service/dpl_rtd.log
log_level=INFO
# Possible log_level values (case insensitive):
# DISABLE
# CRITICAL
# ERROR
# WARNING
# INFO
# DEBUG
# TRACE

[P4RT_RPC_SERVER]
server_address=[::]      # IPv6 "ANY" allows IPv4 connections
server_tcp_port=9559

[DPL_ADMIN_RPC_SERVER]
server_address=[::]      # IPv6 "ANY" allows IPv4 connections
```

```
server_tcp_port=9600

[DPL_NSPECT_RPC_SERVER]
server_address=[::]      # IPv6 "ANY" allows IPv4 connections
server_tcp_port=9560
```

The default logging verbosity that is configured here will be effective when the DPL Runtime Service is started. When the DPL Runtime Service is running, the logging level can be modified with the DPL Admin client, provided in the DPL Dev container. Modifying the logging level from the DPL Amin tool does NOT modify the configuration file, so keep in mind that when the DPL Runtime Service is restarted, the log verbosity will be as specified in the configuration file.

This file also controls the TCP binding of three gRPC servers. It allows you to specify any address (allowing for remote connections from any accessible network interface of the system) or to limit access to a specific IP address that is dedicated for management. Choosing a non-default TCP port for any of the gRPC servers is also possible.

P4RT_RPC_SERVER

This is the server that listens for clients implementing the P4Runtime protocol.

An open-source client is provided in the DPL Dev container.

DPL_ADMIN_RPC_SERVER

This is the server that listens for the p4admin client that is provided in the DPL Dev container.

DPL_NSPECT_RPC_SERVER

This is the server that listens to the DPL Nspect client/debugger.

Performance fine tuning - system.conf

The DPL Runtime Service searches for this path:
`/etc/dpl_rt_service/system.conf`

```
# Example of a possible DPL RT Service system configuration file

[HAL]
queue_size=1024
queues_num=1
burst_size=32
```

These parameters control the internal behavior of how the DPL Runtime Service interfaces with the underlying hardware. Manipulating these values may affect the maximum rate of rule insertion/deletion and its latency. Tuning these parameters for optimal values can be a complex process and is dependent on the use case and configuration of the DPU. For the current release, this file is for internal use and it is not advised for end users to change the default values, unless under the direct guidance of NVIDIA technical support.

Device level configuration - devices.d/ .conf

The DPL Runtime Service searches for this path:
`/etc/dpl_rt_service/devices.d/<device-id>.conf`

e.g. `/etc/dpl_rt_service/devices.d/1000.conf`

A template is available here:
`/etc/dpl_rt_service/devices.d/NAME.conf.template`

In the template, you will also find internal documentation about the meaning of each field.

> **Info**
>
> If you use Scalable Functions (SFs) or SR-IOV Virtual Functions (VFs), be sure to refer to their representors in the configuration file.
>
> The `mac` and `mtu` settings are for future implementation and currently have no effect.

The values are returned back as they appear in the file when queried with the dpl_admin client but they should not be relied upon.

DPL Port ID Assignment

The DPL Port IDs are assigned by the user. The user decides which DPL Port ID is assigned to which DPU interface. This mapping is critical for achieving the desired results when adding P4 table entries. A DPL Port can be:

- Uplink net device interface
- Host PF representor net device interface (`pf<X>hpf`)
- VF representor net device interface
- SF representor net device interface

Note

Make sure all representors ports added to the configuration file belong to the same uplink port used.

DPL Port ID restrictions

The following restrictions must be considered when assigning DPL Port IDs:

- ID of value `UINT32_MAX` is reserved
- For P4 device, the ID must be an integer number greater than zero
- For interfaces, the DPL ID must be an integer number between zero and `UINT32_MAX`
- Currently, only one Uplink port can be added to the configuration file

Example DPL Device Configuration File

```
# Example of a possible DPL RT Service Device configuration file:
#
# This configuration file specifies the DPL device and its interfaces
# and their DPL Port IDs that will be used by a DPL program.
#
# The DPL Port IDs are assigned by the user. The user decides which
# DPL Port ID is assigned to which ConnectX/DPU interface. This mapping
# is critical for achieving the desired results when adding table entries.
# For DPL device, the ID must be an integer number greater than zero.
#
# The configuration file consists of following sections:
# - [DEVICE] section: Must appear only once.
# - [P4_RT_CONTROLLER] section: Must appear only once.
# - [INTERFACE] section: Must be repeated for each DPL Port (network interface).


[DEVICE]
# The DPL Device ID, used for connecting a controller to manage this device's tables.
dpl_device_id=1000
# Cache counter - decrease HW accesses - when expired an HW access will occur upon request.
dpl_counter_cache_timeout=0

[P4_RT_CONTROLLER]
# Packets delivered to the DPL RT Service from a controller will have this source DPL Port ID.
# So, this ID can be used for matching traffic originated from the controller.
p4_controller_port_id=9876

[INTERFACE]
# Interface name on the system to attach to this DPL device.
interface=p0
# DPL Port ID, used to reference this port by the DPL program and/or when updating table entries.
dpl_logical_port_id=0
# Ethernet frame size.
mtu=1514
# Only uncomment and provide this if you wish to override the interface's MAC address.
```

```
# mac=00:00:00:00:00:00

[INTERFACE]
interface=pf0hpf
dpl_logical_port_id=65535
mtu=1514
# mac=00:00:00:00:00:00

[INTERFACE]
interface=pf0vf0
dpl_logical_port_id=1
mtu=1514
# mac=00:00:00:00:00:00

[INTERFACE]
interface=pf0vf1
dpl_logical_port_id=2
mtu=1514
# mac=00:00:00:00:00:00
```

document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.
