DPL Development Container

Table of contents

The DPL Development Container is intended for developers wishing to utilize NVIDIA's DOCA Pipeline Language (DPL) to build data path applications based in a P4 derived programming language. The DPL is supported on of the NVIDIA® BlueField® networking platforms (DPUs, or SuperNICs in DPU-mode).

- DPL Target Architecture is important to read for new DPL developers to understand the target architecture, its capabilities and the main building blocks that DPL applications rely upon.
- P4 Language Support in DPL outlines specific P4-16 language features supported by the DOCA Pipeline Language.
- DPL Installation Guide describes how to install the DPL Development container.
- Compiling DPL Applications describes in detail how to compile a DPL program and the various compilation options.
- Loading DPL Applications demonstrates how to load the resulting binary output of the DPL compiler on NVIDIA BlueField devices.
- Sample DPL Applications provides additional example DPL programs targeting the DPL Model.

DOCA Target Architecture

This section introduces the NVIDIA DPL Model for the NVIDIA® BlueField®-3 networking platforms (DPUs or SuperNICs).

Introduction

The concept of a Target Architecture (TA) is fundamental in the domain of P4 programming, serving as a specification that delineates the programmable components (such as parsers, ingress control flows, etc.) and their interfaces within a specific hardware platform. This specification acts as a formal agreement between the P4 program and the target hardware, ensuring compatibility and efficient utilization of the hardware's data plane capabilities. For each hardware platform, vendors, including NVIDIA, provide a P4 compiler and an architecture definition that is tailored to their specific devices.

This section focuses on the DPL DOCA target architecture, offering a detailed examination of how to program these blocks to enable customized packet processing and forwarding through the various interfaces and ports available on NVIDIA's BlueField devices. Initially, the emphasis is on the BlueField-3 DPU model, with considerations for future expansion to the BlueField 4 model. While some similarities may exist between different Target Architectures, i t is crucial to understand that Target Architectures are generally vendor-specific and are not designed with cross-vendor portability in mind. The following information aims to provide a comprehensive technical overview for utilizing the DPL compiler for BlueField devices, facilitating advanced data plane programming on NVIDIA's hardware platforms.

DOCA Model

The DOCA model encompasses a set of control stages that are supported by the hardware. Each top-level object within this model is designed to receive a specific set of parameters:

- Parser – The parser is responsible for processing the input packet and a header stack. It operates without access to any metadata during the parsing stage, focusing solely on the packet's content and structure.

```
parser NvDocaParser<HEADERS>(
    packet_in packet,
```

```
    out HEADERS headers);
```

- Control – The control stage is designed to handle three distinct types of metadata, each serving a different purpose within the target architecture:
  - Standard Metadata: This is a read-only type of metadata that provides essential information about the packet obtained from prior stages, such as the parser and other fixed hardware units such as the crypto engine.
  - User Metadata: This metadata category is read/write and contains application-specific variables. It allows for the customization and storage of data relevant to the application's logic.
  - Packet Out Metadata: Also a read/write type, this metadata is associated with the packet as it is transmitted from the controller. It carries additional information that may be required in the DOCA pipeline and for packets sent to the controller.

```
control NvDocaMainControl<HEADERS, USER_META, PKT_OUT_META>(
    inout HEADERS headers,
    in nv_standard_metadata_t std_meta,
    inout USER_META user_meta,
    inout PKT_OUT_META pkt_out_meta);
```

An important aspect of BlueField hardware is its capability to perform packet modification and reparsing as needed. Due to this feature, a deparser control stage is not necessary within this target architecture, as the native BlueField hardware can perform packet modifications and handle reparsing tasks in-line. This simplifies the main package by eliminating the need for the user to manually build a separate deparser control or recirculating the packet, streamlining packet processing within the pipeline. The main package is used as the entry point of every P4 program:

```
package NvDocaPipeline<HEADERS, USER_META, PKT_OUT_META>(
    NvDocaParser<HEADERS> parse,
    NvDocaMainControl<HEADERS, USER_META, PKT_OUT_META> main);
```

DOCA Core Library

Basic common types are defined in `doca_core.p4`. These types are used in the DOCA TA in various interfaces such as extern functions.

Typedefs:

The following typedefs are for commonly used header fields and standard metadata fields:

```
typedef bit<32> nv_logical_port_t;
typedef bit<48> nv_mac_addr_t;
typedef bit<32> nv_ipv4_addr_t;
typedef bit<128> nv_ipv6_addr_t;
typedef bit<32> nv_tunnel_id_t;
typedef bit<12> nv_vlan_id_t;
typedef bit<24> nv_vxlan_id_t;
typedef bit<20> nv_mpls_label_t;
typedef bit<8> nv_debug_cookie_t;
```

Constants:

The following constants are the DOCA TA specific values for standard metadata key fields:

```
const bit<2> L2_TYPE_UNICAST   = 2w0;
const bit<2> L2_TYPE_MULTICAST = 2w1;
const bit<2> L2_TYPE_BROADCAST = 2w2;

const bit<2> L3_TYPE_NONE   = 2w0;
const bit<2> L3_TYPE_IPV4   = 2w1;
const bit<2> L3_TYPE_IPV6   = 2w2;

const bit<2> L4_TYPE_NONE   = 2w0;
const bit<2> L4_TYPE_TCP    = 2w1;
const bit<2> L4_TYPE_UDP    = 2w2;
```

```
const bit<2> L4_TYPE_IPSEC   = 2w3;

const bit<4> L4_TYPE_EXT_NONE  = 4w0;
const bit<4> L4_TYPE_EXT_TCP   = 4w1;
const bit<4> L4_TYPE_EXT_UDP   = 4w2;
const bit<4> L4_TYPE_EXT_ICMP  = 4w3;

const bit<2> VLAN_TYPE_NONE  = 2w0;
const bit<2> VLAN_TYPE_SVLAN = 2w1;
const bit<2> VLAN_TYPE_CVLAN = 2w2;

const bit<2> ENCAP_TYPE_NONE      = 2w0;
const bit<2> ENCAP_TYPE_L2_TUNNEL = 2w1;
const bit<2> ENCAP_TYPE_L3_TUNNEL = 2w2;
const bit<2> ENCAP_TYPE_ROCE      = 2w3;

const bit<2> IPSEC_TYPE_NONE     = 2w0;
const bit<2> IPSEC_TYPE_OVER_IP  = 2w1;
const bit<2> IPSEC_TYPE_OVER_UDP = 2w2;

const bit<8> IPSEC_SYNDROME_OK  = 8w0;
const bit<8> PSP_SYNDROME_OK    = 8w0;
```

These values are used to understand the various type codes in the standard metadata.

#defines:

These macros are for industry standard values for various RFC protocol fields (e.g., ethertype):

```
/* IP protocol numbers in the Protocol field of the IPv4 header
 * and the Next Header field of IPv6 header
 */
#define NV_IPV6_HBH_OPTION   0x00      /* IPv6 Hop bu Hop option */
#define NV_ICMP_PROTOCOL     0x01      /* protocol number for icmp */
#define NV_IPV4_PROTOCOL     0x04      /* protocol number for IPv4 over IPv4 encap */
```

```
#define NV_TCP_PROTOCOL      0x06      /* protocol number for tcp */
#define NV_UDP_PROTOCOL      0x11      /* protocol number for udp */
#define NV_IPV6_PROTOCOL     0x29      /* protocol number for IPv6 over IPv6 encap */
#define NV_IPV6_EXT_FRAG_PROTOCOL 0x2C /* IPv6 fragmentation extension header */
#define NV_GRE_PROTOCOL      0x2F      /* protocol number for Generic Routing Encapsulation */
#define NV_ESP_PROTOCOL      0x32      /* protocol number for IPSec ESP */
#define NV_AH_PROTOCOL       0x33      /* protocol number for IPSec AH */
#define NV_ICMP6_PROTOCOL    0x3A      /* protocol number for icmpv6 */
#define NV_IPV6_NO_NEXT_HDR_PROTOCOL 0x3B
#define NV_SCTP_PROTOCOL     0x84      /* protocol number for SCTP */
#define NV_ROCE_PROTOCOL     0xFE      /* protocol number of RoCEv1.5 */

/* Ethertype */
#define NV_TYPE_VLAN_CTAG    0x8100
#define NV_TYPE_VLAN_STAG    0x88A8
#define NV_TYPE_IPV4         0x0800
#define NV_TYPE_IPV6         0x86DD
#define NV_TYPE_ARP          0x0806
#define NV_TYPE_CONTROL      0x0808
#define NV_TYPE_MPLS         0x8847
#define NV_TYPE_MPLS_MC      0x8848
#define NV_TYPE_PTP          0x88F7
#define NV_TYPE_FCOE         0x8906
#define NV_TYPE_ROCE         0x8915  /* v1 */
#define NV_TYPE_MAC          0x6558  /* Transparent Ethernet Bridge for Generic Routing Encapsulation */

/* UDP ports */
#define NV_ROCE_PORT         4791   /* v2 bth */
#define NV_VXLAN_PORT        4789
#define NV_VXLAN_GPE_PORT    4790
#define NV_GENEVE_PORT       6081
#define NV_MPLS_TUNNEL_PORT  6635
#define NV_IPSEC_NAT_PORT    4500
#define NV_PTP_EVENT_PORT    319
#define NV_PTP_GEN_PORT      320
#define NV_IPV4_OPTION_MRI   31
#define NV_GTP_U_PORT        2152
#define NV_PSP_PORT          1000
#define NV_GUE_PORT          666

/* VXLAN GPE protocols */
#define NV_VXLAN_GPE_IPV4    0x01
#define NV_VXLAN_GPE_IPV6    0x02
```

```
#define NV_VXLAN_GPE_MAC     0x03
#define NV_VXLAN_GPE_NSH     0x04   /* RFC8300 */

/* PSP protocols */
#define NV_PSP_IPV4         0x04
#define NV_PSP_IPV6         0x29
#define NV_PSP_TCP          0x06
#define NV_PSP_UDP           0x11
```

Metadata:

The standard metadata (`struct nv_standard_metadata_t`) for the DOCA TA are found in `doca_metadata.p4`. The members of the standard metadata struct are all read only, and are separated into 3 logical categories:

- Pipeline metadata, information set by hardware units before the programmable part of the BlueField pipeline
- Outer packet header metadata, information set by the hardware parser for the outer packet
- Inner packet header metadata, information set by the hardware parser for the inner packet (if present)

User Metadata:

The DPL programmer can create and use their own metadata struct in the DOCA TA. The type can be `bit<>` or a struct, limited in size by the number of internal registers on the device. For conveniences, the TA defines `nv_empty_metadata_t` which the user can pass into the main package declaration if no metadata is needed.

DOCA Packet Parser

In the DOCA Target Architecture, headers represent the various packet protocol formats that are recognized by the parser. Once these headers are identified, the fields within the packet become candidates for further processing. They can be matched against entries in a match-action table, where they may be altered based on user-defined actions. The DOCA TA comes equipped with a default hardware-integrated parser that is capable of understanding a range of protocols standardized by the Internet Engineering Task Force (IETF). To accommodate custom requirements, the architecture allows for the integration of user-defined ("flex") headers. These flex headers are seamlessly incorporated into the pre-existing parser graph, extending its capabilities to recognize and process additional protocol formats as specified by the user. This flexibility enables developers to tailor the data plane processing to specific applications and protocols beyond the standard set, and future-proofs the developer's investment in the hardware.

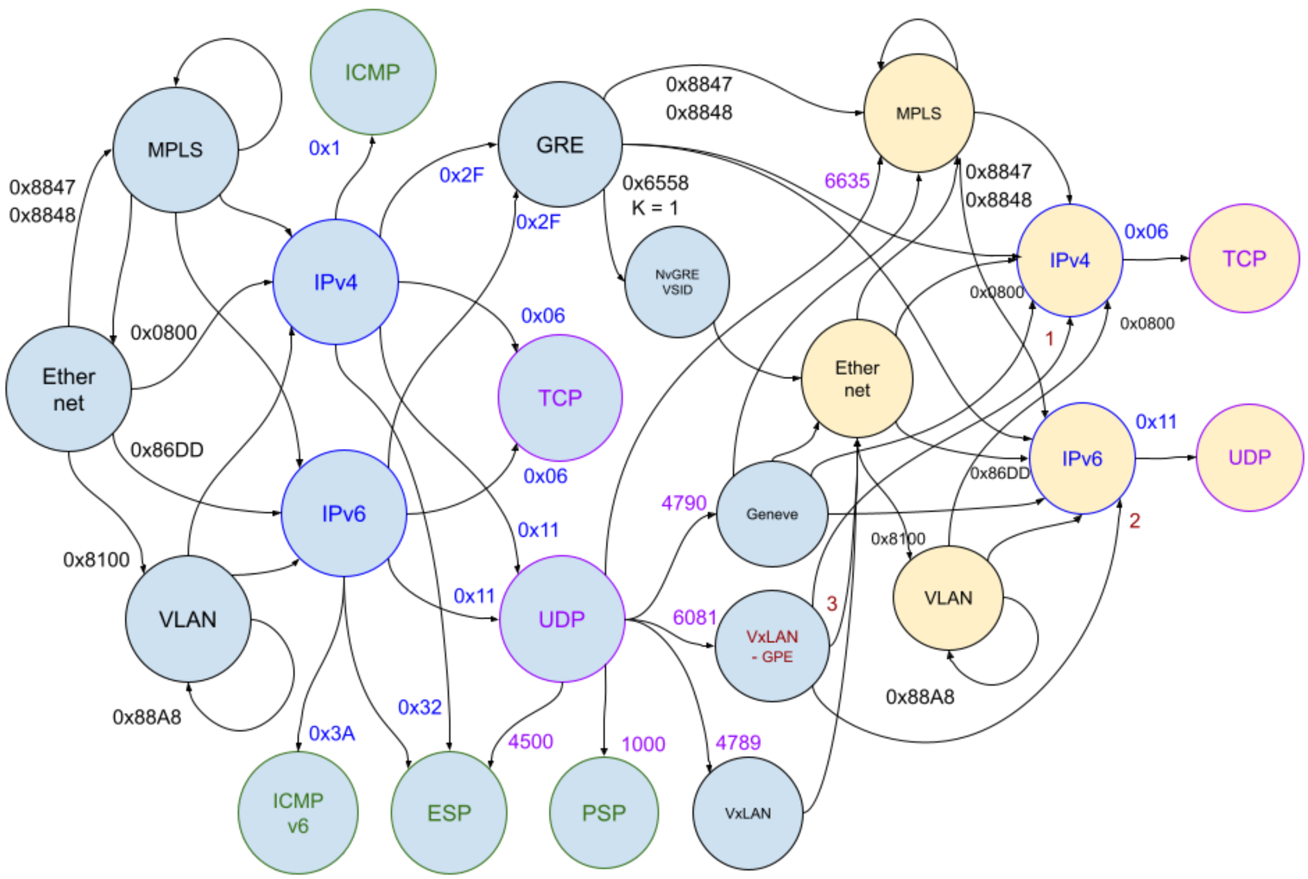
This section on the DPL Packet Parser focuses on BlueField-3 devices and higher. NVIDIA's BlueField architecture combines native and flexible parsing capabilities, allowing the user to enhance the hardware's packet parsing engine with custom protocol headers. This integration is seamless, utilizing built-in parsing for standardized headers.

Flex Parser Configuration

Flex nodes are user-configurable within the DOCA parser, covering elements including the next protocol field, fixed length headers, and calculations for variable header sizes. Arcs link flex nodes and feature properties such as the `is_tunnel` flag, transition value, source node, and destination node. Configurations that are not supported will trigger a compile-time error. Packet fields are extracted from the flex node to the hardware's sampler registers, facilitating the use of fields as match keys and enabling header modification actions.

Use Cases

Examples of custom flex headers include TCP options, SRv6, GTP-U, SCTP, GUE, eCPRI, GENEVE options, and proprietary tunnels. The hybrid architecture ensures smooth integration of user-defined protocols with the fixed parser graph. Below is an illustration of the default fixed parser for BlueField-3:



The DPL compiler enables the user to effectively utilize the P4 language to program packet parsers, taking full advantage of its native and flexible parsing capabilities to meet a variety of network processing needs. The remainder of this section describes the features of the BlueField-3 parser.

Native Parse Graph and Headers

BlueField's parser supports a hybrid architecture that incorporates both predefined (fixed) headers and standard transitions between headers as defined by the Internet Engineering Task Force (IETF). These transitions are determined based on the "next header type" field present within each protocol. For a comprehensive understanding of these transitions and the supported protocols, users are encouraged to refer to the following resources:

- **Service Names and Port Numbers**: IANA Service Names and Port Numbers
- **IEEE 802 Numbers**: IANA IEEE 802 Numbers

- **Protocol Numbers**: IANA Protocol Numbers

These resources provide detailed information on the various service names, port numbers, IEEE standards, and protocol numbers that are recognized and supported by BlueField's parser, facilitating a broad range of packet processing applications. The following table describes the native header and transition support in the DOCA TA:

Header Type	Next Protocol Field	Field Size (bits)	Fixed IETF Transitions	Flex Transition
MAC w VLAN	Ether Type	16	IPv4 (0x0800), IPv6 (0x86DD), MPLS UC (0x8847), MPLS MC (0x8848), VLAN (0x8100), SVLAN (0x88A8)	From, outer, inner
IPv4	Protocol	8	UDP (0x11), TCP (0x6), GRE(0x2F), ICMP(0x1), IPSec AH (0x33), IPSec ESP (0x32), IPv4 encap (0x4)	To, outer, inner
IPv6	Protocol	8	HOPOPT (0x0), UDP (0x11), TCP (0x6),GRE(0x2F), ICMP(0x3A), IPSec AH (0x33), IPSec ESP (0x32), IPv6-Route (0x2B), , IPv6 encap (0x29), IPv6-Frag(0x2C), IPv6-NoNxt (0x3B), IPv6-DestOpts (0x3c)	To, outer, inner
IP	Protocol	8	UDP, TCP, GRE, ICMP, IPSec AH, IPSec ESP	From, outer, inner (both IPv4 and IPv6)
UDP	Destination port	16	VXLAN (4789 + 3 additional custom ports) VXLAN-GPE (4790) GENEVE (6081) MPLS over UDP (6635) IPSEC ESP over UDP (4500) PSP(1000 + 2 additional custom ports)	To, from, outer, inner
TCP	Destination port	16	None	To, from, outer, inner
ICMP	None	N/A	None	Not supported - cannot transition to or from
GRE	Ether Type	16	IPv4 (0x0800), IPv6 (0x86DD) MPLS over GRE (0x8847, 0x8848)	To, from, outer

Header Type	Next Protocol Field	Field Size (bits)	Fixed IETF Transitions	Flex Transition
NVGRE (GRE)	Ether Type	16	Inner MAC (0x6558)	To, from, outer
NVGRE Options	Key present	1	Key present true	Cannot transition to or from
VXLAN	None	N/A	Inner MAC (fixed)	From, outer
VXLAN-GPE	Next Protocol	8	Reserved (0x00), IPv4 (0x01), IPv6 (0x02), Ethernet (0x03), NSH (0x04)	From, outer
GENEVE	Ether Type	16	Inner MAC (0x6558), IPv4 (0x0800), IPv6 (0x86DD), MPLS (0x8847, 0x8848)	From, outer
MPLS	Lookahead	4	IPv4, IPv6	To, from, outer
IPSEC ESP	Next Header	8	Not supported	From, outer
PSP	Next Header	8	Tunnel: IPv4 (4), IPv6 (41) Transport: TCP (6), UDP (17)	To, from, outer

Note that L3 control plane headers ARP (0x0806), MAC control (0x8808), LLDP (0x88CC), PTP (0x88F7) are recognized and steered to a special QP by hardware that are parsed but not matchable in the steering pipeline.

Using Default Native Parser

The DOCA TA defines a default native parser. The user need only include `doca_parser.p4` and reference `nv_fixed_parser` in the DPL package declaration to use the built-in hardware parser. The program should then use the native headers structure, `nv_headers_t`.

Reparsing Capability

A distinctive feature of DPL, setting it apart from other P4 hardware architectures, is its ability to perform reparsing as needed, even midway through the processing pipeline. This flexibility allows for dynamic adjustments to packet processing based on intermediate outcomes during the pipeline's execution. Unlike RMT-based P4 Target Architectures, there are no strict pipeline stages, as the pipeline runs to completion. This means:

- Tables may be applied multiple times, saving scale by not requiring the duplication of tables and their entries at different points in the pipeline.
- Packets do not need to be resubmitted to the pipeline after packet modification, e.g. for encapsulation or decapsulation actions.
- Looping in the pipeline is permitted.

This capability greatly improves the user experience by allowing the DPL developer to focus on a logical view of the pipeline behavior, rather than dealing recirculation logic and saving state in metadata.

Defining a Custom Parser

The P4-16 language does not provide a method to extend a pre-existing, target architecture defined parser. Hence the DPL compiler maintains an internal representation of the default parser (based on a "read only" `doca_parser.p4` source file), and allows headers to be added or removed. In this case, header removal may be implicit, based on excluding a fixed header from the DOCA parser definition. There are 3 steps to defining a custom parser that will be used in conjunction with the native parser:

1. Define the custom header.
2. Add the header to a headers struct.
3. Define the state in a custom parser in the program.

Custom Headers and Headers Struct

A macro, `NV_FIXED_HEADERS`, is provided that references all the headers that the fixed parser can extract. This definition may be used directly as the body of type `Headers_t`, or added to, e.g.:

```
#include <doca_parser.p4>
header custom_header_t {
    bit<16> x;
}
struct custom_headers_t
{
    NV_FIXED_HEADERS
    custom_header_t custom;
}
```

The DPL programmer is expected to use `NV_FIXED_HEADERS` as the basis of their `Headers_t` definition. Failure to do so may result in unexpected behavior, for example:

- Headers/fields that are never extracted at runtime even if extracted by parser states.
- Use of flex extractions for every single referenced header field.
- Compilation errors.

Custom Parser States

A macro, `NV_FIXED_PARSER`, is provided which describes the fixed parser. This definition may be used directly as the body of a parser, or extended by writing additional states, and linking them using the `@nv_transition_from` annotation:

```
parser my_parser(packet_in packet, out custom_headers_t headers)
{
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_ethernet", 0x1234)
    state parse_custom {
        packet.extract(headers.custom);
        transition select(headers.custom.x) {
```

```
                1: nv_parse_ipv4;
                2: nv_parse_ipv6;
                default: accept;
            }
        }
    }
```

The DPL programmer is expected to use `NV_FIXED_HEADERS` as the basis of their parser definition. The macro must take the same variable name use for packet_in and headers used in the parser control definition. Failure to do so may result in unexpected behavior, for example:

- Use of hardware flex parser resources for every parser state, even where hardware fixed parser resources could have been used.
- Definitions of states with names matching states in `NV_FIXED_PARSER` being ignored, and their fixed definition used instead.
- Compilation errors.

Finally, the custom parser should be instantiated in the main DPL package:

```
NvDocaPipeline(
    my_parser(),
    my_main_control()
) main;
```

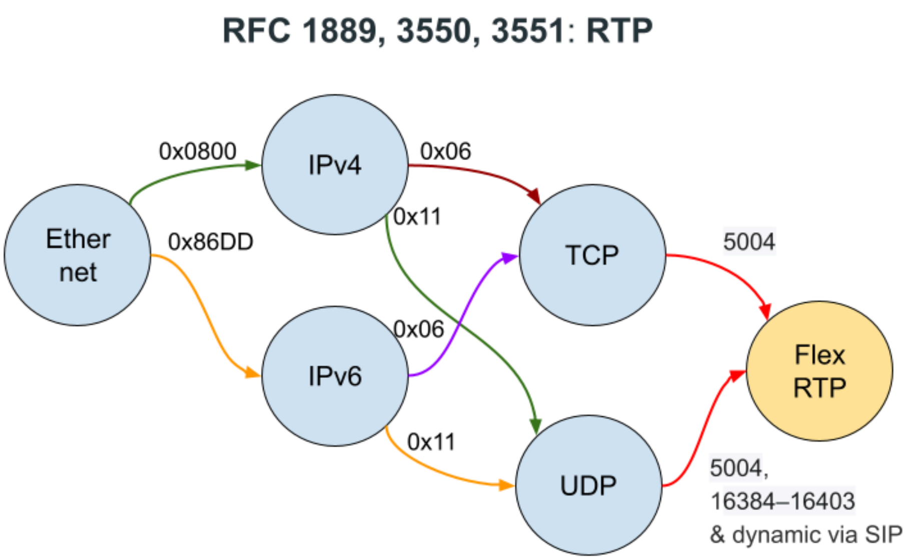
Specifying Transitions to Flex Headers

To extend the native parser with flex headers, the DOCA TA utilizes a combination of macros and DPL annotations to instruct the compiler how to stitch in the customized headers into an existing parse graph.

To create an unconditional transition from a source state, the parser state must be preceded by an `@nv_transition_from` annotation with 1 argument: the source state name. If the parser contains multiple state definitions with an `@nv_transition_from` annotation with the same source state, a single state will be selected (in an undefined manner) to be the target state of the transition. Put another way, each `@nv_transition_from` annotation from a given source state annotation may be considered to override any earlier (in an undefined order) transition destination for that source state.

To create a conditional transition based on the source state's transition field, the parser state must be preceded by an `@nv_transition_from` annotation with 2 arguments: the source state name and transition value. Transitions of this type are prepended to the source state's list of transitions and thus are guaranteed to override any transitions defined by `NV_FIXED_PARSER` in an undefined order. Entries with identical transition values will override each other but may generate warnings due to unreachability.

The following diagram shows an example of extending the native parser to support the RTP protocol:



> **Note**
>
> In the native parser, as an optimization, the user can specify a transition from both IPv4 and IPv6 headers to a flex header using a special 2 argument annotation of the format:
>
> ```
> @nv_transition_from("nv_parse_ipv6, nv_parse_ipv4", 0x1234)
> ```

Where 0x1234 is an example of the custom IP protocol value for some user defined L4 protocol.

Specifying Transitions to Fixed Headers

In a limited number of cases, transitions from fixed headers can be configured. This uses a different annotation, `@nv_transition` , and is always used to annotate the top level parser object. For example, it is a common requirement to add nonstandard transitions from UDP to VxLAN:

```
@nv_transition("nv_parse_udp", "nv_parse_vxlan", 1234)
@nv_transition("nv_parse_udp", "nv_parse_vxlan", 1235)
@nv_transition("nv_parse_udp", "nv_parse_vxlan", 1236)
parser custom_parser(
    packet_in packet,
    out nv_headers_t headers
) {
    NV_FIXED_PARSER(packet, headers)
}
```

where the first parameter specifies the parser node to transition "from", and the second parameter specifies the parser node to transition "to". Note that the annotation `@nv_transition` can be used instead of `@nv_transition_from` for connecting headers.

Defining TLV Parsing for Headers with Optional Data

The DOCA TA allows the user to define protocol headers that have TLVs as sub headers. Note that a custom base "parent" header must be supplied in order to create custom TLVs. It is not supported to create custom TLVs for native headers. For example, the user can define a GENEVE header as follows:

```
header geneve_t {
    bit<2> ver;
    bit<6> opt_len;
    bit<1> o;
    bit<1> c;
    bit<6> reserved;
    bit<16> protocol_type;
    bit<24> vni;
    bit<8> reserved56;
};
```

And then define a struct for the base GENEVE option fields (TYPE and LENGTH) along with a customized VALUE data.

```
// Struct so it can be a field within other headers
struct geneve_option_t {
    bit<24> option_class;
    bit<8> option_type;
    bit<3> reserved24;
    bit<5> length;
};

header geneve_option_int_md_t {
    geneve_option_t base;
    bit<4> ver;
    bit<1> d;
    bit<1> e;
    bit<1> m;
    bit<12> reserved7;
    bit<5> hop_ml;
    bit<8> remaining_hop_count;
    // etc.
```

```
};
```

Next, the user must add the custom headers to the struct of headers and connect them in the parser. Both the base option struct and the newly defined headers must be present in the headers struct so that there is a reference to these types in the parser.

```
struct app_headers {
    NV_FIXED_HEADERS
    geneve_t custom_geneve;
    geneve_option_t geneve_opt;
    geneve_option_int_md_t geneve_opt_int_md;
};
```

The DOCA TA uses an `NvOptionsParser` extern object to define TLV options, since the P4 language itself does not provide a native way to define TLV parsing. An instance of the `NvOptionsParser` must be created in the parser, at the outermost scope, with the following parameters:

- `options_length_field` - the name of the field in the parent header (as a string) that holds the total length of the options. If the total options length is fixed (i.e., not specified in the header's data), then this field should be omitted.
- `options_length_shift` - the value by which the parser should apply a shift (i.e. multiply by power of 2) to the value of the option length field. If the length value is fixed or does not need to be shifted, then this field should be omitted.
- `options_length_add` - the constant to which the parser should add to the value option length field after shifting. If the length value does not need an addend, then this field should be omitted. If the total options length is fixed, then this field must specify the fixed length of the TLV options.
- `option_layout header_type` - the user defined P4 struct that holds the TLV field layout of the option header
- `option_length_field` - the name of the field in the base option struct (as a string) that holds the length of the options. If the options length is fixed (i.e., not specified in the base options struct), then this field should be omitted.

- `option_length_shift` - the value by which the parser should apply a shift (i.e. multiply by power of 2) to the option length field. If the length value is fixed or does not need to be shifted, then this field should be omitted.

- `option_length_add` - the constant to which the parser should add to the option length field. If the length value does not need an addend, then this field should be omitted. If the options length is fixed, then this field must specify the fixed length of the TLV options.

- `options` - a list of tuples, where the first value is the TYPE, and the second value is the name of the child header defined by the user

```
parser geneve_parser(
    packet_in packet,
    out app_headers headers
) {
    NvOptionParser<bit<24>, _>(
        "opt_len",              // options_length_field
        2,                      // options_length_shift
        0,                      // options_length_add
        "geneve_option_t",      // option_layout_header_type
        "length",               // option_length_field
        0,                      // option_length_shift
        4,                      // option_length_add
        "option_class_type",    // option_type_field
        // options data
        (list<tuple<bit<24>, _>>){
            {24w0x010301, "headers.geneve_opt_int_md"}
            // list of additional options ...
        }
    ) geneveOptions;

    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", 6082)
    state parse_custom_geneve {
        packet.extract(headers.custom_geneve);
```

```
        geneveOptions.parseOptions(packet, headers);
        transition accept;
    }
}
```

Restrictions and Unsupported Parser Features

DOCA parser features that either differ from the $P4_{16}$ specification, are restricted in the DOCA TA or are unsupported are listed below.

Unsupported Features

These features are not supported by the DOCA TA in comparison to the $P4_{16}$ specification. Utilizing these features will result in compilation errors.

- Variable Declarations and Extern Instantiations: Parsers and parser states cannot contain any variable declarations or extern instantiations other than `NvOptionsParser`.
- Lookahead and Advance: Programmable lookahead and advance functionalities are not supported.
- Parser Value Sets: Parser value sets for setting transition values from runtime data are not supported.
- State Declarations: States may not contain any declaration, or any statement besides extract, transition statements and calls to `NvOptionsParser` methods.

Restricted Behavior

The following points below highlight how the DOCA TA implementation of DPL parser features differs from the standard $P4_{16}$ specification.

- Reserved State Names: Parser state names beginning with `nv_*` are reserved by the DOCA TA. Custom states using this reserved prefix will be ignored.

- State and Header Coupling: Each state must extract exactly one header, and they are considered to be coupled in a 1:1 fashion.

- Fixed-Size Header Extraction: The extracted header must be fixed-size, known at compile-time, and defined in the P4 header type definition. I.e., `p.extract(headers.xxx)` is allowed, but `p.extract(headers.xxx, someLength)` is not.

- Transition Statements: Besides terminal pre-defined states (`accept`, `reject`), states must transition to another state using a transition statement, which may be unconditional or conditional.

- Transition Select: Transition select statements are limited to using a single field from the header the state extracts or a constant. Lists of expressions, constants, operators, references to declarations other than a field in the header the state extracts, and any other expression not explicitly allowed are disallowed.

- Default Case in Transition Select: Every transition select statement must have a default case, transitioning to the accept state.

- Extract Before Transition: A state's `extract` statement must precede its transition statement.

- Empty Accept and Reject States: The `accept` and `reject` states must be completely empty.

- Loops in Parser Graph: Loops are not allowed in the DPL program's parser graph. Each path through the parser must be acyclic.

Hybrid Parser Behavior

These restrictions and behaviors apply to the unique hybrid fixed/flex parser model supported by the DOCA TA.

- Start state of `nv_parse_ethernet` – The start state cannot be changed or overwritten.

- Redefining states – States may not be redefined. This includes no possibility of applying `@nv_transition_from` to the body of any fixed state.

- `NV_FIXED_PARSER` and `NV_FIXED_HEADERS` – States part of `NV_FIXED_PARSER` extract the pre-defined fixed header, irrespective of the state body. States not part of `NV_FIXED_PARSER` must extract headers not part of `NV_FIXED_HEADERS`.
- Fixed-to-fixed transitions – It is not possible to disable any fixed->fixed transition. These may be overridden to transition to a custom state by applying an `@nv_transition_from` annotation to the destination state.
- MPLSoUDP stops parsing at the inner MAC header if the pseudowire Control Word is set to 0. Additional inner packet fields will not be parsed.

DOCA Extern Objects

Counters

BlueField supports per entry counters, shared and direct. A (shared) counter is an extern object that allows the user to access N independent counters via an index. Whereas a direct counter is an extern object that is directly associated with a P4 table where the index is implied by the entry slot. For both types, the counter value is updated when an action calls the count() method.

NvCounter

Instantiates an indirect counter of the specified width and type.

Signature: `extern NvCounter(bit<32> size, NvCounterType type)`

Parameters:

- `size[in]`: Number of counter indices
- `type[in]`: Enum indicating counter type. Currently only `NvCounterType.PACKETS_AND_BYTES` is supported.

Methods:

count

Increments the counter at the specified index

```
void count(in bit<32> index)
```

Parameters:

- `index[in]`: Index of the counter to increment

NvDirectCounter

Instantiates a direct counter that can be assigned to a table. Direct counters do not have a separate size, they are as wide as the table. Each action invoked by an entry hit should call the count() method to increment the entry count.

Signature: `extern NvDirectCounter(NvCounterType type)`

Parameters:

- `type[in]`: Enum indicating counter type. Currently only `NvCounterType.PACKETS_AND_BYTES` is supported.

Methods:

count

Increments the counter associated the the entry that triggered the action execution

```
void count()
```

> **Note**
>
> Methods of a NvDirectCounter object may only be called by the P4 Table that owns it via the "direct_counter" property bound to the P4 Table. It is an error to attempt to call the count() method from a P4 Table that does not own the direct counter.

Note

If there are no side effects, it is most efficient to call the count method as the first primitive action in a P4 action.

Meters

BlueField supports per-flow metering, following RFC2697, RFC2698, and RFC4115. Similar to counters, both direct meters and shared meters are supported in the DPL target architecture. The packets for a given flow can be colored by the meter into three "colors": red, yellow and green. The meter burst parameters can be specified with units of bytes or packets.

```
typedef bit<8> nv_meter_color_t;

enum nv_meter_color_t NvMeterColor {
    RED = 8w0,
    YELLOW = 8w1,
    GREEN = 8w2,
}

enum NvMeterUnits {
  BYTES,
  PACKETS
}
```

Note

Each configuration parameter of NvMeterPeakTrTCM and NvDirectMeterPeakTrTCM is restricted to certain max values.

When metering by BYTES, max values are the following:

- cir, pir: 255000000000

- cbs, pbs: 2147483648

When metering by PACKETS, max values are the following:

- cir, pir: 1992187500

- cbs, pbs: 16777216

NvMeterPeakTrTCM

Instantiates a Two Rate Three Color Marker shared meter object of the specified size, units and burst parameters. (Implements RFC 2698)

Signature:

```
extern NvMeterPeakTrTCM(bit<32> size, NvMeterUnits units, bit<64>
cir, bit<64> cbs, bit<64> pir, bit<64> pbs)
```

Parameters:

- `size[in]`: Number of meter indices
- `units[in]`: Enum indicating the units used by the meter. Metering by `NvMeterUnits.BYTES` or by `NvMeterUnits.PACKETS` is supported.
- `cir[in]`: Committed Information Rate
- `cbs[in]`: Committed Burst Size
- `pir[in]`: Peak Information Rate
- `pbs[in]`: Peak Burst Size

Methods:

meter

Executes the metering of the flow entry at the specified index

```
nv_meter_color_t meter(in bit<32> index, nv_meter_color_t
initial_color = NvMeterColor.GREEN)
```

Parameters:

- `index[in]`: Index of the meter to execute
- `initial_color[in]`: The initial color to assign to the flow. Coloring of `NvMeterColor.RED`, `NvMeterColor.YELLLOW`, or `NvMeterColor.GREEN`. The default is `NvMeterColor.GREEN`.

Returns:

- The `nv_meter_color_t` of the flow entry at the specified index.

NvDirectMeterPeakTrTCM

Instantiates a Single Rate Three Color Marker indirect meter object of the specified size, units and burst parameters. (Implements RFC 2698)

Signature:

```
extern NvDirectMeterPeakTrTCM(bit<32> size, NvMeterUnits units,
bit<64> cir, bit<64> cbs, bit<64> pir, bit<64> pbs)
```

Parameters:

- `size[in]`: Number of meter indices
- `units[in]`: Enum indicating the units used by the meter. Metering by `NvMeterUnits.BYTES` or by `NvMeterUnits.PACKETS` is supported.
- `cir[in]`: Committed Information Rate
- `cbs[in]`: Committed Burst Size
- `pir[in]`: Peak Information Rate
- `pbs[in]`: Peak Burst Size

Methods:

meter

Executes the metering of the flow entry at the specified index

```
nv_meter_color_t meter(nv_meter_color_t initial_color =
NvMeterColor.GREEN)
```

Parameters:

- `initial_color[in]`: The initial color to assign to the flow. Coloring of `NvMeterColor.RED`, `NvMeterColor.YELLLOW`, or `NvMeterColor.GREEN`. The default is `NvMeterColor.GREEN`.

Returns:

- The `nv_meter_color_t` of the flow entry at the specified index.

> **Note**
>
> Methods of a **NvDirectMeterPeakTrTCM** object may only be called by the P4 Table that owns it via the "direct_meter" property bound to the P4 Table. It is an error to attempt to call the meter() method from a P4 Table that does not own the direct meter. A meter may not be owned by multiple P4 Tables.

Options Parser

DPL supports TLV style parsing for protocols with options.

NvOptionsParser

The NvOptionsParser is an extern object used to define the behavior of the TLV (Type-Length-Value) options parser.

```
extern NvOptionParser<VALUE, OPTION> {
    NvOptionParser(
        string options_length_field = "",
        bit<32> options_length_shift = 32w0,
        bit<32> options_length_add = 32w0,
        string option_layout_header_type = "",
        string option_length_field = "",
        bit<32> option_length_shift = 32w0,
        bit<32> option_length_add = 32w0,
        string option_type_field = "",
        list<tuple<VALUE, OPTION>> options
    );

    void parseOptions<H>(packet_in p, inout H headers);
}
```

Parameters:

- options_length_field[in]: Optional; omit if ***total*** options length is fixed, otherwise provide the field name that contains the header length value
- options_length_shift: Optional; omit if the field can be used without shifting
- options_length_add: Optional; omit if the field can be used without adding
- option_layout_header_type: Mandatory header_type_name
- option_length_field: Optional; omit if ***each*** option is a constant size, otherwise provide the field name that contains the header length value
- option_length_shift: Optional; omit if the field can be used without shifting
- option_length_add: Optional; omit if the field can be used without adding
- option_type_field: Mandatory field_name
- options: Mandatory; variable-length (expression-)list of (type, header field)

Methods:

parseOptions

Parses the TLV

Signature: `void parseOptions<H>(packet_in packet, inout H headers);`

Parameters:

- packet[in]: input packet
- headers[in/out]: packet headers struct

Examples:

See Geneve TLV Parsing Example.

Custom Tunnel Encapsulation

DPL supports the creation of L2 and L3 tunnels using custom tunnel headers.

NvTunnelTemplate

Instantiates a template object used to create tunnel encapsulations. This object is used for controlling the behavior of the actions `nv_set_l2tunnel_underlay` and `nv_set_l3tunnel_underlay`.

```
extern NvTunnelTemplate<HEADER_TYPE> {
    NvTunnelTemplate();
}
```

Type Variables:

- `HEADER_TYPE[in]`: Must be the typename of a user-declared `struct` whose only fields must be `header` types. The entire set of underlay headers should be defined as a struct, for example:

```
struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    my_tunnel_h   custom_tunnel;
}
```

Annotation:

NvTunnelTemplate works slightly differently than other extern objects in that the object has no methods; instead, the object can be passed into one of two methods `nv_set_l2tunnel_underlay` and `nv_set_l3tunnel_underlay`. Declaring a NvTunnelTemplate object does not use any resources in hardware, as the template construct is provided for simplifying the process of specifying header field values for custom tunnels. When declaring a NvTunnelTemplate object, the annotation `@nv_tunnel_fields` must be present. This annotation contains a key-value entry for each header in the specified struct type. For each header, the header fields must be specified in the order that they appear in the header definition. The value assigned to a field must be one of the following:

- a non-negative integer
- the string "variable"
- the string "ignore"

```
@nv_tunnel_fields(
    ethernet = {
        dst_addr = "variable",
        src_addr = "variable",
        ether_type = 0xABCD
    },
    custom_tunnel = {
        field_1 = 1,
        field_2 = "ignore",
        field_3 = "variable"
    }
```

```
)
NvTunnelTemplate<tunnel_headers_t>() my_tunnel;
```

For a complete example, see GTP Tunnel Encapsulation Example.

Note

Certain header fields will automatically be recalculated by the hardware after encapsulation. These fields cannot be assigned by the user. It is good practice to mark them as "ignore".

Header field	Value
ipv4.ecn	0
ipv4.identification	0
ipv4.hdr_checksum	Hardware calculated
ipv4.total_len	Hardware calculated
ipv6.traffic_class (ECN bits only)	0
ipv6.payload_length	Hardware calculated
udp.src_port	Entropy hash
udp.length	Hardware calculated
udp.checksum	0
tcp.checksum	Hardware calculated
gre.key (8 LSB only)	Entropy hash

DOCA Extern Functions

Extern functions serve as a mechanism for exposing DPL target specific functionality that may be beyond a standard P4 model.

Note that many of these externs mutate the packet, for which the HEADERS are a part of the extern signature.

nv_drop

Terminal extern function that stops packet processing and drops the packet

Signature: `extern void nv_drop();`

nv_send_to_port

Terminal extern function that stops packet processing and sends the packet to the specified port

Signature: `extern void nv_send_to_port(in nv_logical_port_t port);`

Paramaters:

- port[in]: This parameter specifies the logical port to which the packet will be sent to

nv_send_to_controller

Terminal extern function that forwards packet metadata to a controller

Signature:

```
extern void nv_send_to_controller<PACKET_META>(in PACKET_META
pkt_in_meta);
```

Paramaters:

- pkt_in_meta[in]: The metadata of the packet being sent to the controller for processing

nv_dec_ip_ttl

Extern function that decrements the ttl value in the IP header. Applies to both IPv4 and IPv6. The ttl value will be set to max value if decremented from zero value.

Signature:

```
extern void nv_dec_ip_ttl<HEADERS>(inout HEADERS headers, in bit<8>
ttl_value);
```

Paramaters:

- headers[in/out]: packet headers struct
- ttl_value[in]: TTL value to decrement

nv_set_ip_dscp

Extern function that sets the DSCP value in the IP header. Applies to both IPv4 and IPv6.

Signature:

```
extern void nv_set_ip_dscp<HEADERS>(inout HEADERS headers, in bit<6>
dscp_value);
```

Paramaters:

- headers[in/out]: packet headers struct
- dscp_value[in]: DSCP value to set

nv_set_ip_ecn

Extern function that sets the ECN value in the IP header. Applies to both IPv4 and IPv6.

Signature:

```
extern void nv_set_ip_ecn<HEADERS>(inout HEADERS headers, in bit<2>
ecn_value);
```

Paramaters:

- headers[in/out]: packet headers struct

- ecn_value[in]: ECN value to set

nv_set_ip_ttl

Extern function that sets the TTL value in the IP header. Applies to both IPv4 TTL and IPv6 hop limit.

Signature:

```
extern void nv_set_ip_ttl<HEADERS>(inout HEADERS headers, in bit<8>
ttl_value);
```

Paramaters:

- headers[in/out]: packet headers struct
- ttl_value[in]: TTL/Hop limit value to set

nv_set_l4_src_port

Extern function that sets the L4 source port value in the TCP/UDP header.

Signature:

```
extern void nv_set_l4_src_port<HEADERS>(inout HEADERS headers, in
bit<16> src_port);
```

Paramaters:

- headers[in/out]: packet headers struct
- src_port[in]: L4 port value to set

nv_set_l4_dst_port

Extern function that sets the L4 destination port value the TCP/UDP header.

Signature:

```
extern void nv_set_l4_dst_port<HEADERS>(inout HEADERS headers, in
bit<16> dst_port);
```

Paramaters:

- headers[in/out]: packet headers struct
- dst_port[in]: L4 port value to set

nv_l2_decap

Extern function that performs Layer 2 decapsulation on the packet headers i.e. removing ethernet/vlan headers.

Signature: `void nv_l2_decap<HEADERS>(inout HEADERS headers)`

Parameters:

- headers[in/out]: packet headers struct that will be decapsulated

nv_l3_decap

Extern function that performs Layer 3 decapsulation on the packet headers i.e. removing ethernet, vlan and IP headers, and appends new L2 headers with specified.

```
void nv_l3_decap<HEADERS>(inout HEADERS headers, bool has_vlan,
nv_mac_addr_t dst_mac, nv_mac_addr_t src_mac, bit<16> l3_ether_type,
nv_vlan_id_t vid)
```

Parameters:

- headers [inout]: The packet headers that will be decapsulated
- has_vlan [in]: Bool flag indicating whether the packet includes a VLAN header.
- dst_mac [in]: The destination MAC address in the Ethernet frame.
- src_mac [in]: The source MAC address in the Ethernet frame.

- l3_ether_type [in]: The EtherType value in the Ethernet frame
- vid [in]: The VLAN ID, only valid if has_vlan is true.

nv_push_vlan

Extern function that inserts a VLAN header immediately after the L2 header.

```
void nv_push_vlan<HEADERS>(inout HEADERS headers, in NvVlanTagId
tpid, in bit<3> pcp, in bit dei, in nv_vlan_id_t vid);
```

Parameters:

- headers [inout]: The packet headers that will be decapsulated.
- tpid [in]: Tag Protocol Identifier.
- pcp [in]: Priority Code Point.
- dei [in]: Drop Eligible Indicator (formerly CFI).
- vid [in]: VLAN Identifier.

nv_pop_vlan

Extern function that removes the outermost VLAN header, immediately after the L2 header.

```
void nv_pop_vlan<HEADERS>(inout HEADERS headers);
```

Parameters:

- headers [inout]: The packet headers that will be decapsulated.

nv_set_vxlan_v4_underlay

Description:

Extern function that encapsulates the packet with an ethernet frame (optionally VLAN tagged), an ipv4 header and a VXLAN header.

Signature:

```
void nv_set_vxlan_v4_underlay<HEADERS>(inout HEADERS headers, in
bool has_vlan, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac,
in nv_vlan_id_t vid, in nv_ipv4_addr_t sip, in nv_ipv4_addr_t dip,
in nv_vxlan_id_t vni)
```

Parameters:

- headers [in/out]: The packet headers that are to be encapsulated with a VXLAN underlay.
- has_vlan [in]: Specifies whether a VLAN tag is present.
- dst_mac [in]: Destination MAC address.
- src_mac [in]: Source MAC address.
- vid [in]: VLAN ID (only valid if has_vlan is set, otherwise ignored)
- sip [in]: Source IPv4 address.
- dip [in]: Destination IPv4 address.
- vni [in]: VXLAN Network Identifier.

nv_set_vxlan_v6_underlay

Description:

Extern function that encapsulates the packet with an ethernet frame (optionally VLAN tagged), an IPv6 header, and a VXLAN header.

Signature:

```
void nv_set_vxlan_v6_underlay<HEADERS>(inout HEADERS headers, in
bool has_vlan, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac,
in nv_vlan_id_t vid, in nv_ipv6_addr_t sip, in nv_ipv6_addr_t dip,
in nv_vxlan_id_t vni)
```

Parameters:

- `headers` [in/out]: The packet headers that are to be encapsulated with a VXLAN underlay.
- `has_vlan` [in]: Specifies whether a VLAN tag is present.
- `dst_mac` [in]: Destination MAC address.
- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set, otherwise ignored).
- `sip` [in]: Source IPv6 address.
- `dip` [in]: Destination IPv6 address.
- `vni` [in]: VXLAN Network Identifier.

nv_set_gre_v4_underlay

Description:

Extern function that configures GRE IPV4 underlay encapsulation, with an optional VLAN tag and optional GRE key.

Signature:

```
void nv_set_gre_v4_underlay<HEADERS>(inout HEADERS headers, in bool has_vlan, in bool has_key, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac, in nv_vlan_id_t vid, in nv_ipv4_addr_t sip, in nv_ipv4_addr_t dip, in NvInnerProtocolType proto_type, in bit<32> key)
```

Parameters:

- `headers` [in/out]: The packet headers that are to be encapsulated with a GRE underlay.

- `has_vlan` [in]: Specifies whether a VLAN tag is present.
- `has_key` [in]: Specifies whether the GRE key is present.
- `dst_mac` [in]: Destination MAC address.
- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set, otherwise ignored).
- `sip` [in]: Source IPv4 address.
- `dip` [in]: Destination IPv4 address.
- `proto_type` [in]: Protocol type for the inner payload.
- `key` [in]: GRE key (only valid if has_key is set, otherwise ignored).

nv_set_gre_v6_underlay

Description:

Extern function that configures GRE IPV6 underlay encapsulation, with an optional VLAN tag and optional GRE key.

Signature:

```
void nv_set_gre_v6_underlay<HEADERS>(inout HEADERS headers, in bool
has_vlan, in bool has_key, in nv_mac_addr_t dst_mac, in
nv_mac_addr_t src_mac, in nv_vlan_id_t vid, in nv_ipv6_addr_t sip,
in nv_ipv6_addr_t dip, in NvInnerProtocolType proto_type, in bit<32>
key)
```

Parameters:

- `headers` [in/out]: The packet headers that are to be encapsulated with a GRE underlay.
- `has_vlan` [in]: Specifies whether a VLAN tag is present.

- `has_key` [in]: Specifies whether the GRE key is present.
- `dst_mac` [in]: Destination MAC address.
- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set, otherwise ignored).
- `sip` [in]: Source IPv6 address.
- `dip` [in]: Destination IPv6 address.
- `proto_type` [in]: Protocol type for the inner payload.
- `key` [in]: GRE key (only valid if has_key is set, otherwise ignored).

nv_set_geneve_v4_underlay

Description:

Extern function that configures GENEVE IPV4 underlay encapsulation with optional VLAN tagging and an optional GENEVE option with 32 bits of data.

Signature:

```
void nv_set_geneve_v4_underlay<HEADERS>(inout HEADERS headers, in
bool has_vlan, in bool has_option, in nv_mac_addr_t dst_mac, in
nv_mac_addr_t src_mac, in nv_vlan_id_t vid, in nv_ipv4_addr_t
src_ip, in nv_ipv4_addr_t dst_ip, in bit<6> opt_fields_len, in
bit<1> oam, in bit<1> critical, in NvInnerProtocolType proto_type,
in bit<24> vni, in bit<16> opt_class, in bit<8> opt_type, in bit<5>
opt_len, in bit<32> opt_data)
```

Parameters:

- `headers` [in/out]: The packet headers that are to be encapsulated with a GENEVE underlay.
- `has_vlan` [in]: Specifies whether a VLAN tag is present.

- `has_option` [in]: Specifies whether a GENEVE option is present.
- `dst_mac` [in]: Destination MAC address.
- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set, otherwise ignored).
- `src_ip` [in]: Source IPv4 address.
- `dst_ip` [in]: Destination IPv4 address.
- `opt_fields_len` [in]: Length of the options field in 4-byte units (must be set to 0 if has_option is false)
- `oam` [in]: OAM flag.
- `critical` [in]: Critical options flag.
- `proto_type` [in]: Protocol type for the inner payload.
- `vni` [in]: Virtual Network Identifier.
- `opt_class` [in]: Option class (ignored if has_option is false)
- `opt_type` [in]: Option type. (ignored if has_option is false)
- `opt_len` [in]: Option length in 4-byte units. (ignored if has_option is false, otherwise must be 1)
- `opt_data` [in]: Option data. (ignored if has_option is false)

nv_set_geneve_v6_underlay

Description:

Extern function that configures GENEVE IPV6 underlay encapsulation with optional VLAN tagging and an optional GENEVE option with 32 bits of data.

Signature:

```
void nv_set_geneve_v6_underlay<HEADERS>(inout HEADERS headers, in bool has_vlan, in bool has_option, in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac, in nv_vlan_id_t vid, in nv_ipv6_addr_t src_ip, in nv_ipv6_addr_t dst_ip, in bit<6> opt_fields_len, in bit<1> oam, in bit<1> critical, in NvInnerProtocolType proto_type, in bit<24> vni, in bit<16> opt_class, in bit<8> opt_type, in bit<5> opt_len, in bit<32> opt_data)
```

Parameters:

- `headers` [in/out]: The packet headers that are to be encapsulated with a GENEVE underlay.
- `has_vlan` [in]: Specifies whether a VLAN tag is present.
- `has_option` [in]: Specifies whether a GENEVE option is present.
- `dst_mac` [in]: Destination MAC address.
- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set, otherwise ignored).
- `src_ip` [in]: Source IPv6 address.
- `dst_ip` [in]: Destination IPv6 address.
- `opt_fields_len` [in]: Length of the options field in 4-byte units (must be set to 0 if has_option is false, otherwise must be set to 2)
- `oam` [in]: OAM flag.
- `critical` [in]: Critical options flag.
- `proto_type` [in]: Protocol type for the inner payload.
- `vni` [in]: Virtual Network Identifier.
- `opt_class` [in]: Option class (ignored if has_option is false)

- `opt_type` [in]: Option type. (ignored if has_option is false)
- `opt_len` [in]: Option length in 4-byte units. (ignored if has_option is false, otherwise must be 1)
- `opt_data` [in]: Option data. (ignored if has_option is false)

nv_set_l2tunnel_underlay

Description:

Extern function that configures custom underlay followed by an L2 encapsulation of the original packet. It is optional, but recommended, that the new tunnel encapsulation be parsable by the DPL parser. For more details see the section on NvTunnelTemplate.

Signature:

```
void nv_set_l2tunnel_underlay<HEADERS, NV_TUNNEL_TEMPLATE, VALUES>(
inout HEADERS headers, NV_TUNNEL_TEMPLATE tunnel, in VALUES values)
```

Parameters:

- `headers` [in/out]: The packet headers that are to be encapsulated.
- `tunnel` [in]: The NvTunnelTemplate object that specifies the underlay headers and the header values.
- `values` [in]: A list expression whose length matches the number of header fields marked "variable" in the NvTunnelTemplate object's annotation. The order of the header fields determines which value a field corresponds to, i.e. 3rd header field marked "variable" will be set to the 3rd value in the list. Each value is required to be runtime constant, meaning it may either be an integer value or an action parameter value.

nv_set_l3tunnel_underlay

Description:

Extern function that configures custom underlay followed by an L3 encapsulation of the original packet. The L3 layer must be IP. It is optional, but recommended, that the new tunnel encapsulation be parsable by the DPL parser. For more details see the section on NvTunnelTemplate.

Signature:

```
void nv_set_l3tunnel_underlay<HEADERS, NV_TUNNEL_TEMPLATE, VALUES>(
inout HEADERS headers, NV_TUNNEL_TEMPLATE tunnel, in VALUES values)
```

Parameters:

- `headers` [in/out]: The packet headers that are to be encapsulated
- `tunnel` [in]: The NvTunnelTemplate object that specifies the underlay headers and the header values.
- `values` [in]: A list expression whose length matches the number of header fields marked "variable" in the NvTunnelTemplate object's annotation. The order of the header fields determines which value a field corresponds to, i.e. 3rd header field marked "variable" will be set to the 3rd value in the list. Each value is required to be runtime constant, meaning it may either be an integer value or an action parameter value.

nv_mirror

Description:

Extern function that duplicates the packet, sending each to a different port. Hence this extern is a terminal action.

Signature:

```
void nv_mirror(in nv_logical_port_t vport, in nv_logical_port_t
mirror_port)
```

Parameters:

- `vport` [in]: Destination for the first copy of the packet

- `mirror_port` [in]: Destination for the second copy of the packet

nv_mirror_vxlan_v4_to_remote

Description:

Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a VXLAN IPV4 underlay

Signature:

```
void nv_mirror_vxlan_v4_to_remote(in nv_logical_port_t vport, in
nv_logical_port_t mirror_port, in bool has_vlan, in nv_mac_addr_t
dst_mac, in nv_mac_addr_t src_mac, in nv_vlan_id_t vid, in
nv_ipv4_addr_t sip, in nv_ipv4_addr_t dip, in nv_vxlan_id_t vni)
```

Parameters:

- `vport` [in]: Destination for the unmodified copy of the packet
- `mirror_port` [in]: The destination for the encapsulated copy of the packet.
- `has_vlan` [in]: Specifies whether a VLAN tag is present.
- `dst_mac` [in]: Destination MAC address.
- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set).
- `sip` [in]: Source IPv4 address.
- `dip` [in]: Destination IPv4 address.
- `vni` [in]: VXLAN Network Identifier.

nv_mirror_vxlan_v6_to_remote

Description:

Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a VXLAN IPv6 underlay.

Signature:

```
void nv_mirror_vxlan_v6_to_remote(in nv_logical_port_t vport, in
nv_logical_port_t mirror_port, in bool has_vlan, in nv_mac_addr_t
dst_mac, in nv_mac_addr_t src_mac, in nv_vlan_id_t vid, in
nv_ipv6_addr_t sip, in nv_ipv6_addr_t dip, in nv_vxlan_id_t vni)
```

Parameters:

- `vport` [in]: Destination for the unmodified copy of the packet.
- `mirror_port` [in]: Destination for the encapsulated copy of the packet.
- `has_vlan` [in]: Specifies whether a VLAN tag is present.
- `dst_mac` [in]: Destination MAC address.
- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set).
- `sip` [in]: Source IPv6 address.
- `dip` [in]: Destination IPv6 address.
- `vni` [in]: VXLAN Network Identifier.

nv_mirror_gre_v4_to_remote

Description:

Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a GRE IPv4 underlay, including optional VLAN tagging and GRE key.

Signature:

```
void nv_mirror_gre_v4_to_remote(in nv_logical_port_t vport, in
nv_logical_port_t mirror_port, in bool has_vlan, in bool has_key, in
nv_mac_addr_t src_mac, in nv_mac_addr_t dst_mac, in nv_vlan_id_t
vid, in nv_ipv4_addr_t src_ip, in nv_ipv4_addr_t dst_ip, in
NvInnerProtocolType proto_type, in bit<32> key)
```

Parameters:

- `vport` [in]: Destination for the unmodified copy of the packet.
- `mirror_port` [in]: Destination for the encapsulated copy of the packet.
- `has_vlan` [in]: Specifies if a VLAN tag is present.
- `has_key` [in]: Specifies whether the GRE key is present.
- `src_mac` [in]: Source MAC address.
- `dst_mac` [in]: Destination MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set).
- `src_ip` [in]: Source IPv4 address.
- `dst_ip` [in]: Destination IPv4 address.
- `proto_type` [in]: Protocol type for the inner payload.
- `key` [in]: GRE key (only valid if has_key is set).

nv_mirror_gre_v6_to_remote

Description:

Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a GRE IPv6 underlay, including optional VLAN tagging and GRE key.

Signature:

```
void nv_mirror_gre_v6_to_remote(in nv_logical_port_t vport, in
nv_logical_port_t mirror_port, in bool has_vlan, in bool has_key, in
nv_mac_addr_t src_mac, in nv_mac_addr_t dst_mac, in nv_vlan_id_t
vid, in nv_ipv6_addr_t sip, in nv_ipv6_addr_t dip, in
NvInnerProtocolType proto_type, in bit<32> key)
```

Parameters:

- `vport` [in]: Destination for the unmodified copy of the packet.
- `mirror_port` [in]: Destination for the encapsulated copy of the packet.
- `has_vlan` [in]: Indicates if a VLAN tag is present.
- `has_key` [in]: Specifies whether the GRE key is utilized.
- `src_mac` [in]: Source MAC address.
- `dst_mac` [in]: Destination MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set).
- `sip` [in]: Source IPv6 address.
- `dip` [in]: Destination IPv6 address.
- `proto_type` [in]: Protocol type for the inner payload.
- `key` [in]: GRE key (only valid if has_key is set).

nv_mirror_geneve_v4_to_remote

Description:

Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a Geneve IPv4 underlay, including optional VLAN tagging and Geneve options.

Signature:

```
void nv_mirror_geneve_v4_to_remote(in nv_logical_port_t vport, in
nv_logical_port_t mirror_port, in bool has_vlan, in bool has_option,
in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac, in nv_vlan_id_t
vid, in nv_ipv4_addr_t src_ip, in nv_ipv4_addr_t dst_ip, in bit<6>
opt_fields_len, in bit<1> oam, in bit<1> critical, in
NvInnerProtocolType proto_type, in bit<24> vni, in bit<16>
opt_class, in bit<8> opt_type, in bit<5> opt_len, in bit<32>
opt_data)
```

Parameters:

- `vport` [in]: Destination for the unmodified copy of the packet.
- `mirror_port` [in]: Destination for the encapsulated copy of the packet.
- `has_vlan` [in]: Indicates if a VLAN tag is present.
- `has_option` [in]: Specifies whether Geneve options are utilized.
- `dst_mac` [in]: Destination MAC address.
- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set).
- `src_ip` [in]: Source IPv4 address.
- `dst_ip` [in]: Destination IPv4 address.
- `opt_fields_len` [in]: Length of the options field in 4-byte units (must be set to 0 if has_option is false, otherwise must be set to 2)
- `oam` [in]: OAM flag.
- `critical` [in]: Critical options flag.
- `proto_type` [in]: Protocol type for the inner payload.
- `vni` [in]: Virtual Network Identifier.

- `opt_class` [in]: Option class (ignored if has_option is false)
- `opt_type` [in]: Option type. (ignored if has_option is false)
- `opt_len` [in]: Option length in 4-byte units. (ignored if has_option is false, otherwise must be 1)
- `opt_data` [in]: Option data. (ignored if has_option is false)

nv_mirror_geneve_v6_to_remote

Description:

Extern function that duplicates the packet, sending each to a different port, with the copy being sent to the mirror port encapsulated with a Geneve IPv6 underlay, including optional VLAN tagging and Geneve options.

Signature:

```
void nv_mirror_geneve_v6_to_remote(in nv_logical_port_t vport, in
nv_logical_port_t mirror_port, in bool has_vlan, in bool has_option,
in nv_mac_addr_t dst_mac, in nv_mac_addr_t src_mac, in nv_vlan_id_t
vid, in nv_ipv6_addr_t src_ip, in nv_ipv6_addr_t dst_ip, in bit<6>
opt_fields_len, in bit<1> oam, in bit<1> critical, in
NvInnerProtocolType proto_type, in bit<24> vni, in bit<16>
opt_class, in bit<8> opt_type, in bit<5> opt_len, in bit<32>
opt_data)
```

Parameters:

- `vport` [in]: Destination for the unmodified copy of the packet.
- `mirror_port` [in]: Destination for the encapsulated copy of the packet.
- `has_vlan` [in]: Indicates if a VLAN tag is present.
- `has_option` [in]: Specifies whether Geneve options are utilized.
- `dst_mac` [in]: Destination MAC address.

- `src_mac` [in]: Source MAC address.
- `vid` [in]: VLAN ID (only valid if has_vlan is set).
- `src_ip` [in]: Source IPv6 address.
- `dst_ip` [in]: Destination IPv6 address.
- `opt_fields_len` [in]: Length of the options field in 4-byte units (must be set to 0 if has_option is false, otherwise must be set to 2)
- `oam` [in]: OAM flag.
- `critical` [in]: Critical options flag.
- `proto_type` [in]: Protocol type for the inner payload.
- `vni` [in]: Virtual Network Identifier.
- `opt_class` [in]: Option class (ignored if has_option is false)
- `opt_type` [in]: Option type. (ignored if has_option is false)
- `opt_len` [in]: Option length in 4-byte units. (ignored if has_option is false, otherwise must be 1)
- `opt_data` [in]: Option data. (ignored if has_option is false)

nv_send_debug_pkt

Description:

Extern function that duplicates the original packet, adds pipeline metadata and packet state, then sends it to the DPL Debugger. The original packet continues on the pipeline.

Signature:

```
void nv_send_debug_pkt(in nv_debug_cookie_t cookie = 8w0x0)
```

Parameters:

- `cookie` [in]: An optional cookie value can be provided by the user, that will show up in the DPL Debugger with the debug packet

> **Note**
>
> You must enable debug mode by compiling your DPL program with the `-g` flag in addition calling `nv_send_debug_pkt` . Debug packets can only be processed when the packet direction is from UPLINK towards VF (Rx direction) and only from the primary wire port P0. Placing this extern call in a Tx packet path will result in no debug packet. Debugging packets from the second wire port P1 is currently not supported.
>
> See DOCA Pipeline Language Developer Tool for more details.

Tables

Match Kinds

The DPL compiler supports following match types with the restrictions below:

- Exact match
  - Support for at least 9 keys of DWORD width and 8 keys of byte width. Note that the actual limit is dependent on the HW and may fail only at load time.
  - See Exact tables in the PSA spec for general guidance and usage from a programmer perspective.
- LPM match
  - Not supported for bool-type keys.
  - Not supported for standard metadata fields.

    - See LPM tables in the PSA spec for general guidance and usage from a controller perspective.
- Ternary match
    - Not supported for bool-type keys.
    - Not supported for standard metadata fields.
    - See Ternary tables in the PSA spec for general guidance and usage from a programmer perspective.
- Range match
    - Support up to a maximum of 4 range match keys per P4 table .
    - Support for slices of header fields and std_meta fields as keys.
    - Not supported for slices of user-declared variables and user metadata fields as keys.
    - Not supported for keys larger than 32 bits (unless sliced down to 32 bits or less, with some key-specific slice alignment restrictions).
    - Range match keys in the same P4 Table with Ternary match keys or LPM keys are currently not supported.
    - When using multiple range keys in a single P4 table, some combinations of keys may not be supported. Note that this may fail only at load time.
    - See Range tables in the PSA spec for general guidance and usage from a programmer perspective.

DPL Key Support

IsValid method call, local variables excluding packet-in struct instance, fixed or flex header fields, and standard metadata can be used as a table key.

However, there are following restrictions:

- A key cannot be used multiple times within one P4 table; this will actively fail during compilation. When IsValid method call is used as a table key, the compiler allocates a register for the key. Thus, using multiple isValid method calls as keys within a table causes the undefined behavior.

- Complex expressions must use name annotation.

The following table lists the match key support for fixed header fields.

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.ethernet.dst_addr`	48	✔	✔	✔	✔ [1]	
`headers.ethernet.src_addr`	48	✔	✔	✔	✔ [1]	
`headers.ethernet.ether_type`	16	□	□	□	□	Last extracted l2 etherType can be matched through std_meta.last_l2_ether_type
`headers.vlan.vlan_pcp`	3	✔	✔	✔	✔	
`headers.vlan.vlan_dei`	1	✔	✔	✔	✔	
`headers.vlan.vlan_id`	12	✔	✔	✔	✔	
`headers.vlan.vlan_ether_type`	16	□	□	□	□	Last extracted l2 etherType can be matched through std_meta.last_l2_ether_type
`headers.inner_ethernet.dst_addr`	48	✔	✔	✔	✔ [1]	
`headers.inner_ethernet.src_addr`	48	✔	✔	✔	✔ [1]	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.inner_ethernet.ether_type`	16	▯	▯	▯	▯	Last extracted inner l2 etherType can be matched through std_meta.inner_last_l2_ether_type
`headers.inner_vlan.vlan_pcp`	3	✔	✔	✔	✔	
`headers.inner_vlan.vlan_dei`	1	✔	✔	✔	✔	
`headers.inner_vlan.vlan_id`	12	✔	✔	✔	✔	
`headers.inner_vlan.vlan_ether_type`	16	▯	▯	▯	▯	Last extracted inner l2 etherType can be matched through std_meta.inner_last_l2_ether_type
`headers.ipv4.version` Alias with `headers.ipv6.version`	4	▯	▯	▯	▯	
`headers.ipv4.ihl`	4	✔	✔	✔	✔	
`headers.ipv4.diffserv` Alias with `headers.ipv6.diffserv`	6	✔	✔	✔	✔	
`headers.ipv4.ecn` Alias with `headers.ipv6.ecn`	2	✔	✔	✔	✔	
`headers.ipv4.total_len`	16	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.ipv4.identification`	16	✔	✔	✔	✔	
`headers.ipv4.flags`	3	✔	✔	✔	✔	
`headers.ipv4.frag_offset`	13	✔	✔	✔	✔	
`headers.ipv4.ttl` Alias with `headers.ipv6.hop_limit`	8	✔	✔	✔	✔	
`headers.ipv4.protocol` Alias with `headers.ipv6.next_header`	8	✔	✔	✔	✔	
`headers.ipv4.hdr_checksum`	16	✔	✔	✔	✔	
`headers.ipv4.src_addr`	32	✔	✔	✔	✔	
`headers.ipv4.dst_addr`	32	✔	✔	✔	✔	
`headers.inner_ipv4.version` Alias with `headers.inner_ipv6.version`	4	▯	▯	▯	▯	
`headers.inner_ipv4.ihl`	4	✔	✔	✔	✔	
`headers.inner_ipv4.diffserv`	6	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
Alias with `headers.inner_ipv6.diffserv`						
`headers.inner_ipv4.ecn` Alias with `headers.inner_ipv6.ecn`	2	✔	✔	✔	✔	
`headers.inner_ipv4.total_len`	16	✔	✔	✔	✔	
`headers.inner_ipv4.identification`	16	✔	✔	✔	✔	
`headers.inner_ipv4.flags`	3	✔	✔	✔	✔	
`headers.inner_ipv4.frag_offset`	13	✔	✔	✔	✔	
`headers.inner_ipv4.ttl` Alias with `headers.inner_ipv6.hop_limit`	8	✔	✔	✔	✔	
`headers.inner_ipv4.protocol` Alias with `headers.inner_ipv6.protocol`	8	✔	✔	✔	✔	
`headers.inner_ipv4.hdr_checksum`	16	✔	✔	✔	✔	
`headers.inner_ipv4.src_addr`	32	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.inner_ipv4.dst_addr`	32	✔	✔	✔	✔	
`headers.ipv6.flow_label`	20	✔	✔	✔	✔	
`headers.ipv6.payload_length`	16	✔	✔	✔	✔	
`headers.ipv6.src_addr`	128	✔	✔	✔	✔ [2]	
`headers.ipv6.dst_addr`	128	✔	✔	✔	✔ [2]	
`headers.inner_ipv6.flow_label`	20	✔	✔	✔	✔	
`headers.inner_ipv6.payload_length`	16	✔	✔	✔	✔	
`headers.inner_ipv6.src_addr`	128	✔	✔	✔	✔ [2]	
`headers.inner_ipv6.dst_addr`	128	✔	✔	✔	✔ [2]	
`headers.mpls.label`	20	✔	✔	✔	✔	
`headers.mpls.tc`	3	✔	✔	✔	✔	
`headers.mpls.bos`	1	✔	✔	✔	✔	
`headers.mpls.ttl`	8	✔	✔	✔	✔	
`headers.inner_mpls.label`	20	✔	✔	✔	✔	
`headers.inner_mpls.tc`	3	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.inner_mpls.bos`	1	✔	✔	✔	✔	
`headers.inner_mpls.ttl`	8	✔	✔	✔	✔	
`headers.icmp.type`	8	✔	✔	✔	✔	
`headers.icmp.code`	8	✔	✔	✔	✔	
`headers.icmp.checksum`	16	✔	✔	✔	✔	
`headers.icmp.identifier`	16	✔	✔	✔	✔	
`headers.icmp.sequence_number`	16	✔	✔	✔	✔	
`headers.inner_icmp.type`	8	✔	✔	✔	✔	
`headers.inner_icmp.code`	8	✔	✔	✔	✔	
`headers.inner_icmp.checksum`	16	✔	✔	✔	✔	
`headers.inner_icmp.identifier`	16	✔	✔	✔	✔	
`headers.inner_icmp.sequence_number`	16	✔	✔	✔	✔	
`headers.icmpv6.type`	8	✔	✔	✔	✔	
`headers.icmpv6.code`	8	✔	✔	✔	✔	
`headers.icmpv6.checksum`	16	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.icmpv6.payload_1`	32	✔	✔	✔	✔	
`headers.icmpv6.payload_2`	32	✔	✔	✔	✔	
`headers.inner_icmpv6.type`	8	✔	✔	✔	✔	
`headers.inner_icmpv6.code`	8	✔	✔	✔	✔	
`headers.inner_icmpv6.checksum`	16	✔	✔	✔	✔	
`headers.inner_icmpv6.payload_1`	32	✔	✔	✔	✔	
`headers.inner_icmpv6.payload_2`	32	✔	✔	✔	✔	
`headers.tcp.src_port` Alias with `headers.udp.src_port`	16	✔	✔	✔	✔	
`headers.tcp.dst_port` Alias with `headers.udp.dst_port`	16	✔	✔	✔	✔	
`headers.tcp.seq_no`	32	✔	✔	✔	✔	
`headers.tcp.ack_no`	32	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.tcp.data_offset`	4	✔	✔	✔	✔	
`headers.tcp.res`	3	□	□	□	□	
`headers.tcp.nonce_sum`	1	✔	✔	✔	✔	
`headers.tcp.ecn`	2	✔	✔	✔	✔	
`headers.tcp.flags`	6	✔	✔	✔	✔	
`headers.tcp.window`	16	✔	✔	✔	✔	
`headers.tcp.checksum`	16	✔	✔	✔	✔	
`headers.tcp.urgent_ptr`	16	✔	✔	✔	✔	
`headers.inner_tcp.src_port` Alias with `headers.inner_udp.src_port`	16	✔	✔	✔	✔	
`headers.inner_tcp.dst_port` Alias with `headers.inner_udp.dst_port`	16	✔	✔	✔	✔	
`headers.inner_tcp.seq_no`	32	□	□	□	□	
`headers.inner_tcp.ack_no`	32	□	□	□	□	
`headers.inner_tcp.data_offset`	4	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.inner_tcp.res`	3	▯	▯	▯	▯	
`headers.inner_tcp.nonce_sum`	1	✔	✔	✔	✔	
`headers.inner_tcp.ecn`	2	✔	✔	✔	✔	
`headers.inner_tcp.flags`	6	✔	✔	✔	✔	
`headers.inner_tcp.window`	16	▯	▯	▯	▯	
`headers.inner_tcp.checksum`	16	✔	✔	✔	✔	
`headers.inner_tcp.urgent_ptr`	16	▯	▯	▯	▯	
`headers.udp.length`	16	✔	✔	✔	✔	
`headers.udp.checksum`	16	✔	✔	✔	✔	
`headers.inner_udp.length`	16	✔	✔	✔	✔	
`headers.inner_udp.checksum`	16	✔	✔	✔	✔	
`headers.gre.checksum_present`	1	✔	✔	✔	✔	
`headers.gre.reserved1`	1	✔	✔	✔	✔	
`headers.gre.key_present`	1	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.gre.sequence_present`	1	✔	✔	✔	✔	
`headers.gre.reserved2`	4	✔	✔	✔	✔	
`headers.gre.reserved3`	5	✔	✔	✔	✔	
`headers.gre.version`	3	✔	✔	✔	✔	
`headers.gre.protocol`	16	✔	✔	✔	✔	
`headers.nvgre_vsid.vsid`	24	✔	✔	✔	✔	
`headers.nvgre_vsid.flow_id`	8	✔	✔	✔	✔	
`headers.esp.security_parameters.index`	32	✔	✔	✔	✔	
`headers.esp.sequence_number`	32	✔	✔	✔	✔	
`headers.esp.next_header`	8	□	□	□	□	
`headers.psp.next_header`	8	✔	✔	✔	✔	
`headers.psp.hdr_ext_len`	8	✔	✔	✔	✔	
`headers.psp.crypt_offset`	8	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.psp.needs_sampling`	1	✔	✔	✔	✔	
`headers.psp.drop`	1	✔	✔	✔	✔	
`headers.psp.version`	1	✔	✔	✔	✔	
`headers.psp.has_virtualization_key`	1	✔	✔	✔	✔	
`headers.psp.one_1`	1	✔	✔	✔	✔	
`headers.psp.security_parameters_index`	32	✔	✔	✔	✔	
`headers.psp.initialization_vector`	64	✔	✔	✔	✔	
`headers.psp.virtualization_key_high`	32	✔	✔	✔	✔	
`headers.psp.virtualization_key_lower`	32	✔	✔	✔	✔	
`headers.vxlan.reserved1`	4	✔	✔	✔	✔	
`headers.vxlan.vni_valid`	1	✔	✔	✔	✔	
`headers.vxlan.reserved2`	3	✔	✔	✔	✔	
`headers.vxlan.reserved3`	24	✔	✔	✔	✔	
`headers.vxlan.vni`	24	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.vxlan.reserved4`	8	✔	✔	✔	✔	
`headers.vxlan_gpe.reserved1`	4	✔	✔	✔	✔	
`headers.vxlan_gpe.vni_valid`	1	✔	✔	✔	✔	
`headers.vxlan_gpe.reserved2`	3	✔	✔	✔	✔	
`headers.vxlan_gpe.reserved3`	16	✔	✔	✔	✔	
`headers.vxlan_gpe.next_proto`	8	✔	✔	✔	✔	
`headers.vxlan_gpe.vni`	24	✔	✔	✔	✔	
`headers.vxlan_gpe.reserved4`	8	✔	✔	✔	✔	
`headers.geneve.ver`	2	✔	✔	✔	✔	
`headers.geneve.opt_len`	6	✔	✔	✔	✔	
`headers.geneve.o`	1	✔	✔	✔	✔	
`headers.geneve.c`	1	✔	✔	✔	✔	
`headers.geneve.reserved1`	6	✔	✔	✔	✔	
`headers.geneve.protocol_type`	16	✔	✔	✔	✔	
`headers.geneve.vni`	24	✔	✔	✔	✔	

Fixed Header Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`headers.geneve.reserved2`	8	✔	✔	✔	✔	

1. Must be a slice of upper 16 bits or lower 32 bits __ __ __ __
2. Must be a slice of one of 4 dwords __ __ __ __

The following table lists the support for "isValid()" for fixed header.

Header IsValid	Notes
`ethernet`	
`inner_ethernet`	
`vlan`	VLAN tag is CVLAN
`inner_vlan`	Inner VLAN tag is CVLAN
`svlan`	VLAN tag is SVLAN
`inner_svlan`	Inner VLAN tag is SVLAN
`ipv4`	L3 header protocol is IPv4
`inner_ipv4`	Inner l3 header protocol is IPv4
`ipv6`	L3 header protocol is IPv6
`inner_ipv6`	Inner l3 header protocol is IPv6
`mpls`	MPLS layer 0 is valid after passing all HW checks
`inner_mpls`	Inner MPLS layer 0 is valid after passing all HW checks
`mpls1`	MPLS layer 1 is valid after passing all HW checks
`inner_mpls1`	Inner MPLS layer 1 is valid after passing all HW checks
`mpls2`	MPLS layer 2 is valid after passing all HW checks
`inner_mpls2`	Inner MPLS layer 2 is valid after passing all HW checks

Header IsValid	Notes
`mpl3`	MPLS layer 3 is valid after passing all HW checks
`inner_mpls3`	Inner MPLS layer 3 is valid after passing all HW checks
`mpls_4`	MPLS layer 4 is valid after passing all HW checks
`inner_mpls4`	Inner MPLS layer 4 is valid after passing all HW checks
`tcp`	L4 type is TCP
`inner_tcp`	Inner l4 type is TCP
`udp`	L4 type is UDP
`inner_udp`	Inner l4 type is UDP
`gre`	• L3 layer is valid after passing all HW checks • IPv4 protocol field value is 47
`nvgre`	• L3 is OK • IPv4 protocol is 47 • GRE key is present • GRE protocol type is 25944
`esp`	L4 type is IPSEC
`icmp`	L4 type is ICMP
`inner_icmp`	Inner l4 type is ICMP
`icmpv6`	L4 type is ICMP
`inner_icmpv6`	Inner l4 type is ICMP
`vxlan`	• L4 type is UDP • headers.udp.dst_port is 4789
`vxlan_gpe`	• L4 type is UDP • headers.udp.dst_port is 4790
`geneve`	• L4 type is UDP • headers.udp.dst_port is 6081

The following table lists the match key support for BlueField standard metadata fields.

Standard Metadata Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`ingress_port`	32	✔	▯	▯	▯	The P4 port ID that the packet ingressed the pipeline
`eth_to_fcs_packet_len`	16	✔	▯	▯	✔	Length of the packet from L2 start to FCS, in bytes
`is_l2_ok`	1	✔	▯	▯	✔	L2 layer is valid after passing all HW checks
`l2_type`	2	✔	▯	▯	✔	• 0: L2_TYPE_UNICAST • 1: L2_TYPE_MULTICAST • 2: L2_TYPE_BROADCAST
`last_l2_ether_type`	16	✔	✔	✔	✔	Last extracted value of etherType within ethernet header or VLAN tags
`vlan_type`	2	✔	▯	▯	✔	• 0: VLAN_TYPE_NONE • 1: VLAN_TYPE_SVLAN • 2: VLAN_TYPE_CVLAN
`is_l3_ok`	1	✔	▯	▯	✔	L3 layer is valid after passing all hardware checks
`l3_type`	2	✔	▯	▯	✔	• 0: L3_TYPE_NONE • 1: L3_TYPE_IPV4 • 2: L3_TYPE_IPv6
`is_ip_fragmented`	1	✔	▯	▯	✔	
`is_ipv4_checksum_ok`	1	✔	▯	▯	✔	IPv4 layer checksum is valid
`is_l4_ok`	1	✔	▯	▯	✔	L4 layer (TCP/UDP) is valid after passing all hardware checks
`l4_type`	2	✔	▯	▯	✔	• 0: L4_TYPE_NONE • 1: L4_TYPE_TCP • 2: L4_TYPE_UDP

Standard Metadata Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
						• 3: L4_TYPE_IPSEC
`l4_type_ext`	4	✔	□	□	✔	• 0: L4_TYPE_EXT_NONE • 1: L4_TYPE_EXT_TCP • 2: L4_TYPE_EXT_UDP • 3: L4_TYPE_EXT_IPSEC
`is_l4_checksum_ok`	1	✔	□	□	✔	L4 layer (TCP/UDP) checksum is valid
`l4_src_port`	16	✔	✔	✔	✔	
`l4_dst_port`	16	✔	✔	✔	✔	
`encap_type`	2	✔	□	□	✔	• 0: ENCAP_TYPE_NONE • 1: ENCAP_TYPE_L2_TUNNEL • 2: ENCAP_TYPE_L3_TUNNEL • 3: ENCAP_TYPE_ROCE
`ipsec_layer`	2	✔	□	□	✔	• 0: IPSEC_TYPE_NONE • 1: IPSEC_TYPE_OVER_IP • 2: IPSEC_TYPE_OVER_UDP
`ipsec_syndrome`	8	✔	□	□	✔	0: CRYPTO_OK
`psp_syndrome`	8	✔	□	□	✔	0: CRYPTO_OK
`is_inner_l2_ok`	1	✔	□	□	✔	Inner L2 layer is valid after passing all HW checks
`inner_l2_type`	2	✔	□	□	✔	• 0: L2_TYPE_UNICAST • 1: L2_TYPE_MULTICAST • 2: L2_TYPE_BROADCAST
`inner_last_l2_ether_type`	16	✔	✔	✔	✔	Last extracted value of etherType within inner ethernet header and VLAN tags

Standard Metadata Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
`inner_vlan_type`	2	✔	✔	✔	✔	• 0: VLAN_TYPE_NONE • 1: VLAN_TYPE_SVLAN • 2: VLAN_TYPE_CVLAN
`is_inner_l3_ok`	1	✔	▯	▯	✔	Inner l3 layer (IPv4/IPv6) is valid after passing all HW checks
`inner_l3_type`	2	✔	▯	▯	✔	• 0: L3_TYPE_NONE • 1: L3_TYPE_IPV4 • 2: L3_TYPE_IPv6
`is_inner_ipv4_checksum_ok`	1	✔	✔	✔	✔	
`is_inner_l4_ok`	1	✔	▯	▯	✔	Inner L4 layer (TCP/UDP) is valid after passing all HW checks
`inner_l4_type`	2	✔	▯	▯	✔	• 0: L4_TYPE_NONE • 1: L4_TYPE_TCP • 2: L4_TYPE_UDP • 3: L4_TYPE_IPSEC
`inner_l4_type_ext`	4	✔	▯	▯	✔	• 0: L4_TYPE_EXT_NONE • 1: L4_TYPE_EXT_TCP • 2: L4_TYPE_EXT_UDP • 3: L4_TYPE_EXT_IPSEC
`is_inner_l4_checksum_ok`	1	✔	▯	▯	✔	
`inner_l4_src_port`	16	✔	✔	✔	✔	
`inner_l4_dst_port`	16	✔	✔	✔	✔	
`random_value`	16	✔	✔	✔	✔	
`ut_clock`	64	✔	✔	✔	✔	UTC time stamp

Standard Metadata Fields	Bit Width	Exact	LPM	Ternary	Range	Notes
						• seconds in the upper 32 bits • nano-seconds in the lower 32 bits
`fr_clock`	64	✔	✔	✔	✔	Free running clock, in units of 1/device frequency
`source_qp`	24	✔	▯	▯	✔	

Size

Each P4 table can specify a size attribute. In the DOCA TA, this size represents the maximum number of entries, excluding miss (default) entry. A user-provided size value must be non-zero value. If the size attribute is not present, a default size of 128 is assigned. Note that the memory model of BlueField means that the size parameter is not a guaranteed size, and runtime insertions may fail even if the entry count is below the maximum.

Default Action

The table can contain default action. When there are no matching entries within this table, it will execute the default action. When the default action is not provided, the missing entry will execute NoAction action (continue to the next logical step in the pipeline).

```
table ipv4_fowarding_table {
        key = {
            headers.ipv4.src_addr : exact;
        }
        actions = {
            forward;
            NoAction;
        }
        default_action = forward(1);
```

```
}
```

Const Entries

Const entries are optional field of P4 table, which can be used to insert entries when loading the blob to BlueField. For LPM or ternary matching, the unmasked bits must be zeros.

```
table ipv4_fowarding_table {
        key = {
            headers.ipv4.src_addr : lpm;
        }
        actions = {
            forward;
            NoAction;
        }
        default_action = forward(1);
        const entries = {
            (32w0x11111100 &&& 32w0xFFFFFF00)  : forward(2);
            (32w0x11111100 &&& 36w0xFFFFFF000) : forward(3);
        }
}
```

Packet IO

Controller Packet in (pipeline to controller) and Packet out (controller to pipeline) are supported in the DOCA TA. To send a packet to the controller with metadata:

- Create a struct with the annotation `@nv_controller_metadata("packet_in")`
- The struct must be exactly 32 bits in width (user must insert padding as needed)

- Use the extern action `extern void nv_send_to_controller<PACKET_META>(in PACKET_META pkt_in_meta);`

To check if a packet that has ingressed the pipeline from the controller:

- add as a match key "`std_meta.ingress_port`" to a P4 table key set
- add an entry that specifies the key value to be the P4 Port ID defined in the DPL Port ID assignment step.

P4 Language Support in DPL

This section contains information on the P4 language features that are supported, known issues and deviations from the P4 Language Specification. All references to the spec refer to $P4_{16}$ language specification version *v1.2.4*.

Introduction

The NVIDIA BlueField networking platform (DPU or SuperNIC) is a specialized Data Processing Unit engineered to significantly enhance data center performance. It achieves this by efficiently offloading, accelerating, and isolating critical tasks related to networking, storage, and security. BlueField-3 merges the advantages of dedicated accelerators with the versatility of general-purpose processors, all within an ASIC-based system-on-a-chip architecture. It boasts impressive connectivity speeds of up to 400G, making it an ideal choice for environments demanding high levels of AI and high-performance computing capabilities.

To ensure BlueField delivers on its promise of optimized functionality, its P4 language implementation has been tailored to leverage the strengths of high-performance ASIC hardware. This strategic focus on performance and efficiency, however, means that BlueField supports DOCA Pipeline Language (DPL), which does not include every feature outlined in the P4 language specification. This is primarily due to the inherent differences in flexibility and programmability between ASICs and other types of hardware, such as CPUs and FPGAs, and the unique pipeline model of the BlueField DPU. This section outlines the P4 Language features that are currently supported in this release of the NVIDIA DPL compiler.

Note

This document refers to features as being `supported` or `unsupported`. Supported features have been tested and should work according to the $P4_{16}$ language specification, subject to any caveats described in this document. Unsupported features have not been fully tested and should not be relied upon. In most cases, the compiler will reject programs that use unsupported features. However, in some cases the compiler may accept a program that uses unsupported features if the feature is not necessary to implement the program. For example, if a program contains an expression that uses an unsupported operator but its operands can be computed at compile-time, the compiler may choose to compute the value of the expression at compile-time and accept the program. This behavior should not be used as an indicator of whether the feature is supported.

P4 Language Features

Identifiers

Identifiers starting with `__` are reserved for internal compiler use. Otherwise, identifiers described in the P4 language spec section *6.4.1. Identifiers* are allowed.

Data Types

See *Operators* section for support of operations on values with these types.

- Bool
  - Supported
- Arbitrary-precision Integer
  - Supported only for literals. See spec *7.1.6.5. Arbitrary-precision integers*.
- Signed integer
  - Unsupported
- Strings literals
  - No operations are allowed or validity checks performed. See spec 6.4.3.3.
- Bit strings
  - Supported, limited by available hardware resources

Derived Types

- Enum
  - Enumeration types are supported as described in section 7.2.1 of the $P4_{16}$ spec, allowing the P4 programmer to either specify an underlying representation or allow the compiler to choose the representation. Note that the set of allowed types for the underlying representation is limited to those otherwise supported by the DPL compiler.
- Header
  - Header types are supported as described in section 7.2.2 of the $P4_{16}$ spec using field types otherwise supported by the DPL compiler with the exception of `varbit<>` fields. Variable-length header types are supported only using the `NvOptionParser` extern type.
- Header stacks
  - Unsupported
- Structs
  - Struct types are supported as described in section 7.2.5 of the $P4_{16}$ spec using field types otherwise supported by the DPL compiler.
- Unions
  - Unsupported
- Tuple/List
  - Tuple types are supported as described in section 7.2.6 of the $P4_{16}$ spec using component types otherwise supported by the DPL compiler.
- Extern types
  - Extern types, including both extern functions and extern objects, as described in section 7.2.9 of the $P4_{16}$ spec are supported only for those declared in the P4 headers distributed with the DPL compiler. P4 programs cannot declare additional extern types.
- Type specialization

- Type specialization is supported as described in section 7.2.10 of the $P4_{16}$ spec.

Statements and Expressions

- Assignment
  - An L-value cannot be used in a method call expression, packet out metadata, flex-header field, standard metadata field, or as an action parameter. Not all fixed header fields can be an L-value of an assignment statement. Please refer to the chart below.
- Conditional
  - Conditional statements are only supported within control apply blocks. Its expression must evaluate to a bit or bool type.
- switch statement
  - The switch statement is only supported within control apply block. Its expression must evaluate to a bool type.
  - The compiler supports empty switch statement, fall through, default case, and non-default cases. See spec section *11.7 Switch statement* for details.

The following tables describe the compiler support for expressions using the built in header fields and standard metadata as L-values and R-values. Note, this is separate of header fields that can be used as match keys.

> **Note**
>
> In the default hardware parser, some fields that are mutually exclusive are extracted to the same buffer location (referred to in the table as an alias). Assignments to and copy from these fields can use either of the aliased field names.

Fixed Header Fields	Assignable	Copyable	Notes
`headers.ethernet.dst_addr`	✔	✔	
`headers.ethernet.src_addr`	✔	✔	
`headers.ethernet.ether_type`	▯	▯	Last extracted outer `etherType` value
`headers.vlan.vlan_pcp`	▯	▯	
`headers.vlan.vlan_dei`	▯	▯	
`headers.vlan.vlan_id`	✔	✔	
`headers.vlan.vlan_ether_type`	▯	▯	Last extracted outer `etherType` value
`headers.inner_ethernet.dst_addr`	▯	✔	
`headers.inner_ethernet.src_addr`	▯	✔	
`headers.inner_ethernet.ether_type`	▯	▯	Last extracted inner `etherType` value
`headers.inner_vlan.vlan_pcp`	▯	▯	
`headers.inner_vlan.vlan_dei`	▯	▯	
`headers.inner_vlan.vlan_id`	▯	▯	
`headers.inner_vlan.vlan_ether_type`	▯	▯	Last extracted inner `etherType` value
`headers.ipv4.version` Alias with headers.ipv6.version	▯	▯	
`headers.ipv4.ihl`	▯	▯	
`headers.ipv4.diffserv`	▯	✔	Can be set through `nv_set_ip_dscp` extern

Fixed Header Fields	Assignable	Copyable	Notes
Alias with `headers.ipv6.diffserv`			
`headers.ipv4.ecn` Alias with `headers.ipv6.ecn`	□	✔	Can be set through `nv_set_ip_ecn` extern
`headers.ipv4.total_len`	□	□	Value is write only by hardware
`headers.ipv4.identification`	□	□	
`headers.ipv4.flags`	□	□	
`headers.ipv4.frag_offset`	□	□	
`headers.ipv4.ttl` Alias with `headers.ipv6.hop_limit`	□	✔	Can be set through `nv_set_ip_ttl` extern
`headers.ipv4.protocol` Alias with `headers.ipv6.next_header`	□	□	
`headers.ipv4.hdr_checksum`	□	□	Value is write only by hardware
`headers.ipv4.src_addr`	✔	✔	
`headers.ipv4.dst_addr`	✔	✔	
`headers.inner_ipv4.version` Alias with `headers.inner_ipv6.version`	□	□	
`headers.inner_ipv4.ihl`	□	□	
`headers.inner_ipv4.diffserv` Alias with `headers.inner_ipv6.diffserv`	□	✔	

Fixed Header Fields	Assignable	Copyable	Notes
`headers.inner_ipv4.ecn` Alias with `headers.inner_ipv6.ecn`	□	□	
`headers.inner_ipv4.total_len`	□	□	
`headers.inner_ipv4.identification`	□	□	
`headers.inner_ipv4.flags`	□	□	
`headers.inner_ipv4.frag_offset`	□	□	
`headers.inner_ipv4.ttl` Alias with `headers.inner_ipv6.hop_limit`	□	✔	
`headers.inner_ipv4.protocol` Alias with `headers.inner_ipv6.protocol`	□	□	
`headers.inner_ipv4.hdr_checksum`	□	□	
`headers.inner_ipv4.src_addr`	□	✔	
`headers.inner_ipv4.dst_addr`	□	✔	
`headers.ipv6.flow_label`	□	□	
`headers.ipv6.payload_length`	□	□	
`headers.ipv6.src_addr`	✔	✔	
`headers.ipv6.dst_addr`	✔	✔	

Fixed Header Fields	Assignable	Copyable	Notes
`headers.inner_ipv6.flow_label`	□	□	
`headers.inner_ipv6.payload_length`	□	□	
`headers.inner_ipv6.src_addr`	□	✔	
`headers.inner_ipv6.dst_addr`	□	✔	
`headers.mpls.label`	✔	✔	
`headers.mpls.tc`	✔	✔	
`headers.mpls.bos`	✔	✔	
`headers.mpls.ttl`	✔	✔	
`headers.inner_mpls.label`	□	✔	
`headers.inner_mpls.tc`	□	✔	
`headers.inner_mpls.bos`	□	✔	
`headers.inner_mpls.ttl`	□	✔	
`headers.icmp.type`	□	□	
`headers.icmp.code`	□	□	
`headers.icmp.checksum`	□	□	
`headers.icmp.identifier`	□	□	
`headers.icmp.sequence_number`	□	□	
`headers.inner_icmp.type`	□	□	
`headers.inner_icmp.code`	□	□	
`headers.inner_icmp.checksum`	□	□	

Fixed Header Fields	Assignable	Copyable	Notes
`headers.inner_icmp.identifier`	□	□	
`headers.inner_icmp.sequence_number`	□	□	
`headers.icmpv6.type`	□	□	
`headers.icmpv6.code`	□	□	
`headers.icmpv6.checksum`	□	□	
`headers.icmpv6.payload_1`	□	□	
`headers.icmpv6.payload_2`	□	□	
`headers.inner_icmpv6.type`	□	□	
`headers.inner_icmpv6.code`	□	□	
`headers.inner_icmpv6.checksum`	□	□	
`headers.inner_icmpv6.payload_1`	□	□	
`headers.inner_icmpv6.payload_2`	□	□	
`headers.tcp.src_port` Alias with `headers.udp.src_port`	□	✔	Can be set through `nv_set_l4_src_port` extern
`headers.tcp.dst_port` Alias with `headers.udp.dst_port`	□	✔	Can be set through `nv_set_l4_dst_port` extern
`headers.tcp.seq_no`	✔	✔	
`headers.tcp.ack_no`	✔	✔	
`headers.tcp.data_offset`	□	□	
`headers.tcp.res`	□	□	

Fixed Header Fields	Assignable	Copyable	Notes
`headers.tcp.nonce_sum`	□	□	
`headers.tcp.ecn`	✔	✔	
`headers.tcp.flags`	✔	✔	
`headers.tcp.window`	□	□	
`headers.tcp.checksum`	□	□	
`headers.tcp.urgent_ptr`	□	□	
`headers.inner_tcp.src_port` Alias with `headers.inner_udp.src_port`	□	✔	
`headers.inner_tcp.dst_port` Alias with `headers.inner_udp.dst_port`	□	✔	
`headers.inner_tcp.seq_no`	□	✔	
`headers.inner_tcp.ack_no`	□	✔	
`headers.inner_tcp.data_offset`	□	□	
`headers.inner_tcp.res`	□	□	
`headers.inner_tcp.nonce_sum`	□	□	
`headers.inner_tcp.ecn`	□	✔	
`headers.inner_tcp.flags`	□	✔	
`headers.inner_tcp.window`	□	□	
`headers.inner_tcp.checksum`	□	□	

Fixed Header Fields	Assignable	Copyable	Notes
`headers.inner_tcp.urgent_ptr`	⎕	⎕	
`headers.udp.length`	⎕	⎕	
`headers.udp.checksum`	⎕	⎕	
`headers.inner_udp.length`	⎕	⎕	
`headers.inner_udp.checksum`	⎕	⎕	
`headers.gre.checksum_present`	⎕	⎕	
`headers.gre.reserved1`	⎕	⎕	
`headers.gre.key_present`	⎕	⎕	
`headers.gre.sequence_present`	⎕	⎕	
`headers.gre.reserved2`	⎕	⎕	
`headers.gre.reserved3`	⎕	⎕	
`headers.gre.version`	⎕	⎕	
`headers.gre.protocol`	⎕	⎕	
`headers.nvgre_vsid.vsid`	⎕	⎕	
`headers.nvgre_vsid.flow_id`	⎕	⎕	
`headers.esp.security_parameters.index`	⎕	⎕	
`headers.esp.sequence_number`	⎕	⎕	
`headers.esp.next_header`	⎕	⎕	Value set by hardware after decryption
`headers.psp.next_header`	⎕	⎕	Value set by hardware after

Fixed Header Fields	Assignable	Copyable	Notes
			decryption
`headers.psp.hdr_ext_len`	□	□	
`headers.psp.crypt_offset`	□	□	
`headers.psp.needs_sampling`	□	□	
`headers.psp.drop`	□	□	
`headers.psp.version`	□	□	
`headers.psp.has_virtualization_key`	□	□	
`headers.psp.one_1`	□	□	
`headers.psp.security_parameters_index`	□	□	
`headers.psp.initialization_vector`	□	□	
`headers.psp.virtualization_key_high`	□	□	
`headers.psp.virtualization_key_low`	□	□	
`headers.vxlan.reserved1`	□	□	
`headers.vxlan.vni_valid`	□	□	
`headers.vxlan.reserved2`	□	□	
`headers.vxlan.reserved3`	□	□	
`headers.vxlan.vni`	✔	✔	
`headers.vxlan.reserved4`	□	□	
`headers.vxlan_gpe.reserved1`	□	□	

Fixed Header Fields	Assignable	Copyable	Notes
`headers.vxlan_gpe.vni_valid`	▯	▯	
`headers.vxlan_gpe.reserved2`	▯	▯	
`headers.vxlan_gpe.reserved3`	▯	▯	
`headers.vxlan_gpe.next_proto`	▯	▯	
`headers.vxlan_gpe.vni`	✔	✔	
`headers.vxlan_gpe.reserved4`	▯	▯	
`headers.geneve.ver`	▯	▯	
`headers.geneve.opt_len`	▯	▯	
`headers.geneve.o`	▯	▯	
`headers.geneve.c`	▯	▯	
`headers.geneve.reserved1`	▯	▯	
`headers.geneve.protocol_type`	▯	▯	
`headers.geneve.vni`	✔	✔	
`headers.geneve.reserved2`	▯	▯	

All the fields of BlueField standard metadata are read only. The following table outlines the current support for using a standard metadata field as an R-value in an expression.

Standard Metadata Fields	Copyable	Notes
`ingress_port`	✔	
`eth_to_fcs_packet_len`	▯	
`is_l2_ok`	▯	
`l2_type`	▯	
`last_l2_ether_type`	✔	Last extracted value of etherType with

Standard Metadata Fields	Copyable	Notes
		in ethernet header or VLAN tags
`vlan_type`	✓	
`is_l3_ok`	✓	
`l3_type`	✓	
`is_ip_fragmented`	✓	
`is_ipv4_checksum_ok`	✓	
`is_l4_ok`	✓	
`l4_type`	✓	

Standard Metadata Fields	Copyable	Notes
`l4_type_ext`	✓	
`is_l4_checksum_ok`	✓	
`l4_src_port`	✓	
`l4_dst_port`	✓	
`encap_type`	✓	ROCE not currently supported
`ipsec_layer`	✓	
`ipsec_syndrome`	✓	Valido

Standard Metadata Fields	Copyable	Notes
		nly after hardware encrypt/decrypt
`psp_syndrome`	￼	Valid only aft

Standard Metadata Fields	Copyable	Notes
		erhardware encrypt/decrypt
`is_inner_l2_ok`	▯	
`inner_l2_type`	▯	
`inner_last_l2_ether_type`	✔	Last extract

Standard Metadata Fields	Copyable	Notes
		edvalueoftherTypewithinnerethernethea

Standard Metadata Fields	Copyable	Notes
		der or VLAN tags
`inner_vlan_type`	□	
`is_inner_l3_ok`	□	
`inner_l3_type`	□	
`is_inner_ipv4_checksum_ok`	□	
`is_inner_l4_ok`	□	
`inner_l4_type`	□	
`inner_l4_type_ext`	□	
`random_value`	□	
`ut_clock`	□	
`fr_clock`	□	
`source_qp`	□	

Operators

The P4-16 language specification lists a wide variety of operations that the language accepts for the supported data types (see Section 8). The table below lists the operators that are officially supported by the NVIDIA P4 compiler:

Operator	Compile-time value	P4Runtime value	Runtime value	Spec section
Bool && Bool	✔	✔	✔	8.5
Bool || Bool	✔	✔	✔	8.5
Bool == Bool	✔	✔ [1]	✔ [1]	8.5
Bool != Bool	✔	✔ [1]	✔ [1]	8.5
Bit<W> == Bit<W>	✔	✔ [1]	✔ [1]	8.6
Bit<W> != Bit<W>	✔	✔ [1]	✔ [1]	8.6
Bit<W> << integer	✔ [2]	✔ [2]	✔ [2]	8.6
Bit<W> >> integer	✔ [2]	✔ [2]	✔ [2]	8.6
Bit<W>[H:L]	✔ [3]	✔ [3]	✔ [3]	8.6
All explicit casts between supported types	✔	✔	✔	8.11.1
All implicit casts between supported types	✔	✔	✔	8.11.2
Bit<W>..Bit<W>	✔ [4]	✔ [4]	▯	8.15.4
Assignment to user struct fields	✔	✔	✔	8.16
Assignment to packet-in struct fields	✔	✔	✔	8.16
All operations on header fields	✔	✔	✔ [5]	8.17
Method calls	✔	✔	▯	8.20

Operator	Compile-time value	P4Runtime value	Runtime value	Spec section
Function calls with positional args	✔	✔ 6	✔ 6	8.20
Extern constructor invocations	✔	▯	▯	8.21
Parser constructor invocations	✔	▯	▯	8.21
Control constructor invocations	✔	▯	▯	8.21
Package constructor invocations	✔	▯	▯	8.21

1. LHS or RHS must be compile-time constant __ __ __ __ __ __ __ __
2. RHS must be compile-time constant. See *spec 8.9.2* __ __ __ __ __ __
3. H and L are subject to the restrictions described in the *spec 8.6.* Assigning to slices (slices as L-values) is not supported. Additionally, slices as R-values are only supported as P4 table keys. __ __ __
4. Only valid in P4 Table entries __ __
5. Limited to those fields that can be a copy source __
6. Limited on a per extern function basis __ __

Variables

Variables are supported in accordance with the following spec items:

- Constants (*spec 11.1*)
    - "Compile-time known values" are evaluated on a best-effort basis. It is possible that a compile-time known value may not be recognized by the compiler as such.

- Variables (*spec 11.2*)
- Instantiations (*spec 11.3*)
    - Instantiations with abstract methods (*spec 11.3.1*) are allowed in BlueField Target Architecture
    - Named arguments are not supported

Variables may be declared in any of the locations described in (*spec 11.2*) and follow the scope rules described there.

The compiler will emit errors for uninitialized values. In some cases where a struct is partially initialized, only a warning may be produced. In some cases there may be no error emitted when an uninitialized struct field is accessed. The accessed field will then contain an undefined value.

Control Apply Block

The following statements are supported in a control's apply block:

- `table.apply()` calls
- `if` statement
- `switch` statement
- extern function and method calls
- assignment statements with the supported operators
- the empty statement
- `return` statements

The `exit` statement is not supported.

All supported expressions are allowed within these statements, where applicable.

Actions

Actions support the same statements as controls except for the following:

- `table.apply()` calls
- Conditional statements - `if` and `switch`

Actions support the same expressions as controls except for the following:

- Boolean logical operators - `&&`, `||`, ternary operator
- Comparisons (`==`, `!=`, etc.)

DPL Installation Guide

Introduction

The DPL Development Container (`dpl_dev`) bundles several tools for compiling, controlling and debugging DPL programs.

Pulling the container image can be done using the following command:

```
docker pull nvcr.io/nvidia/doca/dpl_dev:1.0.1-doca2.10.0-host
```

A set of convenience scripts is provided together with `dpl_dev` , one script per tool.

Each script launches each respective tool, requiring fewer command line options to be specified.

Installing the launch scripts

The `/install.sh` script provided within the `dpl_dev` container copies the launch scripts to the current working directory.

Below is an example command of how it can used to copy the launch scripts from within the container to a mounted directory of your choosing (specifically ${PWD} in this example).

```
docker run --rm -it -v ${PWD}:${PWD} -u $(id -u):$(id -g) -w
${PWD} nvcr.io/nvidia/doca/dpl_dev:1.0.1-doca2.10.0-host
/install.sh
```

Make sure the copied scripts have execute permission with:

Info

`chmod +x scripts/*.sh`

The above command copies the following scripts to the local host file system:

- Tool launch scripts
  - `dplp4c.sh`
  - `dpl_admin.sh`
  - `p4runtime_sh.sh`
  - `dpl_nspect.sh`
  - `dpl_debugger.sh`
- Configuration script
  - `scripts_config.sh`
- General purpose script for internal use only
  - `scripts_utils.sh`

For a detailed explanation on each tool, see DOCA Pipeline Language Developer Tool.

Configuring launch scripts

For your convenience, the `scripts_config.sh` file can be used to avoid specifying commonly used arguments when executing the scripts.

By editing the following file you can avoid passing the `-i <DPL Dev image name:tag>` and `-a <DPL Runtime daemon address:port>` to the various tools.

```
#!/bin/bash

# This configuration file defines various arguments used by the launch scripts.
# These variables are optional, but defining them in advance simplifies the use of the launch scripts.

# Image name and tag that contains the development tools
# Example: nvcr.io/nvidia/doca/dpl_dev:<tag>
DEV_IMAGE=nvcr.io/nvidia/doca/dpl_dev:1.0.1-doca2.10.0-host

# IP where the DPL Runtime daemon (RTD) is running
# Example: 192.168.1.100
# DPL_RTD_IP=

# IP where the DPL RTD is running and the port for the P4 Runtime Server
# Example: ${DPL_RTD_IP}:9559
# P4RT_ADDRESS=${DPL_RTD_IP}:9559

# IP where the DPL RTD is running and the port for the DPL Admin
# Example: ${DPL_RTD_IP}:9600
# DPL_ADMIN_ADDRESS=${DPL_RTD_IP}:9600

# IP where the DPL RTD is running and the port for the DPL Nspect
# Example: ${DPL_RTD_IP}:9560
# DPL_NSPECT_ADDRESS=${DPL_RTD_IP}:9560
```

Next steps

1. Compiling DPL Applications that you wish to examine using the developer tools.
2. Loading DPL Applications can now be done with the compilation outputs.
3. Use the DOCA Pipeline Language Developer Tool to further examine the DPL programs correct operation.

Compiling DPL Applications

This section describes how to use the NVIDIA DPL compiler to compile DPL applications.

Introduction

DOCA Programming Language (DPL) applications are an educational resource provided as a guide on how to program on the NVIDIA® BlueField® networking platform (DPU or SuperNIC) using DPL.

The NVIDIA DPL compiler (dplp4c.sh) is provided as an executable in a self contained docker image. The docker image is available to partners by logging into the NVIDIA NGC system and downloading the most recent DPL docker containers.

Installation

Please refer to the DPL Runtime Service for the system requirements and DPL Installation Guide on how to install the compiler. For information on the development environment, refer to the DOCA Pipeline Language Model and the DPL System Overview.

Dependencies

- BlueField-3 is required
- Ubuntu 22.04 hosts (x86) or greater
- A P4Runtime controller based on API version 1.3.0 or greater

Compiling Applications

The DPL compiler is provided with all its dependencies in the dpl-dev container. For details on how to obtain and install the docker images see DPL Installation Guide .

A shell script, dplp4c.sh, is provided for a convenient way to execute the DPL compiler from the dpl-dev container:

```
./dplp4c.sh [--mount dir]* dplp4c_args*
```

- `--mount` – (optional) directory to be mounted into the container
- `dplp4c_args` – arguments to be passed to the compiler (e.g., `sample.p4`)

For example, compiling the "hello_packet" DPL example is as simple as:

```
local-user@vm-1:~/p4-samples/dpu/hello_packet$ dplp4c.sh hello_packet.p4
Generating compiler output in "_out"
updating: MANIFEST.json (deflated 40%)
updating: hello_packet.program.json (deflated 87%)
updating: hello_packet.debug.json (deflated 96%)
```

The following files in this example are produced in the _out directory:

`compiler.log`	Log of any compiler warnings or errors to the specified program
`hello_packet.dplconfig`	Binary blob, containing all the data needed for the DPL Runtime daemon to load the program
`hello_packet.p4info.txt`	P4Runtime protobuf file, in text format

Additional NVIDIA® BlueField®-specific arguments include:

```
--help                                  Print this help message
--version                               Print compiler version
-I path                                 Specify include path
(passed to preprocessor)
-D arg=value                            Define macro (passed to
preprocessor)
-U arg                                  Undefine macro (passed to
preprocessor)
-E                                      Preprocess only, do not
compile (prints program on stdout)
```

```
-M                                      Output `make` dependency
rule only (passed to preprocessor)
-MD                                     Output `make` dependency
rule to file as side effect (passed to preprocessor)
-MF file                                With -M, specify output
file for dependencies (passed to preprocessor)
-MG                                     with -M, suppress errors
for missing headers (passed to preprocessor)
-MP                                     with -M, add phony target
for each dependency (passed to preprocessor)
-MT target                              With -M, override target
of the output rule (passed to preprocessor)
-MQ target                              Like -Mt, override target
but quote special characters (passed to preprocessor)
-g                                      Enable debugging via DPL
Nspect
--nocpp                                 Skip preprocess, assume
input file is already preprocessed.
--Wdisable[=diagnostic]                 Disable a compiler
diagnostic, or disable all warnings if no diagnostic is
specified.
--Winfo[=diagnostic]                    Report an info message
for a compiler diagnostic.
--Wwarn[=diagnostic]                    Report a warning for a
compiler diagnostic, or treat all info messages as warnings if no
diagnostic is specified.
--Werror[=diagnostic]                   Report an error for a
compiler diagnostic, or treat all warnings as errors if no
diagnostic is specified.
--maxErrorCount errorCount              Set the maximum number of
errors to display before failing.
--target target                         Compile for the specified
target device.
--odir out_directory                    Write output to out
directory
```

```
--enable feature[,feature]*|--help      Enable a feature, or
comma-separated list of features
--disable feature[,feature]*|--help     Disable a feature, or
comma-separated list of features
```

See section Compiling DPL Applications for examples of how to use the dplp4c.sh to compile.

Info

The binary `<sample_name>.` is created under `_out/<sample_name>.dplconfig`.

Info

The P4Runtime file is created under `_out/<sample_name>.p4info.txt`.

Compiling Applications in Debug Mode

To enable packet debugging, you must ensure the following:

- Compile your DPL program with the `-g` flag. This enables parsing for a debug flow.
- Ensure you have at least one `nv_send_debug_pkt()` call in your DPL program. A debug packet will be emitted where `nv_send_debug_pkt()` is placed, and represent the program state at that point.

Please note packet debugging is only available on the RX direction. Debug packets will not be produced on TX. You may optionally include a "cookie", within your debug extern call, i.e.: `nv_send_debug_pkt(8w0x42)`. This cookie may help differentiate between different debug extern calls, and will appear in your debugging output.

Loading DPL Applications

NVIDIA DPL programs are deployed to the NVIDIA® BlueField® networking platform (DPU or SuperNIC) using the P4Runtime API. This allows platform-independent, standards-based integration with SDN controllers.

Introduction

The DPL compiler generates a pipeline binary optimized for execution on BlueField. Pipeline loading and control—such as installing the program and populating P4 tables—are handled via the P4Runtime API, an open and well-defined interface.

The P4Runtime server, running on the BlueField device, enables a P4 Controller to:

- Connect over gRPC
- Set the `ForwardingPipelineConfig` (i.e., install the compiled DPL binary and `p4info`)
- Query the device for its current pipeline config and table state
- Maintain runtime P4 tables as defined in the DPL source

This model enables integration with open-source, proprietary, or custom-built controllers in a standardized way.

Prerequisites

Before loading a DPL application, ensure the following services and components are properly set up:

- DPL Runtime Service is running and configured on the BlueField (Arm side). See the Container Deployment page for setup instructions.
- DPL Development Container and the `p4runtime_sh.sh` launch script are installed on the host. See the DPL Installation Guide for more details.

Loading the Application

The a DPL application can be loaded using:

- A custom P4Runtime controller
- The the NVIDIA-supplied Python controller
- An open source controller

In the following example we'll be using the P4Runtime controller bundled within the DPL Development Container.

Using p4runtime_sh.sh Script

Running the script with no arguments displays the usage information:

```
usage: p4runtime_sh.sh -i <docker image> -p <program_folder> -a
<dpl_rtd_ip:port> [OPTIONS]
```

Example Command

```
p4runtime_sh.sh -i doca_p4_dev:latest -p /root/hello_packet/_out
-a 192.168.1.100:9559 --device-id 1000
```

Arguments:

- `-i` - The pulled DPL Development Container image.
- `-p` - Directory that holds the DPL program compilation outputs (Compiling DPL Applications).
- `-a` - Address of the DPL Runtime daemon and the P4Runtime port as specified in the DPL Runtime Service Configuration.
- `--device-id` – (Optional) ID of the target device

After successful loading, the script launches an interactive Python shell connected to the P4Runtime server. From here, you can inspect and manipulate tables (see p4runtime_sh Usage).

P4Runtime Optimizations

DPL table entries are added via a P4Runtime controller, which may run remotely or locally on BlueField.

- The standard gRPC-based controller model supports ~50K rule insertions per second
- For use cases requiring high-speed (1M+) rule insertions, NVIDIA is introducing a bulk insertion API extension to the P4Runtime protobuf specification

> **Note**
>
> This feature is planned for future releases.

Sample DPL Applications

This section describes NVIDIA DOCA Pipeline Language sample applications.

Sample Prerequisites

The DPL program can be run compiled on any host machine with the DPL compiler installed. The resulting binary can be loaded on either the host machine or on an NVIDIA® BlueField®-3 DPU target where the DPL Runtime daemon is running. See to the following documents:

-
    - DPL Runtime Service for details on how to install BlueField-related software.
    - DPL Release Notes for any issue you may encounter with the installation, compilation, or execution of DPL samples.

Running the Sample

Compile the DPL sample program, locate the generated .p4info and .dplconfig files from the compiler output directory, and load them to the DPL Runtime daemon using a P4Runtime controller.

Sample Applications

Hello Packet

This sample program demonstrates a basic match/action pipeline, with a simple match and a drop or send to port action.

GTP Parsing

This sample program demonstrates how to add a custom protocol to the native BlueField Platform parser and match on those fields.

GTP Tunnel Encapsulation

This sample program demonstrates how to add a custom tunnel encapsulation, GTP-U, that was not natively supported on the BlueField platform.

Geneve TLV Parsing

This sample program demonstrates how to add a custom protocol that contains TLVs, and to match on the TLV fields.

VXLAN Tunnel Gateway

This sample program demonstrates how to create a tunnel gateway between two different VXLAN domains.

Connection Tracking

This sample program shows how to program the data plane portion of a basic TCP connection tracking solution. Packets are sent to a controller to manage the table entries.

Host-based Networking

This sample program provides a stretched L2 tenant pipeline with VXLAN encapsulation/decapsulation on top of the BlueField Platform.

Hello Packet Example

This example demonstrates the most basic pipeline in the DOCA target architecture. It consists of:

- Match on L2 source MAC address
- Action to either forward to a P4 port ID, drop, or no action

Sample Code

The following #includes are required for every DPL program. They define the DOCA target architecture. In particular, they contain things like the default DOCA parser, default headers struct, etc. Note that the symbol names are prefixed with "nv_" and are reserved for NVIDIA DOCA TA usage.

```
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>
```

The DOCA TA features a single control, which requires a headers struct and standard metadata. User metadata and packet out metadata, defined by the user, are optional.

```
control hello_packet(
inout nv_headers_t headers,
in nv_standard_metadata_t std_meta,
inout nv_empty_metadata_t user_meta,
inout nv_empty_metadata_t pkt_out_meta
) {
action drop() {
nv_drop();
}
action forward(bit<32> port) {
nv_send_to_port(port);
}
table forward_table {
key = {
headers.ethernet.dst_addr : exact;
}
actions = {
drop;
forward;
NoAction;
}
default_action = forward(3);
const entries = {
(48w0x001111111111) : forward(1);
```

```
(48w0x002222222222) : forward(2);
(48w0x00dddddddddd) : drop();
(48w0x00aaaaaaaaaa) : NoAction();
(48w0x00bbbbbbbbbb) : NoAction();
}
}
apply {
forward_table.apply();
}
}
```

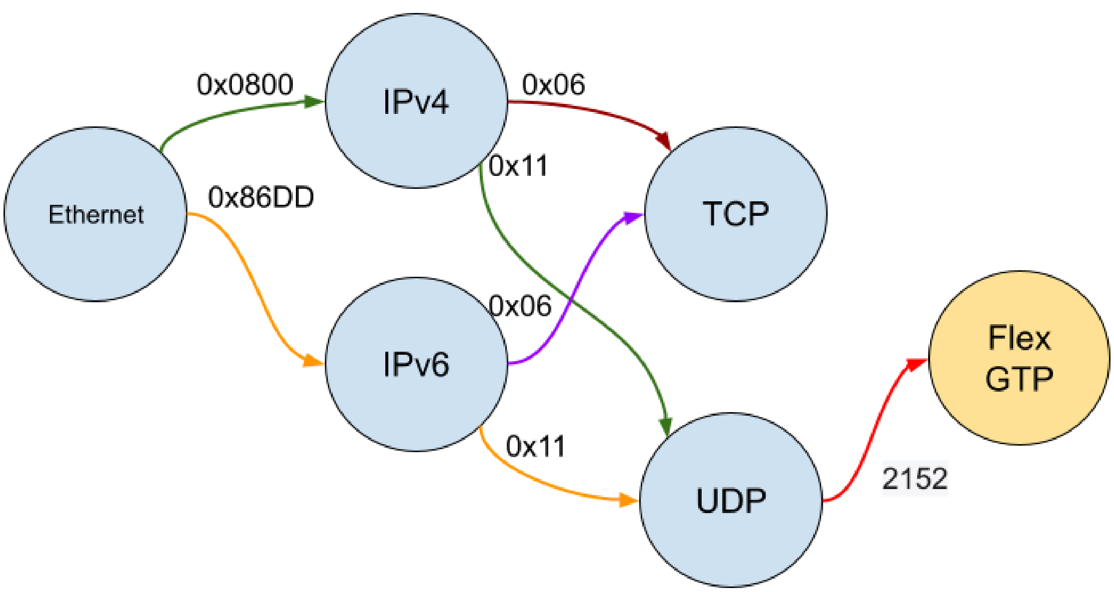
Finally, the main package must be instantiated:

```
NvDocaPipeline(
nv_fixed_parser(),
hello_packet()
) main;
```

See the full DPL example hello_packet.p4

GTP Parsing Example

This example demonstrates how to add a simple flex parser node to the existing hardware defined parse graph.



Sample Code

The example starts with some basic definitions.

```
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

const bit<32> WIRE_PORT = 32w00;
const bit<32> GTP_VPORT = 32w01;
const bit<32> DEFAULT_VPORT = 32w04;

struct metadata_t {
}
#define GTP_U_PORT 2152
```

Then we define the GTP-U version 1 header.

```
header Gtp_v1_h {
    bit<3>        version;                /** For GTPv1, this has a value of 1. */
    bit           protocol_type;          /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag;  /** extension header optional field. */
    bit           seq_number_flag;        /** Sequence Number optional field */
    bit           n_pdu_number_flag;      /** N-PDU number optional field */
    bit<8>        message_type;           /** types of messages are defined in
3GPP TS 29.060 section 7.1 */
    bit<16>       message_length;         /** length of the payload in bytes */
    bit<32>       teid;                   /** Tunnel endpoint identifier */
    bit<16>       sequence_number;        /** optional */
    bit<8>        n_pdu_number;           /** optional */
    bit<8>        next_extension_hdr_type;  /** optional if any of the E, S, or PN
bits are on. The field must be interpreted only if the E bit is on */
}
```

Then we add NV_FIXED_HEADERS to the headers struct, along with the new GTP header.

```
struct headers_t {
    NV_FIXED_HEADERS
    Gtp_v1_h   gtpv1;
}
```

Using the nv_transition_from annotation, the GTP parser state is connected as a select transition from the UDP state.

```
parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
    {
        packet.extract(headers.gtpv1);
        transition accept;
    }
}
```

The control example uses a single flow table that matches on input port and GTP tunnel endpoint ID. The policy is then to forward the GTP packet to a port or drop the packet.

```
/**
 * This control admits GTP packets only if the tunnel ID matches
 *
 */
 control gtp_tunnel(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) gtp_counter;
```

```
action send_to_port(nv_logical_port_t port) {
    gtp_counter.count();
    nv_send_to_port(port);
}

action drop() {
    gtp_counter.count();
    nv_drop();
}

table gtp_table {
    key = {
        std_meta.ingress_port: exact;
        headers.gtpv1.teid: exact;
    }
    actions = {
        send_to_port;
        drop;
    }
    default_action = drop;
    direct_counter = gtp_counter;

    const entries = {
        (WIRE_PORT, 0x000000001) : send_to_port(GTP_VPORT);
        (GTP_VPORT, 0x000000001) : send_to_port(WIRE_PORT);
    }
}

apply {
    if (headers.gtpv1.isValid()) {
        if (gtp_table.apply().miss) {
            nv_send_to_port(DEFAULT_VPORT);
        }
    }
    drop();
```

```
    }
}

NvDocaPipeline(
    packet_parser(),
    gtp_tunnel()
) main;
```

See the full DPL example gtp_parsing.p4

GTP Tunnel Encapsulation Example

The following example demonstrates GTPv1 encapsulation which must be declared as a custom header:

```
header gtp_v1_h {
    bit<3>        version;               /** For GTPv1, this has a value of 1. */
    bit           protocol_type;         /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag; /** extension header optional field. */
    bit           seq_number_flag;       /** Sequence Number optional field */
    bit           n_pdu_number_flag;     /** N-PDU number optional field */
    bit<8>        message_type;          /** types of messages are defined in
3GPP TS 29.060 section 7.1 */
    bit<16>       message_length;        /** length of the payload in bytes */
    bit<32>       teid;                  /** Tunnel endpoint identifier */
    bit<16>       sequence_number;       /** optional */
    bit<8>        n_pdu_number;          /** optional */
    bit<8>        next_extension_hdr_type; /** optional if any of the E, S, or PN
bits are on. The field must be interpreted only if the E bit is on */
}
```

The custom GTPv1 header can optionally be added to the parse graph (see Custom Parser States) is optional, and is only required if any of the custom header fields need to be read from or modified after encapsulation.

```
struct headers_t {
    NV_FIXED_HEADERS
    gtp_v1_h   gtpv1;
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
    {
        packet.extract(headers.gtpv1);
        transition nv_parse_inner_ipv4;
    }
}
```

The extern object NvTunnelTemplate<HEADER_TYPE> requires a struct type, for which the entire underlay header is defined as a struct:

```
struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    nv_ipv4_h     ipv4;
    nv_udp_h      udp;
    gtp_v1_h      gtpv1;
}
```

Finally, when declaring a NvTunnelTemplate object, the annotation `@nv_tunnel_fields` must be present. This annotation contains a key-value entry for each header in the specified struct type. Under each header, each header field must be specified in the order that the field appears in the header. For the GTPv1 example, a NvTunnelTemplate would be declared in the main control:

```
@nv_tunnel_fields(
    ethernet = {
        dst_addr = "variable",
        src_addr = "variable",
        ether_type = 0x0800
    },
    ipv4 = {
        version = 0x4,
        ihl = 0x5,
        diffserv = 0,
        ecn = "ignore",                // Cannot set
        total_len = "ignore",          // reparse will calculate, cannot set
        identification = "ignore",     // Cannot set
        flags = 0,
        frag_offset = 0,
        ttl = 64,
        protocol = 17,
        hdr_checksum = "ignore",       // reparse will calculate, cannot set
        src_addr = "variable",
        dst_addr = "variable"
    },
    udp = {
        src_port = "ignore",           // reparse will set, cannot be set manually
        dst_port = "variable",
        length   = "ignore",           // reparse will calculate, cannot set
        checksum = "ignore"            // Cannot set
    },
    gtpv1 = {
        version = 1,
        protocol_type = 1,
        reserved = 0,
        extension_header_flag = 1,
        seq_number_flag = 1,
        n_pdu_number_flag = 1,
        message_type = 0xFF,      // T-PDU
```

```
        message_length = "variable",
        teid = "variable",
        sequence_number = 0,
        n_pdu_number = 0,
        next_extension_hdr_type = 0
    }
)
NvTunnelTemplate<tunnel_headers_t>() gtpv1Tunnel;
```

The type `tunnel_headers_t` determines the required structure of the annotation. Fields marked "variable" will be determined according to the arguments passed to `nv_set_l2tunnel_underlay` or `nv_set_l3tunnel_underlay`. Some fields will always be overwritten by reparse, such as IPv4 length, IPv4 checksum, UDP length, etc which must be calculated and cannot be set by the user. Such fields can be marked "ignore". Marking other fields "ignore", would set them to zero.

Finally, the instantiated NvTunnelTemplate object can be used in one of the custom tunnel externs. For example:

```
action forward(nv_ipv4_addr_t src_addr) {
    nv_set_l3tunnel_underlay(headers, gtpv1Tunnel, {
        48w0x001A2B3C4D5E,  // ethernet.dst_addr
        48w0x00F1E2D3C4B5,  // ethernet.src_addr
        src_addr,           // ipv4.src_addr
        32w0xC07B7B01,      // ipv4.dst_addr
        16w0x868,           // udp.dst_port for GTP-U
        16w0x4,             // gtpv1.message_length
        32w0x1234567        // gtpv1.teid
    });
    nv_send_to_port(1);
}
```

The third argument to `nv_set_l3tunnel_underlay` is required to be a list expression where each element corresponds to a header field marked "variable" in the

NvTunnelTemplate annotation. The order of the values in the list is the order in which the header fields appear in the packet.

Full example of gtp_tunnel.p4:

```
/*
 * SPDX-FileCopyrightText: Copyright (c) 2025 NVIDIA CORPORATION & AFFILIATES. All rights reserved.
 * SPDX-License-Identifier: LicenseRef-NvidiaProprietary
 *
 * NVIDIA CORPORATION, its affiliates and licensors retain all intellectual
 * property and proprietary rights in and to this material, related
 * documentation and any modifications thereto. Any use, reproduction,
 * disclosure or distribution of this material and related documentation
 * without an express license agreement from NVIDIA CORPORATION or
 * its affiliates is strictly prohibited.
 */

#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

#define GTP_U_PORT 2152

header gtp_v1_h {
    bit<3>        version;                /** For GTPv1, this has a value of 1. */
    bit           protocol_type;          /** GTP (value 1) from GTP' (value 0) */
    bit           reserved;
    bit           extension_header_flag;  /** extension header optional field. */
    bit           seq_number_flag;        /** Sequence Number optional field */
    bit           n_pdu_number_flag;      /** N-PDU number optional field */
    bit<8>        message_type;           /** types of messages are defined in
3GPP TS 29.060 section 7.1 */
    bit<16>       message_length;         /** length of the payload in bytes */
    bit<32>       teid;                   /** Tunnel endpoint identifier */
    bit<16>       sequence_number;        /** optional */
    bit<8>        n_pdu_number;           /** optional */
    bit<8>        next_extension_hdr_type;  /** optional if any of the E, S, or PN
bits are on. The field must be interpreted only if the E bit is on */
```

```
}

struct headers_t {
    NV_FIXED_HEADERS
    gtp_v1_h   gtpv1;
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GTP_U_PORT)
    state parse_gtp
    {
        packet.extract(headers.gtpv1);
        transition nv_parse_inner_ipv4;
    }
}

struct tunnel_headers_t {
    nv_ethernet_h ethernet;
    nv_ipv4_h     ipv4;
    nv_udp_h      udp;
    gtp_v1_h      gtpv1;
}

control c(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    @nv_tunnel_fields(
        ethernet = {
            dst_addr = "variable",
            src_addr = "variable",
            ether_type = 0x0800
```

```
        },
        ipv4 = {
            version = 0x4,
            ihl = 0x5,
            diffserv = 0,
            ecn = "ignore",                   // cannot set
            total_len = "ignore",             // HW will calculate, cannot set
            identification = "ignore",        // cannot set
            flags = 0,
            frag_offset = 0,
            ttl = 64,
            protocol = 17,
            hdr_checksum = "ignore",          // HW will calculate, cannot set
            src_addr = "variable",
            dst_addr = "variable"
        },
        udp = {
            src_port = "ignore",              // HW will calculate entropy, cannot set
            dst_port = GTP_U_PORT,
            length   = "ignore",              // HW will calculate, cannot set
            checksum = "ignore"               // cannot set
        },
        gtpv1 = {
            version = 1,
            protocol_type = 1,
            reserved = 0,
            extension_header_flag = 1,
            seq_number_flag = 1,
            n_pdu_number_flag = 1,
            message_type = 0xFF,
            message_length = "variable",
            teid = "variable",
            sequence_number = 123,
            n_pdu_number = 255,
            next_extension_hdr_type = 0
        }
```

```
)
NvTunnelTemplate<tunnel_headers_t>() gtpv1Tunnel;

action drop() {
    nv_drop();
}

action forward(nv_logical_port_t port) {
    nv_send_to_port(port);
}

action tunnel(nv_ipv4_addr_t dst_addr, bit<32> teid) {
    nv_set_l3tunnel_underlay(headers, gtpv1Tunnel, {
        48w0x1, 48w0x2, 32w0x3, dst_addr,
        16w4, /* 4 bytes of optional GTP-U payload */
        teid
    });
}

table ip_as_key {
    key = {
        headers.ipv4.src_addr : exact;
    }
    actions = {
        NoAction;
        drop;
        forward;
        tunnel;
    }
    size = 128;
    default_action = NoAction;
    const entries = {
        (32w0x01010101) : tunnel(32w0x11111111, 1);
        (32w0x02020202) : tunnel(32w0x22222222, 2);
        (32w0x03030303) : tunnel(32w0x33333333, 3);
    }
```

```
    }

    table teid_as_key {
        key = {
            headers.gtpv1.teid : exact;
        }
        actions = {
            NoAction;
            drop;
            forward;
        }
        size = 128;
        default_action = forward(3);
        const entries = {
            (32w0x01) : forward(1);
            (32w0x02) : forward(2);

        }
    }

    apply {
        ip_as_key.apply();
        if (headers.gtpv1.isValid()) {
            teid_as_key.apply();
        }
        forward(3);
    }
}

NvDocaPipeline(
    packet_parser(),
    c()
) main;
```

Geneve TLV Parsing Example

This example demonstrates how to add a custom protocol header that supports optional Type-Length-Value (TLV) fields. In the P4-16 specification, there is no first class support in the parser primitives, and the user needs to manually.

Geneve is short for Generic Network Virtualization Encapsulation, a common L2 tunnel header. The Geneve header format is:

```
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+
 |Ver|  Opt Len  |O|C|    Rsvd.  |          Protocol Type
|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+
 |        Virtual Network Identifier (VNI)       |    Reserved
|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+
 |
|
 ~                    Variable-Length Options
~
 |
|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+

The Geneve option header is:

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+
```

```
 |          Option Class         |       Type    |R|R|R|  Length
|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+
 |                ~ Variable-Length Option Data ~
|
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+
```

Sample Code

First we define the Geneve header, per RFC 8926.

```
header geneve_t {
    bit<2> ver;
    bit<6> opt_len;
    bit<1> o;
    bit<1> c;
    bit<6> reserved1;
    bit<16> protocol_type;
    bit<24> vni;
    bit<8> reserved2;
};
```

Next we define a struct that contains the base Geneve Option header, shared by all option types.

```
// Struct so it can be a field within other headers
struct geneve_option_t {
    bit<24> option_class_type;
    bit<3> reserved;
    bit<5> length;
```

```
};
```

Lastly, depending on the option class and type, the variable length option data can be defined. In this example, we model the Geneve options used by the P4 Inband Telemetry (INT) specification.

```
// Intermediate nodes (INT Transit Hops) must process this type of INT Header.
header geneve_option_int_md_t {
    geneve_option_t base;
    bit<4> version;
    bit<1> discard;
    bit<1> exceeded_max_hops;
    bit<1> mtu_exceeded;
    bit<12> reserved;
    bit<5> hop_ml;
    bit<8> remaining_hop_count;
    bit<16> instruction_bitmap;
    bit<16> ds_id;
    bit<16> ds_instruction;
    bit<16> ds_flags;
};
// Destination headers can be used to enable Edge-to-Edge communication between
// the INT Source and INT Sink.
header geneve_option_int_destination_t {
    geneve_option_t base;
    // Destination headers must only be consumed by the INT Sink
};

// Intermediate nodes (INT Transit Hops) must process this type of INT Header
// and generate reports to the monitoring system as instructed.
header geneve_option_int_mx_t {
    geneve_option_t base;
    bit<4> version;
    bit<1> discard;
    bit<27> reserved1;
```

```
    bit<16> instruction_bitmap;
    bit<16> ds_id;
    bit<16> ds_instruction;
    bit<16> ds_flags;
};
```

All the above headers and structs must be added to the application's headers struct, along with the NV_FIXED_HEADERS.

```
struct app_headers {
    NV_FIXED_HEADERS

    geneve_t custom_geneve;
    // geneve_option_t is not explicitly used by the P4 program, so a reference
    // placed in the headers the type is not optimized out. The NvOptionParser
    // instantiation references this type it by string name.
    geneve_option_t geneve_opt;
    geneve_option_int_md_t geneve_opt_int_md;
    geneve_option_int_destination_t geneve_opt_int_destination;
    geneve_option_int_mx_t geneve_opt_int_mx;
};
```

The last step for the parser is to define the parser state that will perform Geneve parsing. An extern object, NvOptionsParser, is used to further parse into the Geneve options. In this example, the TLV "type" is a combination of Option class and type:

- Class 0x103, type 1 (INT-MD)
- Class 0x103, type 2 (Destination-type)
- Class 0x103, type 3 (INT-MX)

```
parser geneve_parser(
    packet_in packet,
```

```
    out app_headers headers
) {
    NvOptionParser<bit<24>, _>(
        "opt_len",                       // options_length_field
        2,                               // options_length_shift
        0,                               // options_length_add
        "geneve_option_t",               // option_layout_header_type
        "length",                        // option_length_field
        0,                               // option_length_shift
        4,                               // option_length_add
        "option_class_type",             // option_type_field
        (list<tuple<bit<24>, _>>){       // options
            {24w0x010301, "headers.geneve_opt_int_md"},
            {24w0x010302, "headers.geneve_opt_int_destination"},
            {24w0x010303, "headers.geneve_opt_int_mx"}
        }
    ) geneveOptions;

    NV_FIXED_PARSER(packet, headers)

    @nv_transition_from("nv_parse_udp", GENEVE_PORT)
    state parse_geneve {
        // Fixed geneve is only an example; “base” header must be flex.
        packet.extract(headers.custom_geneve);
        geneveOptions.parseOptions(packet, headers);
        transition select(headers.custom_geneve.protocol_type) {
            NV_TYPE_IPV4 : nv_parse_inner_ipv4;
            NV_TYPE_IPV6 : nv_parse_inner_ipv6;
            NV_TYPE_MAC : nv_parse_inner_ethernet;
            default : accept;
        }
    }
}
```

The control below shows how a match action table can be configured to now match on a specific INT domain, and perform per ID forwarding.

```
control int_over_geneve(
    inout app_headers headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    action forward(bit<32> port) {
        nv_send_to_port(port);
    }

    action drop() {
        nv_drop();
    }

    table geneve_option_int_md_table {
        key = {
            headers.geneve_opt_int_md.ds_id : exact;
        }
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (16w0x100) : forward(1);
        }
    }

    table geneve_option_int_destination_table {
        key = {
            headers.custom_geneve.vni : exact;
        }
```

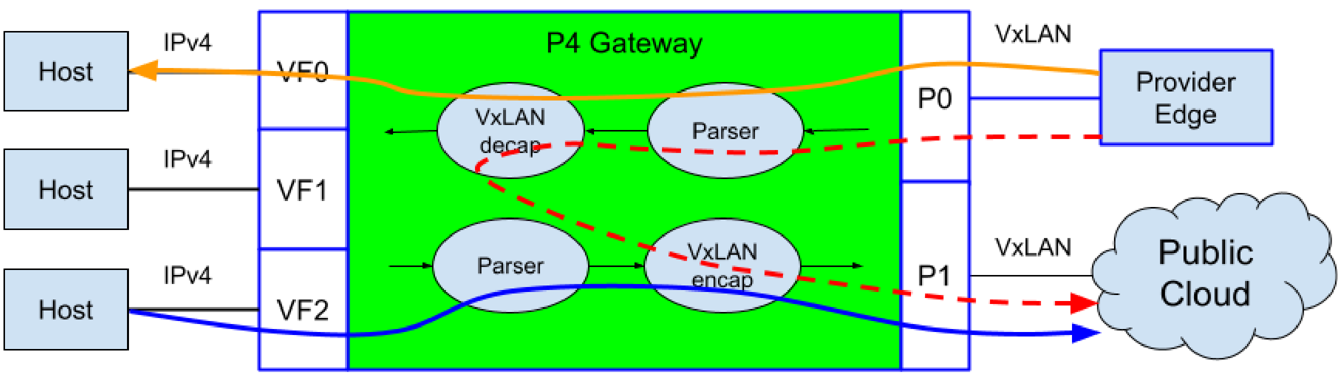
```
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (24w0x200) : forward(2);
        }
    }

    table geneve_option_int_mx_table {
        key = {
            headers.geneve_opt_int_mx.ds_id : exact;
        }
        actions = {
            forward;
            NoAction;
        }
        default_action = NoAction();
        const entries = {
            (16w0x300) : forward(3);
        }
    }

    apply {
        if (headers.custom_geneve.isValid()) {
            geneve_option_int_md_table.apply();
            geneve_option_int_destination_table.apply();
            geneve_option_int_mx_table.apply();
        }
        drop();
    }
}

NvDocaPipeline(
    geneve_parser(),
```

```
    int_over_geneve()
) main;
```

See the full DPL example geneve_tlv.p4

VXLAN Tunnel Gateway Example

This example illustrates how to write a basic VXLAN tunnel gateway using the DOCA target architecture. A tunnel gateway allows programmatic control over how VXLAN traffic can be "stitched" packets across tenant domains. In this example, end point traffic destined to local bare metal hosts can be decapsulated and forwarded to a VF, while gateway traffic can be decapsulated, re-encapsulated and sent back to the wire. For example, this program can be easily extended to be a gateway connecting legacy NVGRE networks to a VXLAN-GPE network.



Sample Code

This example users the native parser and 2 tables, of size 32K each. The wire port is configured with P4 port ID 0. A single bit in the user metadata structure is used to keep the decapsulation state.

```
#include <doca_model.p4>
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

/*
```

```
* Table sizes.
*/
const bit<32> DECAP_TABLE_SIZE = 32768;
const bit<32> ENCAP_TABLE_SIZE = 32768;

/* The directionality is based on network to host
 * The user will configure the P4 port IDs in the OVS configuration
 */
const bit<32> WIRE_PORT = 32w0;

struct metadata_t {
    bit<1> was_decapped;
}

struct headers_t {
    NV_FIXED_HEADERS
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)
}
```

The encapsulation control has a single table, matching on IPv4 destination address. If an entry matches, the packet is VXLAN encapsulated and forwarded to the specified port. If the packet does not hit any entry, then the packet is dropped. It is simple to add more complex policy rules such as a 5 tuple ACL.

```
/*
* This control performs the overlay policy including L2 encap with VXLAN
*/
control overlay_encap(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
```

```
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES)
encap_counter;

    action deny() {
        nv_drop();
    }

    action vxlan_v4_encap(nv_mac_addr_t underlay_src_mac,
nv_mac_addr_t underlay_dst_mac,
                        nv_ipv4_addr_t underlay_sip, nv_ipv4_addr_t
underlay_dip, bit<24> vni, nv_logical_port_t port) {
        encap_counter.count();
        nv_set_vxlan_v4_underlay(headers, false, underlay_dst_mac,
underlay_src_mac, 0, underlay_sip, underlay_dip, vni);
        nv_send_to_port(port);
    }
    table encap_v4_table {
        key = {
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            vxlan_v4_encap;
            deny;
        }
        size = ENCAP_TABLE_SIZE;
        default_action = deny;
        direct_counter = encap_counter;
    }

    apply {
       if (headers.ipv4.isValid() && (user_meta.was_decapped ==
1)) {
            encap_v4_table.apply();
       }
    }
}
```

The decapsulation control simply checks if the packet is VXLAN, and decapsulates it. From there the packet can be sent directly to a port, or hair-pinned back to the wire.

```
/*
* This control is for packets from wire to host (RX)
* and includes policy for L2 decap
*/
control decap_flow(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES)
decap_counter;

    action deny() {
        nv_drop();
    }

    action decap() {
        decap_counter.count();
        nv_l2_decap(headers);
        user_meta.was_decapped = 1;
    }

    action to_port(nv_logical_port_t port) {
        nv_send_to_port(port);
    }

    action decap_to_port(nv_logical_port_t port) {
        decap_counter.count();
        user_meta.was_decapped = 1;
        nv_l2_decap(headers);
        nv_send_to_port(port);
```

```
    }

    table decap_v4_table {
        key = {
            headers.vxlan.vni : exact;
        }
        actions = {
            decap;
            to_port;
            decap_to_port;
            deny;
            NoAction;
        }
        size = DECAP_TABLE_SIZE;
        direct_counter = decap_counter;
        default_action = deny;
    }

    apply {
        if (headers.vxlan.isValid()) {
            decap_v4_table.apply();
        }
    }
}
```

The main control checks the ingress port to determine if the packet is Network to Host, or Host to Network. Depending on the direction, it applies the decap_flow control, or performs an overlay encapsulation.

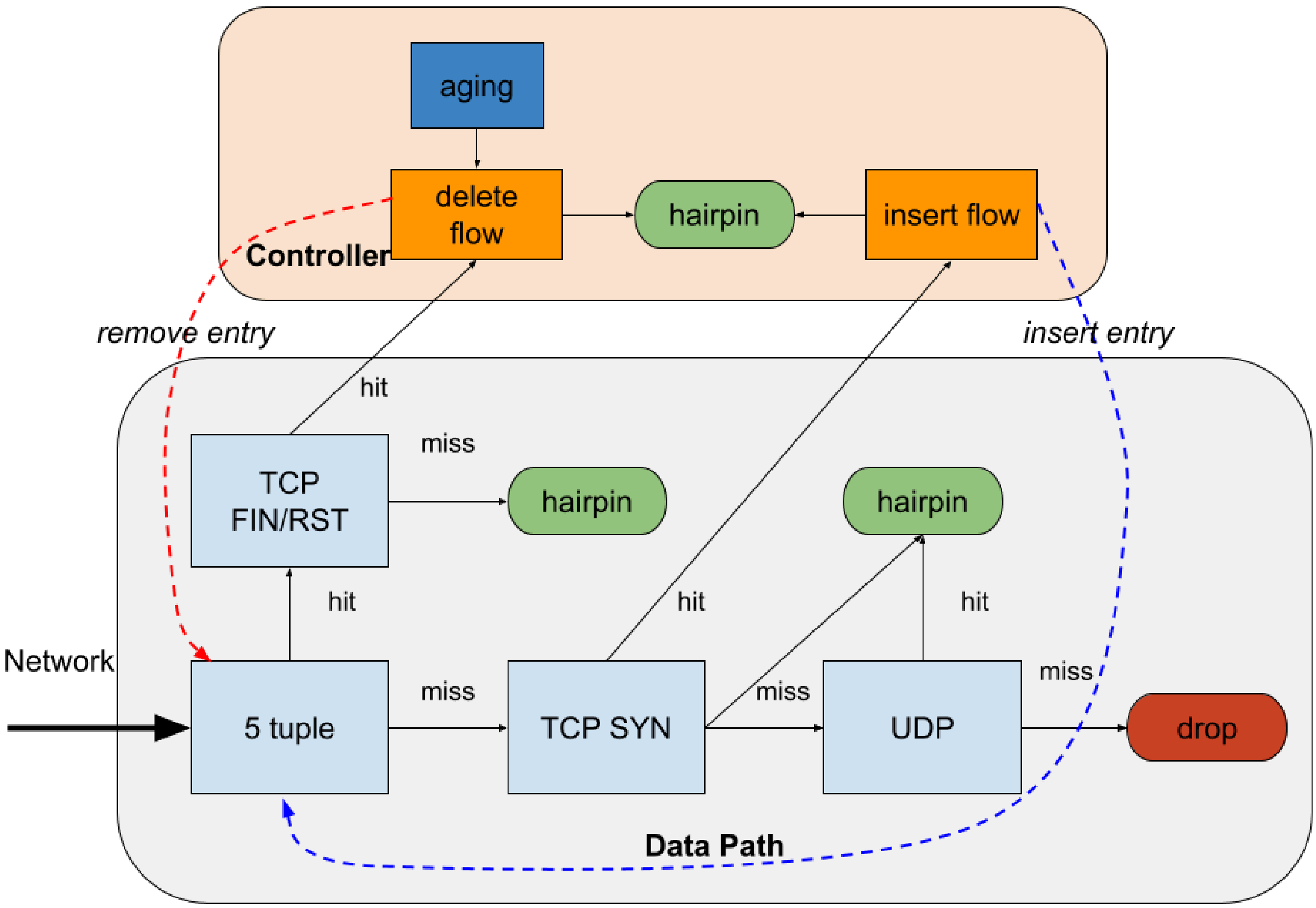
```
control gateway(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
```

```
) {
    overlay_encap() over;
    decap_flow() decap;

    /* user should add entries that correspond to the wire ports
  * A hit means this is an RX packet, miss means a TX packet
  */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        user_meta.was_decapped = 0;
        if (direction_table.apply().hit) {
            decap.apply(headers, std_meta, user_meta,
pkt_out_meta);
        }
        over.apply(headers, std_meta, user_meta, pkt_out_meta);
    }
}

NvDocaPipeline(
    packet_parser(),
    gateway()
) main;
```

See the full DPL example gateway.p4

Connection Tracking Example

Connection tracking is a stateful process that follows the a TCP/UDP session from establishment to termination. It consists of a 5 tuple match on <Source IP address, Destination IP address, IP protocol, L4 Source port, L4 Destination port>. The DPL datapath relies on a controller to insert entries for new 5 tuple flows that have not been seen yet.



Sample Code

Connection metadata is a struct that sent with a packet that is sent to the P4Runtime controller. It requires a special annotation with the label "packet_in". The user can define up to 32 bits of data in this data structure.

```
#include <doca_model.p4>
```

```
#include <doca_headers.p4>
#include <doca_externs.p4>
#include <doca_parser.p4>

@nv_controller_metadata("packet_in")
struct connection_meta_t {
    bit<2> type;
    bit<2> _reserved;
    bit<28> zone;
}
```

In this sample, the packet processing pipeline examines the state of the connection based on the TCP flags and set in the connection metadata an enum type describing the state of the flow.

```
/* Meta connection type */
enum bit<2> ct_type {
    PASS = 0,
    NEW = 1,
    RST = 2,
    FIN = 3
}
```

For readability, constants are used to assign P4 port IDs to more a more meaningful symbol name. The mapping of PFs and VFs to P4 port IDs is done separately in the Service Configuration.

```
/*
  VF port to SW entity that manages the flow insertion,
  deletion and expiry times
*/
const bit<32> WIRE_PORT = 32w0;
const bit<32> HAIRPIN_PORT = 32w1;
```

```
control conn_track(
    inout nv_headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout nv_empty_metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    connection_meta_t ctrl_meta;

    /* 5-tuple matching for L4 TCP/UDP flows */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES)
table_t5_counter;

    action set_connection_id_ct_table_t5(bit<28> zone) {
        table_t5_counter.count();
        ctrl_meta.zone = zone;
        ctrl_meta.type = ct_type.PASS;
    }

    action no_action_ct_table_t5() {
        table_t5_counter.count();
    }

    table ct_table_t5 {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
            headers.ipv4.protocol : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            set_connection_id_ct_table_t5;
            no_action_ct_table_t5;
        }
        direct_counter = table_t5_counter;
```

```
            // on hit table ct_table_known
            // on miss table ct_table_tcp_miss
            default_action = no_action_ct_table_t5;
            size = 1048576;
        }

        /*
  * Known connections handling.
  * Precondition - must be a TCP packet
  *  RST:    TYPE_RST
  *  FIN:    TYPE_FIN
  *  FINRST: TYPE_RST
  *
  * Match: tcp.flags
  * Actions: meta.type, next PIPE
  * MISS: next PIPE
  */
        NvDirectCounter(NvCounterType.PACKETS_AND_BYTES)
table_known_counter;

        action set_connection_type_ct_table_known(bit<2> type) {
            table_known_counter.count();
            ctrl_meta.type = type;
        }

        table ct_table_known {
            key = {
                headers.tcp.flags : exact;
            }
            actions = {
                set_connection_type_ct_table_known;
            }
            direct_counter = table_known_counter;
            default_action =
set_connection_type_ct_table_known(ct_type.PASS);
            const entries = {
```

```
            ( 0x1) :
set_connection_type_ct_table_known(ct_type.FIN);
            ( 0x4) :
set_connection_type_ct_table_known(ct_type.RST);
            (/*NV_TCP_PROTOCOL,*/ 0x5) :
set_connection_type_ct_table_known(ct_type.RST); /* RST+FIN */
        }
    }

    /*
 * Unknown TCP connections handling
 * Precondition - must be a TCP packet
 *  SYN: TYPE_NEW
 *
 * Match: tcp.flags
 * Actions: PIPE, meta.type=NEW
 * MISS: NEXT PIPE
 */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES)
table_tcp_miss_counter;

    action set_connection_type_ct_table_tcp(bit<2> type) {
        table_tcp_miss_counter.count();
        ctrl_meta.type = type;
    }

    action no_action_ct_table_tcp() {
        table_tcp_miss_counter.count();
    }

    table ct_table_tcp {
        key = {
            headers.tcp.flags : exact;
        }
        actions = {
            set_connection_type_ct_table_tcp;
            no_action_ct_table_tcp;
```

```
        }
        direct_counter = table_tcp_miss_counter;
        default_action = no_action_ct_table_tcp;
        const entries = {
            (0x02) :
set_connection_type_ct_table_tcp(ct_type.NEW);   /* SYN=1 ACK=0 */
        }
    }

    /*
  * Unknown UDP connections handling
  *
  * Match: UDP
  * Actions: PIPE, meta.type=NEW
  * MISS: DROP
  */
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES)
table_udp_miss_counter;

    action set_connection_type_ct_table_udp(bit<2> type) {
        table_udp_miss_counter.count();
        ctrl_meta.type = type;
    }

    action drop_ct_table_udp() {
        table_udp_miss_counter.count();
        nv_drop();
    }

    table ct_table_udp {
        key = {
            std_meta.l4_type : exact;
        }
        actions = {
            set_connection_type_ct_table_udp;
            drop_ct_table_udp();
        }
```

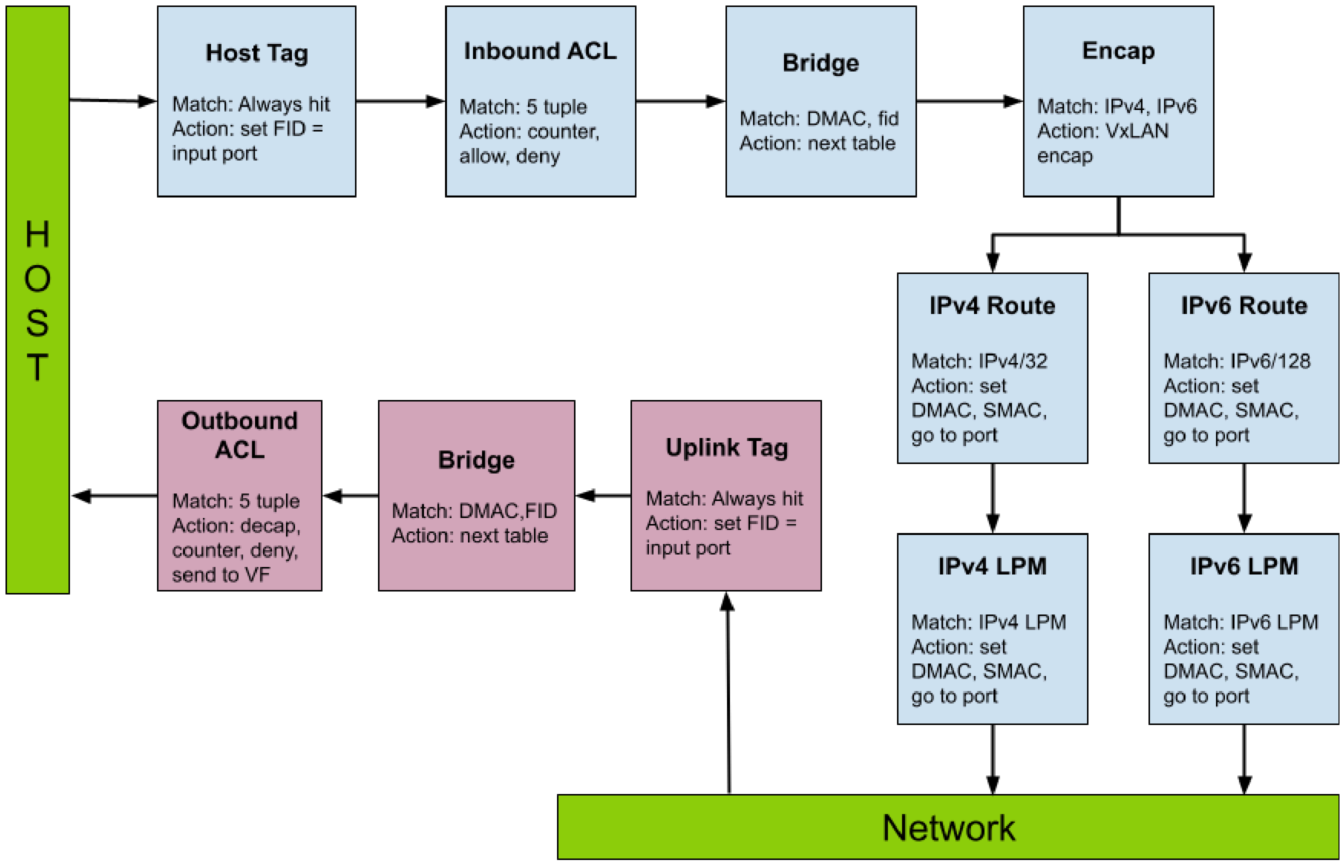
```
        direct_counter = table_udp_miss_counter;
        default_action = drop_ct_table_udp();
        const entries = {
            (L4_TYPE_UDP) :
set_connection_type_ct_table_udp(ct_type.NEW);   /* SYN=1 ACK=0 */
        }
    }

    /* user should add entries that correspond to the wire ports
  * A hit means this is an RX packet, miss means a TX packet
  */
    table direction_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        ctrl_meta.type = ct_type.PASS;
        ctrl_meta._reserved = 0;
        ctrl_meta.zone = 0;

        if (direction_table.apply().hit) {
            if (std_meta.is_l4_ok == 1w1) {
                if (ct_table_t5.apply().hit) {
                    if (ct_table_known.apply().hit) {
                        nv_send_to_controller(ctrl_meta);
                    }
                } else if (headers.tcp.isValid()) {
```

```
                    if (ct_table_tcp.apply().hit) {
                        nv_send_to_controller(ctrl_meta);
                    }
                } else {
                    if (ct_table_udp.apply().hit) {
                        nv_send_to_controller(ctrl_meta);
                    }
                }
            }
        }
        nv_send_to_port(HAIRPIN_PORT);
    }
}

// Instantiate the top-level DOCA Rx package
NvDocaPipeline(
    nv_fixed_parser(),
    conn_track()
) main;
```

See the full DPL example conn_track.p4

Host-based Networking Example

HBN is service that provides controller-less VPC networking solution for bare metal as a service (BMaaS), featuring tenant isolation with EVPN VXLAN and accelerated routing on the host. This example demonstrates how to build a DPL datapath that supports host-based networking (HBN).



Sample Code

This example uses indirect counters to monitor the number of packets that are allowed, denied and encapsulated/decapsulated. A user defined forwarding ID is used for determining how the bridge behaves.

```
enum bit<32> AdmissionCounter_t {
    DENY = 0,
    ALLOW = 1,
    VXLAN_V4_ENCAP = 2,
    VXLAN_V6_ENCAP = 3
}

enum bit<32> DecapFlowCounter_t {
    DENY = 0,
    ALLOW = 1,
}
```

```
/* The directionality is based on network to host
 * The user will configure the P4 port IDs in OVS
 */
const bit<32> WIRE_PORT = 32w0;

/* fid is 32 bits
*/
typedef bit<32> fid_t;

struct metadata_t {
    fid_t fid;
}
```

The default parser is used for this example:

```
struct headers_t {
    NV_FIXED_HEADERS
}

parser packet_parser(packet_in packet, out headers_t headers) {
    NV_FIXED_PARSER(packet, headers)
 }
```

Finally, the main control body invokes three separate sub-controls:

- `overlay_encap` – defines the logic and policies for applying VXLAN encapsulations along with ACLs
- `underlay_route` – applies a custom IPv4/IPv6 routing tables with exact and LPM matching. The next hop is set, TTL decremented, and the packet is routed.

```
/**
* This control performs the overlay policy including L2 encap with VxLAN
*/
control overlay_encap(
```

```
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) fid_counter;
    NvCounter(4, NvCounterType.PACKETS_AND_BYTES)
admission_counter;

    action set_fid(fid_t fid) {
        user_meta.fid = fid;
    }
    action deny() {
        admission_counter.count(AdmissionCounter_t.DENY);
        nv_drop();
    }
    action allow() {
        admission_counter.count(AdmissionCounter_t.ALLOW);
    }
    action vxlan_v4_encap(nv_mac_addr_t underlay_src_mac,
nv_mac_addr_t underlay_dst_mac,
                 nv_ipv4_addr_t underlay_sip, nv_ipv4_addr_t
underlay_dip, bit<24> vni) {
        nv_set_vxlan_v4_underlay(headers, false, underlay_dst_mac,
underlay_src_mac, 0, underlay_sip, underlay_dip, vni);

admission_counter.count(AdmissionCounter_t.VXLAN_V4_ENCAP);
    }
    action vxlan_v6_encap(nv_mac_addr_t underlay_src_mac,
nv_mac_addr_t underlay_dst_mac,
                 nv_ipv6_addr_t underlay_sip, nv_ipv6_addr_t
underlay_dip, bit<24> vni) {
        nv_set_vxlan_v6_underlay(headers, false, underlay_dst_mac,
underlay_src_mac, 0, underlay_sip, underlay_dip, vni);

admission_counter.count(AdmissionCounter_t.VXLAN_V6_ENCAP);
```

```
    }
    // A FID could be as simple as ingress port to FID mapping
    // e.g. ports 1 and 2 are in FID 1, ports 3 and 4 are in FID 2
    table fid_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            set_fid;
            NoAction;
        }
        size = FID_TABLE_SIZE;
        default_action = NoAction;
        direct_counter = fid_counter;
    }

    table admit_v4_table {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
            headers.ipv4.protocol : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            allow;
            deny;
            NoAction;
        }
        size = ADMIT_TABLE_SIZE;
        default_action = NoAction;
    }

    table admit_v6_table {
        key = {
            headers.ipv6.src_addr : exact;
```

```
            headers.ipv6.dst_addr : exact;
            headers.ipv6.next_header : exact;
            headers.tcp.src_port : exact;
            headers.tcp.dst_port : exact;
        }
        actions = {
            allow;
            deny;
            NoAction;
        }
        size = ADMIT_TABLE_SIZE;
        default_action = NoAction;
    }

    table bridge {
        key = {
            headers.ethernet.dst_addr : exact;
            user_meta.fid : exact;
        }
        actions = {
            NoAction;
        }
        size = BRIDGE_TABLE_SIZE;
        default_action = NoAction;
    }

    table encap_v4_table {
        key = {
            headers.ipv4.dst_addr : lpm;
        }
        actions = {
            vxlan_v4_encap;
        }
        size = ENCAP_TABLE_SIZE;
        default_action = vxlan_v4_encap(DEFAULT_DST_MAC,
DEFAULT_SRC_MAC,
```

```
                                        DEFAULT_SRC_IPV4,
DEFAULT_DST_IPV4, DEFAULT_VNI);
    }

    table encap_v6_table {
        key = {
            headers.ipv6.dst_addr : lpm;
        }

        actions = {
            vxlan_v6_encap;
        }
        size = ENCAP_TABLE_SIZE;
        default_action = vxlan_v6_encap(DEFAULT_DST_MAC,
DEFAULT_SRC_MAC,
                                         DEFAULT_SRC_IPV6,
DEFAULT_DST_IPV6, DEFAULT_VNI);

    }
    apply {
        user_meta.fid = 0;
        fid_table.apply();
        if (headers.ipv4.isValid()) {
            admit_v4_table.apply();
            if (bridge.apply().hit) {
                encap_v4_table.apply();
            }
        }
        else if (headers.ipv6.isValid()) {
            admit_v6_table.apply();
            if (bridge.apply().hit) {
                encap_v6_table.apply();
            }
        }
    }
}
```

```
/**
* This control performs the underlay policy with routing
*/
control underlay_route(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    action set_port_and_route(nv_logical_port_t port,
nv_mac_addr_t src_mac, nv_mac_addr_t dst_mac) {
        headers.ethernet.src_addr = src_mac;
        headers.ethernet.dst_addr = dst_mac;
        nv_dec_ip_ttl(headers, 1);
        nv_send_to_port(port);
    }

    table exact_v4_route {
        key = {
            headers.ipv4.dst_addr : exact;
        }
        actions = {
            set_port_and_route;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table lpm_v4_route {
        key = {
            headers.ipv4.dst_addr : lpm;
        }
        actions = {
            set_port_and_route;
            NoAction;
```

```
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table exact_v6_route {
        key = {
            headers.ipv6.dst_addr : exact;
        }
        actions = {
            set_port_and_route;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }
    table lpm_v6_route {
        key = {
            headers.ipv6.dst_addr : lpm;
        }
        actions = {
            set_port_and_route;
            NoAction;
        }
        size = ROUTE_TABLE_SIZE;
        default_action = NoAction;
    }

    apply {
        if (headers.ipv4.isValid()) {
            if (exact_v4_route.apply().miss) {
                lpm_v4_route.apply();
            }
        }
        else if (headers.ipv6.isValid()) {
            if (exact_v6_route.apply().miss) {
                lpm_v6_route.apply();
```

```
                }
            }
        }
}
```

- `decap_flow` – defines the policies for forwarding ID, L2 bridging, decapsulation of VXLAN and VXLAN-GPE tunnels, and forwarding to the VF

```
/**
* This control is for packets from wire to host (RX)
* and includes policy for L2 decap
*/
control decap_flow(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    NvDirectCounter(NvCounterType.PACKETS_AND_BYTES) fid_counter;
    NvCounter(2, NvCounterType.PACKETS_AND_BYTES) decap_counter;

    action deny() {
        decap_counter.count(DecapFlowCounter_t.DENY);
        nv_drop();
    }
    action set_fid(fid_t fid) {
        user_meta.fid = fid;
    }
    action decap_l2_and_send(nv_logical_port_t port) {
        decap_counter.count(DecapFlowCounter_t.ALLOW);
        nv_l2_decap(headers);
        nv_send_to_port(port);
    }
    action decap_ipv4_and_send(nv_logical_port_t port) {
        decap_counter.count(DecapFlowCounter_t.ALLOW);
```

```
        nv_l3_decap(headers, false, 0xffffffffff,
0x112233445566,NV_TYPE_IPV4,0);
        nv_send_to_port(port);
    }
    action decap_ipv6_and_send(nv_logical_port_t port) {
        decap_counter.count(DecapFlowCounter_t.ALLOW);
        nv_l3_decap(headers, false, 0xffffffffff,
0x112233445566,NV_TYPE_IPV6,0);
        nv_send_to_port(port);
    }

    table fid_table {
        key = {
            std_meta.ingress_port : exact;
        }
        actions = {
            set_fid;
            NoAction;
        }
        size = FID_TABLE_SIZE;
        default_action = NoAction;
        direct_counter = fid_counter;
    }

    table bridge {
        key = {
            headers.ethernet.dst_addr : exact;
            user_meta.fid : exact;
        }
        actions = {
            deny;
            NoAction;
        }
        size = BRIDGE_TABLE_SIZE;
        default_action = deny;
    }
```

```
    table decap_v4_table {
        key = {
            headers.ipv4.src_addr : exact;
            headers.ipv4.dst_addr : exact;
            headers.ipv4.protocol : exact;
            headers.udp.src_port : exact;
            headers.udp.dst_port : exact;
        }
        actions = {
            decap_l2_and_send;
            decap_ipv4_and_send;
            deny;
        }
        size = DECAP_TABLE_SIZE;
        default_action = deny;
    }

    table decap_v6_table {
        key = {
            headers.ipv6.src_addr : exact;
            headers.ipv6.dst_addr : exact;
            headers.ipv6.next_header : exact;
            headers.udp.src_port : exact;
            headers.udp.dst_port : exact;
        }
        actions = {
            decap_l2_and_send;
            decap_ipv6_and_send;
            deny;
        }
        size = DECAP_TABLE_SIZE;
        default_action = deny;
    }

    apply {
```

```
            user_meta.fid = (fid_t) 0;
            fid_table.apply();
            if (bridge.apply().hit) {
                if (headers.ipv4.isValid()) {
                    decap_v4_table.apply();
                }
                else if (headers.ipv6.isValid()) {
                    decap_v6_table.apply();
                }
            }
        }
}
```

The main control checks which direction the packet came from.

- if the packet is from network to host, it invokes the decap_flow control
- if the packet is from host to network, it invokes the overlay_encap and routing controls

```
control host_based_networking(
    inout headers_t headers,
    in nv_standard_metadata_t std_meta,
    inout metadata_t user_meta,
    inout nv_empty_metadata_t pkt_out_meta
) {
    overlay_encap() overlay;
    underlay_route() route;
    decap_flow() decap;

    /* user should add entries that correspond to the wire ports
  * A hit means this is an RX packet, miss means a TX packet
  */
    table direction_table {
        key = {
```

```
            std_meta.ingress_port : exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        const entries = {
            (WIRE_PORT) : NoAction();
        }
    }

    apply {
        if (direction_table.apply().miss) {
            overlay.apply(headers, std_meta, user_meta,
pkt_out_meta);
            route.apply(headers, std_meta, user_meta,
pkt_out_meta);
        }
        else {
            decap.apply(headers, std_meta, user_meta,
pkt_out_meta);
        }
    }
}

NvDocaPipeline(
    packet_parser(),
    host_based_networking()
) main;
```

See the full DPL example hbn.p4