P4 Language Support in DPL

Table of contents

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`	✔	✔	

Fixed Header Fields	Assignable	Copyable	Notes
`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` Alias with `headers.ipv6.diffserv`	☐	✔	Can be set through `nv_set_ip_dscp` extern

Fixed Header Fields	Assignable	Copyable	Notes
`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`	☐	✔	
`headers.inner_ipv4.ecn` Alias with `headers.inner_ipv6.ecn`	☐	☐	

Fixed Header Fields	Assignable	Copyable	Notes
`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`	✔	✔	
`headers.inner_ipv6.flow_label`	☐	☐	

Fixed Header Fields	Assignable	Copyable	Notes
`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`	☐	☐	
`headers.inner_icmp.identifier`	☐	☐	

Fixed Header Fields	Assignable	Copyable	Notes
`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`	☐	☐	
`headers.tcp.nonce_sum`	☐	☐	

Fixed Header Fields	Assignable	Copyable	Notes
`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`	☐	

Standard Metadata Fields	Copyable	Notes
`is_l2_ok`	☐	
`l2_type`	☐	
`last_l2_ether_type`	✔	Last extracted value of etherType within ethernet header or VLAN tags
`vlan_type`	☐	
`is_l3_ok`	☐	
`l3_type`	☐	
`is_ip_fragmented`	☐	
`is_ipv4_checksum_ok`	☐	
`is_l4_ok`	☐	
`l4_type`	☐	
`l4_type_ext`	☐	
`is_l4_checksum_ok`	☐	
`l4_src_port`	☐	
`l4_dst_port`	☐	
`encap_type`	☐	ROCE not currently supported
`ipsec_layer`	☐	
`ipsec_syndrome`	☐	Valid only after hardware encrypt/decrypt
`psp_syndrome`	☐	Valid only after hardware encrypt/decrypt
`is_inner_l2_ok`	☐	
`inner_l2_type`	☐	
`inner_last_l2_ether_type`	✔	Last extracted value of etherType within inner ethernet header or VLAN tags
`inner_vlan_type`	☐	

Standard Metadata Fields	Copyable	Notes
`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

Operator	Compile-time value	P4Runtime value	Runtime value	Spec section
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
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.)