P4 Portable NIC Architecture (PNA)

version 0.7

The P4.org Architecture Working Group

2022-12-22

Abstract

P4 is a domain-specific language for describing how packets are processed by a network data plane. A P4 program comprises an architecture, which describes the structure and capabilities of the pipeline, and a user program, which specifies the functionality of the programmable blocks within that pipeline. The Portable NIC Architecture (PNA) is an architecture that describes the structure and common capabilities of network interface controller (NIC) devices that process packets going between one or more interfaces and a host system.

Contents

1. Introduction 2
1.1. Packet processing . . . 3
1.2. Message processing . . . 4
1.3. PNA $P4_{16}$ architecture . . . 5
1.4. Guidelines for Portability . . . 5
2. Naming conventions 5
3. Packet paths 6
4. PNA Data types 7
4.1. PNA type definitions . . . 7
4.1.1. PNA type definition code excerpt . . . 7
4.1.2. PNA port types and values . . . 8
4.2. PNA supported metadata types . . . 9
4.3. Match kinds . . . 10
4.4. Data plane vs. control plane data representations . . . 10
5. Programmable blocks 10
6. Packet Path Details 10
6.1. Initial values of packets processed by main parser . . . 11
6.1.1. Initial packet contents for packets from ports . . . 11
6.1.2. Initial packet contents for packets looped back from host-to-network path . . . 11
6.2. Initial values of packets processed in network-to-host direction by main block . . . 11
6.2.1. Initial packet contents for normal packets . . . 11
6.2.2. Initial packet contents for recirculated packets . . . 11
6.3. Behavior of packets after main block is complete in network-to-host direction . . . 11
6.4. Initial values of packets processed in host-to-network direction by main block . . . 11

6.4.1. Initial packet contents for normal packets 11
6.4.2. Initial packet contents for recirculated packets 11
6.4.3. Initial packet contents for packets looped back after network-to-host main processing 11
6.5. Behavior of packets after main block is complete in host-to-network direction 11
6.6. Contents of packets sent out to ports 12
6.7. Functions for directing packets 12
6.7.1. Extern function `send_to_port` 12
6.8. Packet Mirroring 12
6.9. Packet recirculation 14
7. PNA Extern Objects 14
7.1. Restrictions on where externs may be used 14
7.2. Extern Objects for Inline Accelerators 15
8. PNA Table Properties 16
8.1. Tables with add-on-miss capability 17
8.2. Table entry idle timeout 19
9. Timestamps 20
10. Atomicity of control plane API operations 20
A. Appendix: Revision History 20
A.1. Changes made in version 0.7 20
B. Appendix: Open Issues 21
C. Appendix: Rationale for design 21
C.1. Why a common pipeline, instead of separate pipelines for each direction? 21
C.2. Is it inefficient to have the `MainParser` redo work? 21
D. Appendix: Packet path figures 22
D.1. From network, sent to host 22
D.2. From network, sent to host, with mirror copy to different host 22
D.3. From host, to network 22
D.4. From host, to network, with mirror copy to a different host 22
D.5. From host, to host 22
D.6. From port, to port 22
E. Appendix: Packet ordering 22

1. Introduction

Note that this document is still a working draft. Significant changes are expected to be made before version 1.0 of this specification is released.

The Portable NIC Architecture (PNA) is the P4 architecture that defines the structure and common capabilities for network interface controller (NIC) devices. PNA comprises two main components:

1. A programmable pipeline that can be used to realize a variety of different "packet paths" going between the various ports on the device (e.g., network interfaces or the host system it is attached to), and

2. A library of types (e.g., intrinsic and standard metadata) and $P4_{16}$ externs (e.g., counters, meters, and registers).

PNA is designed to model the common features of a broad class of NIC devices. By providing standard APIs and coding guidelines, the hope is to enable developers to write programs that are portable across multiple NIC devices that are conformant to the PNA[1].

The Portable NIC Architecture (PNA) Model has three programmable P4 blocks and several fixed-function blocks, as shown in Figure 1. The behavior of the programmable blocks is specified using P4. The network ports, packet queues, and (optional) inline accelerators are fixed-function blocks that can be configured by the control plane, but are not intended to be programmed using P4.

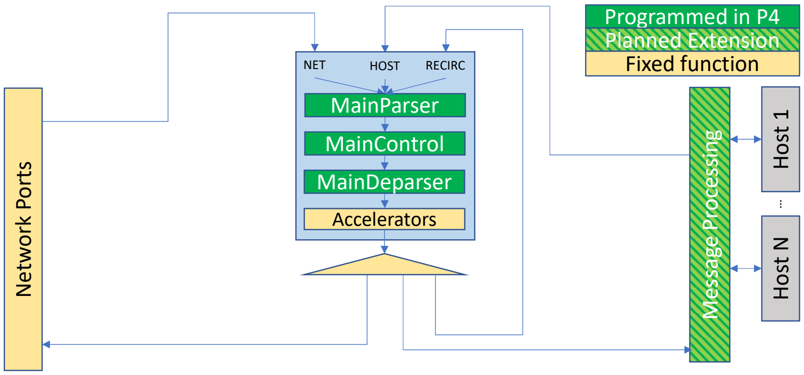

Figure 1. Programmable NIC Architecture Block Diagram

1.1. Packet processing

Packets arriving from a network port or from the hosts first go through a `MainParser`, which is responsible for extracting all relevant packet headers. Then the extracted headers and associated metadata are processed by `MainControl`. The code executed in the `MainControl` transforms headers, updates stateful elements like counters, meters, and registers, and optionally associates additional user-defined meta-

[1] Of course, given the tight hardware resource constraints on NIC devices, there is no promise that a given P4 program that compiles on one device will also compile on another device. However, it should at least be the case that those P4 programs that are able to compile on multiple NIC devices should process packets as described in this document.

data with the packet. The MainDeparser serializes the headers back into a packet that can be sent onwards.

After the MainDeparser, the packet is processed by one or more inline accelerators. The P4 program executed in the MainControl determines whether and how each inline accelerator processes the packet by executing methods of a corresponding extern object(s).

Upon completion of processing in the inline accelerators, a packet may either:

- Proceed to the message processing part of the NIC. If the packet had originally been received through the network, this is a packet being received by the host or a VM. If the packet had originally arrived to the packet processing block from the host, this is enables on-NIC processing of VM-to-VM or host-to-host packets (i.e., on a system with multiple hosts).

- Head towards the network ports. If the packet had originally arrived to the packet processing block from the host, this is a packet transmission by the host or a VM. If the packet had originally been received through the network, this enables on-NIC processing of port-to-port packets without ever traversing the host system.

- Go back into the packet processing block to be processed again (a.k.a. recirculation).

The choice of which network port to go to, or whether to loop back, or whether to proceed to the hosts (and which one) are all controlled from the P4 code running in the MainControl block, via extern functions defined by this PNA specification.

The same MainParser, MainControl, and MainDeparser that process packets from the network are also used to process packets from the host. PNA was designed this way for two reasons:

- It is expected that in many cases, the packet processing in both directions will have many similarities between them. Writing common P4 code for both eliminates code duplication that would occur if the code for each direction was written separately.
- Having a single MainControl in the P4 language enables tables and externs such as counters and registers to be instantiated once, and shared by packets being processed in both directions. The hardware of many NICs supports this design, without having to instantiate a physically separate table for each direction. Especially for large tables used by packet processing in both directions, this approach can significantly reduce the memory required. It is also critical for some stateful features (e.g. those using the table add-on-miss capability defined later in this specification) to access the same table in memory when processing packets in either direction.

Figure 1 shows multiple hosts. Some NICs support PCI Express connections to multiple host CPU complexes. It is also common for NICs to have an array of one or more CPU cores inside of the NIC device itself, and these can be the target for packets received from the network, and/or the source of packets sent to the network, just as the other hosts can be. For the purposes of the PNA, such CPU cores are considered as another host.

1.2. Message processing

The focus in the current version of this specification is on the three P4-programmable blocks mentioned above. The details of how one can use P4 to program the message processing portion of a NIC is left as a future extension of this specification. While there are options for exactly what packet processing functions can be performed in the four primary blocks described above, versus the message processing block, the division is expected to be:

- The primary programmable blocks deal solely with individual network packets, which are at most one network maximum transmission unit (MTU) in size.
- The message processing block is responsible for converting between large messages in host memory and network size packets on the network, and for dealing with one or more host operating systems, drivers, and/or message descriptor formats in host memory.

For example, in its role of converting between large messages and network packets in the host-to-network direction, message processing would implement features like large send offload (LSO), TCP segmentation offload (TSO), and Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE). In the network-to-host direction it would assist in such features as large receive offload (LRO) and RoCE.

In its role of handling different kinds of operating systems, drivers, and message descriptor formats, the message processing block may deal with one or more of the following standards: - VirtIO - SR-IOV

Another potential criteria for dividing packet processing functionality between message processing and the rest of the NIC is for division of control plane responsibilities. For example, in some network deployments the NIC message processing block configuration is tightly coupled with the host operating system, whereas the `MainControl` is controlled by network-focused control plane software.

1.3. PNA P4$_{16}$ architecture

A programmer targeting the PNA is required to provide P4 definitions for each of the programmable blocks in the pipeline (see section 5). The programmable block inputs and outputs are parameterized on the types of user defined headers and metadata. The top-level PNA program instantiates a package named `main` with the programmable blocks passed as arguments (see Section TBD for an example). Note that the `main` package is not to be confused with the `MainControl`.

This document contains excerpts of several P4$_{16}$ programs that use the `pna.p4` include file and demonstrate features of PNA. Source code for the complete programs can be found in the official repository containing the PNA specification[2].

1.4. Guidelines for Portability

A P4 programmer wishing to maximize the portability of their program should follow several general guidelines:

- Do not use undefined values in a way that affects the resulting output packet(s), or for side effects such as updating `Counter`, `Meter` or `Register` instances.

- Use as few resources as possible, e.g. table search key bits, array sizes, quantity of metadata associated with packets, etc.

2. Naming conventions

In this document we use the following naming conventions:

- Types are named using CamelCase followed by `_t`. For example, `PortId_t`.

[2] https://github.com/p4lang/pna in directory `examples`. Direct link: https://github.com/p4lang/pna/tree/main/examples

Description	Processed next by	Resulting packet(s)
packet from network port	main, with direction `NET_TO_HOST`	Zero or more mirrored packets, plus at most one of: a net-to-host recirculated packet, or one to-host packet.
packet from net-to-host recirculate		
packet from port loopback		
packet from message processing	main, with direction `HOST_TO_NET`	Zero or more mirrored packets, plus at most one of: a host-to-net recirculated packet, or one to-net packet.
packet from host-to-net recirculate		
packet from host loopback		

Table 1. Result of packet processed one time by main.

- Control types and extern object types are named using CamelCase. For example `MainParser`.
- Struct types are named using lower case words separated by _ followed by `_t`. For example `pna_input_metadata_t`.
- Actions, extern methods, extern functions, headers, structs, and instances of controls and externs start with lower case and words are separated using _. For example `send_to_port`.
- Enum members, const definitions, and #define constants are all caps, with words separated by _. For example `PNA_PORT_CPU`.

Architecture specific metadata (e.g. structs) are prefixed by `pna_`.

3. Packet paths

Figure 2 shows all possible paths for packets that must be supported by a PNA implementation. An implementation is allowed to support paths for packets that are not described here.
TBD: Create another figure with the updated architecture diagram and names for the paths.

Figure 2. Packet Paths in PNA

Table 1 shows what can happen to a packet as a result of a single time being processed through the four programmable blocks of the packet processing part of PNA, referred to here as "main".

Note that each mirrored packet that results from `mirror_packet` operations will have its own next place that it will go to be processed, independent of the original packet, and independent of any other mirror copies made of the same original packet.

4. PNA Data types

4.1. PNA type definitions

Each PNA implementation will have specific bit widths in the data plane for the types shown in the code excerpt of Section 4.1.1. These widths are defined in the target specific `pna.p4` include file. They are expected to differ from one PNA implementation to another[3].

For each of these types, the P4 Runtime API[4] may use bit widths independent of the targets. These widths are defined by the P4 Runtime API specification, and they are expected to be at least as large as the corresponding `InHeader_t` type below, such that they hold a value for any target. All PNA implementations must use data plane sizes for these types no wider than the corresponding `InHeader_t`-defined types.

4.1.1. PNA type definition code excerpt

```
/* These are defined using `typedef`, not `type`, so they are truly
 * just different names for the type bit<W> for the particular width W
 * shown.  Unlike the `type` definitions below, values declared with
 * the `typedef` type names can be freely mingled in expressions, just
 * as any value declared with type bit<W> can.  Values declared with
 * one of the `type` names below _cannot_ be so freely mingled, unless
 * you first cast them to the corresponding `typedef` type.  While
 * that may be inconvenient when you need to do arithmetic on such
 * values, it is the price to pay for having all occurrences of values
 * of the `type` types marked as such in the automatically generated
 * control plane API.
 *
 * Note that the width of typedef <name>Uint_t will always be the same
 * as the width of type <name>_t. */
typedef bit<unspecified> PortIdUint_t;
typedef bit<unspecified> InterfaceIdUint_t;
typedef bit<unspecified> MulticastGroupUint_t;
typedef bit<unspecified> MirrorSessionIdUint_t;
typedef bit<unspecified> MirrorSlotIdUint_t;
typedef bit<unspecified> ClassOfServiceUint_t;
typedef bit<unspecified> PacketLengthUint_t;
typedef bit<unspecified> MulticastInstanceUint_t;
typedef bit<unspecified> TimestampUint_t;
typedef bit<unspecified> FlowIdUint_t;
typedef bit<unspecified> ExpireTimeProfileIdUint_t;

typedef bit<unspecified> SecurityAssocIdUint_t;
```

[3] It is expected that `pna.p4` include files for different targets will be nearly identical to each other. Besides the possibility of differing bit widths for these PNA types, the only expected differences between `pna.p4` files for different targets would be annotations on externs, etc. that the P4 compiler for that target needs to do its job.

[4] The P4Runtime Specification can be found here: https://p4.org/specs

```
@p4runtime_translation("p4.org/pna/v1/PortId_t", 32)
type PortIdUint_t         PortId_t;
@p4runtime_translation("p4.org/pna/v1/InterfaceId_t", 32)
type InterfaceIdUint_t     InterfaceId_t;
@p4runtime_translation("p4.org/pna/v1/MulticastGroup_t", 32)
type MulticastGroupUint_t MulticastGroup_t;
@p4runtime_translation("p4.org/pna/v1/MirrorSessionId_t", 16)
type MirrorSessionIdUint_t MirrorSessionId_t;
@p4runtime_translation("p4.org/pna/v1/MirrorSlotId_t", 8)
type MirrorSlotIdUint_t MirrorSlotId_t;
@p4runtime_translation("p4.org/pna/v1/ClassOfService_t", 8)
type ClassOfServiceUint_t ClassOfService_t;
@p4runtime_translation("p4.org/pna/v1/PacketLength_t", 16)
type PacketLengthUint_t   PacketLength_t;
@p4runtime_translation("p4.org/pna/v1/MulticastInstance_t", 16)
type MulticastInstanceUint_t MulticastInstance_t;
@p4runtime_translation("p4.org/pna/v1/Timestamp_t", 64)
type TimestampUint_t      Timestamp_t;
@p4runtime_translation("p4.org/pna/v1/FlowId_t", 32)
type FlowIdUint_t      FlowId_t;
@p4runtime_translation("p4.org/pna/v1/ExpireTimeProfileId_t", 8)
type ExpireTimeProfileIdUint_t      ExpireTimeProfileId_t;

@p4runtime_translation("p4.org/pna/v1/SecurityAssocId_t", 64)
type SecurityAssocIdUint_t      SecurityAssocId_t;

typedef error   ParserError_t;

const InterfaceId_t PNA_PORT_CPU = (PortId_t) unspecified;

const MirrorSessionId_t PNA_MIRROR_SESSION_TO_CPU = (MirrorSessiontId_t) unspecified;
```

4.1.2. PNA port types and values

There are two types defined by PNA for holding different kinds of ports: `PortId_t` and `InterfaceId_t`.

The type `PortId_t` must be large enough in the data plane to hold one of these values:

- a data plane id for one network port
- a data plane id for one vport

As one example, a PNA target with four Ethernet network ports could choose to use the values 0 through 3 to identify the network ports, and the values 4 through 1023 to identify vports.

PNA makes no requirement that the numeric values identifying network ports must be consecutive, nor for vports. PNA only requires that for every possible numeric value `x` with type `PortId_t`, exactly one of these statements is true:

- x is the data plane id of one network port, but not any vport
- x is the data plane id of one vport, but not any network port
- x is the data plane id of no port, neither a network port nor a vport

4.2. PNA supported metadata types

```
struct pna_main_parser_input_metadata_t {
    // common fields initialized for all packets that are input to main
    // parser, regardless of direction.
    bool                     recirculated;
    // If this packet has FROM_NET source, input_port contains
    // the id of the network port on which the packet arrived.
    // If this packet has FROM_HOST source, input_port contains
    // the id of the vport from which the packet came
    PortId_t                 input_port;   // network/host port id
}

// is_host_port(p) returns true if p is a host port, otherwise false.
extern bool is_host_port (in PortId_t p);

// is_net_port(p) returns true if p is a network port, otherwise
// false.
extern bool is_net_port (in PortId_t p);

struct pna_main_input_metadata_t {
    // common fields initialized for all packets that are input to main
    // parser, regardless of direction.
    bool                     recirculated;
    Timestamp_t              timestamp;
    ParserError_t            parser_error;
    ClassOfService_t         class_of_service;
    // See comments for field input_port in struct
    // pna_main_parser_input_metadata_t
    PortId_t                 input_port;
}



struct pna_main_output_metadata_t {
  // common fields used by the architecture to decide what to do with
  // the packet next, after the main parser, control, and deparser
  // have finished executing one pass, regardless of the direction.
  ClassOfService_t         class_of_service; // 0
}
```

4.3. Match kinds

PNA supports the `match_kinds` specified in section 4.3 of the PSA specification.

4.4. Data plane vs. control plane data representations

5. Programmable blocks

The following declarations provide a template for the programmable blocks in the PNA. The P4 programmer is responsible for implementing controls that match these interfaces and instantiate them in a package definition.

It uses the same user-defined metadata type `IM` and header type `IH` for all ingress parsers and control blocks. The egress parser and control blocks can use the same types for those things, or different types, as the P4 program author wishes.

```
parser MainParserT<MH, MM>(
    packet_in pkt,
    out   MH main_hdr,
    inout MM main_user_meta,
    in    pna_main_parser_input_metadata_t istd);

control MainControlT<MH, MM>(
    inout MH main_hdr,
    inout MM main_user_meta,
    in    pna_main_input_metadata_t  istd,
    inout pna_main_output_metadata_t ostd);

control MainDeparserT<MH, MM>(
    packet_out pkt,
    in    MH main_hdr,
    in    MM main_user_meta,
    in    pna_main_output_metadata_t ostd);

package PNA_NIC<MH, MM>(
    MainParserT<MH, MM> main_parser,
    MainControlT<MH, MM> main_control,
    MainDeparserT<MH, MM> main_deparser);
```

6. Packet Path Details

Refer to section 3 for the packet paths provided by PNA.
TBD: Need to decide where multicast replication can occur, and in what conditions.
TBD: Need to decide where packet mirroring occurs, and in what conditions, and how the mirrored packets differ from the originals.
TBD: Rewrite this section once the overall architecture is approved

6.1. Initial values of packets processed by main parser

6.1.1. Initial packet contents for packets from ports

Packet is as received from Ethernet port.

User-defined metadata is empty?

6.1.2. Initial packet contents for packets looped back from host-to-network path

Packet is as came out of host-to-net received from Ethernet port.

There can be user-defined metadata included with these packets.

6.2. Initial values of packets processed in network-to-host direction by main block

6.2.1. Initial packet contents for normal packets

The packet should be ...

The user-defined metadata should be ...

The standard metadata contents should be specified in detail here.

6.2.2. Initial packet contents for recirculated packets

Give any differences between this case and previous section.

6.3. Behavior of packets after main block is complete in network-to-host direction

Cases: drop, recirculate, loopback to host-to-net direction, to message processing. Describe the conditions in which each occurs.

6.4. Initial values of packets processed in host-to-network direction by main block

6.4.1. Initial packet contents for normal packets

This is for packets from the message processing block.

6.4.2. Initial packet contents for recirculated packets

Give any differences between this case and previous section.

6.4.3. Initial packet contents for packets looped back after network-to-host main processing

6.5. Behavior of packets after main block is complete in host-to-network direction

Cases: drop, recirculate, to queues. Describe the conditions in which each occurs.

6.6. Contents of packets sent out to ports

6.7. Functions for directing packets

6.7.1. Extern function `send_to_port`

```
extern void send_to_port(PortId_t dest_port);
```

The extern function `send_to_port` is used to direct a packet to a specified network port, or to a vport. Invoking `send_to_port(x)` is supported only within the main control. There are two cases to consider, detailed below.

- `x` is a network port id.

Calling `send_to_port(x)` modifies hidden state for this packet, so that the packet will be transmitted out of the network port with id `x`.

- `x` is a vport id.

Calling `send_to_port(x)` modifies hidden state for this packet, so that the packet will be sent to the vport with id `x` in the host, without being looped back.

6.8. Packet Mirroring

```
extern void mirror_packet(MirrorSlotId_t mirror_slot_id,
                          MirrorSessionId_t mirror_session_id);
```

The extern function `mirror_packet` is used to cause a mirror copy of the packet currently being processed to be created. Invoking `mirror_packet(x)` is supported only within the main control.

PNA enables multiple mirror copies of a packet to be created during a single execution of `MainControl`, by calling `mirror_packet` with different mirror slot id values. PNA targets should support `mirror_slot_id` values in the range 0 through 3, at least, but are allowed to support a larger range.

When `MainControl` begins execution, all mirror slots are initialized so that they do not create a copy of the packet.

After calling `mirror_packet(slot_id, session_id)`, then when the main control finishes execution, the target will make a best effort to create a copy of the packet that will be processed according to the parameters configured by the control plane for the mirror session numbered `session_id`, for mirror slot `slot_id`. Note that this is best effort – if the target device is already near its upper limit of its ability to create mirror copies, then some later mirror copies may not be made, even though the P4 program requested them.

Each of the mirror slots is independent of each other. For example, calling `mirror_packet(1, session_id)` has no effect on mirror slots 0, 2, or 3.

Mirror session id 0 is reserved by the architecture, and must not be used by a P4 developer.

If multiple calls are made to `mirror_packet()` for the same mirror slot id in the same execution of the main control, only the last `session_id` value is used to create a copy of the packet. That is, every call to `mirror_packet(slot_id, session_id)` overwrites the effects of any earlier to `mirror_packet()` with the same `slot_id`.

The effects of mirror_packet() calls are independent of calls to drop_packet() and send_to_port(). Regardless of which of those things is done to the original packet, up to one mirror packet per mirror slot can be created.

The control plane code can configure the following properties of each mirror session, independently of other mirror sessions:

- packet_contents

If PRE_MODIFY, then the mirrored packet's contents will be the same as the original packet as it was when the packet began the execution of the main control that invoked the mirror_packet() function.

If POST_MODIFY, then the mirrored packet's contents will be the same as the original packet that is being mirrored, after any modifications made during the execution of the main control that invoked the mirror_packet() function.

- truncate

true to limit the length of the mirrored packet to the truncate_length. false to cause the mirrored packet not to be truncated, in which case the truncate_length property is ignored for this mirror session.

- truncate_length

In units of bytes. Targets may limit the choices here, e.g. to a multiple of 32 bytes, or perhaps even a subset of those choices.

- sampling_method

One of the values: RANDOM_SAMPLING, HASH_SAMPLING.

If RANDOM_SAMPLING, then a mirror copy requested for this mirror session will only be created with a configured probability given by the sample_probability property.

If HASH_SAMPLING, then a target-specific hash function will be calculated over the packet's header fields resulting in a hash output value H. A mirror copy will be created if (H & sample_hash_mask) == sample_hash_value.

- meter_parameters

If the conditions specified by the sampling_method and other sampling properties are passed, then a P4 meter dedicated for use by this mirror session will be updated. If it returns a GREEN result, then the mirror copy will be created (still with best effort, if the target device's implementation is still oversubscribed with requests to create mirror copies).

If the meter update returns any result other than GREEN, then no mirror copy will be created.

- destination_port

A network port id, or a vport id.

If destination_port is a network port id, that network port is the destination of mirrored copy packets created by this session. If the mirror_packet() call for this session was invoked in the NET_TO_HOST direction, mirror copy packets created will loop back in the host side of the target, and later come back for processing in the main block in the HOST_TO_NET direction, already destined for the network port destination_port. That port can be overwritten by calls to forwarding functions.

If destination_port is a vport id, that vport is the destination of mirrored copy packets created by this session. If the mirror_packet() call for this session was invoked in the HOST_TO_NET direction, mirror copy

Extern type	Where it may be instantiated and called from
`ActionProfile`	MainControl
`ActionSelector`	MainControl
`Checksum`	MainParser, MainDeparser
`Counter`	MainControl
`Digest`	MainDeparser
`DirectCounter`	MainControl
`DirectMeter`	MainControl
`Hash`	MainControl
`InternetChecksum`	MainParser, MainDeparser
`Meter`	MainControl
`Random`	MainControl
`Register`	MainControl

Table 2. Summary of controls that can instantiate and invoke externs.

packets created will loop back in the network port side of the NIC, and later come back for processing in the main block in the `NET_TO_HOST` direction, already destined for the vport `destination_port`. That vport can be overwritten by calls to forwarding functions.

TBD: When a mirror copied packet comes back to the main control, it will have some metadata indicating it is mirror copy. We should define a way in PNA to recognize such mirror copies, e.g. some new extern function call returning true if the packet was created by a `mirror_packet` operation.

6.9. Packet recirculation

7. PNA Extern Objects

7.1. Restrictions on where externs may be used

All instantiations in a P4$_{16}$ program occur at compile time, and can be arranged in a tree structure we will call the instantiation tree. The root of the tree `T` represents the top level of the program. Its child is the node for the package `PNA_NIC` described in Section 5, and any externs instantiated at the top level of the program. The children of the `PNA_NIC` node are the packages and externs passed as parameters to the `PNA_NIC` instantiation. See Figure 3 for a drawing of the smallest instantiation tree possible for a P4 program written for PNA.

Figure 3. Minimal PNA instantiation tree

If any of those parsers or controls instantiate other parsers, controls, and/or externs, the instantiation tree contains child nodes for them, continuing until the instantiation tree is complete.

For every instance whose node is a descendant of the `Ingress` node in this tree, call it an `Ingress` instance. Similarly for the other ingress and egress parsers and controls. All other instances are top level instances.

A PNA implementation is allowed to reject programs that instantiate externs, or attempt to call their methods, from anywhere other than the places mentioned in Table 2.

For example, Counter being restricted to “Pre, Main” means that every Counter instance must be instantiated within the MainControl block, or be a descendant of one of those nodes in the instantiation tree. If a Counter instance is instantiated in Main, for example, then it cannot be referenced, and thus its methods cannot be called, from any block except MainControl or one of its descendants in the tree.

PNA implementations need not support instantiating these externs at the top level. PNA implementations are allowed to accept programs that use these externs in other places, but they need not. Thus P4 programmers wishing to maximize the portability of their programs should restrict their use of these externs to the places indicated in the table.

All methods for type packet_out, e.g., emit, are restricted to be within deparser control blocks in PNA, because those are the only places where an instance of type packet_out is visible. Similarly all methods for type packet_in, e.g. extract and advance, are restricted to be within parsers in PNA programs. P4$_{16}$ restricts all verify method calls to be within parsers for all P4$_{16}$ programs, regardless of whether they are for the PNA.

See the PSA specification for definitions of all of these externs. There is work under way as of this writing that may result in these extern definitions being moved from the PSA specification into a separate standard library of P4 extern definitions, and if this is done, both the PSA and PNA specifications will reference that.

7.2. Extern Objects for Inline Accelerators

A variety of inline accelerators can be present on a PNA target. These accelerators perform specific functions on a packet. These functions are typically implemented in hardware. These accelerators perform specific functions on a packet after the deparser has finished executing.

These hardware functions are represented as extern objects in a P4 program. An extern object representing a specific accelerator E.g. AES-GCM crypto accelerator, can be instantiated in a P4 program. The methods defined by the extern object are used to send and receive information to/from the inline accelerator. Since the accelerators are present after the deparser, the information sent to the accelerator takes effect only when packet reaches the accelerator. Similarly any information received from accelerator is for the previous function performed on the packet.

This section provides one example definition of a crypto acceleration engine. Other extern objects can be defined in future based on the functionality provided by the hardware accelerators.

```
extern crypto_accelerator {
    /// constructor
    /// Some methods provided in this object may be specific to an algorithm used.
    /// Compiler may be able to check and warn/error when incorrect methods are used
    crypto_accelerator(crypto_algorithm_e algo);


    // security association index for this security session
    // Some implementations do not need it.. in that case this method should result in no-op
    void set_sa_index<T>(in T sa_index);

    // Set the initialization data based on protocol used. E.g. salt, random number/ counter for ipsec
    void set_iv<T>(in T iv);
    void set_key<T,S>(in T key, in S key_size);   // 128, 192, 256

    // The format of the auth data is not specified/mandated by this object definition
    // If it is part of the packet, it can be specified using offset/len mothods below
    void set_auth_data_offset<T>(in T offset);
    void set_auth_data_len<T>(in T len);

    // Alternatively: Following API can be used to consturct the auth_data and
    // provide it to the engine.
    void add_auth_data<H>(in H auth_data);

    // Auth trailer aka ICV is added by the engine after doing encryption operation
    // Specify icv location -
 when a wire protocol wants to add ICV in a specific location (e.g. AH)
    // The following apis can be used to specify the location of ICV in the packet
    // A special offset indicates ICV is after the payload
    void set_icv_offset<T>(in T offset);
    void set_icv_len<L>(in L len);

    // setup payload to be encrypted/decrypted
    void set_payload_offset<T>(in T offset);
    void set_payload_len<T>(in T len);

    // crypto accelerator runs at the end of the pipeline (after deparser), the following
    // methods will enable the accelerator to run encrypt/decrypt operations
    // enable_auth flag enables authentication check for decrypt. For encrypt operation,
    // auth data computed, is added to specified icv_offset/len
    void enable_encrypt<T>(in T enable_auth);
    void enable_decrypt<T>(in T enable_auth);

    // disable crypto engine. Between enable and disable methods,
    // whichever method is called last overrides the previous calls
    void disable();

    // get results of the previous operation
    crypto_results_e get_results();
}
```

8. PNA Table Properties

Table 3 lists all P4 table properties defined by PNA that are not included in the base P4$_{16}$ language specification.

A PNA implementation need not support both of a `pna_implementation` and `pna_direct_counter` property on the same table.

Similarly, a PNA implementation need not support both of a `pna_implementation` and `pna_direct_meter`

Property name	Type	See also
add_on_miss	boolean	Section 8.1
pna_direct_counter	one DirectCounter instance name	
pna_direct_meter	one DirectMeter instance name	
pna_implementation	instance name of one ActionProfile or ActionSelector	
pna_empty_group_action	action	
pna_idle_timeout	PNA_IdleTimeout_t	Section 8.2

Table 3. Summary of PNA table properties.

property on the same table.

A PNA implementation must implement tables that have both a pna_direct_counter and pna_direct_meter property.

A PNA implementation need not support both pna_implementation and pna_idle_timeout properties on the same table.

8.1. Tables with add-on-miss capability

PNA defines the add_on_miss table property. If the value of this property is true for a table t, the P4 developer is allowed to define a default action for t that calls the add_entry extern function.

When t.apply() is invoked, t's lookup key is constructed, and the entries of the table are searched. If there is no match, i.e. the lookup results in a miss, t's default action is executed. So far, this is all standard behavior as defined in the $P4_{16}$ language specification.

If t's default action makes a call to add_entry, it causes a new entry to be added to the table with the same key that was just looked up and resulted in a miss, and the action name and action parameters specified by the parameters of the call to the add_entry extern function. Thus, future packets that invoke t.apply() with the same lookup key will get a match and invoke the specified action (until and unless this new table entry is removed). The new table entry will be matchable when the next packet is processed that invoked t.apply().

Some PNA implementations may allow the control plane software to add, modify, and delete entries of such a table, but any entries added via the add_entry function do not require the control plane software to be involved in any way. Other PNA implementations may choose not to support control plane modification of the entries of an add-on-miss table.

It is expected that PNA implementations will be able to sustain add_entry calls at a large fraction of their line rate, but it need not be at the same packet rate supported for processing packets that do not call add_entry.

```
// The bit width of this type is allowed to be different for different
// target devices.  It must be at least a 1-bit wide type.

typedef bit<1> AddEntryErrorStatus_t;

const AddEntryErrorStatus_t ADD_ENTRY_SUCCESS = 0;
const AddEntryErrorStatus_t ADD_ENTRY_NOT_DONE = 1;
```

```
// Targets may define target-specific non-0 constants of type
// AddEntryErrorStatus_t if they wish.

// The add_entry() extern function causes an entry, i.e. a key and its
// corresponding action and action parameter values, to be added to a
// table from the data plane, i.e. without the control plane having to
// take any action at all to cause the table entry to be added.
//
// The key of the new entry added will always be the same as the key
// that was just looked up in the table, and experienced a miss.
//
// `action_name` is the name of an action that must satisfy these
// restrictions:
// + It must be an action that is in the list specified as the
//   `actions` property of the table.
// + It must be possible for this action to be the action of an entry
//   added to the table, e.g. it is an error if the action has the
//   annotation `@defaultonly`.
// + The action to be added must not itself contain a call to
//   add_entry(), or anything else that is not supported in a table's
//   "hit action".
//
// Type T must be a struct type whose field names have the same name
// as the parameters of the action being added, in the same order, and
// have the same type as the corresponding action parameters.
//
// `action_params` will become the action parameters of the new entry
// to be added.
//
// `expire_time_profile_id` is the initial expire time profile id of
// the entry added.
//
// The return value will be ADD_ENTRY_SUCCESS if the entry was
// successfully added, otherwise it will be some other value not equal
// to ADD_ENTRY_SUCCESS.  Targets are allowed to define only one
// failure return value, or several if they wish to provide more
// detail on the reason for the failure to add the entry.
//
// It is NOT defined by PNA, and need not be supported by PNA
// implementations, to call add_entry() within an action that is added
// as an entry of a table, i.e. as a "hit action".  It is only defined
// if called within an action that is the default_action, i.e. a "miss
// action" of a table.
//
// For tables with `add_on_miss = true`, some PNA implementations
// might only support `default_action` with the `const` qualifier.
// However, if a PNA implementation can support run-time modifiable
// default actions for such a table, some of which call add_entry()
// and some of which do not, the behavior of such an implementation is
// defined by PNA, and this may be a useful feature.

extern AddEntryErrorStatus_t add_entry<T>(
    string action_name,
    in T action_params,
    in ExpireTimeProfileId_t expire_time_profile_id);
```

It is expected that many PNA implementations will restrict `add_entry()` to be called with the following restrictions:

- Only from within an action
- Only if the action is a default action of a table with property `add_on_miss` equal to `true`.
- Only for a table with all key fields having match_kind `exact`.
- Only with an action name that is one of the hit actions of that same table. This action has parameters that are all directionless.
- The type `T` is a struct containing one member for each directionless parameter of the hit action to be added. The member names must match the hit action parameter names, and their types must be the same as the corresponding hit action parameters.

The new entry will have the same key field values that were searched for in the table when the miss occurred, which caused the table's default action to be executed. The action will be the one named by the string that is passed as the parameter `action_name`.

If the attempt to add a table entry succeeds, the return value is `ADD_ENTRY_SUCCESS`, otherwise it will be some other value. PNA implementations are free to define additional failure reasons other than `ADD_ENTRY_NOT_DONE`, but it is perfectly acceptable for a PNA implementation to only support those two possible return values.

8.2. Table entry idle timeout

PNA defines the table property `pna_idle_timeout` to enable specifying whether a table should maintain an idle time for each of its entries, and if so, what the data plane should do when a table entry has not been matched for a length of time at least its configured idle time.

The value assigned to `pna_idle_timeout` must be a value of type `PNA_IdleTimeout_t`:

```
/// Supported values for the pna_idle_timeout table property
enum PNA_IdleTimeout_t {
    NO_TIMEOUT,
    NOTIFY_CONTROL,
    AUTO_DELETE
};
```

If the property `pna_idle_timeout` is not specified for a table, its default value is `NO_TIMEOUT`. Such tables need not maintain an idle time for any of its table entries, and will not perform any special action regardless of how long a table entry remains unmatched.

If the property pna_idle_timeout is assigned a value of NOTIFY_CONTROL, the behavior is the same as defined in the Portable Switch Architecture if a table has its property psa_idle_timeout assigned a value of NOTIFY_CONTROL. See the section titled "Table entry timeout notification" in the PSA specification[5].

If the property pna_idle_timeout is assigned a value of AUTO_DELETE, the behavior is similar to the behavior of the value NOTIFY_CONTROL, except that no notification message is generated to the control plane when an entry's idle time is reached. Instead, the data plane deletes the table entry.

PNA implementations may restrict pna_idle_timeout to be AUTO_DELETE only for tables that also have add_on_miss equal to true.

PNA implementations are expected to be able to perform add-on-miss at very high rates relative to line rate, and similarly for such add-on-miss tables, they should be able to perform auto-deletion of entries in the data plane at a similarly large rate. If a P4 developer wishes to use the high rate add-on-miss capabilities for a particular table, it is likely that they do not wish the control plane to be responsible for keeping up with a high rate of deleting idle entries, and thus will often use add_on_miss = true and pna_idle_timeout = PNA_IdleTimeout_t.AUTO_DELETE together.

9. Timestamps

10. Atomicity of control plane API operations

A. Appendix: Revision History

Release	Release Date	Summary of Changes
0.1	November 5, 2020	Skeleton specification.
0.5	May 15, 2021	Initial draft.
0.7	December 22, 2022	Version 0.7

A.1. Changes made in version 0.7

- Added extern functions add_entry_if, set_entry_expire_time_if, and update_expire_info, intended to be more friendly to targets with poor support for if statements inside of actions.
- Added parameter expire_time_profile_id to extern function add_entry, to specify the initial expire time profile id for a new table entry added by the data plane.
- Removed obsolete references to send_to_vport in example programs, replacing with current send_to_port.
- Add crypto_accelerator extern for basic encrypt / decrypt of a specified portion of a packet, and example program ipsec-acc.p4 demonstrating its use.
- Added match_kind optional.
- Clarified restrictions on when extern function add_entry may be called.
- Removed PreControl, leaving MainParser, MainControl, and MainDeparser as the P4-programmable blocks.
- There is no more "loopback" in PNA. A packet can be sent to a network port or a host port, regardless of where it came from, and this does NOT automatically cause the packet leaving the MainDeparser to later come back for another pass of processing. It will only do so if recirculation is explicitly requested.
- Defined possible values of type PNA_HashAlgorithm_t, to match the values defined for PSA.

[5] The Portable Switch Architecture specification can be found here: https://p4.org/specs

- Added descriptions of the possible values of `pna_idle_timeout` table property, and their behaviors. What was originally proposed as a new table property named `idle_timeout_with_auto_delete` was instead defined as a new possible value for `pna_idle_timeout`.
- Removed unused type `PNA_PacketPath_t` from `pna.p4`.
- Modifications to intrinsic metadata fields: Removed `direction`, `pass`, `loopedback`. Added `recirculated`. For something similar to `direction` added extern functions `is_host_port` and `is_net_port` that can be called to determine whether a port is a host port or network port. This also motivated a change to the parameters of extern function `SelectByDirection`.
- Many minor changes in example P4 programs to keep up with changes to the specification and pna.p4 include file.

B. Appendix: Open Issues

C. Appendix: Rationale for design

C.1. Why a common pipeline, instead of separate pipelines for each direction?

TBD: Andy can write this one. Basic reasons are summarized in existing slides.

C.2. Is it inefficient to have the `MainParser` redo work?

If the only changes made by the inline extern in the network-to-host direction were to decrypt parts of the packet that were previously encrypted, but everything before the first decrypted byte remained exactly the same, then it seems like it is a waste of effort that the main parser starts parsing the packet over again from the beginning.

It is true that an IPsec decryption inline extern is unlikely to change an Ethernet header at the beginning of the packet, but it does seem likely that it could make the following kinds of changes to parts of the packet before the first decrypted byte:

- Remove headers: If the received packet was IPsec tunnel mode, it might be useful if the inline extern removes the outer IP header, since it was added to the packet at the point of IPsec encryption. The software sending the packet (before IPsec encryption occurred) did not create that header, and the corresponding layer of software receiving the decrypted packet does not want to see such IPsec-specific headers.
- Modify headers: If the received packet was IPsec transport mode, it might be useful if the IP header whose protocol was equal to the standard numbers for AH or ESP was changed to be the next header after the AH and ESP headers are removed by the inline extern. Again, what an IPsec decryption block does might be useful to make similar to what the IPsec layer of software does in a software IP stack. The layer of software processing the decrypted packet should see what the last layer of software sent before it was encrypted.

If any or all of the above are true of the inline extern block's changes to the packet, then it seems that the only way you could save the main parser some work is to somehow encode the results of the pre parser, and also undo those results for any headers that were modified in the inline extern. Then you would also need the main parser to be able to start from one of multiple possible states in the parser state machine, and continue from there.

That is all possible to do, but it seems like an awkward thing to expose to a P4 developer, e.g. should we require them to write a main parser that has a start state that immediately branches one of 7 ways based upon some intermediate state the the pre parser reached, as modified by the inline extern if it modified or removed some of those headers?

A NIC implementation might do such things, and it seems likely an implementation might use some of the techniques mentioned in the previous paragraph, but hidden from the P4 developer. The proposed PNA design should not prevent this, if an implementer is willing to go to that effort.

D. Appendix: Packet path figures

D.1. From network, sent to host

D.2. From network, sent to host, with mirror copy to different host

D.3. From host, to network

D.4. From host, to network, with mirror copy to a different host

D.5. From host, to host

D.6. From port, to port

E. Appendix: Packet ordering