P4$_{16}$ Intel® Tofino™ Native Architecture – Public Version

Application Note

Apr 2021

Document Number: 631348-0001

Legal Disclaimer

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel, Tofino, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Contents

Figures

Tables

Revision History

Date	Revision	Description
Apr 2019	0001	Initial Release of Public Version of Intel Tofino Native Architecture.

1 Introduction

This document describes the Intel® Tofino™ Native Architecture (TNA), a $P4_{16}$ architecture that provides a P4 programming interface that closely maps to the features of the Tofino family of switch ASICs.

While there are many similarities between TNA and the Portable Switch Architecture (PSA) published by P4.org, they are not the same.

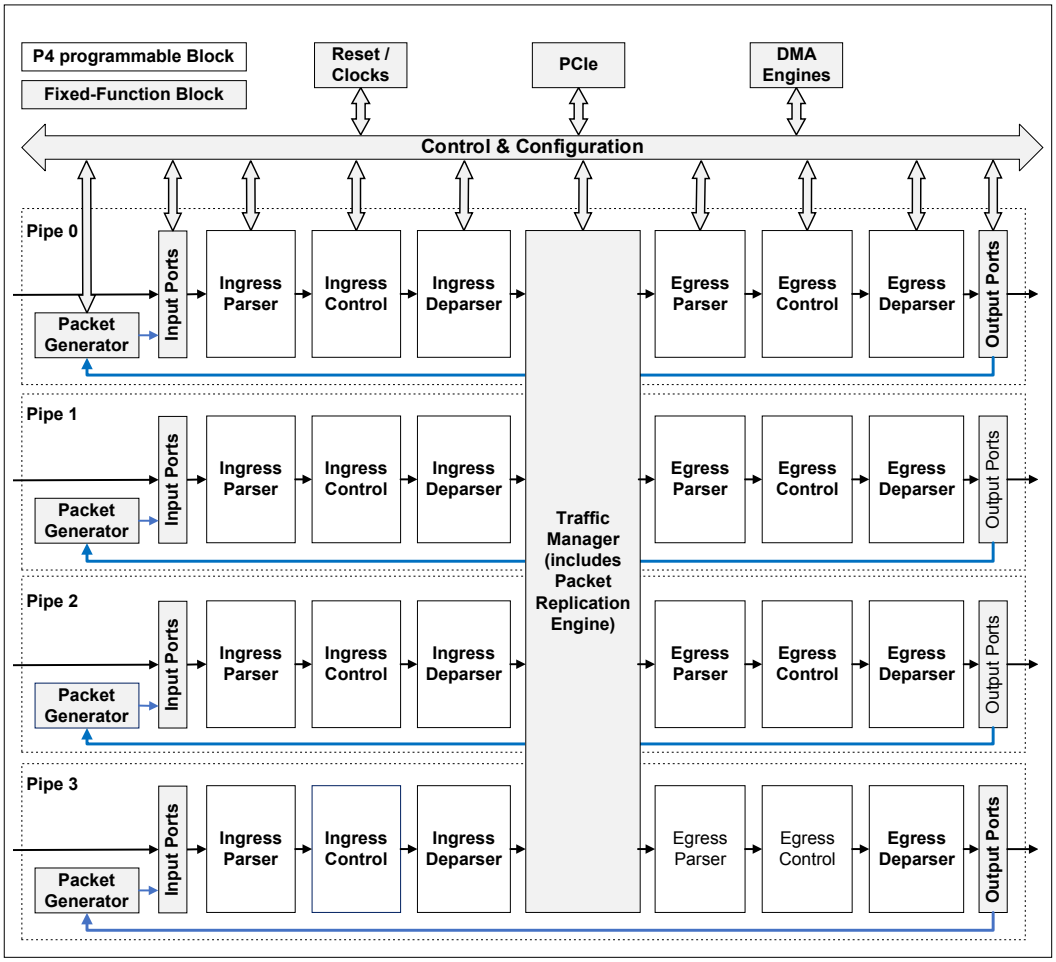
NOTE: The $P4_{16}$ specification is a prerequisite for this document. See https://p4.org/specs for relevant specifications.

2 Intel Tofino

The focus of this document is how to write P4 programs for Intel Tofino. To take advantage of its features, it helps to understand the major parts of Intel Tofino, and how they interact with each other.

The figure below shows a 6.5Tbps Intel Tofino with 4 pipes.

Figure 1: 4-pipe Intel Tofino Block Diagram



Each of the 4 pipes is identical in its internal structure. In each pipe, the blocks labeled "Input Ports" and "Output Ports" contain 16 Ethernet MACs. Each Ethernet MAC can be configured independently as a single 100 Gigabit Ethernet channel, as four 25 Gigabit Ethernet channels, or one of several other modes described in Section 12.

Pipe 0 is also connected to one additional 100Gbps Ethernet port, called the CPU Ethernet port. Pipe 2 is connected to a DMA packet interface, called the CPU PCIe

port. See Section 12 for port numbers. The CPU PCIe port is always connected to a local CPU. The CPU Ethernet port is connected to a local CPU in some systems, or in others can be connected to a remote-control plane (e.g., an SDN controller).

The control plane uses the APIs implemented in the driver software to configure everything through a PCI Express interface, often using DMA channels to increase the rate that large tables can be configured.

In the data plane, a packet is first received by an Input Ports block, and then flows from left to right through the blocks shown in Figure 1: 4-pipe Intel Tofino Block Diagram. The Ingress Parser identifies which headers are present in the packet. The headers of interest are determined by the P4 program. The Ingress Control is responsible for the bulk of data packet processing. Among other things, it is responsible for choosing the desired destination for the packet. By destination we include possibilities such as unicast, multicast, dropping the packet, etc. The Ingress Deparser emits the headers as specified in the P4 program and constructs the packet by prepending these headers to the packet payload, which is the portion of the packet left after parsing.

The Traffic Manager stores packets in a packet buffer, optionally replicates and then enqueues the packet for the selected port(s). Later, the scheduler will dequeue the packet, and send it to the Egress Parser in the selected port's pipe.

The Egress Parser, Egress Control, and Egress Deparser perform similar operations as their counterparts in the ingress pipeline, but the P4 code controlling their behavior is often quite different. For example, the Egress Control cannot change the output port of the packet. The Output Ports block physically transmits the signal representing the output packet to the external physical interface.

Additional differences between ingress and egress will be explained throughout this document, and all available packet paths are described in Section 5.

3 Naming Conventions and Data Types

3.1 Naming Conventions in the Document

In this document we use the following naming conventions:

- Types are named using CamelCase followed by `_t`. For example, `PortId_t`.
- Control types and extern object types are named using CamelCase.
- Struct types are named using lower case words separated by _ followed by `_t`.
- Header types are named using lower case words separated by _ followed by `_h`.
- Actions, extern methods, extern functions, headers, structs, and instances of controls and start with lower case and words are separated using _.
- Enum members, const definitions, and #define constants are set in all caps, with words separated using _.

3.2 Type Definitions

TNA defines the following types with fixed, architecture-specific bit widths. These widths are defined in the architecture file, and it is recommended that these typedef names are used as the types (rather than the hardcoded bit width values) in P4 programs.

NOTE: Do not use these types in the packet headers that are visible outside of Intel Tofino – these can lead to non-portable packet formats

```
typedef bit<9>  PortId_t;           // Port id
typedef bit<16> MulticastGroupId_t; // Multicast group id
typedef bit<5>  QueueId_t;          // Queue id
typedef bit<10> MirrorId_t;         // Mirror session id
typedef bit<16> ReplicationId_t;    // Replication id

const   bit<32>   PORT_METADATA_SIZE = 64;
```

Note: Many intrinsic header definitions shown in this document omit padding fields that are present in the definitions used by the P4 compiler. See Section 11.2 for a description of the `@padding` annotation that is often used for such fields.

4 The Intel Tofino Native Architecture

The $P4_{16}$ language specification describes the syntax for defining parser and control blocks. The specification does not say which parameters these parsers and controls should have on a particular target device. The language specification also gives the syntax for defining extern objects and their methods, but the set of supported externs can vary from one P4-programmable device to another. These variations are described in $P4_{16}$ architectures, and TNA is such an architecture.

This section describes how to write P4 parsers and controls for an Intel Tofino device using TNA. There are a few variations, depending upon whether you want all pipes within an Intel Tofino device to process packets using the same P4 code, or whether you wish some pipes to process packets differently from each other.

Section 5 describes in detail the intrinsic metadata fields that TNA provides. Many are inputs to your P4 program, such as the port on which a packet arrived, or the time when it arrived. Others are outputs from your P4 program, directing the behavior of the device, such as whether the packet should be dropped, or sent to one output port, or replicated to a group of output ports.

The $P4_{16}$ language specification defines several table properties such as `key` and `actions`. Section 6 describes additional table properties provided by TNA that can be used to modify the behavior of the table.

Section 7 defines the extern objects and functions provided by TNA. Section 9 describes options for configuring the behavior of the Intel Tofino packet generators. Section 11 describes several $P4_{16}$ annotations that may help in writing some kinds of programs for TNA.

4.1 Identical P4 code for all pipes

To process packets identically in all pipes, you must write P4 code for two parsers and four controls, corresponding to the six programmable blocks of one of the pipes.

The P4 developer must define a P4 struct that contains all headers of interest for their use case in the ingress pipeline. They must also define a P4 struct that contains all user-defined metadata fields of interest. The type names of these structs is up to the choice of the P4 developer. The examples in this section will use the type names `my_ingress_headers_t` and `my_ingress_metadata_t`. These same type names must be used when defining the ingress parser, ingress control, and ingress deparser in the program.

The sample P4 code below demonstrates defining an ingress parser, ingress control, and ingress deparser. The names of the parsers and controls and the names of parameters are up to the choice of the P4 developer. The things that must be exactly as shown in the example are the order of the parameters, their direction (`in`, `out`, or `inout`), and the types `packet_in` and any type names containing `intrinsic_metadata` as part of their name. The contents of those intrinsic metadata types will be described in Section 5.

```
parser MyIngressParser(
    packet_in pkt,
    out my_ingress_headers_t  ig_hdr,
    out my_ingress_metadata_t ig_md,
    out ingress_intrinsic_metadata_t ig_intr_md)
{
    state start {
        // parser code begins here
        transition accept;
    }
}

control MyIngress(
    inout my_ingress_headers_t  ig_hdr,
    inout my_ingress_metadata_t ig_md,
    in    ingress_intrinsic_metadata_t               ig_intr_md,
    in    ingress_intrinsic_metadata_from_parser_t   ig_prsr_md,
    inout ingress_intrinsic_metadata_for_deparser_t ig_dprsr_md,
    inout ingress_intrinsic_metadata_for_tm_t       ig_tm_md)
{
    apply {
        // ingress control code here
    }
}

control MyIngressDeparser(
    packet_out pkt,
    inout my_ingress_headers_t  ig_hdr,
    in    my_ingress_metadata_t ig_md,
    in    ingress_intrinsic_metadata_for_deparser_t ig_dprsr_md)
{
    apply {
        // emit headers for out-of-ingress packets here
    }
}
```

The egress parser, egress control, and egress deparser must also have two struct types defined for the purposes of holding headers and user-defined metadata. These types can be different than the types used in the ingress code, but you may also use the same type names for ingress and egress code if that is desired. The examples in this section will use the type names `my_egress_headers_t` and `my_egress_metadata_t`.

The sample P4 code below demonstrates defining an egress parser, egress control, and egress deparser. As for ingress, the names of parsers and controls and parameter names may be chosen by the P4 developer.

```
parser MyEgressParser(
    packet_in pkt,
    out my_egress_headers_t  eg_hdr,
    out my_egress_metadata_t eg_md,
    out egress_intrinsic_metadata_t eg_intr_md)
{
    state start {
        // parser code begins here
        transition accept;
    }
```

```
}

control MyEgress(
    inout my_egress_headers_t  eg_hdr,
    inout my_egress_metadata_t eg_md,
    in    egress_intrinsic_metadata_t                 eg_intr_md,
    in    egress_intrinsic_metadata_from_parser_t     eg_prsr_md,
    inout egress_intrinsic_metadata_for_deparser_t    eg_dprsr_md,
    inout egress_intrinsic_metadata_for_output_port_t eg_oport_md)
{
    apply {
        // egress control code here
    }
}

control MyEgressDeparser(
    packet_out pkt,
    inout my_egress_headers_t  eg_hdr,
    in    my_egress_metadata_t eg_md,
    in    egress_intrinsic_metadata_for_deparser_t eg_dprsr_md)
{
    apply {
        // emit desired egress headers here
    }
}
```

After defining one of each of these six blocks, then they are collected together into a definition for a pipe, as shown in the example below. The names of the parameters to the `Pipeline` package must match the names used earlier when defining those controls. The name of the `Pipeline` instance may be chosen by the P4 developer. In the example below uses the instance name `pipe`.

```
Pipeline(MyIngressParser(),
         MyIngress(),
         MyIngressDeparser(),
         MyEgressParser(),
         MyEgress(),
         MyEgressDeparser()) pipe;
```

Now to finish the program, create the top-level instance. Following the $P4_{16}$ language specification, this instance must be named `main`.

```
Switch(pipe) main;
```

This package instantiation for package `Switch` has a single parameter `pipe`. When the package `Switch` is instantiated with a single parameter, it means that the six programmable blocks bundled together in the instance `pipe` will be duplicated in every pipe of the target device.

The word "duplicated" is significant here. If the program above is compiled for a 4-pipe Intel Tofino, any P4 tables or extern objects defined within the controls `MyIngress` and `MyEgress` will have four separate instances, one per pipe. There are no tables or extern objects that can be physically shared across separate pipes. This automatic duplication is provided as a convenience. It is a modification of the $P4_{16}$ language specification.

There are control plane APIs that can make it convenient for control plane software to keep the entries of such a duplicated table the same as each other. For example, there are options to some APIs that perform table add, delete, and modify operations on all four pipes with a single call, without the control plane software having to make a separate call per pipe. There are also APIs that modify the entries of a table in only one of the four pipes, and in that case, the other three pipes are unaffected.

4.2 Different P4 code running on different pipes

Building upon the example of the previous section, it is also possible to define another set of six programmable blocks with different names than the ones defined above. Rather than show nearly identical example code here as given in the previous section, imagine that the example code there is duplicated, but the parser and control names all have the number 2 appended to them. That is, their names are `MyIngressParser2`, `MyIngress2`, `MyIngressDeparser2`, `MyEgressParser2`, `MyEgress2`, and `MyEgressDeparser2`. They may use the same struct type names for headers and user-defined metadata as in the previous section, or they may have their own independent definitions.

The example code below shows the creation of a separate `Pipeline` instance named `pipe2`, which uses this second set of six blocks.

```
Pipeline(MyIngressParser2(),
         MyIngress2(),
         MyIngressDeparser2(),
         MyEgressParser2(),
         MyEgress2(),
         MyEgressDeparser2()) pipe2;
```

The example below has four parameters to the `Switch` package instantiation.

```
Switch(pipe, pipe, pipe2, pipe2) main;
```

This example will cause Intel Tofino pipes 0 and 1 to execute the P4 code of the first set of six programmable blocks shown in the previous section, and pipes 2 and 3 to execute the P4 code of the second set of six programmable blocks whose names end with “2”.

As before, if one of the `pipe` or `pipe2` instances is used multiple times in an instantiation of the package `Switch`, they are duplicated.

Many variations of this example are also supported. For example, to have different code for the ingress and egress control, but identical code for the parsers and deparsers, there is no requirement to copy and paste the parser and deparser code. Instead, one can define `pipe3` as in the example below. Note that all parser and deparser parameters are the same as the ones used when creating instance `pipe`.

```
Pipeline(MyIngressParser(),
         MyIngress2(),
         MyIngressDeparser(),
         MyEgressParser(),
         MyEgress2(),
         MyEgressDeparser()) pipe3;
```

4.3 Programmable block names

Throughout this document, the names of the P4-programmable blocks used will be:

- Ingress parser
- Ingress control – the longer name "ingress match-action control" is a synonym, but ingress control is used for brevity.
- Ingress deparser
- Egress parser
- Egress control
- Egress deparser

The $P4_{16}$ language specification allows parsers to call sub-parsers, and controls to call other controls. TNA supports this.

If a TNA program's ingress control calls control C1, then C1 is considered part of the ingress control, e.g. for the purposes of restrictions on what kinds of extern objects that control C1 may instantiate and call, as documented in Section 7. The same applies for all programmable blocks listed above.

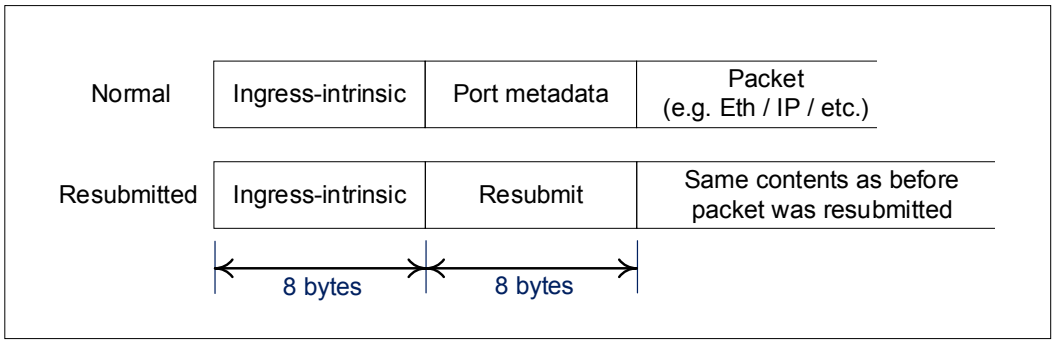
5 Packet Paths

This section describes all packet paths through Intel Tofino and how to exercise them in TNA. It also describes in detail the contents of packets received by the ingress and egress pipeline, and what happens to packets after they finish executing the P4 code of the ingress and egress pipeline. There is also a description of the options and behavior of multicast replication.

5.1 Contents of packets received by ingress parser

TNA provides the following data at the beginning of packets as they begin parsing in the ingress parser.

Figure 2: Ingress Parser Byte Stream Formats



As shown in Figure 2: Ingress Parser, the byte stream format depends on the packet processing path. The first 8 bytes contain the ingress intrinsic metadata which is common to all packet types. The ingress parser is expected to extract this metadata into the argument of the type `ingress_intrinsic_metadata_t` so that this data will be available for use in the ingress control.

The next 8 bytes hold either the port metadata or the resubmit metadata (depending on the `resubmit_flag` bit inside the ingress intrinsic metadata). The contents of the port metadata and resubmit metadata is defined by the P4 program. The size is always 8 bytes. The ingress parser is expected to extract these headers or to skip over them if they are not needed.

For packets from an Ethernet port, the Ethernet header begins immediately after the port metadata or resubmit metadata.

Note: It is possible for a packet received from outside Intel Tofino on an Ethernet port to begin with something other than an Ethernet header. The only absolute requirement is that it be a sequence of bytes treated as valid by Intel Tofino's Ethernet MAC logic.

In TNA, recirculation of packets is accomplished by sending a packet to a port that is in loopback mode. Intel Tofino supports recirculation ports that are never connected to front panel ports and always in loopback mode. The ports that may normally be used

to connect to the front panel may also be configured in loopback mode if desired. The P4 code is in control of making the choice for a packet to be sent to a port in loopback mode.

If a received packet was created by a packet generator, the packet will contain a 6-byte `pktgen` header immediately after the port metadata header. See Section 9 for more details on packet generation, and the contents of the headers and payloads it creates.

The header definition below shows the fields within the ingress intrinsic metadata. Extracting a header with this type will typically be the first step in your ingress parser code.

Note: Padding fields have been omitted from the definition of `ingress_intrinsic_metadata_t` for brevity.

```
header ingress_intrinsic_metadata_t {
    bit<1>   resubmit_flag;       // Flag distinguishing original
                                  // packets from resubmitted packets.
    bit<2>   packet_version;      // Read-only Packet version.
    PortId_t ingress_port;        // Ingress physical port id.

    bit<48>  ingress_mac_tstamp;  // Ingress IEEE 1588 timestamp
                                  // (in nsec) taken at the
                                  // ingress MAC.
}
```

Regardless of whether a packet was received from outside the device, from a packet generator, or looped back internally, the packet may be resubmitted once by your P4 program. See Section 7.14 for instructions on doing so. When the resubmitted packet begins ingress parsing again, it will contain resubmit metadata supplied by the P4 program when the resubmit operation was invoked, instead of port metadata. The rest of the packet after the resubmit metadata will be the same as the original packet.

5.1.1 Parser

TNA supports the `extract`, `lookahead`, and `advance` methods on the type `packet_in` parameter.

The function `verify()` defined in the $P4_{16}$ language specification is not supported. You may instead assign values to user-defined metadata fields in the parser code recording error status when packet header contents are found that you consider illegal or malformed.

TNA provides an extern function `port_metadata_unpack` to aid in extracting the port metadata of a packet. The P4 developer first defines a struct type that contains the fields they wish to appear in the port metadata for their program. The struct must contain no more than 64 bits; it may contain less than 64 bits. This struct need not contain any padding fields. The P4 compiler will choose the bit-level format of the fields.

In the example below, this struct type is named `port_metadata_t`.

```
struct port_metadata_t {
    bit<3>   port_pcp;
    bit<12>  port_vid;
}
```

Like a call to `extract`, calling `port_metadata_unpack` advances the "pointer" in the packet being parsed. `port_metadata_unpack` always advances the pointer by the size of the port metadata, regardless of the definition of the struct type.

A significant benefit of using `port_metadata_unpack` for extracting port metadata is that the compiler generates control plane APIs for the individual port metadata fields that are convenient for the control plane developers. Thus it is recommended that you use `port_metadata_unpack` if you wish to use port metadata, or `advance` to skip over port metadata if you do not wish to use it.

5.1.1.1 Example Ingress Parser

The example P4 code below demonstrates an ingress parser that parses the ingress intrinsic metadata. It also uses the `resubmit_flag` field inside of that header to decide whether port metadata or resubmit metadata appears next. If port metadata appears next, it uses `port_metadata_unpack` to extract the port metadata. If resubmit metadata appears next, it uses `lookahead` to get the resubmit metadata, then `advance` to skip past the resubmit metadata, which works correctly even if the resubmit metadata header definition is smaller than 8 bytes.

```
header resubmit_metadata_t {
    // user-defined fields, up to the maximum size supported
}

struct port_metadata_t {
    // user-defined fields, up to the maximum size supported
}

struct my_ingress_metadata_t {
    port_metadata_t port_meta;
}

struct my_ingress_headers_t {
    resubmit_metadata_t resub;
    ethernet_h ethernet;
}

parser MyIngressParser(
    packet_in pkt,
    out my_ingress_headers_t  ig_hdr,
    out my_ingress_metadata_t ig_md,
    out ingress_intrinsic_metadata_t ig_intr_md)
{
    state start {
        pkt.extract(ig_intr_md);
        transition select (ig_intr_md.resubmit_flag) {
            1: parse_resubmit;
```

```
            0: parse_port_metadata;
        }
    }
    state parse_resubmit {
        ig_hdr.resub = pkt.lookahead<resubmit_metadata_t>();
        // Always skip forward this size, regardless of how large the
        // fields in resubmit_metadata_t are.
        pkt.advance(PORT_METADATA_SIZE);
        transition parse_ethernet;
    }
    state parse_port_metadata {
        ig_md.port_meta = port_metadata_unpack<port_metadata_t>(pkt);
        transition parse_ethernet;
    }
    state parse_ethernet {
        pkt.extract(ig_hdr.ethernet);
        transition accept;
    }
}
```

5.2 Additional intrinsic metadata from the ingress parser

The ingress parser generates additional intrinsic metadata, which becomes an input to the ingress control. It has the type `ingress_intrinsic_metadata_from_parser_t`. Its fields are shown below.

```
struct ingress_intrinsic_metadata_from_parser_t {
    bit<48> global_tstamp;
    bit<32> global_ver;
    bit<16> parser_err;
}
```

The field `global_tstamp` contains the time in nanoseconds when the packet entered the ingress parser. The field `parser_err` contains a code indicating any error that occurred during ingress parsing. See Section 5.2.1 for a list of parser errors and their causes. The field `global_ver` is reserved.

5.2.1 Parser Errors

Intel Tofino's parser may detect errors while parsing a packet. The errors are exposed in the ingress and egress pipeline in the value of an intrinsic metadata field called `parser_err`.

```
// PARSER_ERROR_OK is the value of parser_err if no errors occur
const bit<16> PARSER_ERROR_OK           = 16w0x0000;

const bit<16> PARSER_ERROR_NO_TCAM      = 16w0x0001;
const bit<16> PARSER_ERROR_PARTIAL_HDR  = 16w0x0002;
const bit<16> PARSER_ERROR_CTR_RANGE    = 16w0x0004;
```

```
const bit<16> PARSER_ERROR_TIMEOUT_USER = 16w0x0008;
const bit<16> PARSER_ERROR_TIMEOUT_HW   = 16w0x0010;
const bit<16> PARSER_ERROR_SRC_EXT      = 16w0x0020;
const bit<16> PARSER_ERROR_DST_CONT     = 16w0x0040;
const bit<16> PARSER_ERROR_PHV_OWNER    = 16w0x0080;
const bit<16> PARSER_ERROR_MULTIWRITE   = 16w0x0100;
const bit<16> PARSER_ERROR_ARAM_MBE     = 16w0x0400;
const bit<16> PARSER_ERROR_FCS          = 16w0x0800;
```

If multiple errors occur while parsing a packet, the value of the `parser_err` field will be the bitwise OR of each of the error codes. For example, if both `PARTIAL_HDR` and `SRC_EXT` errors occur, the value of `parser_err` will be (0x0002 | 0x0020) = 0x0022.

If the ingress control P4 code never refers to the `parser_err` field, the compiler configures the ingress parser to drop all packets that experience any parsing error in the list above. For such a P4 program, these parsing error packets will never execute the code of the ingress control.

If the ingress control P4 code does refer to the `parser_err` field, packets that experience a `NO_TCAM` parsing error are configured to be dropped in the ingress parser and will never execute the code of the ingress control. If a `NO_TCAM` parsing error is not encountered by the packet during ingress parsing, the packet will be processed by the ingress control.

For packets that are parsed by the egress parser, the egress control always processes the packet, regardless of whether the egress parser detected an error.

If the P4 code explicitly performs a transition to the `reject` state, it will terminate parsing, but does not cause any of the parser error flags to be set.

The P4 developer can choose to match on the `parser_err` field and perform any desired actions. One simple approach to handling parser errors in TNA is to drop the packet. In the ingress pipeline there are additional safe options: mirroring the packet, or resubmitting.

Processing a packet that was parsed with errors can lead to unpredictable results, because the hardware does not always complete all operations that appear in the P4 parser code before the parsing error was detected.

See below for descriptions of the reasons that each of these errors can occur.

`PARSER_ERROR_OK`: This is not an error. It is the value of `parser_err` when no parsing error has occurred.

`PARSER_ERROR_NO_TCAM`: This error occurs if the parser executes a `select` expression, and none of the cases matched. A `select` expression with a `default` case at the end will never cause this error. In some situations, you may wish to write `select` expressions where packets with unexpected values in some header fields cause this parsing error to occur, by design. Then the egress control code can be written to check when a parsing error occurs and drop the packet (packets experiencing this error are always dropped by the ingress parser).

`PARSER_ERROR_PARTIAL_HDR`: (partial header) This error occurs if there is a call to the `extract`, `lookahead`, or `advance` method on a packet, where the operation attempts to read or advance beyond the end of the packet.

PARSER_ERROR_CTR_RANGE: (match loop counter range) This error occurs if a ParserCounter extern instance is ever changed to hold a value outside of the range configured by the P4 developer. See 7.11 for documentation on the ParserCounter extern.

PARSER_ERROR_TIMEOUT_USER: (user timeout) This error occurs for a similar reason as the PARSER_ERROR_TIMEOUT_HW error. The elapsed parsing time required for this error to occur is configurable and can be set shorter than the time required for a PARSER_ERROR_TIMEOUT_HW error to occur.

PARSER_ERROR_TIMEOUT_HW: (hardware limit timeout) This error occurs if the time elapsed while parsing a packet exceeds a maximum hardware limit built into the device. This is a quite long time that no packet should require to complete parsing.

PARSER_ERROR_SRC_EXT: (source extraction) If this error occurs, it is a sign of an error in the P4 compiler's output.

PARSER_ERROR_DST_CONT: (destination container) This error status bit is reserved, and currently not used.

PARSER_ERROR_PHV_OWNER: (PHV owner) This error occurs if the ingress parser attempts to write to a field owned by the egress control, or the egress parser attempts to write to a field owned by the ingress control. If this error occurs, it is a sign of an error in the P4 compiler's output.

PARSER_ERROR_MULTIWRITE: (multiple write) If this error occurs, in most cases it is a sign of an error in the P4 compiler's output.

PARSER_ERROR_ARAM_MBE: (action RAM multi-bit error) This error occurs if the parser read a configuration memory during parsing, and error detection hardware logic indicated that the contents must have been corrupted after it was written by driver software when loading the compiled P4 program. In most cases this error can occur only very briefly, as driver software will detect and correct the problem.

PARSER_ERROR_FCS: This error can occur for several different reasons. These reasons all have in common that there was some error detected before the parser, and this error is signaled to the parser. The parser then uses this error flag to pass on the presence of any of those errors. One example is that an Ethernet frame was detected to be corrupted because its FCS (Frame Check Sequence) did not match the one calculated from the frame contents.

5.3 Field validity

The $P4_{16}$ language specification defines validity for headers. TNA adds to this a feature where some intrinsic metadata fields have their own validity, independent of any headers or other fields. The following is a complete list of the fields with this feature.

Table 1: Fields with Validity

Fields with validity	Struct type name in which the fields occur

`ucast_egress_port`
`mcast_grp_a`
`mcast_grp_b`
	`ingress_intrinsic_metadata_for_tm_t`

`digest_type` `resubmit_type`	`ingress_intrinsic_metadata_for_deparser_t`
`mirror_type`	`egress_intrinsic_metadata_for_deparser_t`

When the ingress parser begins executing, all ingress fields with validity are initialized to invalid. When the egress parser begins executing, all egress fields with validity are initialized to invalid.

When the P4 code assigns a value to a field with validity, the field automatically becomes valid. If the P4 program never assigns a value to a field with validity while processing a packet, the field remains invalid. Calling the TNA extern function `invalidate` on a field with validity forces it to become invalid, e.g. `invalidate(ig_tm_md.ucast_egress_port)`.

The validity of these fields is significant in determining what happens to the packet, as will be described later.

5.4 Destinations

When a P4 program sends a packet to the Traffic Manager, there is a group of intrinsic metadata fields that control what copies will be made of the packet, and where each of these copies will go. This group of metadata fields is called a destination in this document.

A destination contains fields to specify the following places to which a packet will be sent; there are conditions described later in which a packet will be sent to only some of the following places, not all of them.

- A unicast output port `ucast_egress_port`. If it is invalid, or if it is valid but the numeric value is not the number of any port on the device, no unicast copy of the packet will be made. See Section 5.3 for the meaning of field validity, and Section 12 for port numbers.
- Two multicast group ids, `mcast_grp_a` and `mcast_grp_b`. Each may be valid or invalid, independently of the other. Only the ones that are valid cause multicast copies to be created. See Section 5.3 for the meaning of field validity, and Section 5.7.3 for details on multicast packet replication. Note that even a valid `mcast_grp` field can refer to a multicast group that is configured to make 0 copies of the packet.
- A 1-bit copy-to-CPU flag indicating whether to create a copy to send to the CPU (control plane). The CPU port number is configurable via the corresponding traffic manager APIs. It must be a single port, not a multicast group.

Note that a single packet sent to a destination can be sent to any subset of the four places (unicast, multicast group A, multicast group B, copy-to-CPU).

Below are most of the fields in the structure of type `ingress_intrinsic_metadata_for_tm_t` that is an output from the ingress control. The other fields are described in Section 5.5. The fields here are those describing a destination.

```
struct ingress_intrinsic_metadata_for_tm_t {
    // Only fields that are part of the destination are included here.

    // fields related to unicast
    PortId_t  ucast_egress_port;
    QueueId_t qid;

    // fields controlling multicast replication
    MulticastGroupId_t mcast_grp_a;
    MulticastGroupId_t mcast_grp_b;
    bit<13>   level1_mcast_hash;
    bit<13>   level2_mcast_hash;
    bit<16>   level1_exclusion_id;
    bit<9>    level2_exclusion_id;
    bit<16>   rid;

    // fields to control copy-to-CPU
    bit<1>    copy_to_cpu;
    bit<3>    icos_for_copy_to_cpu;

    // other fields used by Traffic Manager
    bit<3>    ingress_cos;
    bit<2>    packet_color;
}
```

5.5 Intrinsic metadata for the ingress deparser

The following struct is an input and output of the ingress control, and an input to the ingress deparser, where its fields are used to control several operations mentioned below.

```
struct ingress_intrinsic_metadata_for_deparser_t {
    bit<3>     drop_ctl;
    bit<3>     digest_type;
    bit<3>     resubmit_type;
    MirrorType mirror_type;
}
```

As mentioned in Section 5.3, fields `digest_type` and `resubmit_type` have field validity and are initially invalid. TNA initializes `drop_ctl` and `mirror_type` to 0 for each packet.

See Section 7.7 for details on `digest_type`, Section 7.14 for details on `resubmit_type`, Section 7.10 for details on `mirror_type`, and Section 5.6 for details on `drop_ctl`.

5.6 Behavior of packets after ingress processing

The behavior of a packet after ingress processing is complete is summarized in the following pseudocode.

```
if (ingress deparser invoked a digest operation) {
    generate a digest message to the control plane software;
}
if (ingress deparser invoked a resubmit operation) {
    resubmit the packet;
} else {
    if (ingress deparser invoked a mirror operation) {
        send ingress-to-egress mirror packet to the Traffic Manager;
    }
    if (drop_ctl[0:0] == 1) {
        invalidate the ucast_egress_port, mcast_grp_a, and mcast_grp_b
            fields of the destination;
    }
    if (drop_ctl[1:1] == 1) {
        assign 0 to copy_to_cpu bit of destination;
    }
    send the normal packet output by the deparser to the Traffic Manager;
}
```

Note that if a packet is resubmitted, it will only be resubmitted, and no packet will go to the Traffic Manager. If the packet is not resubmitted, then both a mirror packet and the normal packet will go to the Traffic Manager if a mirror operation is invoked.

See Section 7.7 for how to invoke a digest operation, Section 7.14 for how to invoke a resubmit operation, and Section 7.10 for how to invoke a mirror operation. A mirror packet's destination is determined by looking up its mirror session id in a table in the Traffic Manager.

A normal packet's destination is determined by fields in the structure with type `ingress_intrinsic_metadata_for_tm_t` that is an output of the ingress control. The fields relevant for the destination are described in Section 5.3. In the structure with type `ingress_intrinsic_metadata_for_deparser_t` that is an output of the ingress control, there is a field `drop_ctl`. The `drop_ctl` field modifies the normal packet's destination as follows.

- If bit 0 of `drop_ctl` is 1, then the unicast egress port and both multicast groups of the normal packet's destination are invalidated. If bit 0 of `drop_ctl` is 0, then the validity of the unicast egress port and multicast groups are left as specified in the destination.
- If bit position 1 of `drop_ctl` is 1, then no copy-to-CPU will occur for the normal packet. If bit position 1 of `drop_ctl` is 0, then a copy-to-CPU will occur if the `copy_to_cpu` field is 1.

Below are a few of the fields in the structure with type `ingress_intrinsic_metadata_for_tm_t` that is an output from the ingress control. The other fields are described in Section 5.3. The fields here are only those that are not part of the destination.

```
struct ingress_intrinsic_metadata_for_tm_t {
    // Fields that are part of the destination are omitted here.

    bit<1>  bypass_egress;
    bit<1>  deflect_on_drop;
    bit<1>  disable_ucast_cutthru;
    bit<1>  enable_mcast_cutthru;
}
```

The egress processing can be skipped by assigning the `bypass_egress` field to 1 in the ingress pipeline. This applies to all copies that are made of the packet according to the values in its destination.

See Section 5.7.2 for the behavior of the `deflect_on_drop` field.

The default behavior of Intel Tofino is store-and-forward. In this mode, egress processing will not begin for a packet until after its last byte has arrived, and the packet is completely stored in the packet buffer.

Cut-through forwarding is an optimization to reduce latency through the switch. When a packet arrives on an input port and is destined for one or more output ports with the same link speed, or a slower link speed, cut-through forwarding will cause egress processing to begin for a packet starting shortly after its first few bytes have arrived to the Traffic Manager, without having to wait for the packet's last byte to arrive on the input port. There are control plane configuration options to enable cut-through forwarding, either globally for the device, or for packets arriving on, and/or destined for, particular ports.

The fields `disable_ucast_cutthru` and `enable_mcast_cutthru` enable overriding the Traffic Manager's configured behavior on a per-packet basis.

5.7 Traffic Manager

This is a brief description of the major functions of the Traffic Manager. They are given in the logical order that they are performed on packets received by the Traffic Manager. Such packets can come from three places:

- Normal packets from ingress. That is, packets from ingress that were not created by a mirroring operation (see Section 7.10).
- Mirrored packets from the ingress deparser
- Mirrored packets from the egress deparser

5.7.1 Mirror Session Lookup

Every mirrored packet received by the Traffic Manager has a mirror session id associated with it (see Section 7.10). The mirror session id value is selected per packet by the P4 program when it performs the mirror operation. This mirror session id is used as in index in a table of mirror sessions within the Traffic Manager. Each mirror session is configured by the control plane software with values for all fields of a destination (see Section 5.3). Thus, a single mirrored packet can be replicated in the

Packet Replication Engine, using the same multicast mechanisms that normal packets from ingress may use.

For normal packets from ingress, the P4 program sends the destination fields in the `ingress_intrinsic_metadata_for_tm_t` structure. This structure is an output from the ingress control.

5.7.2 Write Admission Control

The packet buffer's capacity is partitioned into several pools. Write admission control determines the pool into which a packet may be stored based upon a variety of inputs, e.g. the `ingress_cos` field value associated with the packet.

If write admission control decides not to store the packet in the chosen pool, e.g. because that pool is too full, then the rest of the packet's destination is ignored, and the packet is typically dropped.

If a packet from ingress cannot be stored in the chosen pool, and has the `deflect_on_drop` field assigned to 1, the Traffic Manager will attempt to store the packet in an alternate pool. If there is room in that alternate pool, the packet will be stored in the alternate pool and sent to an alternate destination. The alternate pool and destination for deflect-on-drop packets are configured by the control plane software.

5.7.3 Packet Replication Engine

The packet replication engine (PRE) makes copies of a packet by creating multiple packet descriptors. Each packet descriptor contains a pointer to a location where the packet is stored in the packet buffer. This saves space in the packet buffer, since the packet content is stored only once, with potentially many packet descriptors pointing to it.

Every packet arriving to the PRE already has a destination (as described in Section 5.4): regular packets get their destination from the intrinsic metadata, and mirrored packets get their destination from the a mirror session lookup (see Section 5.7.1).

PRE examines the following three components from the destination:

- `mcast_grp_a`, which has field validity (see Section 5.3)
- `mcast_grp_b`, which has field validity (see Section 5.3)
- `copy_to_cpu` flag

If the field `mcast_grp_a` is invalid, no copies of the packet descriptor are made for `mcast_grp_a`. If `mcast_grp_a` is valid, then copies of the packet descriptor are made based upon the configuration of multicast group `mcast_grp_a`, described in the next section.

Independently, `mcast_grp_b` is considered, and may create copies of the packet descriptor following the same rules as described for `mcast_grp_a`.

Every destination includes a `copy_to_cpu` flag. If `copy_to_cpu` is 1, a copy of the packet descriptor is made with `egress_rid` equal to 0, `egress_rid_first` equal to 1, and its `egress_port` is assigned a value configured into the Traffic Manager by the control plane software.

5.7.3.1 Multicast group configuration

The Intel Tofino PRE is designed to support protocol independent multicast replication. Each multicast group is logically divided into two levels. Level 1 represents a list of networks to which a packet will be replicated, and each of these can replicate a packet to a list of Level 2 destinations, which are physical ports or Link Aggregation Groups (LAGs).

Multicast groups are configured by the control plane. For each multicast group id, the control plane configures a list of Level 1 nodes (this list may be empty).

Each Level 1 node has one of these two types:

- Individual Level 1 node
- Equal Cost Multi-Path (ECMP) node, configured as a list of Individual level 1 nodes (must have at least one node, duplicates are not allowed)

Each Level 2 node has one of these two types:

- Individual port
- A LAG, configured as a list of individual ports (must have at least one port, duplicates are not allowed)

Each individual Level 1 node `n1` is configured with the following properties:

- `n1.RID` - A 16-bit replication id. The value 0 may be used, but if one avoids using 0, then the `egress_rid` field of a packet's egress intrinsic metadata (see Section 5.8.1) makes it easy to distinguish multicast replicated packets from other packets during egress processing.
- Level 2 nodes, consisting of both of the following lists.
    - `n1.dev_port_list` - A list of individual ports (may be empty, duplicates are not allowed)
    - `n1.lag_list` - A list of LAGs (may be empty, duplicates are not allowed)

Multicast groups contain a list of Individual Level 1 nodes (may be empty), and a list of ECMP nodes (may be empty). When a Level 1 node (either individual or ECMP) is associated with a multicast group, the following information must be provided as a part of that association:

- Level 1 Node ID – This ID is used for identifying the node in the control plane API and does not affect the data plane behavior.
- `L1_XID` - a 16-bit level 1 exclusion id
- `L1_XID_VALID` - a 1-bit field that is 1 if `L1_XID` is valid

5.7.3.2 Packet replication for one multicast group

When a packet is replicated for a multicast group, the PRE iterates through the multicast group's list of level 1 nodes, as described in the pseudocode below. In this pseudocode, names beginning with `pkt` such as `pkt.level1_mcast_hash` refer to fields in the packet's destination (see Section 5.4).

A brief summary is that the PRE iterates over all level 1 nodes configured for the multicast group, and for each level 1 node, the PRE iterates over all level 2 nodes configured for that level 1 node. If a level 1 node is type ECMP, or a level 2 node is type LAG, then a hash value from the packet's destination is used to select exactly one of the individual nodes in the ECMP/LAG node. Each node also has a "prune" condition, which if true causes the PRE to make no copies of the packet for that node.

The name `PRE.L2_exclusion_table` in the pseudocode is a PRE table configured by the control plane. For each Level 2 exclusion ID `X`, `PRE.L2_exclusion_table[X]` is a set of port numbers.

Note: Only values in the range [0, 287] are supported for `pkt.level2_exclusion_id`.

```
for (n1 in multicast group's list of level 1 nodes) {
    if (n1 is an ECMP node) {
        sel1 = select one of the Individual level 1 nodes in
               n1's list using the value pkt.level1_mcast_hash,
               with equal weight;
    } else {
        sel1 = n1;
    }
    prune1 = n1.L1_XID_VALID && (pkt.level1_exclusion_id == n1.L1_XID);
    if (prune1) {
        // Make no copies of the packet descriptor for n1
    } else {
        for (n2 in concatenate(n1.dev_port_list, n1.lag_list)) {
            if (n2 is a LAG) {
                sel2 = select one of the individual ports in the LAG's
                       list using the value pkt.level2_mcast_hash,
                       with equal weight;
            } else {
                sel2 = n2;
            }
        }
        prune2 = (pkt.rid == n1.RID) &&
                 (sel2.port in the set
```

```
                        PRE.L2_exclusion_table[pkt.level2_exclusion_id]);
        if (prune2) {
            // Make no copies of the packet descriptor for port n2.port
        } else {
            // Make one copy of the packet descriptor for port n2.port.
            // If it does not bypass egress processing, its egress
            // intrinsic metadata fields will be initialized like this:
            egress_port = sel2.port;
            egress_rid = sel1.RID;
            egress_rid_first = see text below;
        }
    }
}
```

The selection among multiple nodes in an ECMP node is always a deterministic stateless selection that is a function of only the list of Individual Level 1 nodes in the ECMP node, and `pkt.level1_multicast_hash`. The selection among multiple ports in a LAG may be configured to be a function of only the list of ports in the LAG and the `pkt.level2_multicast_hash`. (Note that there are other configuration options that take into account the current up/down status of ports, which will be described in a future version of this document.) This enables load balancing over multiple paths such that all packets with the same values of `pkt.level1_multicast_hash` and `pkt.level2_multicast_hash` will follow the same path through the network, during time intervals when no changes are being made to the multicast group configuration.

The field `egress_rid_first` is will be documented in a future revision of this document.

5.7.4 Queue Admission Control

By the time that a packet descriptor reaches the queue admission control step, all of them contain a queue id. Queue admission control considers each packet descriptor independently and decides whether to enqueue the packet descriptor on the queue, or to discard the packet descriptor.

Every packet arriving to the Traffic Manager has a `packet_color` field, which comes from the destination of the packet (see Section 5.4). The numeric encoding of the `packet_color` field is the same as the default green, yellow, and red encodings for meter colors (see Section 7.9.2).

Packets with color green are the highest importance. Packets with color yellow are lower importance than green. Packets with color red are the lowest importance. The ingress P4 code decides the value of the `packet_color` field for each packet, except for mirrored packets, where the value of `packet_color` comes from the configuration of the mirror session.

Every queue has three queue depth threshold values configured by control plane software, in units of 80-byte cells. The red threshold is the smallest. The yellow threshold is larger than the red threshold. The green threshold is the largest.

The table below describes the behavior of queue admission control when a packet descriptor arrives to a queue, based upon that queue's current depth, the queue's configured thresholds, and the `packet_color` of the packet.

Table 2: Queue Admission Control drop behavior

Current depth of target queue	Queue Congestion State	Description
Smaller than red threshold	0	No congestion. All arriving packet descriptors will be enqueued, regardless of `packet_color`.
Larger than red threshold, smaller than yellow threshold	1	Light congestion. Lowest priority red packet descriptors will be dropped. Yellow and green packet descriptors will be enqueued.
Larger than yellow threshold, smaller than green threshold	2	High congestion. Red and yellow packet descriptors will be dropped. Green packet descriptors will be enqueued.
Larger than green threshold	3	Extreme congestion. All arriving packet descriptors will be dropped, regardless of `packet_color`.

The Queue Congestion State number is used for the `enq_congest_stat` and `deq_congest_stat` intrinsic metadata fields. See Section 5.8.1.

5.7.5 Packet Queues

There are 32 queues per 100 Gigabit port. Every queue is destined for a single port and operates in first in first out manner. If a packet descriptor is enqueued on a queue, it will at some later time be dequeued by the packet scheduler.

5.7.6 Packet Scheduler

It is the job of the packet scheduler to decide which of the queues destined to an output port to dequeue a packet descriptor from next, and precisely when.

There are many options for configuring the packet scheduler. For example, one of the queues for an output port can be configured as strictly higher priority than all other queues for the same output port. Multiple queues for the same output port may be configured to use weighted fair queuing with relative weights configured for each queue. Maximum shaping rates in bits per second may be configured for queues. The full details will be documented in the future.

After dequeuing a packet descriptor, the packet scheduler fills in the egress intrinsic metadata fields for this copy of the packet and appends after it the contents of the packet read from the packet buffer. The space occupied by the packet in the packet buffer is deallocated only when the last packet descriptor that points to it has been dequeued.

5.8 Contents of packets received by egress parser

The layout of packets received at the egress parser is shown in Figure 3: Normal & Mirrored Layout of Packets Received at the Egress Parser. There are two formats of packets that can be received by the egress parser: normal packets and mirrored packets.

A normal packet is one that is the result of ingress processing, and the packet was unicast, multicast, copied via the `copy_to_cpu` flag, or deflected via the `deflect_on_drop` flag.

A mirrored packet is either an ingress mirror packet created in the ingress deparser, or an egress mirror packet created in the egress deparser. The details on mirroring will be explained in Section 7.10.

All packets begin with egress intrinsic metadata. The egress parser P4 code should extract the egress intrinsic metadata. After the intrinsic metadata, the rest of the packet's contents are as it was deparsed, either in the ingress or egress deparser.

Note that in TNA there is no data about a packet calculated during ingress processing that is automatically available for use during egress processing. For a metadata field calculated during ingress processing to be available in egress processing, the field must either be included in the packet received by the egress parser (and thus it must have been deparsed in the ingress deparser), or the field must be calculated in the egress processing code.

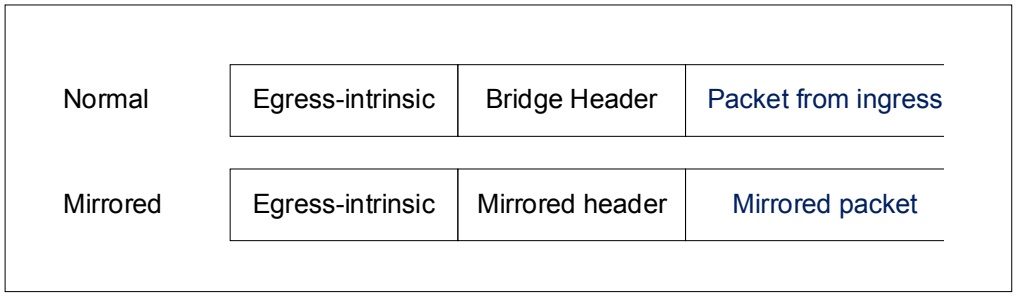
For this reason, it is common for there to be one or more user-defined headers added at the beginning of packets in the ingress deparser, to carry such fields. It is customary to call these bridge header(s). Bridge headers can be defined, manipulated, emitted in the ingress deparser, and parsed in the egress parser, just as any other headers. TNA does not distinguish them from other headers in any way.

Bridge headers are optional. There is no size limit on bridge headers. However, using a large bridge header can reduce the packet throughput. Intel Tofino is engineered to maintain full packet throughput if the length of the bridge headers plus the packet from ingress is at most 28 bytes longer than the original packet received by the ingress parser (not counting the ingress intrinsic metadata and port metadata).

Similarly, it is common for a P4 developer to wish to preserve some fields calculated during ingress processing with an ingress-to-egress mirrored packet and use those fields in egress processing. And it is common to preserve some fields calculated during egress processing with an egress-to-egress mirrored packet and use those fields when the packet does egress processing again. Such fields are preserved in headers that are customarily called mirror headers.

As for bridge headers, mirror headers are optional. Mirror headers may be at most 32 bytes long.

Figure 3: Normal & Mirrored Layout of Packets Received at the Egress Parser



Since bridge and mirror headers are optional, and their contents are user defined, it is up to the P4 developer to design these header formats in such a way that one can distinguish different kinds of packets arriving at the egress parser from each other. For example, a P4 developer could choose a design where the first 4 bits of all packets sent to the Traffic Manager contain a packet type value that identifies the format of the packet's first header.

TNA does provide a few intrinsic metadata fields to distinguish between some kinds of packets. See the description of the fields `deflection_flag`, `egress_rid`, and `egress_rid_first` in the next section.

5.8.1 Egress Intrinsic Metadata

The Traffic Manager prepends metadata to packets before they are sent to the egress parser. The metadata is represented as a header with type `egress_intrinsic_metadata_t`. It is recommended that the egress parser extracts this metadata into the provided parameter of the same type, so that this data will be available for use in the egress control. (Note that the padding fields have been omitted from the definition of `egress_intrinsic_metadata_t` for brevity.)

```
header egress_intrinsic_metadata_t {
    PortId_t  egress_port;
    bit<19>   enq_qdepth;
    bit<2>    enq_congest_stat;
    bit<18>   enq_tstamp;
    bit<19>   deq_qdepth;
    bit<2>    deq_congest_stat;
    bit<8>    app_pool_congest_stat;
    bit<16>   egress_rid;
    bit<1>    egress_rid_first;
    QueueId_t egress_qid;
    bit<3>    egress_cos;
    bit<1>    deflection_flag;
    bit<16>   pkt_length;
}
```

All egress intrinsic metadata is read-only in the egress control P4 code.

The `deflection_flag` field is 1 if this packet is the result of the deflect-on-drop behavior in the Traffic Manager's Write Admission Control block described in Section 5.7.2. Otherwise `deflection_flag` is 0.

The `egress_port` field specifies the outgoing port of the packet.

For unicast packets, `egress_rid` is 0 and `egress_rid_first` is 0. For copy-to-CPU packets, `egress_rid` is 0 and `egress_rid_first` is 1. See Section 5.7.3 for how the fields `egress_rid` and `egress_rid_first` are assigned values for multicast replicated packets. If the suggestion to avoid using RID 0 for multicast replicated packets is followed, then these cases can be easily distinguished during egress processing using these two fields.

The `enq_qdepth` and `deq_qdepth` fields give the depth of the queue that the packet passed through at the time the packet was enqueued, and dequeued, respectively. Queue depths are in units of cells, where a cell contains 80 bytes.

The `enq_congest_stat` and `deq_congest_stat` fields provide congestion level information about the queue that the packet was enqueued on when it was in the Traffic Manager. The numeric encodings of the values of these fields is shown as the Queue Congestion State value in the table of Section 5.7.3.2. `enq_congest_stat` contains the value of Queue Congestion State determined from the depth of the queue when the packet was enqueued. `deq_congest_stat` contains the value of Queue Congestion State determined from the depth of the queue when the packet was dequeued.

The `enq_tstamp` field gives the time when the packet was enqueued, in nanoseconds.

The `egress_qid` is determined from the hardware queue id that the packet went through, via a hardware remapping table.

The control plane software configures a 3-bit egress Class Of Service value for each hardware queue. This value is put into the field `egress_cos`.

The field `app_pool_congest_stat` contains 8 bits, divided into four 2-bit sub-fields. One sub-field is in bit positions [7:6], another in bit positions [5:4], the third in bit positions [3:2], and the last in bit positions [1:0]. Each sub-field is configured to correspond with one application pool in the Traffic Manager's packet buffer. See Section 5.7.2.

Each Traffic Manager pool is configured with three thresholds, in units of 80-byte cells. These three thresholds are similar to the queue thresholds discussed in Section 5.7.3.2, and they are also called a red threshold, a yellow threshold, and a green threshold, each larger than the previous one.

At the time a packet is dequeued, the four configured application pools are checked for their total occupancy, and each is compared against the three thresholds. A 2-bit numeric value that has the same encoding and meaning as the Queue Congestion Thresholds in Section 5.7.3.2 is calculated for the pool, and put into the configured sub-field of `app_pool_congest_stat`.

5.9 Additional intrinsic metadata from the egress parser

The egress parser generates additional intrinsic metadata, which becomes an input to the egress control. It has the type `egress_intrinsic_metadata_from_parser_t`. Its fields are shown below.

```
struct egress_intrinsic_metadata_from_parser_t {
    bit<48> global_tstamp;
    bit<32> global_ver;
    bit<16> parser_err;
}
```

The field `global_tstamp` contains the time in nanoseconds when the packet entered the egress parser. The field `parser_err` contains a code indicating any error that occurred during egress parsing. See Section 5.2.1 for a list of parser errors and their causes. The field `global_ver` is reserved.

5.10 Intrinsic metadata for the egress deparser

The purpose of the structure below is to control the behavior of the egress deparser. The struct is an input and output of the egress control, and an input to the egress deparser, where its fields are used to control operations mentioned below.

```
struct egress_intrinsic_metadata_for_deparser_t {
    bit<3>     drop_ctl;
    MirrorType mirror_type;
    bit<1>     coalesce_flush;
    bit<7>     coalesce_length;
}
```

As mentioned in Section 5.3, field `mirror_type` has field validity and is initially invalid. TNA initializes `drop_ctl` to 0 for each packet.

See Section 7.10 for details on `mirror_type`, and Section 5.12 for details on `drop_ctl`. The behavior of the other fields is an advanced topic to be documented in the future. It is recommended to leave them with the values that TNA initializes them to.

5.11 Intrinsic metadata for the output port

The following struct is an input and output of the egress control, and an input to the output ports.

```
struct egress_intrinsic_metadata_for_output_port_t {
    bit<1>     capture_tstamp_on_tx;
    bit<1>     update_delay_on_tx;
    bit<1>     force_tx_error;
}
```

The behavior of these fields is an advanced topic to be documented in the future. It is recommended to leave them with the values that TNA initializes them to.

5.12 Behavior of packets after egress processing

The behavior of a packet after egress processing is complete is summarized in the following pseudocode.

```
if ((egress deparser invoked a mirror operation) && (drop_ctl[2:2] == 0))
{
    send egress-to-egress mirror packet to the Traffic Manager;
}
if (drop_ctl[0:0] == 0) {
    send the normal packet output by the deparser to the output port
        number egress_port;
}
```

Note that it is possible for both a mirror packet to be sent to the Traffic Manager, and the normal packet to go to the output port.

See Section 7.10 for how to invoke a mirror operation. A mirror packet's destination is determined by looking up its mirror session id in a table in the Traffic Manager (see Section 5.7.1).

6 Table Properties

Table properties specify the semantics of a P4 table. Some of the table properties are standard (defined by the $P4_{16}$ language specification), and the rest are TNA-specific extensions. The TNA-specific properties extend a P4 table beyond the match-action semantics with additional operations such as action selection, metering, and counting. For example, a P4 table can be associated with an `ActionProfile` to optimize for the case where the table has a large number of entries, but the actions associated with those entries are expected to range over a small number of distinct values. Further, a P4 table can be associated with one of the 'direct' resources which is executed when a table match occurs. While it is possible to define multiple properties for a single table, not all combinations of table properties are supported.

The table below lists the supported table properties in TNA.

Table 3: Summary of TNA Table Properties

Property name	Standard or Extension	Supported type
`key`	standard	see P4_16 specification
`actions`	standard	see P4_16 specification
`entries`	standard	see P4_16 specification
`default_action`	standard	see P4_16 specification
`size`	standard	see P4_16 specification
`implementation`	extension	ActionProfile, ActionSelector
`meters`	extension	DirectMeter
`counters`	extension	DirectCounter
`registers`	extension	DirectRegister
`idle_timeout`	extension	Boolean

All table properties must be assigned a compile-time known value of the supported type. This example shows using a previously instantiated extern as the value of a table property.

```
ActionProfile (1024) ap1;
table t1 {
    actions = { <code omitted> }
    implementation = ap1;
}
```

This is the recommended way to assign extern instances to table properties, because it provides an explicit instance name for use by the generated control plane API.

6.1 Property Compatibility

All table properties listed in Table 3: Summary of TNA Table Properties are mutually compatible with each other, except that the same table may not have both meters and registers.

6.2 Match Types

TNA supports two additional `match_kind` types on top of the standard P4$_{16}$ `match_kind` types.

```
match_kind {
    range,      // Used to represent min..max intervals
    selector    // Used for implementing ActionSelector
}
```

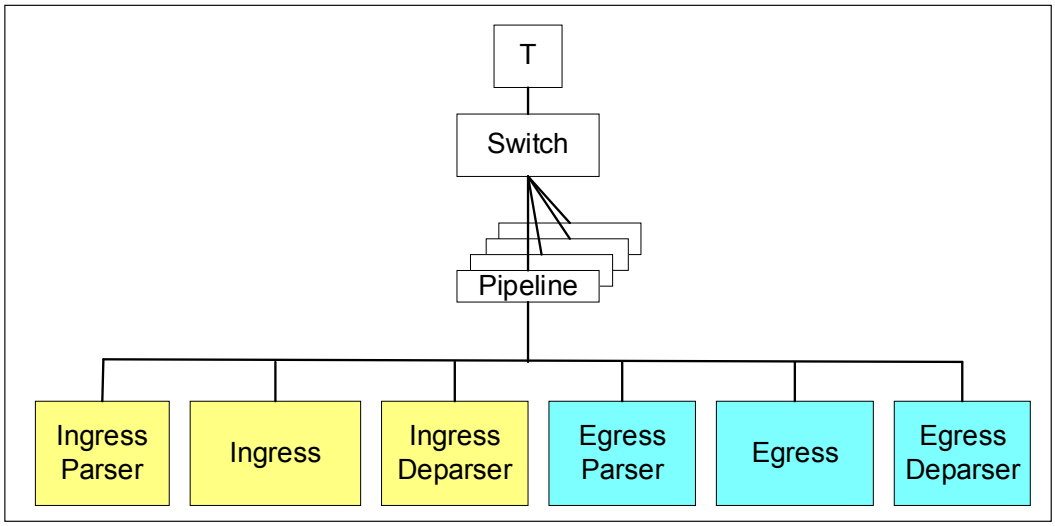
A table key with `range` match type can match on values within a range `[start, end]`. The range is inclusive at the `start` and `end`, meaning any search key value `x` will match if `(start <= x) && (x <= end)`.

A table key field with `selector` match type may only be used in a table implemented via an `ActionSelector` (see Section 7.4 Action Selector). Such a table contains key fields with other match types (e.g., `exact` , `ternary` , `range`), but must also have at least one key field with the `selector` match type. The regular match types are used in the match operation. The field(s) with match type `selector` are used to select one of the multiple actions within one group of the action selector. If a key is used both as a regular match type and a selector match type, it must be listed twice in the list of table key fields.

7 Externs

An extern encapsulates a hardware feature into an object with methods. The object can be instantiated at compile time and the methods can be invoked at packet processing time. TNA provides the externs in Table 4: Summary of TNA Externs and Where They May be Instantiated to represent various Intel Tofino hardware features. (Note: There are several TNA externs not yet described in this document. They will be added to this table when they are documented.) Each extern has its own restrictions on where the instantiation and invocation can happen.

Figure 4: TNA Instantiation Tree



The compiler rejects programs that instantiate externs, or attempt to call their methods, from anywhere other than the places mentioned in Table 4: Summary of TNA Externs and Where They May be Instantiated. For example, a Counter extern can only be instantiated in either the ingress control or egress control. As described in Section 4.3, any sub-parser or sub-control called by one of the six P4 programmable blocks is considered to be part of it.

TNA does not support using the same extern instance from both Ingress and Egress, nor from more than one of the parsers or controls defined in TNA.

Table 4: Summary of TNA Externs and Where They May be Instantiated

Extern Type	Where the Extern May be Instantiated
`ActionProfile`	Ingress Control, Egress Control
`ActionSelector`	Ingress Control, Egress Control
`Checksum`	Ingress Parser, Egress Parser Ingress Deparser, Egress Deparser
`Counter`	Ingress Control, Egress Control
`CRCPolynomial`	Ingress Control, Egress Control

Extern Type	Where the Extern May be Instantiated
`Digest`	Ingress Deparser
`DirectCounter`	Ingress Control, Egress Control
`DirectMeter`	Ingress Control, Egress Control
`DirectRegister`	Ingress Control, Egress Control
`DirectRegisterAction`	Ingress Control, Egress Control
`Hash`	Ingress Control, Egress Control
`MathUnit`	Ingress Control, Egress Control
`Meter`	Ingress Control, Egress Control
`Mirror`	Ingress Deparser, Egress Deparser
`ParserCounter`	Ingress Parser, Egress Parser
`Random`	Ingress Control, Egress Control
`Register`	Ingress Control, Egress Control
`RegisterAction`	Ingress Control, Egress Control
`Resubmit`	Ingress Deparser

7.1 Direct and indirect externs

Several of the externs have two different variants, called direct and indirect. Occasionally indirect externs are called indexed, but this terminology is less common.

The externs that have these variants are counters, meters, and registers. In all cases the direct variant of the extern has a name beginning with `Direct`, e.g. `DirectCounter`, and the indirect variants of the extern has a name with no such prefix, e.g. `Counter`.

7.1.1 Direct externs

All direct externs are associated with a single P4 table. This table is called the extern's owner. A direct extern effectively extends every table entry of its owner table, adding extra fields to it that are not part of the key or action data. The contents, meaning, and methods of updating these extra fields is determined by the extern. For example, a `DirectCounter` extern instance `m1` with owner table `t1` extends each of `t1`'s table entries with packet and/or byte count fields.

Each direct extern has at least one method that can update its fields of a table entry, e.g. `count()` for a `DirectCounter`. The fields belonging to a direct extern can only be accessed from the P4 program via these methods, and these methods may only be called from an action of the direct extern's owner table. Every time the owner table is applied, and an entry is matched, the action associated with the matching table entry is executed. If an action calls a direct extern method, that method can only access the extra fields which are part of that matching table entry.

7.1.2 Indirect externs

All indirect externs are instantiated with a compile-time known size N, and are effectively an array of N entries, or elements. The entries are numbered from 0 up to N-1. Indirect externs are not associated with a table, the way that direct externs are.

Each indirect extern has at least one method that can update one of its entries, e.g. `execute(index)` for a `DirectMeter`. An entry of an indirect extern can only be accessed from the P4 program via these methods. These methods may be called from within actions of a table, or they may also be called from within the `apply` block of a P4 `control`. Every call to such a method must specify the index of the entry to be accessed.

7.2 Packet lengths used by externs

Several externs use the length of the current packet implicitly, i.e. not as an explicit parameter to the method calls used to access these externs. For example, all updates to byte counters, and updates to meters based on byte rates, both use the length of the packet currently being processed, in bytes.

In the ingress control, the packet length used by these externs includes all bytes from the first byte after the port metadata or resubmit header, up to the end of the Frame Check Sequence at the end of the Ethernet frame. For packets received from an external Ethernet interface, the packet length is thus always at least 64 bytes, the minimum Ethernet frame length.

In the egress control, the packet length used by these externs includes all bytes emitted by the ingress deparser (or the egress deparser in the egress-to-egress mirroring case) when the packet was sent to the Traffic Manager, plus the payload, including the Frame Check Sequence. It does not include the egress intrinsic metadata.

Note that the ingress packet length used does not include the effects of any headers added or removed by your P4 program during the current execution of the ingress control. If the packet was recirculated, it does include the effects of any headers added or removed before the current execution of the ingress control.

The egress length used does include headers added or removed before it was last sent to the Traffic Manager but does not include the effects of any headers added or removed by your P4 program during the current execution of the egress control.

7.3 Action Profile

Action profiles provide a mechanism to populate table entries with action specifications that have been defined outside the table entry specification. An action profile extern can be instantiated as a resource in the P4 program. A table that uses this action profile must specify its implementation attribute as the action profile instance.

```
extern ActionProfile {
    // Construct an action profile of 'size' entries.
    ActionProfile(bit<32> size);
}
```

Action profiles are used as table implementation attributes.

Figure 5: Action Profiles in TNA

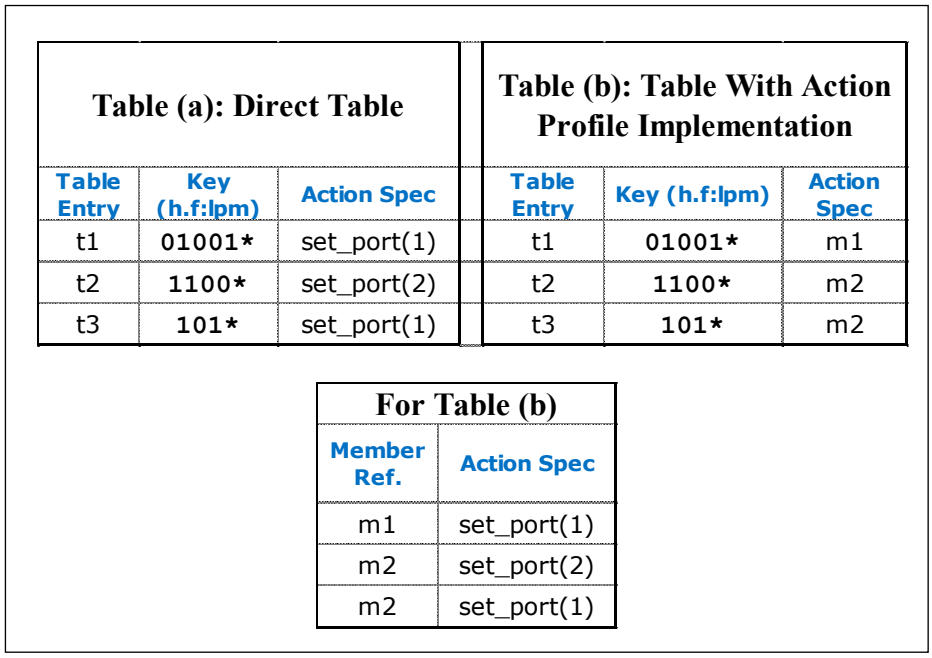


Table (a): Direct Table

Table Entry	Key (h.f:lpm)	Action Spec
t1	**01001***	set_port(1)
t2	**1100***	set_port(2)
t3	**101***	set_port(1)

Table (b): Table With Action Profile Implementation

Table Entry	Key (h.f:lpm)	Action Spec
t1	**01001***	m1
t2	**1100***	m2
t3	**101***	m2

For Table (b)

Member Ref.	Action Spec
m1	set_port(1)
m2	set_port(2)
m2	set_port(1)

Figure 5: Action Profiles in TNA, above, contrasts a direct table with a table that has an action profile implementation. A direct table, as seen in Figure 5: Action Profiles in TNA part (a) contains the action specification in each table entry. In this example, the table has a match key consisting of an LPM on header field `h.f`. The action is to set the port. As we can see, entries `t1` and `t3` have the same action, i.e. to set the port to 1. Action profiles enable sharing an action across multiple entries by using a separate table as shown in Figure 5: Action Profiles in TNA part (b).

A table with an action profile implementation has entries that point to a member reference instead of directly defining an action specification. A mapping from member references to action specifications is maintained in a separate table that is part of the action profile instance defined in the table implementation attribute. When a table with an action profile implementation is applied, the member reference is resolved and the corresponding action specification is applied to the packet.

Action profile members may only specify action types defined in the `actions` attribute of the implemented table. An action profile instance may be shared across multiple tables only if all such tables define the same set of actions in their actions attribute. The default action for such tables is implicitly set to `NoAction`.

The control plane can add, modify or delete member entries for a given action profile instance. The controller-assigned member reference must be unique in the scope of the action profile instance. An action profile instance may hold at most size entries as defined in the constructor parameter. Table entries must specify the action using the controller-assigned reference for the desired member entry. Directly specifying the action as part of the table entry is not allowed for tables with an action profile implementation.

7.3.1 Example

The P4 control block `MyIngress` in the example below instantiates an action profile `ap1` that contains up to 1000 member entries. Table `ipv4_dest` uses instance `ap1` by assigning it to the `implementation` attribute. The control plane can add member entries to `ap1`, where each member can specify either a `send` or `drop` action (plus the action data). When a member is successfully added, the switch software returns a member id. When adding or modifying entries in table `ipv4_dest`, the control plane software must specify the action using this member id.

```
control MyIngress(
    inout my_ingress_headers_t  ig_hdr,
    inout my_ingress_metadata_t ig_md,
    in    ingress_intrinsic_metadata_t               ig_intr_md,
    in    ingress_intrinsic_metadata_from_parser_t   ig_prsr_md,
    inout ingress_intrinsic_metadata_for_deparser_t ig_dprsr_md,
    inout ingress_intrinsic_metadata_for_tm_t       ig_tm_md)
{
    ActionProfile(1000) ap1;

    action send (PortId_t port, mac_addr_t new_dest_mac) {
        ig_tm_md.ucast_egress_port = port;
        ig_hdr.ethernet.dst_addr = new_dest_mac;
    }
    action drop () {
        ig_dprsr_md.drop_ctl = 1;
    }
    table ipv4_dest {
        key = { ig_hdr.ipv4.dst_addr: exact; }
        actions = { send; drop; }
        const default_action = drop;
        implementation = ap1;
        size = 32768;
    }
    apply {
        if (ig_hdr.ipv4.isValid()) {
            ipv4_dest.apply();
        }
    }
}
```

7.4 Action Selector

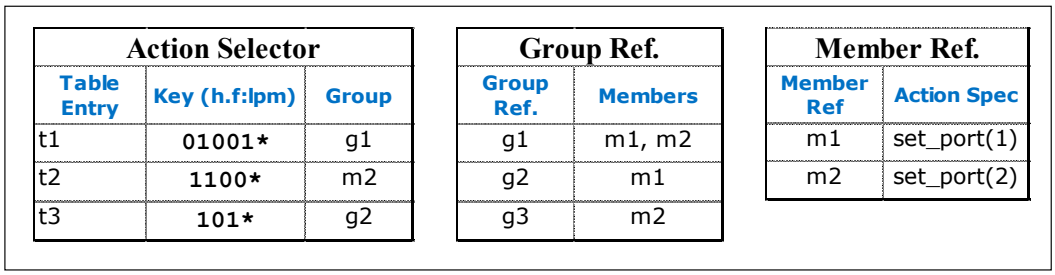
Action selectors implement yet another mechanism to associate table entries with action specifications that have been defined outside the table entry. They are more powerful than action profiles because they also provide the ability to dynamically select the action specification to apply upon matching a table entry.

```
extern ActionSelector {
    // Construct an action selector of 'size' entries
    ActionSelector(bit<32> size, Hash<_> hash, SelectorMode_t mode);

    // Stateful action selector.
    ActionSelector(bit<32> size, Hash<_> hash, SelectorMode_t mode,
                   Register<bit<1>, _> reg);

    /// Construct a selection table for a given ActionProfile.
    ActionSelector(ActionProfile action_profile,
                   Hash<_> hash,
                   SelectorMode_t mode,
                   bit<32> max_group_size,
                   bit<32> num_groups);

    /// Stateful action selector.
    ActionSelector(ActionProfile action_profile,
                   Hash<_> hash,
                   SelectorMode_t mode,
                   Register<bit<1>, _> reg,
                   bit<32> max_group_size,
                   bit<32> num_groups);
}
```

Action selectors are used as table implementation attributes.

An action selector extern can be instantiated as a resource in the P4 program, like action profiles. To specify that a table use an action selector instance named `as1`, for example, assign it to its implementation table property, `implementation = as1`.

Figure 6: Action Selector in TNA



Action Selector

Table Entry	Key (h.f:lpm)	Group
t1	01001*	g1
t2	1100*	m2
t3	101*	g2

Group Ref.

Group Ref.	Members
g1	m1, m2
g2	m1
g3	m2

Member Ref.

Member Ref	Action Spec
m1	set_port(1)
m2	set_port(2)

Figure 6: Action Selector in TNA, above, illustrates a table that has an action selector implementation. In this example, the table has a match key using a longest-prefix match on header field `h.f`. A second match type selector is used to define the fields that are used to look up the action specification from the selector at runtime.

A table with an action selector implementation consists of table entries that point to either an action profile member reference or an action profile group reference. An action selector instance can be logically visualized as two tables as shown in Figure 6: Action Selector in TNA. The first table contains a mapping from group references to a

set of member references. The second table contains a mapping from member references to action specifications.

In the figure, table entry `t1` points at group `g1`, and group `g1` contains member `m1` and `m2`. Table entry `t2`, however, points directly at member `m2`, without referring to any groups. This flexibility of allowing some table entries to point at groups, and others directly at members, can be useful when implementing routing tables. You may also notice that there is a group `g3` in the group table, but there are no table entries that point at this group. This is a common condition, at least temporarily, during routing table updates. New groups must be created before table entries can point to them.

When a packet matches a table entry at runtime, the controller-assigned reference of the action profile member or group is read. If the entry points to a member then the corresponding action specification is applied to the packet. However, if the entry points to a group, a dynamic selection algorithm is used to select a member from the group, and the action specification corresponding to that member is applied. The dynamic selection algorithm is specified as the 'hash' parameter when instantiating the action selector.

Action selector members may only specify action types defined in the actions attribute of the implemented table. All actions in a group must have the same action name. The action parameters for actions in the same group may differ, and the action names used by different groups in a selector may be different. An action selector instance may be shared across multiple tables only if all such tables define the same set of actions in their actions attribute. Furthermore, the selector match fields for such tables must be identical and must be specified in the same order across all tables sharing the selector. Tables with an action selector implementation cannot define a default action. The default action for such tables is implicitly set to `NoAction`.

The dynamic selection algorithm requires a field list as an input for generating the index to a member entry in a group. This field list is created by using the match type selector when defining the table match key. The match fields of type selector are composed into a field list in the order they are specified. The composed field list is passed as an input to the action selector implementation. It is illegal to define a selector type match field if the table does not have an action selector implementation.

The control plane can add, modify or delete member and group entries for a given action selector instance. An action selector instance may hold at most size member entries as defined in the constructor parameter. The number of groups may be at most the size of the table that is implemented by the selector. Table entries must specify the action using a reference to the desired member or group entry. Directly specifying the action as part of the table entry is not allowed for tables with an action selector implementation.

7.4.1 Action Selector Example

The P4 control block Ctrl in the example below instantiates an action profile `ipv4_ecmp_ap` that can contain at most 128 member entries. Then it instantiates a `Hash` extern with a CRC-16 algorithm with output width of 8 bits to select `hash_fn` to calculate a hash to select one of up to 256 different ECMP paths for a single IPv4 prefix in the table `ipv4_lpm`. The action selector `ipv4_ecmp` uses the action profile `ipv4_ecmp_ap`.

Table `ipv4_lpm` uses this instance by specifying the `implementation` table property as shown. The control plane can add entries with action names and parameters to the action profile `ipv4_ecmp_ap`. Each member can specify either a `set_port` or `drop` action.

Then the control plane can add groups to the action selector `ipv4_ecmp`, and members to those groups, where each member is a reference to an entry in `ipv4_ecmp_ap`. When programming the table entries in table `ipv4_lpm`, the control plane *does not* include the fields with `match_kind selector` in the key. The `selector` fields are instead given as input to the `hash_fn` extern. In the example below, the fields {`hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.ipv4.protocol`} are passed as input to the `CRC16` hash algorithm used for member selection by the action selector.

```
control Ctrl(inout H hdr, inout M meta) {

    action set_port(PortId_t port) {
        ig_tm_md.ucast_egress_port = port;
    }
    action drop() {
        ig_dprsr_md.drop_ctl = 1;
    }

    ActionProfile(128) ipv4_ecmp_ap;
    Hash<bit<8>>(HashAlgorithm_t.CRC16) hash_fn;
    ActionSelector(action_profile = ipv4_ecmp_ap,
                   hash = hash_fn,
                   mode = SelectorMode_t.FAIR,
                   max_group_size = 256,
                   num_groups = 2048) ipv4_ecmp;

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
            hdr.ipv4.srcAddr: selector;
            hdr.ipv4.dstAddr: selector;
            hdr.ipv4.protocol: selector;
        }
        actions = { set_port; drop; }
        implementation = ipv4_ecmp;
        default_action = drop;
        size = 16384;
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        } else {
            drop();
        }
    }
}
```

7.5 Checksum

Intel Tofino checksum engines can verify the header-only checksums such as IPv4 header checksum at the parser and re-compute the checksum using header or metadata fields at the deparser.

For checksums that include the payload such as TCP checksum which is computed over the entire packet, Intel Tofino assumes that the original checksum is correct and incrementally updates the checksum value as described in RFC 1624. For an incremental checksum update, the P4 program must subtract any fields that are modified in the program in the parser (including the checksum field itself), and then add the new values to the checksum update at the deparser.

Intel Tofino checksum engines support only 16-bit ones' complement checksums.

```
extern Checksum {
    // Constructor.
    Checksum();

    // Add data to checksum.
    // @param data : List of fields to be added to checksum calculation.
    //     The data must be byte aligned.
    void add<T>(in T data);

    // Subtract data from existing checksum.
    // @param data : List of fields to be subtracted from the checksum.
    //     The data must be byte aligned.
    void subtract<T>(in T data);

    // Verify whether the complemented sum is zero, i.e. the checksum is
    // valid.
    // @return : Boolean flag indicating whether the checksum is valid or
    //     not.
    bool verify();

    // Subtract all header fields after the current state and
    // return the calculated checksum value.
    // Marks the end position for residual checksum header.
    // All header fields extracted after will be automatically
    // subtracted.
    // @param residual: The calculated checksum value for added fields.
    void subtract_all_and_deposit<T>(out T residual);

    // Get the calculated checksum value.
    // @return : The calculated checksum value for added fields.
    bit<16> get();

    // Calculate the checksum for a  given list of fields.
    // @param data : List of fields contributing to the checksum value.
    // @param zeros_as_ones : encode all-zeros value as all-ones.
    bit<16> update<T>(in T data, @optional in bool zeros_as_ones);
}
```

7.5.1 Checksum Examples

The partial program below demonstrates one way to use the Checksum extern to verify whether the checksum field in a parsed IPv4 header is correct and set the flag if it is wrong. P4 programs may choose to handle packets with parser errors in other ways than shown in this example – it is up to the P4 program author to choose and write the desired behavior. Note that the Checksum `verify` method will not terminate parsing, if the header checksum is wrong. However, the Boolean flag can be passed to the ingress control block to be acted on if checksum is incorrect.

```
parser IngressParser(
    packet_in pkt,
    out header_t hdr,
    out metadata_t md,
    out ingress_intrinsic_metadata_t ig_intr_md)
{
    Checksum() ipv4_csum;

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        // This is equivalent to passing all the IPv4 fields
        ipv4_csum.add(hdr.ipv4);
        md.checksum_err = ipv4_csum.verify();
        . . .
    }
}
```

The partial program below demonstrates one way to use the Checksum extern to calculate and then update the IPv4 header checksum. In this example, the checksum is calculated in the ingress deparser block, with the assumption that all IPv4 modification is performed at ingress. In general, it is up to the P4 program author to make sure that all relevant field modification shall precede the checksum update location (ingress or egress).

```
control IngressDeparser(
    packet_out pkt,
    inout header_t   hdr,
    in    metadata_t md,
    in    ingress_intrinsic_metadata_for_deparser_t md_for_dprs)
{
    Checksum() ipv4_csum;
    apply {
        if (hdr.ipv4.isValid()) {
            hdr.ipv4.hdrChecksum = ipv4_csum.update({
                hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
                hdr.ipv4.total_len,
                hdr.ipv4.identification,
                hdr.ipv4.flags, hdr.ipv4.frag_offset,
                hdr.ipv4.ttl, hdr.ipv4.protocol,
                /*  skip hdr.ipv4.hdr_checksum, */
                hdr.ipv4.srcAddr,
                hdr.ipv4.dstAddr});
        }
        pkt.emit(hdr);
    }
```

```
}
```

The example shows how to use the Checksum extern to compute an incremental checksum for the TCP header for a common use case, like NAT, where the only fields that are modified in the header are src/dst IP address and src/dst TCP port.

```
parser IngressParser(
    packet_in pkt,
    out header_t   hdr,
    out metadata_t md,
    out ingress_intrinsic_metadata_t ig_intr_md)
{
    Checksum() tcp_csum;
    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        tcp_csum.subtract({hdr.ipv4.src_addr, hdr.ipv4.dst_addr});
        transition select(hdr.ipv4.protocol) {
                  6 : parse_tcp;
            default : accept;
        }
    }
    state parse_tcp {
        pkt.extract(hdr.tcp);
        tcp_csum.subtract({hdr.tcp.checksum});
        tcp_csum.subtract({hdr.tcp.src_port, hdr.tcp.dst_port});
        tcp_csum.subtract_all_and_deposit(md.checksum);
        transition accept;
    }
}

control IngressDeparser(
    packet_out pkt,
    inout header_t   hdr,
    in    metadata_t md,
    in    ingress_intrinsic_metadata_for_deparser_t md_for_dprs)
{
    Checksum() tcp_csum;
    apply {
        hdr.tcp.hdr_checksum = tcp_csum.update({
            hdr.ipv4.src_addr,
            hdr.ipv4.dst_addr,
            hdr.tcp.src_port,
            hdr.tcp.dst_port,
            md.checksum});
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
    }
}
```

The checksum update can be done in the Ingress Deparser and/or the Egress Deparser. If the residual metadata `md.checksum` is assigned in the Ingress Parser and is needed in the Egress Deparser, then the metadata needs to be sent from ingress to egress using bridge metadata. If such incremental checksum update happens for a mirrored packet, the P4 programmer may need to add the metadata as a part of the mirror header. More details of mirroring are given in Section 7.10.

Certain checksum fields need to be aligned at either the high or low byte in the 2-byte (16-bit) alignment in order to produce correct calculation, e.g. IPv4.protocol needs to be aligned at the low byte of the 2-byte alignment (see the definition of the TCP pseudo-header for checksum calculation in RFC 793, and the UDP pseudo-header in RFC 768). In such cases, the P4 programmer needs to manually insert padding in the "add"/"subtract" statements in the parser, and the "update" statement in the deparser (see example below). The compiler will infer the 2-byte alignment from these statements rather than the header format.

```
parser IngressParser(
    packet_in pkt,
    out header_t hdr,
    out metadata_t md,
    out ingress_intrinsic_metadata_t ig_intr_md)
{
    Checksum() tcp_csum;
    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        tcp_csum.subtract({hdr.ipv4.src_addr,
                           hdr.ipv4.dst_addr,
                           8w0,       // zero byte for TCP pseudo-header
                           hdr.ipv4.protocol,
                           hdr.ipv4.total_len});
       <code omitted>
}

control IngressDeparser(
    packet_out pkt,
    inout header_t   hdr,
    in    metadata_t md,
    in    ingress_intrinsic_metadata_for_deparser_t md_for_dprs)
{
    Checksum() tcp_csum;
    apply {
        if (ipv4.tcp.isValid() && hdr.tcp.isValid()) {
            hdr.tcp.hdr_checksum = tcp_csum.update({
                hdr.ipv4.src_addr,
                hdr.ipv4.dst_addr,
                8w0,               // zero byte for TCP pseudo-header
                hdr.ipv4.protocol,
                hdr.ipv4.total_len,
                hdr.tcp.src_port,
                hdr.tcp.dst_port,
                hdr.tcp.seq_no,
                hdr.tcp.ack_no,
                hdr.tcp.data_offset,
                hdr.tcp.res,
                hdr.tcp.flags,
                hdr.tcp.window,
                hdr.tcp.urgent_ptr
                md.checksum});
        }
        // other ingress deparser code goes here
    }
}
```

7.6 Counters

Counters are a mechanism for keeping packet count and/or byte count statistics. The counter values stored within counter externs can only be read by the control plane software, while the P4 program can only increment the values. If one requires counters that can be read from the P4 program, Register externs can be used instead (see Section 7.13).

TNA supports both direct and indirect variants of counters. See Section 7.1 for a discussion of the differences between these variants.

7.6.1 Counter Types

Every `Counter` and `DirectCounter` instance has entries that contain either:

- A packet count
- A byte count
- Both a packet count and a byte count

All constructor methods for creating instances of a counter extern take a parameter of type `CounterType_t` to specify this choice.

```
enum CounterType_t {
    PACKETS,
    BYTES,
    PACKETS_AND_BYTES
}
```

The byte counts are always incremented by the length of the packet currently being processed, as defined in Section 7.2, minus the value of the `adjust_byte_count` parameter, if such a parameter is given.

7.6.2 Indirect Counter

The code below can be found in the TNA include file. It defines the types of parameters and return values for the constructor and `count` methods for an indirect counter.

```
// Indirect counter with `size' independent counter entries, where
// every entry has a data plane size specified by type W.
extern Counter<W, I> {
    // Constructor
    // @type_param W : width of the counter entry.
    // @type_param I : width of the counter index.
    // @param type : counter type. Packet and byte counters are
    //     supported.
    Counter(bit<32> size, CounterType_t type);

    // Increment the counter.
    // @param index : index of the counter to be incremented.
    // @param adjust_byte_count : optional parameter indicating value
    //     to be subtracted from counter value.
    void count(in I index, @optional in bit<32> adjust_byte_count);
```

```
}
```

7.6.3 Direct Counter

The declarations for the constructor and count method of a `DirectCounter` extern are shown here:

```
extern DirectCounter<W> {
    DirectCounter(CounterType_t type);
    void count(@optional in bit<32> adjust_byte_count);
}
```

A direct counter instance must be assigned as the value of the `counters` table property for exactly one table. That table is the direct counter's owner. It is an error to call the `count` method for a direct counter instance anywhere except inside an action of its owner table. TNA requires that every action of the direct counter's owner table must call the direct counter's `count` method exactly once.

The counter value updated by an invocation of count is always the one associated with the table entry that matched.

A `DirectCounter` instance must have a counter value associated with its owner table that is updated when there is a default action assigned to the table, and a search of the table results in a miss. If there is no default action assigned to the table, then there need not be any counter updated when a search of the table results in a miss.

By "a default action is assigned to a table", we mean that either the table has a `default_action` table property with an action assigned to it in the P4 program, or the control plane has made an explicit call to assign the table a default action. If neither of these is true, then there is no default action assigned to the table.

7.7 Digest

This section describes digest generation, a mechanism to send a message from the data plane to the control plane.

The other main technique to send messages from data plane to control plane is sending a packet to the CPU Ethernet port or CPU PCIe port. Sending packets to these ports typically sends most of the original packet headers, and perhaps also the payload, each as a separate message to be received and processed by the control plane.

Using digests provides several advantages over sending packets to CPU ports:

- Digest messages are smaller than even the minimum sized Ethernet packets.
- Data can be automatically and flexibly packed into digest messages, without the need to define header formats that can be placed into an Ethernet packet.
- TNA can aggregate multiple digest messages into larger batch messages in the hardware, reducing the rate of receiving these messages in the control plane software.

- Intel Tofino hardware has automatic techniques to eliminate duplicate digest messages, i.e. digest messages with the same field values as recently generated digest messages. This further reduces the rate of messages that the control plane software must process, in the common use cases where the control plane software only needs to be notified of the first message with distinct values for its fields.

A digest message may contain any value from the data plane that is an input to the ingress deparser. If a desired field is not already an input to the ingress deparser, a P4 developer may add the field to their user-defined metadata type, and assign the desired value to the field in the ingress control. When a Digest instance is created, the P4 programmer specifies a type that holds the desired value(s), often a P4 struct type. The compiler determines a good serialization format to send the digest contents to the control plane software. TNA provides the ability to the control plane software to distinguish the messages created by different Digest instances from each other.

A TNA program can instantiate up to 8 different Digest instances in the ingress deparser block. Each digest message can hold up to 47 bytes of data. At most one digest message can be created each time a packet performs ingress processing.

By default, no digests are generated. To generate a digest, assign a value in the range 0 through 7, inclusive, to the `digest_type` field in the ingress control, and then use that value in the ingress deparser to choose between the values to send in the digest, as shown in the example code below.

The `digest_type` field has field validity (see Section 5.3) and is initially invalid at the beginning of ingress processing. If at one point in your ingress P4 code you assign a value to `digest_type`, and then later want to undo the decision to generate a digest, call the `invalidate` extern function on the `digest_type` field.

A digest is created by calling the `pack` method on an instance of the Digest extern. The argument is the value to be included in the digest, and the type of this argument must be the same as the type given when the Digest instance was constructed. Every `pack` method call must be enclosed in an `if` statement with a condition of the form `(digest_type == constant)`. The recommended P4 code pattern to use the Digest extern is as follows:

```
/* A simple digest of a header fields and metadata. */
struct digest_a_t {
    mac_addr_t dst_addr;
    PortId_t   port;
    mac_addr_t src_addr;
}

/* A second digest of header fields and metadata. */
struct digest_b_t {
    mac_addr_t dst_addr;
    bit<16>    f1;
}

struct my_ingress_metadata_t {
    // user-defined ingress metadata
    PortId_t port;
    bit<16>  metadata_to_learn;
}
```

```
control MyIngressDeparser(
    packet_out pkt,
    inout my_ingress_headers_t  ig_hdr,
    in    my_ingress_metadata_t ig_md,
    in    ingress_intrinsic_metadata_for_deparser_t ig_dprsr_md)
{
    Digest<digest_a_t>() digest_a;
    Digest<digest_b_t>() digest_b;

    apply {
        // Generate a digest, if digest_type is set in ingress control.
        if (ig_dprsr_md.digest_type == 1) {
            // The fields in the list correspond to the fields of the
            // struct with type digest_a_t.
            digest_a.pack({ig_hdr.ethernet.dst_addr, ig_md.port,
                    ig_hdr.ethernet.src_addr});
        }
        if (ig_dprsr_md.digest_type == 2) {
            // The fields in the list correspond to the fields of the
            // struct with type digest_b_t.
            digest_b.pack({ig_hdr.ethernet.dst_addr,
                    ig_md.metadata_to_learn});
        }
        // the rest of the ingress deparser code goes here
        pkt.emit(ig_hdr.ethernet);
    }
}
```

Regardless of whether a digest is generated or not, the rest of the ingress deparser code still takes effect for determining whether a packet is resubmitted, mirrored, and/or a normal packet is sent to the Traffic Manager, as described in Section 5.5.

It is possible to have eight `if` statements comparing `digest_type` to each of the values 0 through 7. Because the `digest_type` has field validity (see Section 5.3), if `digest_type` is invalid, all eight of those conditions will evaluate to false.

7.8 Hash

The Hash extern can take any collection of header or metadata fields and calculate a deterministic hash function of those values. The output of the Hash extern can be used in arbitrary expressions, e.g. as an index for an indirect counter, meter, or register. The Hash output can also be used as an input to an `ActionSelector` extern, often used for implementing features like ECMP or LAG to load balance traffic across several available network paths.

```
extern Hash<W> {
    // Constructor
    // @type_param W : width of the calculated hash.
    // @param algo : The default algorithm used for hash calculation.
    Hash(HashAlgorithm_t algo);

    // Constructor
    // @param poly : The default coefficient used for hash algorithm.
    Hash(HashAlgorithm_t algo, CRCPolynomial<_> poly);
```

```
    // Compute the hash for the given data.
    // @param data : The list of fields contributing to the hash.
    // @return The hash value.
    W get<D>(in D data);
}

extern CRCPolynomial<T> {
    CRCPolynomial(T coeff, bool reversed, bool msb, bool extended,
                  T init, T xor);
}
```

7.8.1 Predefined Hash Algorithms

The `HashAlgorithm_t enum` type contains values for all the hash algorithms available for use via the `Hash` extern.

Supported hash algorithms:

```
enum HashAlgorithm_t {
    IDENTITY,
    RANDOM,
    CRC8,
    CRC16,
    CRC32,
    CRC64,
    CUSTOM
}
```

7.8.2 User-Defined Hash Algorithms

User can define any custom CRC polynomial with the `CRCPolynomial` extern. The next table can be used as a reference for hash parameters to use for some standard CRC functions.

Table 5: Constants Used to Instantiate Pre-Defined CRC Polynomials

Name	Coeff	Rev-ersed	Init	xor
crc_8	0x107		0x00	0x00
crc_8_darc	0x139	Y	0x00	0x00
crc_8_i_code	0x11D		0xFD	0x00
crc_8_itu	0x107		0x55	0x55
crc_8_maxim	0x131	Y	0x00	0x00
crc_8_rohc	0x107	Y	0xFF	0x00
crc_8_wcdma	0x19B	Y	0x00	0x00
crc_16	0x18005	Y	0x0000	0x0000
crc_16_buypass	0x18005		0x0000	0x0000
crc_16_dds_110	0x18005		0x800D	0x0000
crc_16_dect	0x10589		0x0001	0x0001
crc_16_dnp	0x13D65	Y	0xFFFF	0xFFFF
crc_16_en_13757	0x13D65		0xFFFF	0xFFFF
crc_16_genibus	0x11021		0x0000	0xFFFF
crc_16_maxim	0x18005	Y	0xFFFF	0xFFFF

Name	Coeff	Rev-ersed	Init	xor
crc_16_mcrf4xx	0x11021	Y	0xFFFF	0x0000
crc_16_riello	0x11021	Y	0x554D	0x0000
crc_16_t10_dif	0x18BB7		0x0000	0x0000
crc_16_teledisk	0x1A097		0x0000	0x0000
crc_16_usb	0x18005	Y	0x0000	0xFFFF
x_25	0x11021	Y	0x0000	0xFFFF
xmodem	0x11021		0x0000	0x0000
modbus	0x18005	Y	0xFFFF	0x0000
kermit	0x11021	Y	0x0000	0x0000
crc_ccitt_false	0x11021		0xFFFF	0x0000
crc_aug_ccitt	0x11021		0x1D0F	0x0000
crc_32	0x104C11DB7	Y	0x00000000	0xFFFFFFFF
crc_32_bzip2	0x104C11DB7		0x00000000	0xFFFFFFFF
crc_32c	0x11EDC6F41	Y	0x00000000	0xFFFFFFFF
crc_32d	0x1A833982B	Y	0x00000000	0xFFFFFFFF
crc_32_mpeg	0x104C11DB7		0xFFFFFFFF	0x00000000
posix	0x104C11DB7		0xFFFFFFFF	0xFFFFFFFF
crc_32q	0x1814141AB		0x00000000	0x00000000
jamcrc	0x104C11DB7	Y	0xFFFFFFFF	0x00000000
xfer	0x1000000AF		0x00000000	0x00000000
crc_64	0x1000000000000001B	Y	0x0000000000000000	0x0000000000000000
crc_64_we	0x142F0E1EBA9EA3693		0x0000000000000000	0xFFFFFFFFFFFFFFFF
crc_64_jones	0x1AD93D23594C935A9	Y	0xFFFFFFFFFFFFFFFF	0x0000000000000000

NOTE: In Table 5: Constants Used to Instantiate Pre-Defined CRC Polynomials, if "Reversed" = "Y", then set the `CRCPolynomial` reversed argument to 1; otherwise set reversed to 0.

The supported parameters on `CRCPolynomial` are:

- **poly**: the polynomial to use. Note: for the polynomial of the power N, bit N must be set. In other words, the bit width of the string must be at least `N+1`.
- **init**: the initial value of the CRC shift register. Default value is 0

- **reversed**: Boolean true to specify that the data bits to be hashed are sent in reverse order into the (conceptual) shift register performing the CRC; Boolean false to send the data bits in the normal order.
- **xor**: a value to XOR with the final CRC computed. Default value is 0x0
- **msb**: indicates to use the most significant bits of the hash output
- **extend**: indicates to repeat the hash output width until the desired bit width is achieved

When using the `CRCPolynomial` extern with the Hash extern, the `HashAlgorithm_t` parameter should be `HashAlgorithm_t.CUSTOM`.

```
// instantiate a Hash extern named hash_1
Hash<bit<16>>(HashAlgorithm_t.CRC16) hash_1;

// calculate the hash of the concatenation of two fields of the packet,
// storing the calculated hash in packet.hash_result1.
packet.hash_result1 = hash_1.get({ packet.hash_field1,
                                   packet.hash_field2 });
```

7.9 Meters

Meters (RFC 2698) provide a mechanism for measuring and detecting when a sequence of packets in a flow have recently been arriving slower or faster than a configured average rate (or two configured rates, for two rate meters). Each packet is categorized into one of the three colors green, yellow, or red at the time the meter is accessed and updated. See RFC 2698 for details on the conditions under which one of these three results is returned.

Often the resulting color is used to decide whether to perform a modification to a packet, or to drop the packet. The meter extern only updates the meter state accessed and returns a color. It is up to the P4 developer to decide what further actions they wish to perform on the packet based upon the result.

The value returned by an uninitialized meter is always green. This is in accordance with the P4 Runtime specification.

RFC 2698 describes "color aware" and "color blind" meters. The `Meter` and `DirectMeter` externs implement both behaviors. The only difference is in which `execute` method you call when updating them. See the comments on the extern definitions below.

Like counters, there are two variants of meters: direct and indirect. See Section 7.1 for a description of these variants.

7.9.1 Meter Types

You must provide a value of type `MeterType_t` when constructing a meter extern. You may create a meter that operates on packet rates, independent of the size of the packets, or on the number of bytes each packet contains.

```
enum MeterType_t {
    PACKETS,
    BYTES
}
```

The packet length used to update a byte-based meter is always the length of the packet currently being processed, as defined in Section 7.2, minus the value of the `adjust_byte_count` parameter, if such a parameter is given.

7.9.2 Meter Colors

The MeterColor_t type gives names for the default numeric encoding of green, yellow, and red results returned from executing a meter. It is also the encoding that must be used for the input color when performing a color-aware meter update.

```
enum bit<8> MeterColor_t {
    GREEN  = 8w0,
    YELLOW = 8w1,
    RED    = 8w3
}
```

Table 6: Meter Color Encoding

Color	Encoding (binary)	Encoding (decimal)
GREEN	0b00	8w0
YELLOW	0b01	8w1
YELLOW	0b10	8w2
RED	0b11	8w3

Note that TNA recognizes two different numeric values as an encoding for the color yellow. TNA does not treat these two values any differently from each other when a color value is needed.

7.9.2.1 User-Defined Colors

There is a translation table in TNA that may be used to convert the 8-bit numeric encodings of the colors green, yellow, and red given above, to different 8-bit numeric encodings that you configure. This translation table may be useful in cases where the actions your program wishes to perform based upon the green/yellow/red result can be made more efficient by using a different encoding. For example, it may reduce the number of if statements in your P4 program if you change the color encoding in a way that enables your program to do a bitwise OR operation on the meter color and the `drop_ctl` intrinsic metadata field.

This translation only exists for converting the return value from the Meter and DirectMeter `execute` method calls. If you use an `execute` method that performs a color-aware meter update, your program must use the numeric encoding documented above.

7.9.3 Indirect Meter

The code below can be found in the TNA include file. It defines the types of parameters and return values for the constructor and `execute` methods for an indirect meter.

```
extern Meter<I> {
    Meter(bit<32> size, MeterType_t type);
    Meter(bit<32> size, MeterType_t type, bit<8> red, bit<8> yellow,
          bit<8> green);

    // Use this method call to perform a color aware meter update (see
    // RFC 2698). The color of the packet before the method call was
    // made is specified by the color parameter.
    bit<8> execute(in I index, in MeterColor_t color,
                   @optional in bit<32> adjust_byte_count);

    // Use this method call to perform a color-blind meter update (see
    // RFC 2698). It may be implemented via a call to execute(index,
    // MeterColor_t.GREEN), which has the same behavior.
    bit<8> execute(in I index, @optional in bit<32> adjust_byte_count);
}
```

7.9.4 Direct Meter

The declarations for the constructor and count method of a `DirectMeter` extern are shown here:

```
extern DirectMeter {
    DirectMeter(MeterType_t type);
    DirectMeter(MeterType_t type, bit<8> red, bit<8> yellow,
                bit<8> green);
    bit<8> execute(in MeterColor_t color,
                   @optional in bit<32> adjust_byte_count);
    bit<8> execute(@optional in bit<32> adjust_byte_count);
}
```

A direct meter instance must appear as the value of the `meters` table property for exactly one table. That table is the direct meter's owner. It is an error to call the `execute` method for a direct meter instance anywhere except inside an action of its owner table. TNA requires that every action of the direct meter's owner table must call the direct meter's `execute` method exactly once.

7.10 Mirror

This section describes packet mirroring, a mechanism to create a copy of a packet and send the copies to a specified destination (see Section 5.3 for the definition of a destination). TNA supports two packet paths for mirroring: ingress-to-egress mirroring and egress-to-egress mirroring.

In the ingress-to-egress mirroring path, the decision to mirror the packet is made in the ingress control, and the Mirror extern `emit` method is called in the ingress

deparser. The Mirror extern makes a copy of the packet's contents as it was received by the ingress parser, before any modifications were made in the ingress control or ingress deparser (the ingress intrinsic metadata and port metadata or resubmit header are not included in the ingress-to-egress mirrored packet). The Mirror extern `emit` method then prepends a mirror header that you specify and sends this packet to the Traffic Manager.

The Traffic Manager uses a mirror session identifier (specified as the first parameter to the Mirror extern `emit` call) to look up a mirror session table that contains the values of intrinsic metadata indicating where the mirror packet should be sent. All mirror sessions are configured by the control plane. The mirrored packet(s) will later begin processing in the egress pipeline.

The egress-to-egress mirroring path is similar. The decision to mirror the packet is made in the egress control, and the Mirror extern `emit` method is called in the egress deparser. The main difference, compared to ingress-to-egress mirroring, is that the Mirror extern makes a copy of the packet's contents as it is produced by the egress deparser, after all modifications made in the egress control and egress deparser are completed. The Mirror extern then prepends a mirror header that you specify and sends this packet to the Traffic Manager. The Traffic Manager operates on such packets in the same way described above for the ingress-to-egress mirroring path.

When your code initiates a mirror operation, you specify one of 1023 *mirroring sessions* to use. (Note: Please, do not use Mirror Session 0, which is reserved.) Each mirroring session is independently configured by the control plane software. This configuration data resides within the Traffic Manager.

For example, suppose a P4 program would like to send a packet replica to the CPU Ethernet port. The P4 program and control plane software developers agree on using mirroring session 27 for this purpose. The control plane software configures mirroring session 27 with the `ucast_gress_port` attribute set to the CPU Ethernet port number. Later the P4 program processes a packet and calls a Mirror extern `emit` method with session 27, which causes the Traffic manager to read session 27's configuration data and find the `ucast_gress_port` equal to the CPU Ethernet port number. The Traffic Manager enqueues the mirrored packet for the CPU Ethernet port. Note that the P4 program cannot change the mirror session attributes.

The Traffic Manager can prepend up to 32 bytes of user-defined header to the mirrored packet. Intel Tofino supports up to 8 possible user-defined headers. Each mirrored packet can use one of the 8 user-defined headers, or none. (Note: Please, do not use mirror type 0 for ingress-to-egress mirroring.)

TNA provides the Mirror extern to choose a mirror session and a user-defined header for each mirrored packet. This ability to choose a mirror session is controlled by the `session_id` argument of the `emit` method. The ability to choose a user-defined header is controlled by the `mirror_type` intrinsic metadata field sent to ingress and egress deparser and the `hdr` argument in the `emit` method. Next, we explain the restrictions on the `Mirror` extern and the `mirror_type` field.

To mirror a packet in ingress, you must assign a value in the range 1 through 7, inclusive, to the `mirror_type` intrinsic metadata field. `mirror_type` is initialized to 0 at the beginning of ingress, and 0 should not be used to mirror packets in ingress. If at one point in your ingress P4 code you assign a value 1 through 7 to `mirror_type`,

and then later want to undo the decision to create an ingress-to-egress mirror, assign a value of 0 to `mirror_type`.

To mirror a packet in egress, you must assign a value in the range 0 through 7, inclusive, to the `mirror_type` intrinsic metadata field. In egress, the `mirror_type` field has field validity (see Section 5.3), and is initially invalid at the beginning of egress. If at one point in your egress P4 code you assign a value to `mirror_type`, and then later want to undo the decision to create an egress-to-egress mirror, call the `invalidate` extern function on the `mirror_type` field.

The Mirror extern may only be instantiated in the ingress deparser, egress deparser, or in both. The compiler will reject any attempt to instantiate the `Mirror` extern elsewhere.

The `Mirror` extern provides two `emit` methods. Calling the 1-argument `emit` method will create a mirrored packet with no user-defined header, using the mirror session specified by `session_id`. Calling the 2-argument `emit` method will create a mirrored packet with a user-defined header `hdr`, using the mirror session specified by `session_id`.

```
extern Mirror {
    // Constructor
    Mirror();

    // Mirror the packet.
    void emit(in MirrorId_t session_id);

    // Write @hdr into the ingress/egress mirror buffer.
    // @param hdr : T can be a header type.
    void emit<T>(in MirrorId_t session_id, in T hdr);
}
```

A P4 program may instantiate one or more Mirror externs. However, only one Mirror extern may be used for any given packet. Every `emit` method call must be enclosed in an `if` statement with a condition of the form `(mirror_type == constant)`. The recommended P4 code pattern to use the Mirror extern is as follows:

```
Mirror() mirror;   // construct an instance of Mirror extern
apply {
    if (ig_dprsr_md.mirror_type == 1) {             // check mirror_type
        mirror.emit(session_id, mirror_hdr_type1);  // call emit
    }
    if (ig_dprsr_md .mirror_type == 2) {
        mirror.emit(session_id, mirror_hdr_type2);
    }
    // the rest of ingress deparser code goes here
    pkt.emit(hdr.ethernet);
    pkt.emit(hdr.ipv4);
    // etc.
}
```

It is possible to have eight `if` statements comparing `mirror_type` to each of the values 0 through 7 in the egress deparser. Because the `mirror_type` has field validity

in egress (see Section 5.3), if `mirror_type` is invalid, all eight of those conditions will evaluate to false and no mirror packet will be created.

7.11 Parser Counter

The ParserCounter extern can be used to extract header stacks or headers with variable length. Intel Tofino has a single 8-bit signed counter that can be initialized with an immediate value or a header field. A limited number of operations such as masking and circular rotation can be performed on the header field when initializing the counter using the `set` method.

```
extern ParserCounter {
    // Constructor
    ParserCounter();

    // Load the counter with an immediate value or a header field.
    void set<T>(in T value);

    // Load the counter with a header field.
    // @param max : Maximum permitted value for counter (pre
    //     rotate/mask/add).
    // @param rotate : Right rotate (circular) the source field by this
    //     number of bits.
    // @param mask : Mask the rotated source field by 2 ^ (mask + 1) - 1.
    // @param add : Constant to add to the rotated and masked lookup
    //     field.
    void set<T>(in T field,
                in bit<8> max,
                in bit<8> rotate,
                in bit<3> mask,
                in bit<8> add);

    // @return true if counter value is zero.
    bool is_zero();

    // @return true if counter value is negative.
    bool is_negative();

    // Add an immediate value to the parser counter.
    // @param value : Constant to add to the counter.
    void increment(in bit<8> value);

    // Subtract an immediate value from the parser counter.
    // @param value : Constant to subtract from the counter.
    void decrement(in bit<8> value);
}
```

7.12 Random

The Random extern provides generation of pseudo-random numbers with a uniform distribution. If one wishes to generate numbers with a non-uniform distribution, you may do so by first generating a uniformly distributed random value, and then using

appropriate table lookups and/or arithmetic on the resulting value to achieve the desired distribution.

```
// Random number generator.
extern Random<W> {
    // Constructor
    Random();

    // Return a random number with uniform distribution.
    // @return : random number between 0 and 2**W - 1
    W get(W max);
}
```

7.13 Registers

Registers are stateful memories whose values can be read and written during packet forwarding under the control of the P4 program. They are like counters and meters in that their state can be modified as a result of processing packets, but they are far more general in the behavior they can implement.

Like counters, there are two variants of registers: direct and indirect. See Section 7.1 for a description of these variants. Below are the definitions of the constructor methods for creating instances of these externs.

```
extern Register<T, I> {
    Register(bit<32> size);
    Register(bit<32> size, T initial_value);
}

extern DirectRegister<T> {
    DirectRegister();
    DirectRegister(T initial_value);
}
```

The type argument `I` specifies the type of the index of an indirect register extern. This type can typically be inferred by the compiler.

The type argument `T` specifies the type of each entry, i.e. the type of state stored in each entry of the register. Register entries in TNA may be bit<8>, int<8>, bit<16>, int<16>, bit<32>, or int<32> values, or may be pairs of one of those types. Any struct with exactly two fields of identical type will be recognized as a pair. In addition, entries containing a single value of type bit<1> can also be used.

7.13.1 RegisterAction extern

Registers are accessed via `RegisterAction` externs, which contain a function named `apply` that can read and update the value of one entry of a Register. Up to four separate `RegisterAction`s may be defined for a single Register extern, but only one `RegisterAction` may be executed per packet for a given Register.

The apply method in a `RegisterAction` may be declared with either one or two arguments; the first inout argument is the value of the Register entry being read and updated, while the second optional out argument is the value that will be returned by the execute method when it is called in a table action.

Together `Register` and `RegisterAction` externs form the Stateful ALU portion of the pipeline. `RegisterAction`s are triggered from table actions by calling their execute method, which causes the Stateful ALU to read the specified entry of the Register storage, run the `RegisterAction` code, and write back the modified value to the same entry of the Register storage. The simplest example Stateful ALU would be a simple packet counter

```
control example<M>(M meta) {
    Register<bit<32>, _>(4096) counters;
    RegisterAction<_, _, void> increment_counter = {
        void apply(inout bit<32> value) {
            value = value + meta.increment_amount;
        }
    };
    action trigger_counter() {
        increment_counter.execute(meta.index);
    }
}
```

This illustrates the basic structure and use of a Stateful ALU. The `RegisterAction.apply` method encapsulates all the behavior in the stateful ALU, and the `execute` method triggers it to run from some other construct in the P4 program.

Within the Stateful ALU there are two comparison ALUs and four arithmetic/logical ALUs that can be used for computation. The results of the comparisons may be used to condition the outputs of the other ALUs. All the ALUs operate on the same word size for a given `RegisterAction` which may be 8, 16, or 32 bits. The inputs to all the ALUs may come from the memory word (either half if it is a pair) or from either of two PHV registers accessed by the stateful ALU, or from up to 4 constant values. All `RegisterAction`s sharing a single Register also share these resources, so the compiler will attempt to efficiently pack and reuse values used by `RegisterAction`s in the same stateful ALU.

The four arithmetic/logical ALUs are organized as two pairs of two, with one pair writing its result to each word of the memory. The compiler will schedule the operations specified in the `apply` method to the appropriate ALUs to perform the specified computation, and will attempt to reuse values and pack things as efficiently as possible. The optional output of the stateful ALU (second argument to the `apply` method which will be returned by the execute method) must be a copy of one of the values read as input by the stateful ALU or one of the values being written to memory.

The restrictions above are significant, but they can be used to construct very useful stateful data plane algorithms. Once learned, writing `RegisterAction` code is reasonably straightforward, using if-else statements and simple assignments to the parameters of the `apply` call. The constant ALUs can compare a memory value, a packet value and a constant added together, allowing tests for thresholds or limits of various kinds triggering different operations depending on them. For example, you can measure inter-packet timestamps by storing the previous timestamp in the memory and look for bursts exceeding some burst size:

```
struct burst_data {
    bit<32> timestamp;
    bit<32> count;
};

#define BURST_INTERPACKET_DELAY 10
#define BURST_SIZE 5

Register<burst_data, _>(0x1000) reg;

RegisterAction<burst_data, _, bit<1>>(reg) bursts = {
    void apply(inout burst_data data, out bit<1> burst) {
        burst = 0;
        if (meta.current_time - data.timestamp
                <= BURST_INTERPACKET_DELAY)
        {
            if (data.count >= BURST_SIZE) {
                burst = 1;
            }
            data.count = data.count + 1;
        } else {
            data.count = 1;
        }
        data.timestamp = meta.current_time;
    }
}
```

This RegisterAction can then be called from an action and the return value tested to see if this packet belongs to a burst of 5 or more packets. The above code shows a few basic principles for structuring the code that will minimize problems

- Always have tests involving the values from memory before modifying them
- Only one modification of each memory item on any given path through the code
- When outputting a flag, set it to zero at the top of the apply method, and set it to one when the condition you want to test is true

The arithmetic ALUs support normal addition and subtraction operations. They also support saturating addition and subtraction operations (see the $P4_{16}$ language specification for the "`|+|`" and "`|-|`" operators). All combinations of bitwise logical operations are supported, as well as `min` and `max`.

If the RegisterAction entry type is bit<1>, most of the stateful ALU functionality is disabled. No comparisons or PHV reads are allowed, and the only operations allowed are setting the memory bit value to 0 or to 1, along with optionally returning the previous bit value. This is mainly useful for updating `ActionSelector`s or various bloom filter like constructs.

7.13.1.1 MathUnit Extern

The `MathUnit` extern allows access to the Intel Tofino math unit block in the Stateful ALU. The extern can be programmed with the raw parameters (`exponent_invert`, `exponent_shift`, `output_scale` and 16 x bit<8> raw data), or can be programmed by giving it a function and a scale constant.

```
enum MathOp_t {
    MUL,          // x
    SQR,          // x^2
    SQRT,         // sqrt(x)
    DIV,          // 1/x
    RSQR,         // 1/x^2
    RSQRT         // 1/sqrt(x)
};

extern MathUnit<T> {
    // Configure a math unit for use in a register action
    MathUnit(MathOp_t op, int factor);  // configure as factor * op(x)
    MathUnit(MathOp_t op, int A, int B);  // configure as (A/B) * op(x)
    T execute(in T x);
};
```

An instance of the `MathUnit` extern can be invoked in a RegisterAction: e.g.:

```
    Register<bit<16>, bit<16>>(1) accum;
    MathUnit<bit<16>>(MathOp_t.SQR, 1) square;
    RegisterAction<bit<16>, bit<1>, bit<16>>(accum) run = {
        void apply(inout bit<16> value, out bit<16> rv) {
            value = square.execute(hdr.data.h1);
            // Can't output the math unit directly -- math unit can
            // only go in alu2-lo, which writes to memory.
            rv = value;
        }
    };
```

A given `Register` can only have one `MathUnit` extern, so if there are multiple `RegisterAction`s sharing a `Register`, they can only use one `MathUnit` between them (it may be used in more than one `RegisterAction`).

7.14 Resubmit

Packet resubmission is a mechanism to repeat ingress processing on a packet. One example where this can be useful is when parsing MPLS packets, where the first header that appears after the last MPLS header might be Ethernet or IPv4. The first time the packet is parsed, after parsing the last MPLS header, if the next 4 bits are the value 4 in decimal, then the next header may be either:

- an IPv4 header
- an Ethernet header where the value of the most significant 4 bits of the destination MAC address are the value 4 in decimal

There is not enough information in the packet contents to distinguish between these cases. A good approach is to guess that the header is IPv4 during parsing and proceed.

During ingress processing, a lookup in a table with a key of the packet's MPLS label(s) may give a result indicating that the control plane created this stack of MPLS labels for encapsulating Ethernet frames. If so, we now know that we made an incorrect guess

that the next header was IPv4. A good approach is to resubmit the packet, with a resubmit header indicating that the parser should parse the first header after the last MPLS header as Ethernet.

All not-resubmitted packets begin in the ingress parser with `resubmit_flag=0` in the ingress intrinsic metadata. This includes packets from a front panel port, the CPU, the packet generator, and recirculated packets. All resubmitted packets have `resubmit_flag=1`, to enable the P4 code to distinguish resubmitted packets from the original packet.

The input port holds a copy of all received packets until the end of ingress processing. When your program resubmits a packet, the input port sends the buffered packet to the ingress pipeline again, with only small modifications described below.

To enable your P4 program to use some additional facts about what happened during the first time processing the packet, during that first time you may make a method call `emit` on an instance of the Resubmit extern and pass it an argument with type header (a header type that you define) that is up to 64 bits long. The contents of this header will overwrite the port metadata of the original packet, and this resubmit header can be extracted by the ingress parser when the resubmitted packet is parsed.

Other than the value of the `resubmit_flag` field and the contents of the port metadata header being overwritten by the resubmit header, the rest of the contents of the original packet remain as they were the first time it was processed.

You may resubmit each received packet at most once. Attempting to perform a resubmit operation on a packet with `resubmit_flag=1` causes the packet to be dropped.

TNA supports up to 8 different user-defined resubmit headers. To select which one to use for a particular packet, your P4 program should assign a value to the `resubmit_type` field in the ingress control, and then use that value in the ingress deparser to choose between your user-defined resubmit headers, as shown in the example code below.

To resubmit a packet, you must assign a value in the range 0 through 7, inclusive, to the `resubmit_type` intrinsic metadata field. The `resubmit_type` field has field validity (see Section 5.3) and is initially invalid at the beginning of ingress processing. If at one point in your ingress P4 code you assign a value to `resubmit_type`, and then later want to undo the decision to resubmit the packet, call the `invalidate` extern function on the `resubmit_type` field.

The Resubmit extern may only be instantiated in the ingress deparser. The Resubmit extern provides two `emit` methods. Invoking the 0-argument `emit` method will create a resubmitted packet with all zeroes in the resubmit header. The 1-argument `emit` method takes an argument `resub_hdr` with type header. Invoking the 1-argument `emit` method will create a resubmitted packet with `resub_hdr` as the contents of its resubmit header.

```
extern Resubmit {
    // Constructor
    Resubmit();
```

```
    // Resubmit the packet.
    void emit();

    // Resubmit the packet and prepend it with @hdr.
    // @param hdr : T can be a header type.
    void emit<T>(in T hdr);
}
```

A P4 program may instantiate multiple Resubmit externs. One Resubmit extern instance suffices, regardless of the number of different resubmit header formats you use. Every `emit` method call must be enclosed in an `if` statement with a condition of the form `(resubmit_type == constant)`. The recommended P4 code pattern to use the Resubmit extern is as follows:

```
Resubmit() resubmit;   // construct an instance of Mirror extern
apply {
    if (ig_intr_md_for_dprsr.resubmit_type == 1) { // check resubmit_type
        resubmit.emit(resubmit_hdr_type1);         // call emit
    }
    if (ig_intr_md_for_dprsr.resubmit_type == 2) {
        // showing a different style of specifying resubmit header
        // contents, assuming resubmit_hdr_type2 has 3 fields.
        resubmit.emit<resubmit_hdr_type2>(
            {meta.field1, hdr.ethernet.ether_type, meta.field7});
    }
    // the rest of ingress deparser code goes here
    pkt.emit(hdr.ethernet);
    pkt.emit(hdr.ipv4);
    // etc.
}
```

If the packet is resubmitted, no packet will be sent to the Traffic Manager. All calls to `pkt.emit` in the code example above will be executed, but the value of `pkt` is discarded.

It is possible to have eight `if` statements comparing `resubmit_type` to each of the values 0 through 7. Because the `resubmit_type` has field validity (see Section 5.3), if `resubmit_type` is invalid, all eight of those conditions will evaluate to false.

The resubmit operation is only supported for ports in quads 0 up to 15 of each Intel Tofino pipe (see Section 12). It is not supported on ports in quads 16 or 17.

8 Packet Truncation

Intel Tofino supports packet truncation for mirrored packets (see Section 7.10). For each mirror session id, the control plane can configure a maximum packet length in bytes. When a packet is mirrored using that session id, any bytes after the maximum configured for the session are removed. For the purposes of packet truncation, any mirrored header, but not egress intrinsic metadata, is included as part of the packet (see Section 5.8.1).

9 Packet Generation

There is one packet generator in each pipe. See Section 12 for the port numbers they are attached to.

Each packet generator can generate up to 8 independent streams of packets (called applications), numbered 0 through 7. Each application is initially disabled, and each can be configured and enabled independently by the control plane software. No packets will ever be generated by a disabled application.

When control plane software enables an application, it must specify what kind of an event (called a trigger) will start the process of packet generation. The following four trigger types are available:

- One-time timer
- Periodic timer
- Port down
- Packet recirculation

Each packet generator also has a 16 KByte buffer that is shared by all of its applications. The control plane configures the contents of this buffer. Each application is configured with a starting byte offset O and length L in bytes within this buffer. All packets created by the packet generator begin with a 6-byte pktgen header, followed by L bytes copied from this buffer, starting at offset O, as configured for the application that generated the packet.

Packets generated by the packet generator will enter the corresponding pipe through the ports they are attached to as normal (non-resubmitted) packets.

9.1 Common behavior for all triggers

Regardless of which trigger is used by a given application, once the specified event takes place, it will initiate the creation of one or more batches of packets, where each batch consists of a configured number of packets.

The control plane configures the following parameters for any application that it wishes to generate packets when the specified event occurs.

- the number of batches B to create, up to 64K (the current control plane API requires you to supply the maximum batch id B-1)
- the time interval IBT between batch starts (in nanoseconds)
- the number of packets N to generate per batch, up to 64K (the current control plane API requires you to supply the maximum packet id N-1)
- the time interval IPT between creating packets in the same batch (in nanoseconds)
- the starting byte offset O and length L in bytes within the packet generator's buffer.

The `pktgen` header for all triggers contains the following fields:

- `pipe_id` - the pipe number of the packet generator.
- `app_id` - the application number of the application that created the packet.
- `packet_id` - a number identifying which packet within a batch this one is.

Some triggers also populate a `batch_id` value to allow the P4 program to distinguish packets from different batches

For the simplest case of B=1 and N=1, the only packet created will have `batch_id` and `packet_id` equal to 0 (recall that the control plane API requires you to configure a value 1 less for B and N, so both 0 for this case).

In general, B*N packets will be created for each trigger event. These B*N packets differ only in the time they are created, and the values in their `batch_id` field (if they have one) and their `packet_id` field. The following pseudocode describes when each packet is created, and what values of `batch_id` and `packet_id` will be used to create that packet.

```
// Values B, IBT, N, IPT are configured for this application,
// as described earlier.

// trigger_event_time is the time of the event that caused this
// group of packet batches to be generated.  This varies from one
// trigger type to another.  See the later sections on each trigger.

// next_batch_start_time and next_packet_time are variables in
// this pseudocode that are not visible from a P4 program.

next_batch_start_time = trigger_event_time;
for (batch_id = 0; batch_id < B; batch_id++) {
    next_packet_time = max(time_now, next_batch_start_time);
    for (packet_id = 0; packet_id < N; packet_id++) {
        wait until time next_packet_time;
        // Note: do not wait at all if next_packet_time is in the past
        create packet with current batch_id and packet_id values;
        next_packet_time += IPT;
    }
    next_batch_start_time += IBT;
}
```

Note that it is possible to configure IPT to a time interval smaller than the time required to create and transmit one packet. If it is configured this way, the pseudocode statement “wait until time next_packet_time” will not wait any time at all, because next_packet_time will already have passed. In this case, packets within a batch will have no idle time gap between them.

Similarly, it is possible to configure IBT to a time interval smaller than the time to create one batch’s worth of packets. In this case, any later batch will begin immediately after the last packet of the previous batch was created.

The Intel Tofino ASIC also has configuration options where the IBT and IPT values can be pseudo-randomly generated within a configured range of values.

9.2 One-time timer trigger

When a control plane configures an application with a one-time timer trigger, it configures all the common values described in Section 9.1, plus the following:

- the time T when the trigger will fire

When time T is reached, the application's trigger fires once, generating B batches of N packets each as described in Section 9.1. Then the application is then disabled; it will not generate any more packets until the control plane enables it again.

The `pktgen` header for packets generated with a one-time timer trigger contains the following fields:

```
header pktgen_timer_header_t {
    bit<2>  pipe_id;         // Pipe id
    bit<3>  app_id;          // Application id
    bit<16> batch_id;        // Start at 0 and increment to a
                             // programmed number
    bit<16> packet_id;       // Start at 0 and increment to a
                             // programmed number
}
```

(Note that padding fields have been omitted from the definition of `pktgen_timer_header_t` for brevity.)

9.3 Periodic timer trigger

When a control plane configures an application with a periodic timer trigger, it configures all the common values described in Section 9.1, plus the following:

- the time interval T between consecutive trigger firing events for this application

When time T is reached, the application's trigger fires, generating B batches of N packets each as described in Section 9.1. The application remains enabled, and its next trigger event will be at time 2*T, its third trigger event will be at time 3*T, and so on. The application will continue to cause periodic trigger events until the control plane disables it.

The `pktgen` header for packets generated with a periodic timer trigger is the same as the one described in Section 9.2.

9.4 Port down trigger

When a control plane configures an application with a port down trigger, it configures all of the common values described in Section 9.1. No additional configuration is required for the port down trigger.

When a port that is in the same pipe as the packet generator changes state from up to down, the application's trigger fires once, generating B batches of N packets each as described in Section 9.1. The application is still enabled, but cannot fire again for the same port, until the control plane enables it to fire again for that port.

The control plane must make a call for the (application id, port number) combination to enable a port down event on that port number to cause packet generation to occur. This is necessary to enable the trigger to fire the first time. It is necessary to repeat a control plane call for an (application id, port number) combination after it has fired for that port number, or it will not fire again for that port number.

The `pktgen` header for packets generated with a port down trigger contains the following fields. (Note that the padding fields have been omitted from the definition of `pktgen_port_down_header_t` for brevity.)

```
header pktgen_port_down_header_t {
    bit<2>   pipe_id;       // Pipe id
    bit<3>   app_id;        // Application id
    PortId_t port_num;      // Port number
    bit<16>  packet_id;     // Start at 0 and increment to a
                            // programmed number
}
```

The `port_num` field contains the full port number, including the pipe id in the most significant, as described in Section 12. Note that while a port down trigger can generate more than one batch of packets, there is no `batch_id` field in the `pktgen` header.

Configuring a port down trigger for an application id of a packet generator cannot cause it to generate packets for port down events of ports in other Intel Tofino pipes.

9.5 Packet recirculation trigger

When a control plane configures an application with a packet recirculation trigger, it configures all of the common values described in Section 9.1, plus the following:

- the 32-bit value V
- the 32-bit mask M

The packet generator examines all packets that are recirculated on its port (but not any other recirculation ports). Let R be the first 32 bits of a recirculated packet as it was created by the egress deparser, just before it was recirculated. R[31:24] is the first byte, and R[7:0] is the fourth byte. For the packet generator, the first byte of a recirculated packet does not include the ingress intrinsic or port metadata shown in Section 5.1.

If (R & M) is equal to (V & M), the recirculated packet matches and the trigger fires, initiating the creation of B batches of packets as described in Section 9.1. The application remains enabled and will cause additional trigger events if later recirculated packet match the configured value/mask, until the control plane disables it.

The `pktgen` header for packets generated with a port down trigger contains the following fields (note that the padding fields have been omitted from the definition of pktgen_recirc_header_t for brevity).

```
header pktgen_recirc_header_t {
    bit<2>  pipe_id;        // Pipe id
    bit<3>  app_id;         // Application id
    bit<24> key;            // Key from the recirculated packet
    bit<16> packet_id;      // Start at 0 and increment to a
                            // programmed number
}
```

The field `key` is filled in as follows, where R is the value described above from the matching recirculated packet:

- `key[23:16]` is equal to R[23:16]
- `key[15:0]` is equal to (R[15:0] + `batch_id`)

The value of R[31:24] is not included in generated packets.

10 Time Synchronization

Switches can optionally synchronize times between them by participating in the PTP protocol, also known as IEEE 1588. Intel Tofino provides hardware assist for switches to maintain and distribute highly synchronized times. When these are enabled, all time stamp values described in all the intrinsic metadata fields use this synchronized value, and thus can be used to make accurate latency measurements between switches in a network.

Regardless of whether these features are enabled, all time stamp metadata fields within a single Intel Tofino device are synchronized with each other and can be used to make precise latency measurements between packets reaching different points within the same device.

For example, in egress processing one could use a formula like the following to calculate the time interval starting when a packet began ingress parsing, until the packet began egress parsing, in nanoseconds. Most of the variation in such a latency measurement would be in the time the packet spent waiting in a queue in the Traffic Manager.

```
eg_prsr_md.global_tstamp[31:0] - ig_prsr_md.global_tstamp[31:0]
```

Where `ig_prsr_md` the intrinsic metadata struct described in Section 5.2, and `eg_prsr_md` is the struct described in Section 5.9. Making this calculation requires including the value of `ig_prsr_md.global_tstamp[31:0]` with packets from ingress to egress, e.g. in a bridge header (see Section 5.8).

11 Annotations

11.1 @flexible

The annotation `@flexible` may be used on a field of a header or struct, or on an entire header or struct definition. A header with the `@flexible` annotation is an extension of the $P4_{16}$ language standard, since the compiler is allowed to change the order of fields within such a header, or add padding fields in arbitrary places, to optimize the format for efficient `emit` and `extract` calls on values with this header type.

Since the changes made by the P4 compiler to such a header's format cannot be predicted by the P4 developer, such headers should not be sent to an external device. The other device will have no way to know the bit-level format of a flexible header. Flexible headers are only intended to be used for headers sent and received within the same Intel Tofino device. For example:

- In bridge metadata headers for unicast and multicast packets sent from ingress to egress
- In mirrored packets
- In recirculated packets

11.2 @padding

The annotation `@padding` may be used on a field of a header or struct. Such a field's value can be changed at any time by the compiler. It is typically used on fields whose only reason for being included is to make the total length of it and the following field become a multiple of 8 bits long, or the size of one PHV container. In many cases, the compiler can produce more efficient results if it knows that it can overwrite these fields with arbitrary values at any time.

Note: using the `@padding` annotation on fields within headers that are transmitted on an external Ethernet port can cause arbitrary bit patterns to be filled in for this field, which will be visible to the device receiving such packets.

12 Intel Tofino Part Numbers

There are several part numbers of Intel Tofino switch ASIC. These part numbers are described in the table below.

Table 7: Intel Tofino Part Numbers and Port Numbers

Intel Tofino Part Number	Pipes	Packet Buffer (MBytes)	Front Panel Ports (100 GE)	Front Panel Ports (25 GE)
BFN-T10-064Q	4	22	64	256
BFN-T10-032Q	4	20	32	128
BFN-T10-032D	2	20	32	128
BFN-T10-032D-024	2	20	24	96
BFN-T10-032D-020	2	20	20	80
BFN-T10-032D-018	2	20	18	72

All Intel Tofino parts also have:

- 1 Ethernet CPU port that is typically attached to a general-purpose CPU on the same board
- 1 PCI Express CPU port.

Each 100 Gigabit Ethernet port can be configured as:

- 1 lane of 100 Gigabit Ethernet, or 40 Gigabit Ethernet
- 2 lanes, where each lane is 50 Gigabit Ethernet, or 40 Gigabit Ethernet
- 4 lanes, where each lane is 25 Gigabit Ethernet, or 10 Gigabit Ethernet

12.1 Dev port numbers

Intel Tofino port numbers, often called `devports`, are represented using 9-bit numbers in the data plane. The most significant 2 bits are the pipe identifier, and the least significant 7 bits are the port number within the pipe.

Within a pipe, ports are organized in 18 quads. Quads 0 through 15 are often connected to front panel ports. Quad 16 is connected either to a CPU port or a recirculation port, depending upon which pipe it is in (see the tables below). Quad 17 is connected to a packet generator and can also be used as a recirculation port.

When quad number Q within a pipe is configured with any of 1, 2, or 4 lanes, the port number within the pipe of its first lane is always 4*Q. The port number within the pipe of each lane depends upon the number of lanes:

- For 1 lane, the quad has only one port number within the pipe: 4*Q
- For 2 lanes, the quad has two port numbers within the pipe: 4*Q and 4*Q+2.
- For 4 lanes, the quad has four port numbers within the pipe: 4*Q, 4*Q+1, 4*Q+2, and 4*Q+3.

For a port number within a pipe A, in pipe number P, the port number is (128*P + A).

The tables below show all port numbers in 4-pipe and 2-pipe Intel Tofino part numbers.

Table 8: 4-pipe Intel Tofino Port Numbers

Pipe	Quads 0..15	Quad 16	Quad 17
0	Ethernet 0..63	CPU Ethernet 64..67	Recirc & PktGen 68..71
1	Ethernet 128..191	Recirc 192..195	Recirc & PktGen 196..199
2	Ethernet [Note 1] 256..319	CPU PCIe 320	Recirc & PktGen 324..327
3	Ethernet [Note 1] 384..447	Recirc 448..451	Recirc & PktGen 452..455

Note 1: For Intel Tofino part number BFN-T10-032Q, all ports in pipes 2 and 3, quads 0 through 15, are recirculation ports.

Table 9: 2-pipe Intel Tofino Port Numbers

Pipe	Quads 0..15	Quad 16	Quad 17
0	Ethernet 0..63	CPU Ethernet 64..67	Recirc & PktGen 68..71
1	Ethernet 128..191	CPU PCIe 192	Recirc & PktGen 196..199

It is possible to configure an arbitrary subset of the quads 0..15 in any pipe in loopback mode. Such quads function as additional recirculation ports.

12.2 Multicast port numbers

There is another port numbering convention used in several of Intel Tofino's fixed function tables. These are called `mcports` for "multicast ports", because the primary places where this alternate port numbering is used is in configuring tables related to multicast.

For a port number within a pipe A (as described in the previous section), in pipe number P, the `mcport` number is `(72*P + A)`.

Everywhere in this document that port numbers are involved, it will be explicitly stated if it must be an `mcport` number. Otherwise it should be a `devport` number.