# NPL

Accelerates high performance networking.

Network Programming Language

v1.3

# NPL – Network Programming Language Specification

# v1.3

January 23, 2019

# Table of Contents

# Table of Tables

# Table of Figures

# 1 Scope

This document describes the Network Programming Language (NPL) constructs and usage.

The main objective of NPL is to describe Data Plane Packet Processing behavior using an appropriate set of constructs. A packet processing application in NPL includes high level constructs for functions such as parsing, match action tables, and packet editing. It can also include specific constructs for other features, such as functions.

Since a primary requirement of the language is to map onto flexible hardware, the overall set of available constructs tends to be focused on features found in flexible hardware. Application development must be done using these constructs.

This document is intended for System Architects, Design Engineers, and Software Engineers so that they can understand NPL logic details, make modifications to an NPL program, or develop their own custom packet processing applications. Test Engineers should understand NPL to enable them to develop comprehensive test plans.

This document does not cover any particular programmable device architecture or the operation of the NPL Front End or Back End compilers.

# 2 Terminology

The following lists the terms, concepts, symbols, and acronyms used in this document.

| Term | Description |
|---|---|
| **NPL Compiler** | Consists of Front End and Back End Compilers |
| **Front End (FE) Compiler** | A component of the compiler that parses and syntax checks NPL source code, and generates an Intermediate Representation (IR) |
| **Back End (BE) Compiler** | A component of the compiler that takes an IR and generates the personality for various hardware components |
| **IR files** | Intermediate Representation files |
| **Constructs** | Built-in components which provide particular functions |
| **Metadata** | Bus, header, and table fields that are not created in the NPL but are still present and accessible. |

# 3 Overview

The growth of Software Defined Networks (SDN) raised the expectations for network programmability and automation. Initially SDN focused on control plane issues, with users seeking to overcome limitations of traditional management models. Subsequently users wanted more flexible solutions capable of adapting to changing network needs, such as new overlay protocols and advanced telemetry features. This enlarged the scope of SDN to include programming the data plane. However, new flexible switching solutions must be capable of delivering full line-rate performance with optimized switch resources and power.

While different programing languages could be designed for programming the data plane, a language with appropriate constructs that take advantage of advanced programmable hardware architecture capabilities is key. For that reason a new language was developed: NPL (Network Programming Language) is intended to be an open, high level language that addresses the unique requirements of efficiently programming packet forwarding planes. NPL includes constructs to express networking behavior that take advantage of advanced features of the underlying programmable hardware.

In its first incarnation, NPL provides all the necessary features to implement robust network switching solutions. NPL building blocks range from data types that allow the specification of individual control signals to high-level constructs which allow interfacing with complex hardware blocks.

In a typical fixed function switch the set of tables and objects for packet processing are designed in, with only limited changes possible after manufacturing. In a programmable switch, packet processing elements can be determined by the user. NPL allows the user to specify details of tables and other objects to achieve the desired behavior. NPL is specialized for programming the data plane. It relies on conventional programming languages to specify how the control plane utilizes features in the switch data path.

NPL includes the following core abstractions:

- Data Types: specifies the basic building blocks of any object field.
- Parser: specifies the allowed headers within received packets and extracts those headers from the packets.
- Logical Bus: specifies the fields and overlays of a logical bus. Logical bus connects various other NPL objects.
- Logical Table (Match Action table): describes a particular table with the associated keys and actions. NPL supports index, hash, tcam, lpm, alpm tables.
- Editor: provides the ability to add, remove or replace a header.
- Special Function: mechanism to call a particular hardware function that may be treated as intellectual property. This provides a structured mechanism to define the interface into these functions without revealing the contents of the function.
- Function: provides programmable decision logic without the overhead of a table. For example, can be used to resolve the results of multiple Match Actions, or to resolve the Match Action key selection.
- Strength Resolution: mechanism to resolve multiple tables updating the same object in parallel.
- Packet Drop, Packet Trace and Packet Count: built-in functions to drop, trace and count packets.
- Create Checksum and Update Packet Length: built-in functions to create checksum and update packet lengths.
- Metadata for MA and Parser: data not created in the NPL, yet still exists at runtime with the packet and can be used by the NPL.

The NPL language is not intrinsically bound to any specific hardware architecture. It is intended to be implemented on multiple hardware platforms such as programmable ASICs, programmable network interface cards (NICs), FPGAs and pure software switches. While certain language constructs are intended to optimize use of specific hardware features on certain targets, these would not prevent mapping to targets that do not support these features.

Like any high level programming language, NPL requires a set of compilers and associated tools to map the programs written in NPL to target hardware objects. The front-end compiler is responsible for checking the syntax and semantics of the user-written program in the NPL language generating an Intermediate Representation (IR). The back-end compiler is responsible for mapping these intermediate representations into specific hardware objects. It also generates an API that the control plane uses to manage the behavior of the switch.

The compilers provide a level of parallelization which is determined by the NPL and underlying hardware.

## 3.1 Benefits

The development of NPL began with a review of existing programmable products and languages. This review clearly highlighted that the products and languages that existed in the public domain were missing key elements. Primarily, they were missing the ability to expose and take advantage of architectural efficiencies of the underlying hardware. These inefficiencies impacted multiple areas, including (but not limited to) increases in latency, increased power usage, and increased area. As such it was decided to create a new language that would enable users to specify Data Plane networking functionality with capabilities to expose target efficiencies with clear user intent. The result is NPL.

Compared to configurable and other programmable solutions available today, NPL provides many advantages. The language has sophisticated features that promote:

- customized table pipelines
- intelligent action processing
- parallelism
- advanced logical table capabilities
- an integrated instrumentation plane
- simple, intuitive control flow

NPL also provides constructs that provide for the inclusion of component libraries that implement fixed function hardware blocks. These features enable describing a range of dataplane application in NPL, from simple table-based architectures to more advanced architectures that incorporate a range of highly efficient building blocks.

The language constructs allow for the expression of these capabilities, which help significantly improve power efficiency and reduce cost in the resulting hardware implementation. Similarly, NPL language constructs promote software reuse that help in building a family of switching solutions ranging from simple to increasingly complex.

## 3.2 Architecture Model

NPL is a high level language designed to provide all the building blocks necessary to implement a robust network switching solution. These building blocks range from data types that allow the specification of individual control signals to high-level constructs which allow the interface into a complex hardware block.

Figure 1: Architecture Model shows a block diagram of the basic NPL architectural components and how they

relate to each other.

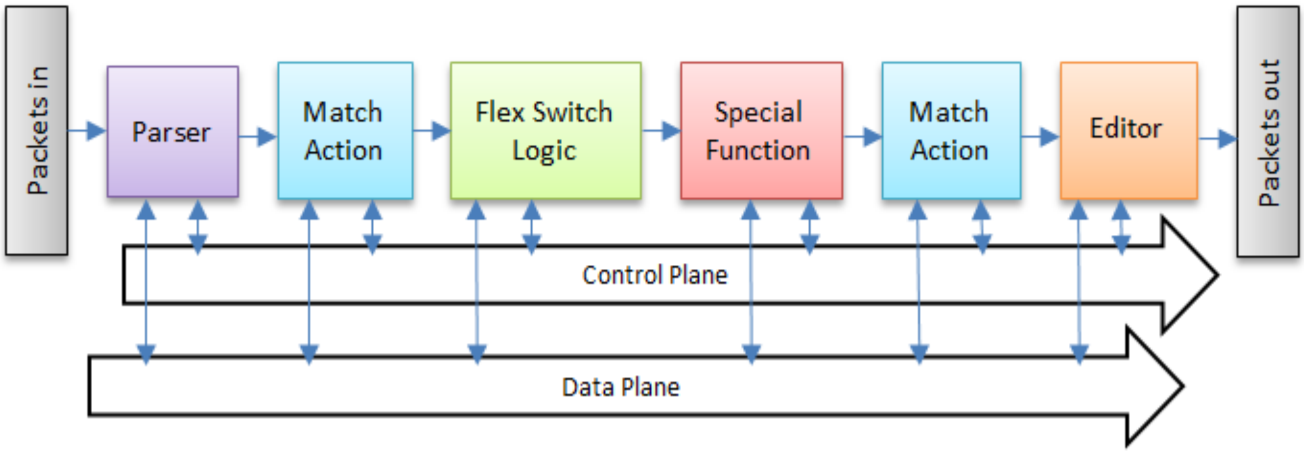



**Figure 1: Architecture Model**

Each functional block interacts with its neighbors by reading or writing one or more buses. A bus contains a number of fields specified using NPL. Logically speaking, buses flow through the blocks to form a pipeline. A block has the option of modifying bus fields as they pass through. For example, Match Action tables, Functions and Special Functions typically read and write bus fields. The Parser block takes the packet as input and writes parsed fields to the bus, while the Editor uses the bus fields to update or create an output packet.

An instance of this architecture model can be created with zero or more of these sub-components in arbitrary order.

The following sections describe the language constructs used to program these core abstractions, with examples.

# 4 NPL Language Components

## 4.1 Constructs Supported

This document is divided into the specific functional sections of a switch pipeline.

- Data Types
- Program Constructs
- Parser Constructs
- Bus Constructs
- Match Action Table Constructs
- Functions
- Editor Constructs
- Special Function Constructs
- Metadata for match action tables and parser

Definitions for NPL identifiers and constants, as well as the complete NPL grammar, are provided in the Appendix. Briefly:

- Identifiers must start with [a-z A-Z _] and can contain characters from [a-z A-Z _ 0-9]
- Decimal and hexadecimal literals
- String literals (e.g., “foobar”)

The remainder of this section provides details for each of the supported constructs.

## 4.2 Data Types

NPL has the following base data types: bit, varbit, list, const and auto_enum.  NPL has derived data type : struct.

### 4.2.1 bit type

The **bit** type is basic data type. It stores value 0 or 1. bit is used to describe fields in derived data types such as struct. bit is also used in logical_table, logical_register, special_function and other constructs.

#### 4.2.1.1 bit-array

A multi-bit field description would use the bit-array type to represent the field’s width. NPL imposes no restriction on the size of a bit array. Note that NPL does not support arrays of bit arrays.

**Example**

```
bit         cfi;          // specify single bit field
bit[3]      pri;          // specify 3 bit pri
bit[12]     vid;          // specify 12 bit vid
bit[128]    bit_map;      // specify a 128 bit map field

bit[8]      label[5];     // NOT allowed, cannot create an array of bit/bits
```

#### 4.2.1.2 bit-array Indexing

NPL allows static and variable indexing of arrays. Initially, support is limited to bit arrays.

Ensure that array size and index size match.

**Static Indexing of a bit-array**

An index is specified as unsigned integer values. It can be one bit or a range of bits.

**Example**

```
local.rst1 = local.rpa_id_profile1[3:2];
local.rst1 = local.rpa_id_profile1[0:0];
```

**Variable Indexing of bit-array**

Variable array indexing is usually used for bitmaps.

**Example**

```
local.rst1 = local.rpa_id_profile1[ip_tmp_bus.idx:ip_tmp_bus.idx];
```

### 4.2.2 varbit type

Varbit is used to specify a variable sized bit-array. A few networking protocols have fields in the headers which  can vary in size from packet to packet. varbit[X] is used to denote a variable whose width can be no more than X bits.

**Example**

```
varbit[120]     options; // options could be up to 120b wide.
```

### 4.2.3 const

const data type is used to denote a constant. integer or enum value.

**Example**

```
usage_mode_create(in const index,
                  in bit[2] in_pkt_color,
                  in varbit[14] meter_action_set,
                  in varbit[10] color_table_index0,
                  in varbit[8] color_pdd_sbr_index0,
                  out bit[2] color
                 );
```

### 4.2.4 list

Some NPL constructs may require combining a variable number of arguments into a list; for example, dynamic_table, strength_resolve and create_checksum. The list data type is used to represent such cases.

Syntax: Use curly braces to specify list datatype

```
{ipv4.protocol, ipv4.dip}
```

A list is used in the following constructs:

- dynamic_table to specify variable number of fields as inputs/outputs.
- update_checksum to specify a variable number of fields to perform the checksum.
- strength_resolve to specify a list of objects which generates a strength entry.

**Example**

The dynamic_table arguments use lists to provide the fields which can be used in the preselect template.

```
flex_digest_lkup.presel_template(
        {
        ing_cmd_bus.l2_iif_opaque_ctrl_id,
        ing_cmd_bus.vfi_opaque_ctrl_id,
        ing_cmd_bus.l2_iif_flex_digest_ctrl_id_a,
        ing_cmd_bus.l2_iif_flex_digest_ctrl_id_b,
        ing_cmd_bus.fixed_hve_iparser1_0,
        ing_cmd_bus.flex_hve_iparser1_1,
        ing_cmd_bus.fixed_hve_iparser2_0,
        ing_cmd_bus.flex_hve_iparser2_1,
        ing_cmd_bus.my_station_hit
        });
```

The create_checksum construct contains a list argument which provides a list of fields to be used in the checksum generation.

```
create_checksum(egress_pkt.fwd_l3_l4_hdr.udp.checksum,
        {
        egress_pkt.fwd_l3_l4_hdr.ipv4.sa,
        egress_pkt.fwd_l3_l4_hdr.ipv4.da,
        editor_dummy_bus.zero_byte,
        egress_pkt.fwd_l3_l4_hdr.ipv4.protocol,
        egress_pkt.fwd_l3_l4_hdr.udp.udp_length,
        egress_pkt.fwd_l3_l4_hdr.udp.src_port,
        egress_pkt.fwd_l3_l4_hdr.udp.dst_port,
        egress_pkt.fwd_l3_l4_hdr.udp.udp_length,
        egress_pkt.fwd_l3_l4_hdr.udp._PAYLOAD
        });
```

### 4.2.5 struct

Struct is used to specify an ordered aggregate of fields. Struct is the basic building block used in a variety of constructs. Only bit and struct are allowed inside a struct. (Refer to 6.1 struct for more information). Subsequent sections describe the usage of structs.

Struct allows overlays to reference fields in a struct in more than one way..

**Table 1: struct Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| struct | | Specify a new struct with, its name and fields. |

| | | |
|---|---|---|
| | fields | bit, bit[n], varbit and struct are allowed. No other data type or construct is allowed here. |
| | overlays | Specify the overlays among the fields of the struct. Only one "overlays" construct is allowed per struct. All the struct overlays are contained in the overlays construct. |

### 4.2.6 struct arrays

Single dimensional struct arrays are allowed in NPL. NPL imposes no size restrictions on struct arrays.

The following examples show supported usage of struct array.

```
obj_bus.struct1[arr1].field = field;
obj_bus.struct1[arr1].struct2[arr2].field = field;
cmd_bus.struct1 = obj_bus.struct1[arr];
```

**Example**

**SPECIFY A SIMPLE STRUCT**

```
struct vlan_s {
    fields {
        bit            cfi;    // specify single bit field
        bit[3]         pri;    // specify 3 bit pri
        bit[12]        vid;    // specify 12 bit vid
    }
}
```

**SPECIFY A STRUCT WITH OVERLAYS**

```
struct  switch_bus_s {
    fields {
        bit[4]       otpid_enable;
        bit          olp_enable;
        bit          ts_enable;
        bit[10]      ing_port_num; // Base field for the overlay fields, defined later.
        bit          svp_enable;
    }
    overlays {
        ing_svp :       ing_port_num[7:0];
        ing_pri :       ing_port_num[9:8]; // multiple overlays on same base field
        exp     :       ing_port_num[9:8];
    }
}
```

**SPECIFY AN ARRAY OF STRUCT**

```
struct mpls_header_stack_t {
    fields {
        mpls_t mpls[3];   // means 3 mpls_t headers can be there.
    }
}
```

The members of the array can be referenced as mpls[0], mpls[1], mpls[2].

### 4.2.7 enum

NPL supports the enum construct for defining enumeration types. Values of enum members must be provided by the user. NPL enums are simple identifier lists providing a subset of what is provided in C/C++. enum is not a datatype in NPL. It is used to represent constants. enum defined constants are used as arguments in function calls and rvalue in an assignment.

**Example**

```
enum drop_reason{
    NO_DROP = 0,
    MEMBERSHIP_DROP = 1,
    TTL_DROP = 2
}
packet_drop(drop_bus.disable_drop, drop_reason.TTL_DROP, 5);
```

### 4.2.8 auto_enum

NPL supports auto_enum data type for defining enumeration types. Values of auto_enum members is assigned by compilers. Target vendor may decide how to map and assign values of auto_enum.

Typical usage for auto_enums are Logical Table Lookup, Multi-Data View and Strength Based Resolution Index.

Target compiler may assign auto_enum values based on the context in which they are used.  Thus auto_enums must be global and used in single instance. For example, auto_enums used in logical table lookup cannot be used in special functions.

**Example**

```
auto_enum qos_entry {
    QOS_DISABLE,
    QOS_L3_TUNNEL,
    QOS_L2_TUNNEL
}
qos_sfc.sf_profile_entry("sfc_qos_profile", qos_entry.QOS_L3_TUNNEL,
    {
    obj_bus.mapping_ptr,
    cmd_bus.effective_exp
    },
    {
    cmd_bus.int_pri,
    cmd_bus.pri
    } );
```

## 4.3 Expressions

### 4.3.1 Number Notation

NPL allows only decimal and hex literal constants. Number notation is used to assign a value to a field. NPL does not provide a bool data type, a value of zero indicates false, a non-zero value indicates true.

**Example**

```
a = 5;                              //decimal
ipv4.ttl = 0xF;                     //hex
ipv6.dip = 0x01234567;
ipv6.dip[63:0] = 0x0123456789abcdef;.
if (ipv4.protocol == 0x23)
ipv4.protocol = 0x231;
```

### 4.3.2 Conditional Statements

NPL supports conditional statements in multiple constructs.

Here is the list of supported conditional statements:

**Table 2: Conditionals in NPL**

| *Conditional* | *Description* |
|---|---|
| if, else if, else | If statements |
| switch | Switch statement |

### 4.3.3 Operators

NPL allows multiple operations. Usage restrictions are described in the various construct sections.

**Table 3: Operators in NPL**

| *Operator* | *Symbol* | *Description* |
|---|---|---|
| Arithmetic operators | + | Addition |
| | - | Subtraction |
| | * | Multiplication |
| | / | Division |
| | % | Modulus |
| Relational operators | == | Equal |
| | != | Not Equal |

| | | |
|---|---|---|
| | < | Less than |
| | <= | Less than or equal to |
| | > | Greater than |
| | >= | Greater than or equal to |
| Concatenation operator | <> | Concatenation |
| Logical operators | && | Logical AND |
| | \|\| | Logical OR |
| Shift operators | << | Shift left |
| | >> | Shift right |
| Bitwise Logical operators | & | AND |
| | \| | OR |
| | !A | Logical NOT |
| | ~A | One's Complement |
| | ^ | XOR |
| Unary operators | &A | Reduction AND (all bits 1) |
| | \|A | Reduction OR (all bits 0) |
| Assignment operators | = | Assignment - Note that the left and right hand sides of an assignment are both sized.<br>If the lvalue is larger than the rvalue then the rvalue is zero extended to the target.<br>If the lvalue is smaller than the rvalue then the compiler shall issue an error. |
| Mask operator | mask | Used in a switch statement case, treated as an AND. |

### 4.3.4 Variable Scope

#### 4.3.4.1 Global Scope

In NPL, the following variable names are in global namespaces. Unique names must be used while specifying these:

- enum
- auto_enum
- struct
- bus
- packet
- logical_table
- logical_register
- parser_node
- function
- special_function

- dynamic_table
- strength

#### 4.3.4.2 Local Scope

Multiple constructs use local scope to specify member names.

- logical_table - fields, keys
- struct - fields, structs
- logical_register - fields
- enum - elements
- auto_enum - elements
- special_function - methods
- dynamic_table - methods

## 4.4 Program Construct

A program represents the NPL application. The name of the program is the entry point, analogous to "main" in C. The program construct defines the order of execution for the other constructs of a packet processing pipeline. Familiar if-then-else syntax is used to specify conditional control flow.

A program may call the following constructs using the keywords in Table 4: Program (Order of Execution) Construct.

- Parser Tree
- Table Lookups (logical_table, dynamic_table)
- Packet Processing Functions
- Special Function
- Strength Resolution

Assignments are not allowed in the Program construct.

These constructs are described in subsequent sections of this document.

**Table 4: Program (Order of Execution) Construct**

| ***Construct*** | ***Arguments/Options*** | ***Description*** |
|---|---|---|
| program | | Specify the order of execution of packet processing components. |
| | conditional | Above mentioned constructs may be called under conditional statements within program.<br>● if/else/else if and switch are allowed.<br>● All NPL Operators are allowed. |
| | parse_begin(<node name>) | Execute a parser tree from the root node.. e.g. for loopback processing we want a different parsing tree than Ethernet. |

| | | |
|---|---|---|
| | parse_continue(<node name>) | Return control back to parser tree traversal to the node specified as the argument. |
| | <table>.lookup(<lookup_num>) | Lookup a table. Automatically, the methods associated with that table are also executed. |
| | packet processing function calls | Calls of packet processing functions are allowed. |
| | special function calls | Calls for special function constructs are also allowed. |
| | strength resolution | Calls for strength resolution among multiple tables. |

**Example**

```
program mim_main ()  {
     parse_begin (ethernet);   /* start parsing from "parser_node" Ethernet. */
     port.lookup (0);          /* lookup the port table. */
     iif.lookup (0);           /* lookup the iif table. */
     my_station.lookup (0);    /* lookup the my_station table. */
     isid.lookup (0);          /* lookup the isid table */
     mim_isid_switch_logic1(); /* execute logic function. */
     ...
     if (cmd_bus.do_l3) {      /* conditional lookup. */
         l3_host.lookup (0);
     }
     ...
     l3_switch_logic1();       /* packet processing function call */
     ...
     next_hop.lookup (0);     /* lookup next_hop table. */
     do_packet_edits();        /* editor function. */
}
```

## 4.5 Parser Constructs

The Parser constructs define:

- Header, consists of an ordered set of fields with fixed or variable lengths
- Header group, consists of different header
- Packets, consists of header groups
- Connectivity among the header types, to form the parse tree

NPL allows specifying basic protocol header types using struct. A parser specification would use header types to declare header group level structs. A parser specification would construct a packet level struct using header groups.

NPL programs must have packets defined as packet.header_group.header.

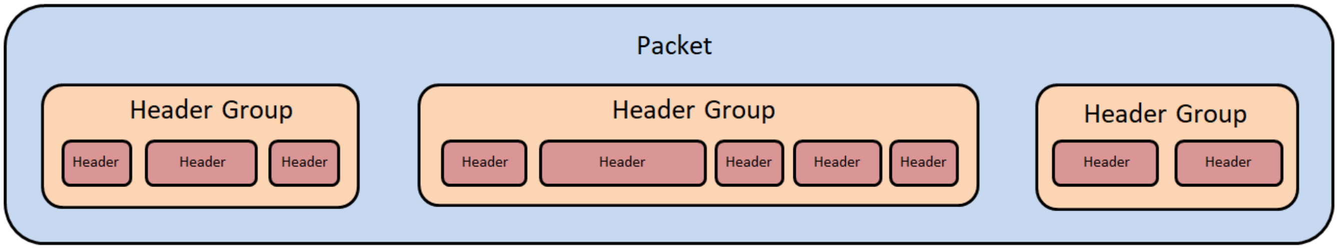



**Figure 2: Packet, Header Group, Header Representation**

### 4.5.1 Header (struct)

Header is defined using struct data type. Header fields are defined using bit and bit-array.

Two additional options are available:

- varbit - to specify a variable length field.
- header_length_exp - to specify header length in case of varbit usage.

In all other usage of struct these two additional options are invalid.

NPL allows header array.

**Table 5: Header (struct) Construct**

| ***Arguments/Options*** | ***Description*** |
|---|---|
| struct | Specify a new header type |
| varbit | If one of the fields is of variable length,<br>● Use varbit to declare it.<br>● Specify the max possible field length with varbit.<br>● There can only be one varbit in a header.<br>● The varbit must be the last field.<br>● NPL imposes no size restrictions on length. |
| header_length_exp | For variable lengths, specifies the equation with which the length can be determined.<br>Operations supported in the equation are +,*.<br>This expression should take the form: var * c0 + c1<br>Where var is a header field and c0 and c1 are constants.<br><br>For static length headers this field is not required. |

**Example**

**SPECIFY A STATIC HEADER**

```
struct vlan_t {
    fields {
        bit[3]       pcp;
        bit          cfi;
        bit[12]      vid;
        bit[16]      ethertype;
    }
}
```

Specifies width and the order in which the fields pack in the header.

**SPECIFY A VARIABLE LENGTH HEADER**

```
struct ipv4_t {
    fields {
        bit[4]          version;
        bit[4]           hdr_len;
        bit[8]           tos;
        bit[32]          sa;
        bit[32]          da;
        varbit[320]      option;    // specify max allowed size of the variable length field
    }
    header_length_exp:    hdr_len*4; // specifies how to compute the number of bytes in the header.
}
```

In this example:

- ”option" is the variable length field.
- "header_length_exp" specifies how to compute the length of the header.
- header_length_exp : (payload_len*4)+2; //another example
- Arithmetic operators allowed : (+,*)
- varbit[num] - num specifies the maximum allowed width for the field.
- Each Variable length header must have a header_length_exp attribute.
- A struct with multiple variable length fields is not supported. Specify individual structs in that case.

## 4.5.2 Header Group (struct)

Header Group is used to combine a group of headers. Header group is specified using struct. Header group level structs are packed to specify a packet (see 4.6.3. Packet Construct). No additional options are allowed. Header groups are required to define a packet in the NPL. A header group level struct allows mass manipulation/reference of constituent headers in a packet. Headers are packed in a header group in the order specified in NPL.

- A header group struct must only instantiate header structs. bit/bit-array are not allowed.
- Arrays of header groups are not allowed.
- Headers and header groups must be listed in the sequence in which they would appear in a packet. This sequence is important.

**Example**

**SPECIFY A HEADER GROUP WITH MULTIPLE STRUCTS (HEADER)**

```
struct l2_header_t {
    fields {
        bit[48]     macda;
        bit[48]     macsa;
        bit[16]     ethertype;
    }
}

struct vlan_tag_t {
    fields {
        bit[3]     pcp;
        bit        cfi;
        bit[12]    vid;
    }
}

struct group0_t {
    fields {
        l2_header_t    l2_header;
        vlan_tag_t     ovlan;
    }
}
```

The header group struct specifies the order in which headers are packed in a header group.

**INSTANTIATING AN ARRAY OF STRUCT (HEADER)**

```
struct group1_t {
    fields {
        mpls_t mpls[3];   // means 3 mpls_t headers are there.
    }
}
```

The members of the array can be referenced as mpls[0], mpls[1], mpls[2].

### 4.5.3 Packet Construct

Packet consists of Header Groups. In NPL, packet is struct of header groups.

Packet is instantiated using keyword **packet** :

**packet** *struct-name instance-name*;

This statement declares a packet with the name *instance-name* which is described by the struct declared as *struct-name*. The associated struct (*struct-name*) is referred to as a packet level structure. The members of a packet level structure must be header group structures.

Packet cannot contain bit, bit-array members. Packet instances cannot be arrays.

A packet level struct is used to aggregate header groups into a packet. The header groups must follow the order in which the groups appear across all packets.

**Example**

DEFINE AND INSTANTIATE A PACKET LEVEL STRUCT

```
struct macs_t {    // Header structure because member of header group structure and
                   // all members are bit-arrays
    fields {
        bit[48] dmac;
        bit[48] smac;
    }
}
struct vlan_t {    // Header structure
    fields {
        bit[16] tpid;
        bit[3]  pcp;
        bit     dei;
        bit[12] vid;
    }
}
struct ethertype_t {    // Header structure
    fields {
        bit[16] type;
    }
}
struct mpls_t {    // Header structure because member of header group structure and
                   // all members are bit arrays
    fields {
      bit[20] label;
      bit[3]  tc;
      bit     s;
      bit[8]  ttl;
    }
}
struct mpls_grp_t {    // Header group structure
    fields{
       mpls_t mpls[3];
    }
}
struct ipv4_t {    // Header structure
    fields {
        // IPv4 field definitions (bit-arrays only)...
    }
}
struct ipv6_t {    // Header structure
    fields {
        // IPv6 field definitions (bit-arrays only)...
    }
}
struct l2_t {     // Header group structure because member of packet level structure and
                  // all members are structures
    fields {
        macs_t macs;
        vlan_t ctag;
```

```
        ethertype_t etype;
    }
}
struct l3_t {    // Header group structure
    fields {
        ipv4_t ipv4;
        ipv6_t ipv6;
    }
}
struct ingress_packet_t {    // Packet level structure because instantiated as packet (below)
    fields {
        l2_t        l2;
        mpls_grp_t  mpls_grp;
        l3_t        l3;
    }
}
packet  ingress_packet_t  ing_pkt; // This declaration identifies ingress_packet_t as a
                                   // packet level structure
```

Header and fields in a packet should be referenced according to the following example:

```
ing_pkt.l2.macs.da
```

Header arrays should be referenced according to the following example:

```
ing_pkt.mpls_grp.mpls[0].label
```

Target may have restrictions on Packet access and modification. For example, for a split Ingress and Egress target architecture, there may be 2 packets, ingress packet and egress packet. Ingress packets may be read-only and can not be modified. All Packet modifications are performed on egress packets.

### 4.5.4 Header Metadata

Each header has a 1 bit "_PRESENT" metadata field associated with it.

The _PRESENT bit indicates whether a particular header instance is valid in the current packet. This makes _PRESENT a dynamic metadata. If in an incoming packet, a header is present, then _PRESENT is evaluated as 1.

_PRESENT is a read-only metadata field and will maintain its state during the life of the packet.

**Example**

```
struct tcp_t {
    fields {
        ...
    }
}

struct group0_t {
    fields {
        tcp_t tcp;
```

```
        ...
    }
}

struct packet_t {
    fields {
        group0_t group0;
        ...
    }
}

packet packet_t ing_pkt;

program l3 () {
    ...
    if (ing_pkt.group0.tcp._PRESENT) {
        l2_table.lookup(0);
    }
    ...
}
```

### 4.5.5 Parse Tree Connectivity (parser_node)

The parser_node construct specifies how the header instances are connected to each other in packets. Essentially, it specifies the parsing nodes and transitions. Conditional operations are supported in parser_nodet to specify the connection to the next parser_node.

**Table 6: parser_node Construct**

| ***Construct*** | ***Arguments/Options*** | ***Description*** |
|---|---|---|
| parser_node | | Specify a parser node and its connectivity to the next parser nodes. |
| | name | Name of this parser_node |
| | root_node | There can be only one root node in a parser tree. parse_begin() function can reference root node only. In order to break/re-enter the parser tree, parse_break/parse_continue should be used. |
| | next_node <node> | Specify the next header that this parser_node is connected to. |
| | switch | Specify a switch statement to describe the condition to reach to the next parser_node. The keyword "mask" can also be used in the "switch" statements. Field bit-array are allowed in the switch statement. The case value in a switch statement can be a constant or a constant with mask value which is ANDed with it during comparison. |

| | | |
|---|---|---|
| | if/else | Supports if/else/else if to specify the equations to reach to the next node.<br>● Comparison operators supported are (==, !=) if rvalue is a Constant.<br>● Logical Operation supported - &&, \|\|, ! |
| | extract_fields(packet.header) | Specify which header instance to parse. Current packet parsing position moves beyond this header. |
| | parse_break(<node>) | Specify that control will break from parser tree and go back to program. Node specified as argument is the next node to resume from, once control returns to parser tree. |
| | end_node | Specify that this is the last node of this tree. |
| | latest | Refers to the last header parsed in this parser_node The latest.field name can be used in the parser tree. |
| | current | Refers to the ongoing packet bytes to specify an equation. For example, to look at the next 2 bytes in the packet to determine the way forward, use "current".<br>When extract_fields is called, the “current” pointer is moved beyond the parsed header.<br>● current (starting bit offset, number of bits to pick) |
| | default | In a switch statement, "default" is allowed to cover all other cases. |

**Example**

SPECIFY A PARSER TREE

```
struct vlan_t {
    fields {
        bit[3]      pcp;
        bit[1]      cfi;
        bit[12]     vid;
        bit[16]     ethertype;
    }
}

struct l2_t {
    fields {
        bit[48]     macda;
        bit[48]     macsa;
        bit[16]     ethertype;
    }
}

struct group1_t {
```

```
    fields {
        l2_t         l2;
        vlan_t       vlan;
    }
}

struct ing_pkt_t {
    fields {
        group1_t group1;
    }
}

parser_node start {
    root_node : 1;
    next_node ethernet;
}

parser_node ethernet {
    extract_fields(ing_pkt.group1.l2);
    switch (latest.ethertype) {
        0x8100       : {next_node ctag};
        default      : {next_node ingress};
    }
}

parser_node ctag {
    extract_fields(ing_pkt.group1.vlan);
    if (current(0,16) == r_ing_outer_tpid_0.tpid) {
        next_node otag;
    }
    next_node ingress;
}

parser_node ingress {
   end_node : 1;
}
```

## 4.6.6. Re-entrant Parse Tree (parse_break / parse_continue)

NPL provides mechanism to perform table lookups or other packet processing during parsing. This may be required when some decision in parser node requires output from table lookup. Thus flow needs to break from parser tree traversal and return control to program and then continue from the same node after the lookup is done. This is done using parse_break and parse_continue constructs.

**Example**

Perform a table lookup before parsing ethernet packet and mpls packet.

```
program mpls_switch() {
    parse_begin(start);
    port_table.lookup(0);
```

```
    // ethernet node consumes output from port_table i.e logical_bus.otpid_enable
    parse_continue(ethernet);
    // mpls_label node consumes output from mpls_table i.e logical_bus.mpls_table_result_type
    mpls_table.lookup(0);
    parse_continue(mpls_label);
}
parser_node start {
    root_node : 1;
    switch(logical_bus.rx_port_parse_ctrl) {
        0x0:    next_node ppd;
        0x2:    next_node sobmh;
        0x3:    parse_break(ethernet);
        default: next_node ingress;
    }
}
parser_node ethernet {
    extract_fields(ingress_pkt.outer_l2_hdr.l2);
    if (logical_bus.otpid_enable[3:3] && latest.ethertype == 0x8100) {next_node otag;} //0x8100
    if (logical_bus.otpid_enable[2:2] && latest.ethertype == 0x8100) {next_node otag;} //0x8100
    if (logical_bus.otpid_enable[1:1] && latest.ethertype == 0x8100) {next_node otag;} //0x8100
    if (logical_bus.otpid_enable[0:0] && latest.ethertype == 0x8100) {next_node otag;} //0x8100
}

parser_node mpls_0 {
    extract_fields(ingress_pkt.outer_l3_l4_hdr.mpls[0]);
    switch (latest.stack) {
        0x0: next_node mpls_1;
        0x1: parse_break(mpls_label);
        default: next_node ingress;
    }
}

parser_node mpls_label {
    extract_fields(ingress_pkt.outer_l3_l4_hdr.mpls[4]);
    switch (logical_bus.mpls_table_result_type) {
        0x0: next_node mpls_cw;
        0x1: next_node inner_ethernet;
        0x2: next_node inner_l3_speculative;
        default: next_node ingress;
    }
}

parser_node ingress {
    end_node:1;
}
```

**Note:**

end_node:1 marks the end of parsing, once control reaches here parse tree traversal will end.

# 4.6 Logical Bus Constructs

Logical bus constructs are used to define collection of fields (variables).

## 4.6.1 Bus Definition

Use a struct to define the fields and overlays that make up a logical bus.

Field ordering is maintained in the logical bus as defined in the struct.

## 4.6.2 Bus Instantiation (bus)

The bus construct is used to declare a logical bus by instantiating a bus definition struct. Note that a bus may have overlay fields that can be individually referenced in the NPL program.

**Example**

INSTANTIATE A BUS

```
struct control_bus_t {
    fields {
        bit    ts_enable;
        bit    olp_enable;
        bit[4] otpid_enable;
    }
}

bus  control_bus_t   control_id;

parser_node pkt_start{
    root_node : 1;
    next_node ethernet;
}

parser_node ethernet {
    extract_fields(ing_pkt.group0.l2);
    if (control_id.ts_enable == 0) {  // control_id is a logical bus.
                                      // ts_enable is a field in the bus.
        if (control_id.otpid_enable != 0 ) {
            switch (latest.ethertype) {
                0xABCD             : {next_node vntag};
                0x8888             : {next_node etag};
                0x8100             : {next_node otag};
                0x9100             : {next_node itag};
                0x0000 mask 0xFC00  : {next_node llc};
                default            : {next_node payload};
            }
        }
    }
}
```

```
parser_node otag{
    extract_fields(ing_pkt.group0.ovlan);
    end_node:1;
}
```

## 4.7 Logical Table Constructs

The logical table constructs are used to define a table with keys, fields, key_construct, fields_assign, table type minsize and maxsize. Logical tables also have a built-in lookup() method.

### 4.7.1 Logical Table (logical_table)

The logical_table construct is used to declare a match action table. It allows user to define a data structure which control-plane or data-plane can modify. User can specify key and policy fields to be stored in the logical_table. User can also specify key construction mechanisms using logical bus fields. Finally logical_table allows user to specify fields_assign method to handle fields.

Logical table Keys and Fields are locally scoped. NPL logical tables can be looked-up multiple number of times, depending on the target architecture capabilities.

All declared logical tables must be called with the <logical table name>.lookup(lookup_num) construct. lookup_num = 0 means first lookup and so on.

**Table 7: logical_table Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| logical_table | | Specify a new table |
| | table_name | Specifies the name of the table. |
| | table_type | Specifies the type of table from the user perspective. The compiler may map these on different tiles. Valid values - index, tcam, hash, alpm. More types may be added by target device. |
| | keys | Specifies the keys used to access the logical table. Define key widths using the bit declaration.<br>● bit and bit array are allowed here.<br>● structs are not allowed here. |
| | fields | Specifies the policy fields of this logical_table. Define field widths using a bit declaration.<br>● bit, bit array and auto_enum are allowed here.<br>● structs are not allowed here<br>● auto_enum are used to specify multiple data views. |

| | |
|---|---|
| key_construct() | Specifies the logic to construct the logical table keys.<br>Rules to generate logical table keys:<br>● Can support multiple lookups on same table. Refer to 4.8.3. Multiple Lookups on Same Logical Table<br>● For multiple logical table lookups, conditional statements with metadata _LOOKUP0 and _LOOKUP1 and so on, are allowed.<br>● Constructed from bus fields. |
| fields_assign() | Method to describe the functionality to process and assign fields to logical bus.<br>● Can support multiple lookups on same table. Refer to 4.8.3. Multiple Lookups on Same Logical Table<br>● For multiple lookups on this logical table, conditional statements with metadata _LOOKUP0 and _LOOKUP1 are allowed.<br>● Assignments can additionally be bounded by _VALID and multi-view(auto_enum)<br>● No other conditions or operations are allowed in fields_assign block at this time. |
| minsize | Minimum guaranteed size. Physical table must have this number of entries. |
| maxsize | Maximum allowed size. This is mainly for SDK populate purposes, in case maxsize and minsize have different values.<br>If minsize and maxsize are the same, then it is treated as constant "size" for that table. The compiler must find a physical table of the specified size. |

**Example**

**DEFINE AN INDEX TABLE**

```
logical_table port {
    table_type : index;
    minsize : 128;
    maxsize : 128;
    keys {
        bit[7] port_num;
    }
    fields {
        bit[1]  l3_enable;
        bit[1]  otag_enable;
        bit[8]  src_modid;
        bit[12] default_vid;
    }
```

```
    key_construct() {
        port_num = obj_bus.port_num;
    }
    fields_assign() {
        if (_LOOKUP0 == 1) {
            cmd_bus.port_l3_enable = l3_enable;
            ...
        }
    }
}
```

### DEFINE A TCAM TABLE

```
logical_table my_station_hit {
    table_type : tcam;
    maxsize : 512;
    minsize : 512;
    keys {
        bit[48] macda;
        bit[12] vid;
        bit[8]  src_modid;
    }
    fields {
        bit[2] mpls_tunnel_type;
        bit    local_l3_host;
    }
    key_construct() {
        macda     = ing_pkt.l2_grp.l2.macda;
        vid       = obj_bus.vlan_id;
        src_modid = obj_bus.source_logical_port;
    }
    fields_assign() {
        if (_LOOKUP0 == 1) {
            l3_cmd_bus.local_l3_host = local_l3_host;
            ...
        }
    }
}
```

### Calling a TABLE

```
program ingress {
     port.lookup(0); //calls port logical table
     if (cmd_bus.vlan_valid == 1) {
        my_station_hit.lookup(0); //calls my_station_hit logical table first time
        my_station_hit.lookup(1); //calls my_station_hit logical table second time
     }
}
```

### 4.7.2 Logical Table Metadata

In NPL, each logical table has the following metadata. For each packet, the metadata value is implicitly assigned using the following rules.

- _LOOKUPx - 1 bit value, set when a table is looked up for a packet.
- _HIT_INDEXx - 32 bit value. It indicates the specific table entry that a packet matched. Format of _HIT_INDEXx can vary based on target. Must have 1b to indicate if lookup hit a valid entry or not.
- _VALID - 1 bit value. It is set, if lookup hits a valid entry.

where "x" represents lookup_num (0, 1 and so on).

**Example**

```
logical_table table_a {
    ...
    fields_assign() {
        if (_LOOKUP0) {
            obj_bus.src_hit_index = _HIT_INDEX0;
        }
        if (_LOOKUP1) {
            obj_bus.dst_hit_index = _HIT_INDEX1;
        }
    }
}
```

As with the header metadata, it increases application readability and provides a basis for instrumentation.

### 4.7.3 Multiple Lookups on Same Logical Table

NPL allows specifying multiple lookups on the same logical_table. With multiple lookups, _LOOKUP0, _LOOKUP1 and so on may be used to distinguish keys and fields handling in the `key_construct() and fields_assign()` block.

**Example**

```
//define a logical table
logical_table mac_table {
    table_type : hash;
    minsize : 64;
    maxsize : 64;
    keys {
        bit[48] macda;
    }
    fields {
        bit[16] port;
        bit[1]  dst_discard;
        bit[1]  src_discard;
    }
    key_construct() {
        if (_LOOKUP0==1) {
            macda = ing_pkt.l2_grp.l2.da;
```

```
        }
        if (_LOOKUP1==1) {
            macda = ing_pkt.l2_grp.l2.sa;
        }
    }
    fields_assign() {
        if (_LOOKUP0==1) { //e.g. Entry 100
            obj_bus.dst = port;
            obj_bus.dst_discard = dst_discard;
        }
        if (_LOOKUP1==1) { //e.g. Entry 200
            temp_bus.src_port = port;
            obj_bus.src_discard = src_discard;
        }
    }
}

program {
    if ((ing_pkt.l2_grp.l2._PRESENT) & (ing_pkt.l2_grp.vlan.vid != 0)) { //Condition is supported.
        mac_table.lookup(0);
        mac_table.lookup(1);
    }
}
```

## 4.7.4 Multiple Data Types (Data Width Modes)

Within logical tables, fields can be packed in multiple formats. These formats may be required due to width reasons or overlaying different information. This is done to achieve higher utilization.

NPL writer must specify all the fields, across different data types, in the fields{} construct.

For example, Logical Table NHI has following 2 Data Views.
Data View 1 - Fields A, B, C
Data View 2 - Fields A, D, E, F

NPL _VALID Rules - (aka hit):

- If a Logical Table has Multiple Data Types, there will be a single _VALID (_VALID = 0) occurrence.
- If some fields have Strength, then they must be specified in the _VALID=0 section within the `fields_assign()` block. Strength is called only for _VALID=1 cases.

**Example**

```
auto_enum multi_view {
    UC_VIEW,
    MC_VIEW,
    BC_VIEW
}

logical_table NHI {
    ...
    fields {
        bit[3]  A;
        bit[15] B;
```

```
        bit[7]  C;
        bit[10] D;
        bit[4]  E,
        bit[4]  F;
        bit[16] strength_object_G;
        multi_view  X; //the data_type field for denoting different views. Use auto_enum.
    }
    fields_assign() {
        if (_LOOKUP0 == 0) {
            if (_VALID == 1) { //_VALID is same as hit.
                if (X == UC_VIEW) {
                    bus.A = A;
                    bus.B = B;
                    bus.C = C;
                }
                if (X == MC_VIEW) {
                    bus.A = A;
                    bus.D = D;
                    bus.E = E;
                    bus.F = F;
                }
            } //end _VALID == 1
            else { //_VALID == 0 //specify only data_type = 0 fields.
                bus.A = 0;
                bus.B = 0;
                bus.C = 5; //example of a non-zero constant
                bus.G = 0;
            }//end _VALID == 0
        }
        if (_LOOKUP1 == 1) {
            ...
        }
    }
}
```

## 4.8 Logical Register Constructs

The Logical Register construct is used to specify a one deep, multi-field entity. The logical_register provides an interface to software to setup controls. Unlike logical table, a logical register has no lookup key. Fields can be initialized at compile time, or populated at run time.

### 4.8.1 Define Single Level Storage

The logical_register construct is used to define a single, deep, multi-field entity. The result is always available for the calling function. No index is required. logical_registers cannot be used to maintain state across packets. They are only populated by control-plane configuration.

A logical_register is often useful in the parse tree, functions, etc. They can be used in lieu of constants, which may be configurable by control-plane.

**Table 8: logical_register Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| logical_register | | Specify a new register. Logical register could be of any width. |
| | fields | Specifies the data fields of this logical_register. Define the width of the fields using bit declaration.<br>reset value must be specified for each field |

**Example**

DEFINE A LOGICAL_REGISTER

```
// For example, this construct can be used to specify values like TPID.
logical_register tpid_values {
    fields {
      bit[16] tpid0 = 0x8100;
      bit[16] tpid1 = 0x9100;
      bit[16] tpid2 = 0x7100;
      bit[16] tpid3 = 0x8868;
    }
}
```

Specifies the width and reset value of each field.

## 4.9 Packet Processing Function (function)

Functions are used in NPL, to describe generic packet processing, to process results of parser, logical tables, special_function and other constructs. Functions are imperative constructs, which can do data transformation and support application modularity. Multiple other NPL constructs can be invoked inside functions.

Function allow conditional statements, assignment statements and complex operations for data transformation of the logical buses. Functions can access and update logical bus data. Functions can be nested within other functions. Declarations are not allowed inside functions.

Here are a few scenarios where function could be used. Functions can be used to implement flexible decision logic. For example, decode lookup results to determine whether a packet is unicast or multicast.  Function can be used to extract packet data. Function can be used to invoke logical table lookups, special functions.

Function can also be used to specify application modularity. For such usage, function may have logical table lookups, strength resolve, special function calls, dynamic table calls etc. All items which can be specified in "program" construct can be specified in such function. It is suggested to maintain separate NPL functions for modularity and packet processing.

Target may choose to limit scope and usage of function depending on hardware architecture.

## Table 9: function Construct

| function | Specify a new packet processing function |
|---|---|
| function_name | Name of the function |
| | All conditional, arithmetic and logical operators are allowed. Manipulation of all logical bus are allowed. parser, logical_table lookup, special_function, editor, strength invocation is allowed. |

### Example

```
// Bus to be used internal to functions.
struct switch_logic_t {
    fields {
        bit no_l3_switch;
        bit l2_same_port_drop;
    }
}

// Logical registers can be accessed in functions.
logical_register cpu_control {
    fields {
        bit   tunnel_to_cpu = 0; // Initialized to zero
    }
}

bus switch_logic_t temp;

function l3_switch_logic1 () {
     temp.no_l3_switch = 0;
     if (port.l3_enable &&
          (ingress_pkt.outer_l3_l4_hdr.ipv4._PRESENT || ingress_pkt.outer_l3_l4_hdr.ipv6._PRESENT)
        ) {
        if (obj_bus.tunnel_pkt || obj_bus.tunnel_error) {
            if (obj_bus.tunnel_error) {
                obj_bus.tunnel_decap = 0;
                temp.no_l3_switch = 1;
                if (cpu_control.tunnel_to_cpu) {
                    obj_bus.copy_to_cpu = 1;
                }
            } else {
                obj_bus.tunnel_decap = 1;
            }
        } else { //Not a tunneled pkt
             obj_bus.tunnel_decap = 0;
        }
     }
     // Trace packets
```

```
    packet_trace(temp.no_l3_switch, cpu_reason.NO_SWITCH);

    temp.l2_same_port_drop = obj_bus.src_prune_en && (obj_bus.l2_oif == obj_bus.l2_iif);
    // Drop packets
    packet_drop(temp.l2_same_port_drop, drop_reason.L2_SAME_PORT_DROP, L2_SAME_PORT_DROP_STR);
  }
```

# 4.10 Editor Constructs

Editor constructs are defined for the following:

- Add a new header. The fields of new header are constructed using function.
- Remove a header from the packet.
- Modify header fields from logical bus.

The modified packets (after editing) must conform to one of the packets described in the parsing graph. Thus, editor commands must use the same names for headers as in the parsing tree.

Construction of new headers is not in scope of the editor constructs. Editor constructs must be invoked from within a function.

Ingress packets are read only and cannot be modified.  Packet modifications are performed on egress packets.

The editor constructs can only be used in function constructs.

## 4.10.1 Add a Header

A new header is constructed using functions. After a new header is constructed, it is added to the packet. The editor compiler will recognize and work on it.

**Table 10: add_header Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| add_header | | Specify addition of new header to the packet. |
| | new_header_name | Specify the new header to be added. This is the "same" name as in the packet specification. |

**Example**

**ADD A HEADER WITHIN THE PACKET**

If an application needs to add an otag and construct it before calling add_header(otag), do the following;

```
egr_pkt.group1.otag.vid = ing_pkt.itag.vid+100; /*both otag and itag are defined as headers in the
packet construct.*/
egr_pkt.group1.otag.pcp = obj_bus.egr_port_table_pcp; /*from the object bus.*/
egr_pkt.group1.otag.tpid = 0x9100;
add_header(egr_pkt.group1.otag);
```

**ADD A TUNNEL HEADER OUTSIDE OF THE PACKET**

If the application is trying to add a Tunnel L2 and VLAN ID -

```
egr_pkt.group2.tunnel_l2.dmac = 0xff;
egr_pkt.group2.tunnel_l2.smac = obj_bus.l3_interface_smac;
egr_pkt.group2.tunnel_l2.vid  = obj_bus.l3_next_hop_vid;
add_header(egr_pkt.group2.tunnel_l2);
```

## 4.10.2 Delete a Header

Deletes a header from the packet header stack. It is used in certain applications, such as in an egress edge switch, to remove the tunnel header.

**Table 11: delete_header Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| delete_header | | Specify deletion of a header from the packet. Use parse specification to identify the location of the header. |
| | header_name | Specify the header or header group to be deleted. This will be the "same" name as in the packet level structure. |

**Example**

If an application is trying to remove an otag from the packet:

```
delete_header(egr_pkt.group1.otag);
```

This works on this instance of the packet and removes otag from this packet. It has no impact on the parsing tree.
If an application is trying to remove a header group from the packet:

```
delete_header(egr_pkt.group1);
```

This works on this instance of the packet and removes header group1 from this packet. It has no impact on the parsing tree.

## 4.10.3 Rewrite a Header

For certain protocols, packet processing needs to modify just the header fields.

**Table 12: replace_header_field Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|

| | | |
|---|---|---|
| replace_header_field | | Replace a header field with a bus field or another header field. |
| | dest_field | Specify the destination header field to be modified. |
| | src_field | Specify the source field which will be used to modify the destination field. Allowed fields are bus.field and header.field. |

**Example**

Say an application would like to modify the dscp field.

```
replace_header_field(egr_pkt.ipv4.dscp, obj_bus.new_dscp);
```

### 4.10.4 Create Checksum

The Create Checksum construct can only be used in function constructs.

The create checksum supports TCP/UDP checksum calculations.

Create Checksum uses the following construct:

**Table 13: create_checksum Construct**

| ***Construct*** | ***Arguments/Options*** | ***Description*** |
|---|---|---|
| create_checksum | | Create a checksum. |
| | checksum_field | Specify the checksum packet field name. This can be: <packet.field> |
| | <packet_field_list> | Ordered list of packet fields associated with the checksum. <packet._PAYLOAD> is treated as a flag to include the payload in the checksum. |

```
create_checksum(egress_pkt.group2.ipv4.hdr_checksum,
        {egress_pkt.group2.ipv4.version, egress_pkt.group2.ipv4.hdr_len,
         egress_pkt.group2.ipv4.tos, egress_pkt.group2.ipv4.v4_length,
         egress_pkt.group2.ipv4.id, egress_pkt.group2.ipv4.flags,
         egress_pkt.group2.ipv4.frag_offset, egress_pkt.group2.ipv4.ttl,
         egress_pkt.group2.ipv4.protocol, egress_pkt.group2.ipv4.sa,
         egress_pkt.group2.ipv4.da});

create_checksum(egress_pkt.fwd_l3_l4_hdr.udp.checksum,
        {egress_pkt.fwd_l3_l4_hdr.ipv4.sa,
         egress_pkt.fwd_l3_l4_hdr.ipv4.da,
         editor_dummy_bus.zero_byte,
         egress_pkt.fwd_l3_l4_hdr.ipv4.protocol,
         egress_pkt.fwd_l3_l4_hdr.udp.udp_length,
         egress_pkt.fwd_l3_l4_hdr.udp.src_port,
```

```
        egress_pkt.fwd_l3_l4_hdr.udp.dst_port,
        egress_pkt.fwd_l3_l4_hdr.udp.udp_length,
        egress_pkt.fwd_l3_l4_hdr.udp._PAYLOAD});
```

### 4.10.5 Update Packet Length Construct

The Update Packet Length construct can only be used in function constructs.

Update Packet Length uses the following construct:

**Table 14: update_packet_length Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| update_packet_length | | Update a packet length. |
| | packet_length_field | Specify the packet length field name.<br>This can be: <packet.field> |
| | update_type | Specify the type of packet length update.<br>This can be: constant<br>● Value of 0 means use only payload length.<br>● Value of 1 means use header and payload length. |
| | truncate_mode | Specify if the packet is to be truncated.<br>This can be: constant<br>● Value of 0 means not to truncate the packet<br>● Value of 1 means to truncate the packet. |

```
    update_packet_length(egress_pkt.group2.ipv4.v4_length, 1);
```

**Example**

USE OF CREATE_CHECKSUM AND UPDATE_PACKET_LENGTH

The following is an example using create_checksum and update_packet_length constructs. Here the packet length must be updated and checksum on egress.

```
function do_checksum_update() {
    create_checksum(egress_pkt.group2.ipv4.hdr_checksum,
        {egress_pkt.group2.ipv4.version, egress_pkt.group2.ipv4.hdr_len,
        egress_pkt.group2.ipv4.tos, egress_pkt.group2.ipv4.v4_length,
        egress_pkt.group2.ipv4.id, egress_pkt.group2.ipv4.flags,
        egress_pkt.group2.ipv4.frag_offset, egress_pkt.group2.ipv4.ttl,
        egress_pkt.group2.ipv4.protocol, egress_pkt.group2.ipv4.sa,
        egress_pkt.group2.ipv4.da});
    create_checksum(egress_pkt.group4.ipv4.hdr_checksum,
        {egress_pkt.group4.ipv4.version, egress_pkt.group4.ipv4.hdr_len,
        egress_pkt.group4.ipv4.tos, egress_pkt.group4.ipv4.v4_length,
        egress_pkt.group4.ipv4.id, egress_pkt.group4.ipv4.flags,
        egress_pkt.group4.ipv4.frag_offset, egress_pkt.group4.ipv4.ttl,
```

```
        egress_pkt.group4.ipv4.protocol, egress_pkt.group4.ipv4.sa,
        egress_pkt.group4.ipv4.da});
}

function do_packet_length_update() {
    update_packet_length(egress_pkt.group2.ipv4.v4_length, 1);
    update_packet_length(egress_pkt.group4.ipv4.v4_length, 1);
}

program app {
    ...
    do_packet_length_update();
    do_checksum_update();
    ...
}
```

# 5 Target Vendor Specific Constructs

A target packet processing pipeline may have generic utilities, accelerators, configurable components that provide an efficient implementation of some specific network function. NPL provides constructs to define and call such components along with the rest of the logical functionality. Target vendor defines these and NPL writer calls them in the application. They are of 3 kinds -

1. Target Extern Functions
2. Special Function
3. Dynamic Table

Target extern functions are the basic functions in a target architecture. Extern functions may be called repeatedly in the application along with other NPL constructs. e.g. packet drop extern function called as part of logical_table lookup.

Special function constructs are used to specify target accelerators or IP, and their usage modes. Special functions require specific-connectivity in terms of inputs and outputs. The internals of the special functions is not specified in NPL. Target vendor defines special function. NPL writer calls them.

The Dynamic Table constructs are used to specify target specific “runtime” logical tables. Target vendor specifies basic structure of dynamic table. NPL writer specifies a menu of logical signals such that target-SDK can create runtime logical tables.

After NPL is converted to a simulation language such as C++, sample behavior of the Special Functions and Target Extern functions may be provided by target.

## 5.1 Target Extern Functions

Each networking device has fundamental functions which repeat multiple times. E.g. drop a packet, copy it to CPU or other port for tracing reasons or to count a packet and so on. These basic functions can be associated with logical_table, function or other NPL constructs. e.g. dropping a packet as part of logical table lookup, or mirroring a packet as part of processing function.

NPL allows target vendor to specify such functions as Extern Functions. Target vendor defines extern function template with information required to better utilize hardware.Following sections cover definition and call process.

## 5.1.1 Target Extern Function - Definition

Target vendor defines the extern function template. Template is identical to any standard programming language.

**Table 15: extern Construct**

| ***Construct*** | ***Arguments/Options*** | ***Description*** |
|---|---|---|
| extern <function name> | | Target Vendor specified extern function. |
| | direction | in or out |
| | field name | Specify fields with width and type. |

**Example**

**Extern function Definition for Packet Drop**

Defined by Target vendor-

```
    extern packet_drop(in bit[1] trigger, in const value, in const drop_code);
```

```
Where trigger means any bus field which can trigger packet drop. And other properties associated
with packet drop.
```

## 5.1.2 Target Extern Function Usage

NPL writer can call Extern functions in the application. Extern calls are allowed inside logical_table and function.

**Example**

```
logical_table packet_integrity {
….
    fields {
        bit copy_to_cpu;
        bit pkt_integrity_drop;
    }
……
    fields_assign() {
       ….
        packet_drop(pkt_integrity_drop, drop_reason.PKT_INTEGRITY_CHECK_FAILED, 2);
    }
```

```
}
```

## 5.2 Special Function Constructs

### 5.2.1 Special Function - Definition Construct

The special_function construct is used to define the interface to a target architecture IP block. The internal functionality of the IP block is not specified in the NPL. The interface definition of the IP block must be provided by target vendor using template methods. The NPL writer calls the IP block in the program by using the defined template methods and using appropriate arguments. The special_function is an extensible construct where target vendor can add custom methods.

**Table 16: special_function Construct**

| ***Construct*** | ***Elements*** | ***Description*** |
|---|---|---|
| special_function | | Define a IP block interface. |
| | special_function_name | Specify the name of the IP block. |
| | <generic method>(<br>[in/out]<br>[const] or<br>[auto_enum] or<br>[str] or<br>[bit/varbit][size] or<br>[list]<br>[name]<br>) | IP block method prototype which specifies the direction and type of arguments.<br>Provided by Target Vendor. Target compiler must be able to process the definition and call. Multiple methods are supported here.<br>varbit indicates a maskable argument. |

#### 5.2.1.1 Sample special_function definition for a Target Vendor

A target vendor may specify the following methods to define IP block interface.

**Example**

```
special_function flex_qos_phb {
    usage_mode_create(in const index,
            in bit[10]   qos_base,
            in varbit[6] qos_attr,
            out bit[4]   int_pri
            );
    usage_mode_select(in bit[6] eindex);
}
```

## 5.2.2 Special Function Usage

The NPL writer connects target IP blocks to logical functionality specified in NPL program using

- Methods provided in special_function library by target device.
- Using NPL built-in 'execute()' method.

### 5.2.2.1 Special Function Methods

The NPL writer uses the special_function methods, provided by the target architecture, to specify connections to the IP blocks. The position of the these methods call in the NPL does not reflect the call sequence.

Syntax to call such method is as below. Argument type and their order MUST match from template provided.

The special function method arguments are passed by reference.

<special_function_name>.<method_name>(<arguments>)

### 5.2.2.2 execute()

The execute() is a built-in method that is not specified in the special_function construct.  This method activate IP block and provide IP block relative position in logical functionality. This method doesn't take any argument.

**Table 17: execute()**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| execute() | | A built-in method to activate the IP block. |

### 5.2.2.3 Sample special_function Usage for a Target Vendor

The NPL writer can access the target vendor IP block by using the special_function method prototypes.

**Example**

```
program app {
    ...
    flex_qos_phb.usage_mode_create(flex_qos_entry.QOS_MPLS_EXP_L3_TUNNEL_ECN,
            ing_obj_bus.mpls_exp_mapping_ptr[9:0],
            ing_cmd_bus.mpls_effective_exp_for_phb,
            ing_cmd_bus.int_pri
            );
    flex_qos_phb.usage_mode_create(flex_qos_entry.QOS_MPLS_EXP_L2_TUNNEL_ECN,
            ing_obj_bus.mpls_exp_mapping_ptr[9:0],
            ing_cmd_bus.mpls_effective_exp_for_phb1,
            ing_cmd_bus.int_pri1
            );
    ...
    flex_qos_phb.usage_mode_select(phb_select_lts_tcam_key.entry_index);
    flex_qos_phb.execute();
    ...
}
```

# 5.3 Dynamic Table Constructs

## 5.3.1 Dynamic Table Definition Construct

The Dynamic Table construct is used to specify runtime logical tables. The dynamic_table construct is target-driven. The NPL writer specifies a menu of logical signals such that target-specific SDK can create runtime logical tables.

Target vendor provides a dynamic_table definition. This definition does not have any explicit input or output connections. It indicates to the subsequent SDK how to use the dynamic tables.

Dynamic table can have multiple methods to support different targets.

The dynamic_table table size is defined as 1.

**Table 18: dynamic_table Construct**

| *Construct* | *Elements* | *Description* |
|---|---|---|
| dynamic_table | | Define a dynamic_table. |
| | dynamic_table_name | Specify the name of the dynamic table block. |
| | <generic method>(<br>[in/out]<br>[list]<br>[name]<br>) | Method prototype which specifies the direction and list of the arguments.<br>Provided by Target Vendor. Target compiler must be able to process the definition and call. Multiple methods are supported here. |

### 5.3.1.1 Sample dynamic_table definition for a Target Vendor

A target vendor may specify the following methods to define a dynamic table interface.

**Example**

```
dynamic_table ing_fp {
    presel_template(in list presel_menu);
    rule_template(in list rule_menu);
    action_template(out list action_menu);
}
```

## 5.3.2 Dynamic Table Usage

The NPL writer uses the dynamic table method templates to correlate logical signals to the dynamic table.

### 5.3.2.1 <dynamic_table_name>.<method_name>(<argument_list>)

The NPL writer uses the dynamic_table methods, provided by the target architecture, to connect logical signals to the dynamic table blocks. The position of the method calls in the NPL does not reflect the call sequence.

The template method call uses the following format:
<dynamic_table_name>.<method_name>(<argument_list>)

The dynamic table method arguments are passed by reference.

### 5.3.2.2 lookup()

The lookup() is a built-in method that is not specified in the dynamic_table construct. This method is used to specify the location of the dynamic_table call.

**Table 19: lookup()**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| lookup() | | A built-in method to call the dynamic_table block. |

### 5.3.2.3 Sample dynamic_table Usage for a Target Vendor

The NPL writer can access the target vendor dynamic_table block by using the dynamic_table method prototypes.

**Example**

```
program {
    ...
    ing_fp.presel_template(
        {
            ing_cmd_bus.l2_iif_opaque_ctrl_id,
            ing_cmd_bus.l3_iif_opaque_ctrl_id,
             ...
        });
    ing_fp.rule_template(
        {
            pkt_fwd_field_bus.macda,
            pkt_fwd_field_bus.macsa,
            ...
        });
    ing_fp.action_template(
        {
            ifp_scratch_bus.ifp_drop_action,
            ifp_scratch_bus.ifp_drop_code,
            ...
        });
    ing_fp.lookup();  //lookup is the constructor for the dynamic_table
}
```

# 6 Strength Resolution Constructs

In the NPL paradigm, multiple tables may be looked up in parallel. When multiple tables assign the same object, a mechanism is needed to resolve which value to use. NPL uses a strength-based mechanism for resolution when a numerical comparison is sufficient to decide the winning object.

NPL provides constructs to associate strength values with table lookup results. Each lookup generates one strength profile. Strengths can be static (per table based) or dynamic (per entry based).

NPL uses the following constructs for strength based resolution:

- strength logical tables entries
- table fields_assign() to indicate where strength resolution is needed
- a strength resolve function

## 6.1 Strength Logical Table Creation

The strength construct declares a prototype for strength logical table entries. The table can contain one or more strength fields.

Use the struct construct to create the strength table fields. These are the fields that require strength resolution.

- The struct can only have bit fields and not nested structs

Use the strength construct to instantiate the strength table

- The strength table name must be globally unique

The entries in the strength table for resolving lookups will be specified in the arguments to the strength_resolve construct.

**Table 20: strength Construct**

| ***Construct*** | ***Arguments/Options*** | ***Description*** |
|---|---|---|
| strength | | Instantiate a Strength Table. |
| | strength struct name | Name of the Strength table structure with field names representing the fields of strength table entry.<br>This should be a struct. |
| | strength table name | Name of the Strength Table.<br>This can be a string |

## 6.2 Strength Table Connectivity

In a logical table fields_assign(), use the use_strength construct in place of an assignment statement to specify strength resolution. The use_strength construct specifies an index into the strength table. This index selects an entry that will be used to assign a strength value to the lookup result from this table.

**Table 21: use_strength Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| use_strength | | Attaching the strength table to a logical table |
| | strength table name | Name of a strength table declared with the strength construct. |
| | index | Index into the strength table which will be used to resolve the result from this source logical table.<br>The size of the strength profile table is determined by the bit width of this argument. In the case of constant indices the width is the number of bits required to accommodate the largest value used. Thus given a bit width of B the table size will be $2^B$.<br>Index may be -<br>● Constant - Index is fixed at compile time.<br>● Table.Field - Field of this logical table to be used as the index. Provides per entry dynamic strength.<br>● Bus.Field - Field of a bus to be used as the index in strength profile table. |

## 6.3 Strength Resolve Construct

The strength_resolve construct specifies the bus object that is to be assigned using strength resolution. It also specifies (strength_list) a strength value for each logical_table lookup result to be resolved. Corresponding entries (source_field_list) associate these strength values with candidate table lookup result values that need to be resolved (use_strength indices). The table with the “strongest” strength value will supply the bus object value.

**Table 22: strength_resolve Construct**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| strength_resolve | | Specifying how to resolve object assignment among multiple sources. |
| | destination bus.field | bus.field to which the object must be assigned after the strength resolution is done. |
| | destination bus.field strength | The strength associated with the destination bus.field. |

| | | |
|---|---|---|
| | | Possible values: bus.field or NULL |
| | <strength_entry_0> | A list of properties describing the first strength item. See strength_entry description below for details. |
| | <strength_entry_1> | A list of properties describing the second strength item. See strength_entry description below for details. |
| | ... | |
| | <strength_entry_n> | A list of properties describing the nth strength item. See strength_entry description below for details. |

| ***strength_entry*** | | |
|---|---|---|
| Format: | ***{table_lookup, user_defined_view_type, strength, source_field}*** | |
| | table_lookup | Specifies the table lookup when the source_field is a table.field.<br>Possible values: table._LOOKUP0 or table._LOOKUP1. |
| | user_defined_view_type | Specifies the fields view type, which is user defined. Use same auto_enum values as specified in fields_assign() block.<br>Possible values: auto_enum or NULL. |
| | strength | Strength associated with logical_table result that corresponds to the object being resolved.<br>Possible values: strength table.field. |
| | source_field | source field which can assign to the destination bus.field.<br>Possible values: logical_table.field. |

**Example**

**SPECIFYING STATIC STRENGTH FOR AN OBJECT BEING DRIVEN FROM 2 TABLES**

For example, Say, Pseudo-code for Strength Resolve is:

```
if (obj_strength_profile[1].obj_k_str > obj_strength_profile[2].obj_k_str)
    cmd_bus.obj_k = Table_A.obj_k;
else
    cmd_bus.obj_k = Table_B.obj_k;
```

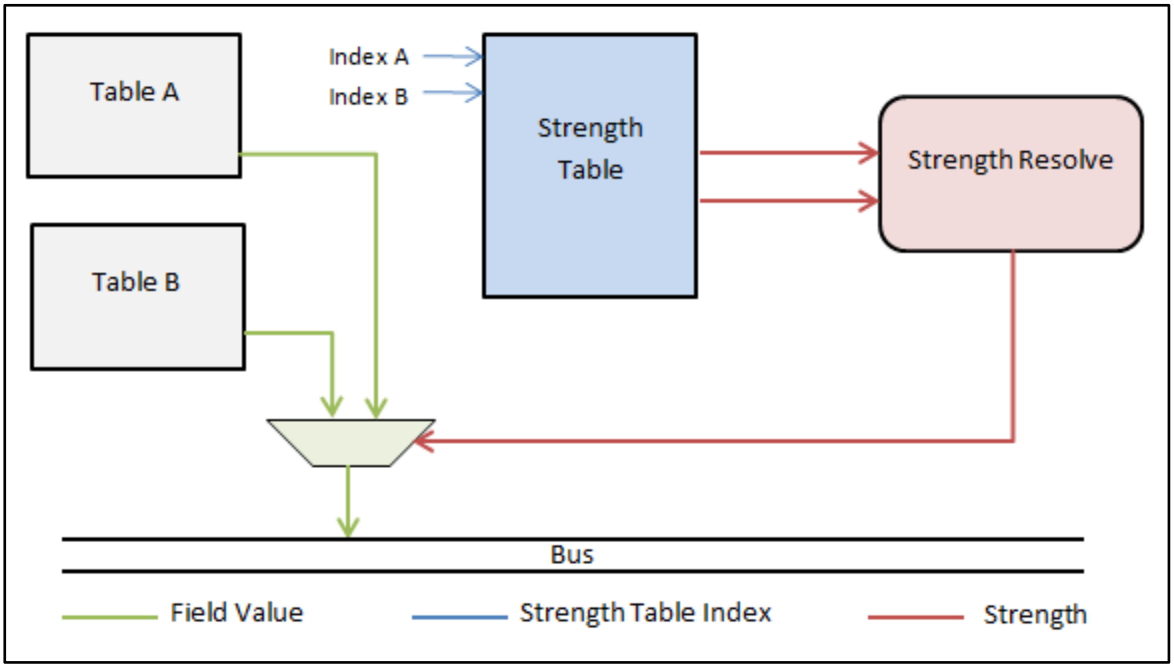



**Figure 3: Strength Using Tables with Static Indexing**

Construct Usage will be:

```
struct cmd_bus_t {
    fields {
        bit[8]    obj_k;
    }
}

struct obj_strength_t {
    fields {
        bit[4]    obj_k_str;
    }
}

bus   cmd_bus_t   cmd_bus;
strength  obj_strength_t   obj_strength_profile; // strength profile table

logical_table Table_A {
    ...
    fields {
        bit[8]    obj_k;
    }
    fields_assign() {
        use_strength(obj_strength_profile, 1);  // reference to strength profile table entry
```

```
    }
}

logical_table Table_B {
    ...
    fields {
        bit[8]    obj_k;
    }
    fields_assign() {
        use_strength(obj_strength_profile, 2);  // reference to strength profile table entry
    }
}
program app {
    ...
    strength_resolve(cmd_bus.obj_k, NULL,
         { Table_A._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_A.obj_k},
         { Table_B._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_B.obj_k});
    ...
}
```

### SPECIFYING DYNAMIC STRENGTH FOR AN OBJECT BEING DRIVEN FROM 2 TABLES

For example, Pseudo-code for Strength Resolve is:

```
if (obj_strength_profile[index_A].obj_k_str > obj_strength_profile[index_B].obj_k_str)
    cmd_bus.obj_k = Table_A.obj_k;
else
    cmd_bus.obj_k = Table_B.obj_k;
```

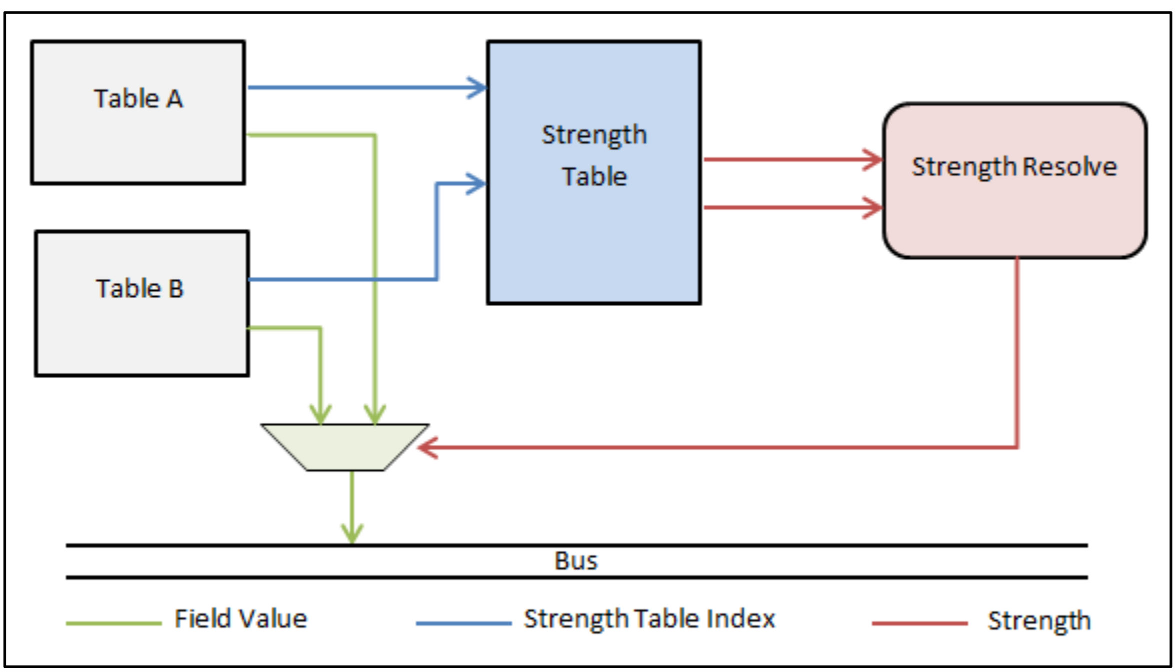



**Figure 4: Strength Using Tables with Dynamic Indexing**

Construct Usage will be:

```
struct cmd_bus_t {
    fields {
        bit[8]    obj_k;
    }
}

struct obj_strength_t {
    fields {
        bit[4]    obj_k_str;
    }
}

bus cmd_bus_t cmd_bus;
strength obj_strength_t obj_strength_profile;

logical_table Table_A {
    ...
    fields {
        bit[8]     obj_k;
        bit[5]     strength_index;

    }
    fields_assign() {
        use_strength(obj_strength_profile, strength_index);
    }
}

logical_table Table_B {
    ...
    fields {
        bit[8]     obj_k;
        bit[5]     strength_index;
    }
    fields_assign() {
        use_strength(obj_strength_profile, strength_index);
    }
}
program app {
    ...
    strength_resolve(cmd_bus.obj_k, NULL,
        { Table_A._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_A.obj_k},
        { Table_B._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_B.obj_k});
    ...
}
```

**SPECIFYING STRENGTH FOR AN OBJECT BEING DRIVEN FROM A TABLE AND BUS**

For example, if Pseudo-code for Strength Resolve is:

```
if (obj_strength_profile[a_strength_index].obj_k_str > cmd_bus.obj_k_str)
    cmd_bus.obj_k = Table_A.obj_k;
else
```

```
    cmd_bus.obj_k = cmd_bus.obj_k;
```

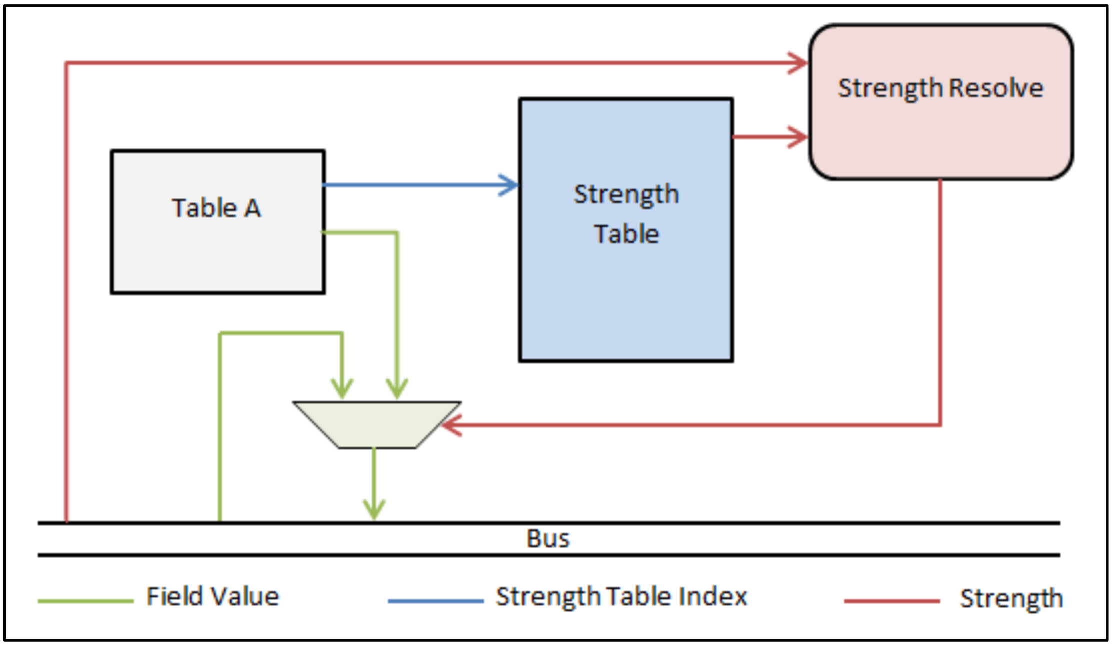



**Figure 5: Strength Using Table and Bus**

Construct Usage will be:

```
struct cmd_bus_t {
    fields {
        bit[8]      obj_k;
        bit[4]      obj_k_str;
    }
}

struct obj_strength_t {
    fields {
        bit[4]    obj_k_str;
    }
}

bus cmd_bus_t cmd_bus;
strength obj_strength_t obj_strength_profile;

logical_table Table_A {
    ...
    fields {
        bit[8]    obj_k;
        bit[3]    a_strength_index;
    }
    fields_assign() {
          use_strength(obj_strength_profile, a_strength_index);
     }
}
```

```
program () {
    strength_resolve(cmd_bus.obj_k, cmd_bus.obj_k_str,
        {Table_A._LOOKUP0, NULL, obj_strength_profile.obj_k_str, Table_A.obj_k});
}
```

### SPECIFYING STRENGTH USING 2 MULTI-LOOKUP TABLES

```
struct obj_bus_t {
    fields {
        bit[12]      dst_vlan;
        bit[12]      src_vlan;
        bit[11]      dst_vfi;
        bit[11]      src_vfi;
    }
}

struct cmd_bus_t {
    fields {
        bit      dst_discard;
        bit      src_discard;
    }
}

struct UAT_strength_profile_t {
    fields {
        bit[4] obj_src_discard_str;
        bit[4] obj_K_str;
    }
}

strength UAT_strength_profile_t UAT_strength_profile;
bus obj_bus_t obj_bus;
bus cmd_bus_t cmd_bus;

//multi lookup case
logical_table Table_A {
    ...
    field {
        bit[12] vlan;
        bit     discard; //strength resolve field
    }
    fields_assign() {
        if (_LOOKUP0)
            obj_bus.dst_vlan = vlan;
        if (_LOOKUP1)
            obj_bus.src_vlan = vlan;
        use_strength(UAT_strength_profile, 10);
    }
}

//single lookup case. No need to add _LOOKUP0 in fields_assign().
logical_table Table_C {
    ...
```

```
    field {
        bit[11] vfi;
        bit     discard; //strength resolved field
    }
    fields_assign() {
        obj_bus.dst_vfi = vfi;
    }
}

//Same as Table_A but specified here for strength logic below
logical_table Table_B {
    ...
    field {
        bit[11] vfi;
        bit     discard; //strength resolved field
    }
    fields_assign() {
        if (_LOOKUP0) {
            obj_bus.dst_vfi = vfi;
        }
        if (_LOOKUP1) {
            obj_bus.src_vfi = vfi;
        }
        use_strength(UAT_strength_profile, 20);
    }
}

program () {
    ...
    //multi (2) lookups
    Table_A.lookup(0);
    Table_A.lookup(1);

    //Single Lookup
    Table_C.lookup(0);

    //multi (2) lookups
    Table_B.lookup(0);
    Table_B.lookup(1);

    strength_resolve(cmd_bus.src_discard, NULL,
        {Table_A._LOOKUP1, NULL, UAT_strength_profile.obj_src_discard_str, Table_A.discard},
        {Table_B._LOOKUP1, NULL, UAT_strength_profile.obj_src_discard_str, Table_B.discard});

    strength_resolve(cmd_bus.dst_discard, NULL,
        {Table_A._LOOKUP0, NULL, UAT_strength_profile.obj_K_str, Table_A.discard},
        {Table_B._LOOKUP0, NULL, UAT_strength_profile.obj_K_str, Table_B.discard});
    ...
}
```

## 6.4 Strength Resolve Using Functions

The function construct may also be used to specify the Strength Resolve for tables in series. In this coding style the compiler might choose a different mapping scheme.

# 7 Generic Constructs

## 7.1 NPL Attributes

NPL Attributes are used to transfer NPL writer's intent to yaml files. These yaml files are then accessed to extract pertinent information.

Attributes are represented by <! and !>.

The compiler makes no effort to verify anything written as NPL Attribute.

### 7.1.1 Positional Attributes

Positional attributes are used to attach documentation to NPL code to describe various items. This is different from the comments (// and /* */) support in NPL although both of them do not have any impact on code / compilation.

NPL Attributes currently support the following NPL items -

- logical_table, keys, fields
- logical_register, fields
- bus (struct), fields, overlays
- packet header (struct), field
- special_function and dynamic_table tables

Compilers can propagate NPL Attributes to the target vendor output files. e.g. target compiler can dump documentation about logical_table in output file called, Regsfile or Map File. The following Table summarises the use of Positional Attributes.

**Table 23: Positional Attributes**

| ***Descriptor*** | ***Destination*** | ***Description*** |
|---|---|---|
| REGSFILE, DESC: <desc> | logical_regsfile.yml<br>header.yml | Attribute applies to:<br>logical_table,<br>logical_table.field,<br>logical_table.key,<br>logical_register,<br>logical_register.field,<br>struct, (bus/header)<br>struct.field. (bus/header)<br><br>Attribute <desc> used as:<br>TABLE <tbl> DESC,<br>TABLE <tbl.fld> DESC,<br>REGISTER <reg> DESC,<br>REGISTER <reg.fld> DESC,<br>FORMAT <struct> DESC,<br>FORMAT <struct.fld> DESC<br>HEADER <struct> DESC,<br>HEADER <struct.fld> DESC. |

| | | |
|---|---|---|
| REGSFILE, ENCODING: <enum> | logical_regsfile.yml | Attribute applies to: logical_table.field.<br><br>Attribute <enum> used as: Table field, ENCODINGS |
| REGSFILE, ENCODING: DESC: enum.field = <desc> | logical_regsfile.yml | Attribute applies to: logical_table.field enum.field.<br><br>Attribute <desc> used as: logical_table field, ENCODINGS field DESC. |
| REGSFILE, FIELD_NAME: <field> | logical_sftblfile.yml | Attribute applies to: dynamic_table.arg.<br><br>Attribute <field> used as: dyamic_table.field name |

**Example (REGSFILE, DESC: <desc>)**

NPL Code:

```
<!(REGSFILE, DESC: "This table is looked up using Layer 2 incoming interface packet was received on.
  This table provides incoming Layer 2 interface attributes for the packet.") !>
logical_table ing_l2_iif_table {
    ...
    fields {
        <!(REGSFILE, DESC: "If set, IPV6 Tunnel is enabled on this interface.") !>
        bit ipv6_tunnel_enable;
    }
}
```

Logical Regsfile Representation:

```
TABLE = {
    ing_l2_iif_table:
        DESC: |-
            This table is looked up using Layer 2 incoming interface packet was received on.
            This table provides incoming Layer 2 interface attributes for the packet.
        FIELDS:
            ipv6_tunnel_enable:
                DESC: |-
                    If set, IPV6 Tunnel is enabled on this interface.
                MAXBIT: 13
                MINBIT: 13
                ORDER: 4
                TAG: data
                WIDTH: 1
```

**Example (REGSFILE, ENCODING: <enum>)/(REGSFILE, ENCODING: DESC: enum.field = <desc>)**

NPL Code:

```
enum pvlan_port {
       PVLAN_PROMISCUOUS_PORT = 0,
       PVLAN_COMMUNITY_PORT = 1,
       PVLAN_ISOLATED_PORT = 2
}

logical_table ing_vfi_table {
    ...
    fields {
        <!REGSFILE, ENCODING: pvlan_port !>
        <!REGSFILE, ENCODING: DESC: pvlan_port.PVLAN_PROMISCUOUS_PORT = Pvlan Promiscuous port  !>
        bit[FIELD_2_WD] src_pvlan_port_type;
    }
}
```

Logical Regsfile Representation:

```
ing_vfi_table:
    FIELDS:
        src_pvlan_port_type:
            ENCODINGS:
                pvlan_port__PVLAN_PROMISCUOUS_PORT:
                    DESC: |-
                        Pvlan Promiscuous port
                    VALUE: 0
                pvlan_port__PVLAN_COMMUNITY_PORT:
                    DESC: ‘’
                    VALUE: 1
                pvlan_port.PVLAN_ISOLATED_PORT
                    DESC: ‘’
                    VALUE: 2
```

**Example (REGSFILE, FIELD_NAME: <field>)**

NPl Code:

```
dynamic_table egr_flex_ctr {

    presel_template (in list presel_menu);
    object_template (in list object_menu);
}
    egr_flex_ctr.presel_template(
    {
        <!REGSFILE, FIELD_NAME: "mirror_pkt_ctrl_0" !>
        egr_cmd_bus.mirror_pkt_ctrl
    }
    );
```

Logical Regsfile Representation:

```
    dt_egr_flex_ctr_presel_template:
        FIELDS:
            mirror_pkt_ctrl_0:
                BUS_SELECT_WIDTH: 4
                DESC: |-
```

```
        Input - egr_cmd_bus_mirror_pkt_ctrl
    MAXBIT: 65
    MINBIT: 2
    ORDER: 2
    TAG: bus_select
    WIDTH: 64
```

## 7.1.2 Non-Positional Attributes

Non-Positional attributes are used to allow the full intent for the NPL writer.

All Non-Positional Attributes are collected in a IFILE (Intent File) yaml file. The IFILE will only contain non-positional attributes. These attributes will be grouped based on the specified functionality. The non-positional attributes can support different functionality.

It is the responsibility of the NPL writer to:

- Specify the content-format of the Non-Positional Attributes

It is the responsibility of the Consumer to:

- Parse and transform the attributes in desired format.
- Perform any necessary verification of attributes.

Consumer of the IFILE must write utility to consume compiler outputs and convert them in desired format along with checks, if any.

NPL Non-Positional Attributes are propagated to the IFILE File based on the Description. Following Table summarises the use of Non-Positional Attributes.

**Table 24: Non-Positional Attributes**

| ***Descriptor*** | ***Destination*** | ***Description*** |
|---|---|---|
| (IFILE, <blk>: <code>) | ifile.yml | The <code> is added to the <blk> block in the IFILE. |

### 7.1.2.1 Initialization

The NPL writer may specify initialization for different logical tables, strength tables or any other element inside.

**Example**

NPL:

```
//assigning an NPL logical_table
<! IFILE, INIT: ”lt ing_l3_next_hop_1_table nhop_index_1=0 dvp=0x0 l3_oif_1=0x0” !>

//assigning a physical resource if required.
<! IFILE, INIT: ”pt IFTA150_SBR_PROFILE_TABLE_0_INDEX=10 strength=0x5” !>

//assigning to a symbolic item (in future)
<! IFILE, INIT: ”lt _SYMBOL=TIMESTAMP _INDEX=10 data=0x5” !>
```

```
//using NPL Enum
<! IFILE, INIT: ”pt EPOST_FMT_AUX_BOTP_IN_DATA mtu_drop=NPL_EGR_MTU_DROP_ENUM” !>
```

Output IFILE:

```
INIT:
    0: |-
        lt ing_l3_next_hop_1_table nhop_index_1=0 dvp=0x0 l3_oif_1=0x0
    1: |-
        pt IFTA150_SBR_PROFILE_TABLE_0_INDEX=10 strength=0x5
    2: |-
        lt _SYMBOL=TIMESTAMP _INDEX=10 data=0x5
    3: |-
        pt EPOST_FMT_AUX_BOTP_IN_DATA mtu_drop=NPL_EGR_MTU_DROP_ENUM
```

#### 7.1.2.2 Relational

The NPL writer may want to specify additional functionality related to an NPL symbol or call.

**Example**

NPL:

```
<! IFILE, REL: ”ecmp_level0:npl_ecmp_level0_member_table” !>
```

Output IFILE:

```
REL:

    0: |-
        ecmp_level0:npl_ecmp_level0_member_table
```

## 7.2 Preprocessor Constructs

### 7.2.1 Include - (#include)

Use #include to include other NPL source code files into the NPL program.

```
#include "bus.npl"              // NPL located in same directory
#include "../lib/header.npl"    // NPL located in different directory
```

### 7.2.2 If-else-endif (#if - #endif)

NPL supports C/C++ style conditional compilation.

```
#ifdef XYZ
```

### 7.2.3 Define - (#define)

NPL supports C/C++ style macro definitions for variables. Parameterized definitions are not allowed. Use # and Capital Names in the definitions.

```
#define CPU_PORT        5
#define MAC_ADDR_BCAST  0xffffffffffff
```

## 7.3 Comments

NPL allows 2 kinds of comments:

- Multi line - Use /* and */ at beginning and end of the comment. They can be multi-line or in-line.
- Single line - Use // at the beginning of the comment.

## 7.4 Print

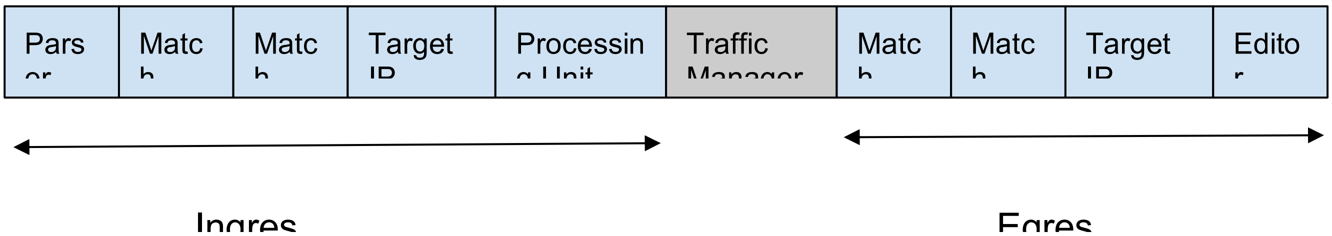
NPL uses `print` construct to print value of any variable in the program. print command is translated to the Behavioral C Model. It does not have significance from compilation point of view. print invocation is called in the C model in the same sequence as in an NPL program. Only fields can be printed, no structs. Internally, print command works in a similar way as the C printf command.

Construct is `print - print in behavioral model.`

```
print("Value of the SVP is %d, VFI is %d\n", obj_bus.svp, obj_bus.vfi);
```

# 8 Appendix A: Example Switch Pipeline

An example Switch Pipeline is shown below. The example illustrates NPL Construct usage and an example Switch Pipeline. NPL as a language can support multiple different architectures. Following diagram is for illustration only.




Parser - NPL Parser Constructs define and modify the behavior of the HW Parser Block.

Match Action - NPL Logical Tables and Logical Register define and modify the behavior of the Match Action Blocks.

Target IP - An Intellectual Property (IP) belonging to a Target vendor is represented by special_function construct.

Processing Unit - This is a generic target specific processing element. NPL functions, strength resolve define and modify the behavior of this block.

Editor - NPL editor constructs define the behavior of the editor block.

A target may choose to have multiple of each of these units and in any order.

# 9 Appendix B: Usage Guideline

## 9.1 struct as headers

High-level objects, for example: logical_table, logical_register, and enums can NOT be part of struct.

NPL allows struct nesting. However, depending on the different use cases for the structs there may be valid or invalid use cases. They are marked below.

**Table 25: struct usage guide**

| ***Struct Usage*** | ***bit/bit[n]*** *(can structs have bit)* | ***overlays*** *(can structs have overlays)* | ***1-level Structs*** *(can structs have one level of structs)* | ***Multi-level Nested Structs*** |
|---|---|---|---|---|
| Examples | | | item is struct.field | item is struct.struct.field |
| as header type | Y | NA | NA | NA |
| as header_group | NA | NA | Y | NA |
| packet | NA | NA | Y | NA |
| bus | Y | Y | Y | Y |

**Table 26: Referencing struct in Packet**

| ***Valid packet construction*** | ***Is Valid*** | ***packet reference description*** |
|---|---|---|
| packet.struct.struct.field | Y | packet.group.header.field |
| packet.struct.struct | Y | packet.group.header |
| packet.struct.field | N | no group/header |
| packet.field | N | packet must have group/header. |
| no packet | N | |

## 9.2 Function

Functions with arguments are not supported. Functions must operate on logical bus and packet data.

## 9.3 Overlay Rules

Overlays can be used in multiple constructs and they must follow certain rules which are applicable for all constructs.

1. Overlay fields can be defined on Base Fields of type bit/bit[n]
2. Overlay fields can be defined on Base Fields of type struct. (i.e., fields specified as struct in the bus).
However partial overlay of the struct is not allowed.
3. Overlays can overlap.
4. Overlay fields can NOT be defined on other overlay fields.
5. Overlay fields can NOT be of type struct.

## 9.4 Bit-Array Slicing

NPL allows bit-array slicing (ranges) to be specified in the assignments, equations, Special Functions. Bit ranges are allowed on both lvalue and rvalue of equations.

**Example**

```
a = b[7:4];
if (b[5:3])...
obj_bus.a[5:4] = ing_pkt.ipv4.ecn;  // mostly in function, action.
a = b[0:0];                         // single bit access uses range of one bit
```

**Use Cases**

```
1.      if (a[5:3]) //supported
2.      a[5:3] = b[5:3]; //supported
```

## 9.5 Concatenation Rules

- Concatenations are allowed in Only in rvalue in assignments. Generally used with varbit fields.
  ```
  a = b<>c;
  ```
- Concatenations are NOT allowed on the lvalue.
  ```
  b<>c = a[5:0];
  ```
- Concatenations are allowed on the same type of fields. i.e., struct<>struct
  ```
  two_mpls_hdrs = mpls_hdr_0<>mpls_hdr_1;
  ```
- Concatenations are allowed on different types of fields. i.e., struct<>bit
  ```
  new_mpls_hdr = mpls_hdr_0<>c[3:0];
  ```
- Concatenations are allowed on part of the fields. i.e., da[5:0]<>sa[5:3]
  ```
  a[5:0] = b[3:2]<>c[3:0];
  ```
- If the comparison is with "register value" only the (==) operator is supported.

# 10 Appendix C: NPL Reserved Words

The following table lists all NPL reserved words.

| | | | |
|---|---|---|---|
| **fields_assign** | **add_header** | **auto_enum** | **bit** |
| **bus** | **create_checksum** | **default** | **delete_header** |
| **define** | **dynamic_table** | **else** | **enum** |
| **extract_fields** | **fields** | **function** | **hash** |
| **header_length_exp** | **if** | **index** | **keys** |
| **latest** | **logical_register** | **logical_table** | **mask** |
| **maxsize** | **minsize** | **next_node** | **NPL_PRAGMA** |
| **overlays** | **packet** | **keys_construct** | **parse_break** |
| **parse_continue** | **parse_begin** | **end_node** | **parser_node** |
| **print** | **program** | **replace_header_field** | **root_node** |
| **special_function** | **strength** | **strength_resolve** | **struct** |
| **switch** | **table_type** | **tcam** | **update_packet_length** |
| **use_strength** | **varbit** | **_HIT_INDEXx** | **_LOOKUPx** |
| **_PRESENT** | **_VALID** | **extern** | **true** |
| **false** | | | |

# 11 Appendix D: NPL Grammar

This section provides the NPL grammar using the yacc notation.

The lexer tokenizes identifiers (ID) for user defined names by the regular expression "[A-Za-z_][\w_]*". Decimal constants must be natural numbers. Hex constants are as usual (e.g., 0x0f, 0X0f, 0x0F, etc.). Width constants are like hex constants with printf-style sizes (e.g., 8x00). String constants must be enclosed in double quotes and not contain a new line (e.g., "foo.bar").

**Identifiers:**

```
primary_types :   constant
                  | identifier

                  | STR_CONST      /* r'\"([^\\\n]|(\\.))*?\"'  */
constant :        | DEC_CONST      /* ([0-9][0-9]*)  */
                  | HEX_CONST      /* (0[xX][0-9A-Fa-f]+) */


identifier : ID

dir :      IN
           | OUT
```

**Declarations**

```
npl_node :         npl_declaration_specifier
                   | empty

npl_declaration_specifier :        npl_declaration
                                   | npl_declaration_specifier npl_declaration

npl_declaration  :         struct_definition_specifier
                           | packet_definition_specifier
                           | strength_definition_specifier
                           | bus_definition_specifier
                           | sp_definition_specifier
                           | function_definition_specifier
                           | special_func_defintion_specifier
                           | enum_defintiion_specifier
                           | program_definition_specifier
                           | table_definition_specifier
                           | register_definition_specifier
                           | parsernode_definition_specifier
                           | print_command
                           | generic_block

array_access_format : '[' postfix_expression ']'

range_access_format : '[' postfix_expression ':'   postfix_expression ']'
```

declaration_expn : BIT identifier ‘;’
| BIT array_access_format identifier ‘;’
| VARBIT array_access_format identifier ‘;’
| identifier identifier ‘;’
| identifier identifier array_access_format ‘;’
| BIT array_access_format identifier ‘==’ constant ‘;’

field_declarator : FIELDS ‘{’ field_declaration_list ‘}’

field_declaration_list : declaration_expn
| field_declaration_list declaration_expn

key_declarator : KEYS ‘{’ field_declaration_list ‘}’

**Overlay**

overlay_declarator : OVERLAYS ‘{’ overlay_declaration_list ‘}’

overlay_declaration_list : overlay_expression
| overlay_declaration_list overlay_expression

overlay_expression : identifier ‘:’ concat_format_list ‘;’

**Signal Concatenation**

concat_format_list : concat_format
| concat_format_list CONCAT concat_format

concat_format : identifier
| identifier range_access_format

**Structure**

struct_definition_specifier : STRUCT identifier ‘{’ struct_body ‘}’
| STRUCT identifier ‘{’field_declaration_list ‘}’
| STRUCT ‘{’struct_body‘}’

struct_body : field_declarator header_len_opt
| struct_body overlay_declarator

header_len_opt : HEADER_LENGTH_EXP ‘:’ expression_statement
| empty

**Enumerator**

enum_defintiion_specifier : ENUM postfix_expression args_list_format

**Bus declaration**

bus_definition_specifier : BUS identifier identifier ‘;’

**Table declaration**

table_definition_specifier : LOGICAL_TABLE identifier ‘{’ table_body_block ‘}’
table_type
table_body_block : table_body
| table_body_block table_body

table_body : TABLE_TYPE ':' table_type
| key_declarator
| field_declarator
| KEY_CONSTRUCT key_construct_definition_block
| FIELDS_ASSIGN fields_assign_definition_block
| MAXSIZE ':' constant ';'
| MINSIZE ':' constant ';'

table_type : INDEX ';'
| HASH ';'
| TCAM ';'
| ALPM ';'

table_keys_list : table_keys_expression
| table_keys_list table_keys_expression

table_keys_expression : postfix_expression ';'

key_construct_definition_block : '{' generic_statement_list '}'

fields_assign_definition_block : '{' generic_statement_list '}'

**Register Declaration**

register_definition_specifier : LOGICAL_REGISTER identifier '{' field_declarator '}'

**Packet Declaration**

packet_definition_specifier : packet_instance

packet_instance : PACKET identifier identifier ';'

**Strength**

strength_definition_specifier : strength_instance

strength_instance : STRENGTH identifier identifier ';'

**Statements Block**

generic_statement_list : generic_block
| generic_statement_list generic_block

generic_block : statement

statement : expression_statement
| select_statement
| compound_statement
| label_statement
| header_command
| parser_statement
| pragma_call

compound_statement : '{' generic_statement_list '}'

**Conditional Statements**

```
select_statement :        IF ‘(’ expression ‘)’ statement ELSE statement
                          | IF ‘(’ expression ‘)’ statement
                          | SWITCH ‘(’ expression ‘)’ statement

label_statement :         postfix_expression ‘:’ statement
                          | DEFAULT ‘:’ statement
                          | constant MASK constant ‘:’ next_node
```

**Expressions**

```
expression_statement : expression ‘;’
expression :       assignment_expression
                   | lookup_statement
                   | parse_init
                   | function_call

assignment_expression : generic_expression
                          | generic_expression assignment_operator assignment_expression

generic_expression : binary_expression

binary_expression :       unary_expression
                          | function_call
                          | binary_expression  ‘!=’  binary_expression
                          | binary_expression  ‘==’  binary_expression
                          | binary_expression  ‘&’  binary_expression
                          | binary_expression  ‘<’  binary_expression
                          | binary_expression  ‘<=’  binary_expression
                          | binary_expression  ‘>=’  binary_expression
                          | binary_expression  ‘>’  binary_expression
                          | binary_expression  ‘|’  binary_expression
                          | binary_expression  ‘^’  binary_expression
                          | binary_expression  ‘&&’ binary_expression
                          | binary_expression  ‘||’  binary_expression
                          | binary_expression  ‘<<’  binary_expression
                          | binary_expression  ‘>>’  binary_expression
                          | binary_expression  ‘*’  binary_expression
                          | binary_expression  ‘+’  binary_expression
                          | binary_expression  ‘-’  binary_expression
                          | binary_expression  ‘/’  binary_expression
                          | binary_expression  ‘%’  binary_expression
                          | binary_expression  ‘<>’  binary_expression

unary_expression  :  unary_operator unary_expression
                     | args_format_specifier
                     | parser_access_latest

unary_operator  : ‘~’
                  | ‘!’
                  | ‘|’
                  | ‘&’
```

```
assignment_operator : '=='

primary_expression  : '(' expression')'
primary_expression : primary_types
                   | metainfo
                   | header_position
                   | profile_type

postfix_expression : primary_expression
                   | postfix_expression '.' identifier
                   | postfix_expression '.' metainfo
                   | postfix_expression array_access_format
                   | postfix_expression range_access_format
```

**Program**

```
program_definition_specifier : PROGRAM postfix_expression '{' generic_statement_list '}'
                   | PROGRAM postfix_expression '{' '}'
```

**Parser**

```
parsernode_definition_specifier : PARSER_NODE identifier '{' generic_statement_list '}'

parser_statement : next_node
                   | root_node
                   | parsing_done
                   | parser_field_extract

root_node : ROOT_NODE ':' constant ';'

next_node : NEXT_NODE identifier ';'

parse_init : PARSE_BEGIN '(' postfix_expression ')'

parsing_done : END_NODE':' constant';'

parser_field_extract : EXTRACT_FIELDS '(' postfix_expression ')' ';'

parser_access_latest :  LATEST '.' postfix_expression
```

**Packet Header/s update**

```
 header_command : CREATE_CHECKSUM '(' args_format_specifier ')' ';'
                   | UPDATE_PACKET_LENGTH '(' args_format_specifier ')' ';'
                   | ADD_HEADER '(' postfix_expression ')' ';'
                   | DELETE_HEADER '(' postfix_expression ')' ';'
                   | COPY_HEADER '(' postfix_expression ',' postfix_expression ')' ';'
                   | REPLACE_HEADER_FIELD '(' postfix_expression ',' postfix_expression ')' ';'
```

**Table lookup**

```
 lookup_statement : LOOKUP args_access_format
```

**Function**

```
function_call : postfix_expression '(' ')'
              | postfix_expression args_access_format


function_definition_specifier : FUNCTION postfix_expression '(' func_def_args ')' \
                                '{' function_code_block '}'

func_def_args : args_format_specifier
              | empty

function_code_block : statement
                    | function_code_block statement
```

**Special Function**

```
sp_definition_specifier : SFC postfix_expression postfix_expression ';'

special_func_defintion_specifier : SPECIAL_FUNCTION postfix_expression '{'  special_func_def_list '}'

special_func_def_list : special_func_def
                      | special_func_def_list special_func_def

special_func_def : function_call  ';'
```

**Function Arguments**

```
args_access_format : '(' args_format_specifier ')'

args_type_specifier : dir LIST postfix_expression
                    | dir STR  postfix_expression

args_format_specifier : args_type_specifier
                      | args_format_specifier ',' args_type_specifier
                      | args_list_format
                      | args_format_specifier ',' args_list_format
                      | postfix_expression
                      | args_format_specifier ',' postfix_expression
                      | args_size_dir
                      | args_format_specifier ',' args_size_dir

args_list_format : '{' args_format_specifier '}'
                 | '{' '}'

args_def_list_format : '{' args_size_multi '}'

args_size_dir : dir args_def_list_format
              | dir args_size

args_size_multi : args_size
                | args_size_multi ',' args_size

args_size :     BIT postfix_expression
              | BIT array_access_format postfix_expression
```

**Print Command**

print_command : PRINTLN '(' STR_CONST ')'

**Pragmas**

pragma_call : directive NPL_PRAGMA pragma_access_format

directive : PRAGMA

pragma_access_format : '(' postfix_expression ',' pragma_format_specifier ')'

pragma_format_specifier : pragma_format_specifier ',' \
| postfix_expression ':' postfix_expression
| postfix_expression

# 12 Appendix E: Directives (@NPL_PRAGMA)

To assist compilers, Directives may be specified in the application program. These Directives are used by FE and BE compilers to do placement and other functions. Directives are NOT part of NPL. However, to better aid understanding, directive syntax and usage examples will be described.

## 12.1 Directives

Directives assist in specifying desired behavior which may have hardware dependencies. Directives assist in BE Mapping.

Rules:

- Directives may be specified in the NPL logic file or in a separate file.
- There is no positioning associated with directives.
- Directives must start with newline.
- Use keyword, "null", in case there is no object associated with a directive or multiple objects require a directive.

Syntax and keyword to specify the directives is -

```
@NPL_PRAGMA(object_name, property_name:property_value);
```

**Example**

**SPECIFYING THE BUS_TYPE DIRECTIVE**

Say in an application user wants to define fixed bus here is how to specify it -

```
bus mpls_fixed_bus_s mpls_fixed_bus;
@NPL_PRAGMA(mpls_fixed_bus, bus_type:ing_obj_fixed);
```

**SPECIFYING THE MAPPING DIRECTIVE**

To map a particular table, function, sfc or bus_field to a specific hardware block -

```
function vlan_assign_functions()
@NPL_PRAGMA(vlan_assign_functions, mapping:"hw_proc_block_20")
```

# 13 Examples of Target Extern Functions

### Packet Drop

Packet Drop uses the following packet_drop extern function.

**Table 27: packet_drop**

| *Construct* | *Arguments/Options* | *Description* |
|---|---|---|
| packet_drop | | Drop the packet. |
| | bus.field_name | Specify the Field Name which if set means the packet must be dropped.<br>This can be: <bus.field><br>Within logical_table fields_assign(), this must be a <table_field>. |
| | drop_code | Specify a constant associated with this drop reason.<br>This can be: constant/enum<br>● Valid values: 0-255<br>● Value of 0 means, No Drop Code. |
| | strength | Specify the priority or strength associated with this drop.<br>This can be: constant or register field |

```
packet_drop(
    <bus.field_name>,  // signal name on which user wants to connect drop
    <drop_code>,       // constant drop opcode
    <strength>         // Priority for drop opcode
 );
```

### Packet Trace

Packet Trace uses the following packet_trace extern function.

**Table 28: packet_trace**

| *Construct* | *Arguments/ Options* | *Description* |
|---|---|---|
| packet_trace | | Trace the packet. |
| | bus.field_name | Specify the Field Name which if set means the packet must be traced.<br>This can be: <bus.field><br>Within logical_table fields_assign(), this must be a <table_field>. |

| | | |
|---|---|---|
| | trace_code | Specify the trace code associated with this trace.<br>This can be: constant/enum<br>● Valid values: 0-47<br>● Value of 0 means, no trace code. |

```
packet_trace(
    <bus.field_name>,    // signal name on which user wants to connect trace
    <trace_code>         // constant trace opcode
 );
```

**Packet Count**

Packet Count uses the following packet_count extern function.

## Table 29: packet_count

| ***Construct*** | ***Arguments/ Options*** | ***Description*** |
|---|---|---|
| packet_count | | Count the packet. |
| | bus.field_name | Specify the Field Name which if set means the packet must be counted.<br>This can be: <bus.field><br>Within logical_table fields_assign(), this must be 0 or 1 |
| | counter_id | Specify a constant counter id associated with this counter.<br>This can be: constant/enum<br>● Valid values: 1-63<br>Within logical_table fields_assign(), this can be a <table_field>. |

```
packet_count(
    <bus.field_name>,    // signal name on which user wants to connect count
    <counter_id>         // constant counter id
 );
```

**Example**

USE OF PACKET_DROP, PACKET_TRACE AND PACKET_COUNT

The following is an example of count, trace and drop. The following is needed:

- Drop all IPV4 packets with ttl as 0; used drop_code 11 and priority 5.
- Trace all IPV4 packets with ttl as 1
- Count all IPV4 packets with ttl as 2

```
struct cond_bus_s {
    bit    ttl_0;
    bit    ttl_1;
```

```
    bit    ttl_2;
}

bus cond_bus_s cond_bus;
#define TTL0 11

function func_ttl_proc () {
    if (header_ipv4.ttl == 0) {
        cond_bus.ttl_0 = 1;
    }
    if (header_ipv4.ttl == 1) {
        cond_bus.ttl_1 = 1;
    }
    if (header_ipv4.ttl == 2) {
        cond_bus.ttl_2 = 1;
    }
    packet_drop(cond_bus.ttl_0, TTL0, 5);
    packet_trace(cond_bus.ttl_1, 3);
    packet_count(cond_bus.ttl_2, 1);
}

program ipv4() {
    ...
    func_ttl_proc();
    ...
}
```