Proposal for serializable enums in P4

P4 Serializable Types Subcommittee

Abstract

We propose extending the enumeration type to allow specification of a backing type and representation, allowing such enums to be used as serializable data.

1. Making serializable enums

P4's enum type is currently left up to the compiler to determine the representation and backing type for the enum, however, as the current P4 spec acknowledges, there are instances where an enum is exposed to the control plane, and a compiler must specify how it will handle such enumerations in communication with the control plane.

Instead of having this be a place where one implementation might vary from another, we propose that the enum type be extended to allow specifying both the backing type and the numeric representation of enumeration items.

We propose an extending enum as follows:

```
enumDeclaration
    : optAnnotations ENUM name '{' identifierList '}'
    | optAnnotations ENUM typeRef name '{' specifiedIdentifierList '}'

specifiedIdentifierList
    : specifiedIdentifier
    | specifiedIdentifierList ', ' specifiedIdentifier

specifiedIdentifier
    : name '=' initializer
```

This keeps the original enum for use internal to the P4 program, but also provides a serializable enum with both the backing type and the numeric representation specified. In this case, we expect the `initializer` to be a compile-time known values.

We imagine a simple example looking something like the following:

```
enum bit<8> example_t {
  first  = 0,
  second = 1,
  third  = 2
}
```

This specifies that `example_t` is a serializable `enum` serializable to the type `bit<8>` (following the syntax for `typedef` of `typeRef` followed by `name`) with three entries `first`, which serializes to `0`; `second`, which serializes to `1`; and `third`, which serializes to `2`.

If we added a new entry that was not representable in the `bit<8>`, we would expect the compiler to raise an error.

```
enum bit<8> example_t {
  first  = 0,
  second = 1,
  third  = 2,
  more   = 300       // compiler would raise error.
}
```

One use case we can imagine is for specifying things like ethernet types to use within an ethernet header:

```
enum bit<16> etherType_t {
  BF_FABRIC = 0x9000,
  VLAN      = 0x8100,
  QINQ      = 0x9100,
  MPLS      = 0x8847,
  IPV4      = 0x0800,
  IPV6      = 0x86dd,
  ARP       = 0x0806,
  RARP      = 0x8035,
  NSH       = 0x894f,
  ETHERNET  = 0x6558,
  ROCE      = 0x8915,
  FCOE      = 0x8906,
  TRILL     = 0x22f3,
  VNTAG     = 0x8926,
  LLDP      = 0x88cc,
  LACP      = 0x8809
  // ...
}

typedef bit<48> macAddr_t;

header ether_t {
  macAddr_t   dstAddr;
  macAddr_t   srcAddr;
  etherType_t etherType;
}
```

2. Handling incomplete enums

An incomplete enumeration is one that does not have a symbol for every numeric value expressible in the backing type. The etherType_t enum example above is an illustration of this. This is an issue for any operation that translates a numeric value that is does not have an associated symbol in the enum. For example, both packet_in.extract and a cast from bit<16> to the etherType_t could result in the enum having a value that does not correspond to the enum.

There are a few ways to handle this situation. First, we could simply disallow serializable enum types in headers and disallow casts to enums from bit<n> values. This eliminates the problem, but limits the usefulness of enum values in the data plane. Second, we could allow the result of an extract or a cast to hold an undefined value, in the same way an unitialized value contains an unsigned value. Thus:

```
etherType_t et;

// et has an undefined value here

ether_t h;

in.extract(h);
```

```
// h.etherType may have an undefined value here

bit<16> x = /* some valid 16-bit value */;
etherType_t et = x;

// et may have an undefined value here
```

Finally, we could introduce a `convert` extern to allow for a safe cast to a serializable enum. The `covert` function would be defined as follows

```
extern void convert<S, D>(in S data, in T default, out T value);
```

With the `S` type being the `bit<n>` type we are converting from and `T` representing the seralizable enum type we are converting to. The extern would take three arguments, `data` the `bit<n>` value to be converted, `default` a default value when the `data` value falls outside the symbolic values of the enum, and an output `value` argument that will contain the converted value. For example, we could use this with by extending `etherType_t` with an `UNRECOGNIZED` type, using this as the default:

```
enum bit<16> etherType_t {
  BF_FABRIC    = 0x9000,
  VLAN         = 0x8100,
  QINQ         = 0x9100,
  MPLS         = 0x8847,
  IPV4         = 0x0800,
  IPV6         = 0x86dd,
  ARP          = 0x0806,
  RARP         = 0x8035,
  NSH          = 0x894f,
  ETHERNET     = 0x6558,
  ROCE         = 0x8915,
  FCOE         = 0x8906,
  TRILL        = 0x22f3,
  VNTAG        = 0x8926,
  LLDP         = 0x88cc,
  LACP         = 0x8809
  // ...
  UNRECOGNIZED = 0xFFFF
}
```

```
etherType_t et;

bit<16> x = /* some valid 16-bit value */;
convert<bit<16>,etherType_t>(x, UNRECOGNIZED, et);

// et is a valid etherType_t value, either one corresponding to x or
// UNRECOGNIZED in the case where x does not correspond to a known ether-
Type.
```

In the case of `packet_in.extract` we might alternatively choose to raise an error when an unsupported value is raised.

3. Handling multiple symbols for a type

Another issue is that an `enum` could contain multiple numeric values for a single symbol. If these symbols are intended to not be equivalent, than this raises an issue in the representation. We propose either disallowing multiple symbols from having the same value, or treating the two symbols as interchangeable in the program.