

# 4 Constant Fields and Constant when Subtypes

You can use the **const** field modifier to identify a field whose value must be kept constant. Doing so can result in significantly improved performance.

The **const** field modifier is described in the following section.

## 4.1 const Field Modifier

---

### Purpose

Identify a field whose value should be kept constant throughout the lifetime of the object.

### Category

Keyword

### Syntax

**[private | protected | package] const** [!][%]*field-name*[: *scalar* | *struct type*]

### Syntax Example

```
extend packet {  
    const kind : packet_kind;  
};
```

## Description

Identifies a field whose value should be kept constant throughout the lifetime of the object and enforces the constant value.

The *e* compiler takes advantage of **const** declarations to optimize memory performance. You can expect significant memory performance improvement if a field that serves as a **when** determinant is declared as **const**.

## Initializing const Fields

Initialization of fields declared as **const** must be completed during the creation phase of a struct.

- Your code can assign a value to a **const** field:
  - During generation, with **pre\_generate()** or **post\_generate()**
  - During unpacking, with **do\_unpack()**
  - With an assignment action while creating an object, with a **new...with** action block or the **init()** method

In these contexts, a **const** field of the newly created struct may figure on the left side of an assignment operator.

- You can also initialize a **const** field with a built-in initialization mechanism using the **copy()** or **read\_binary\_struct()** method.

A **const** field can be assigned a value only once:

- A compile-time error occurs if a **const** field is initialized with any construct other than those listed above.
- A run-time error occurs if a **const** field is assigned more than one value.

## Notes

- The **const** modifier cannot be applied to fields that are accessed by tick access notation. (This notation is undocumented.)
- Constant fields cannot be passed by reference.
- Fields of **list** type cannot be declared **const**.
- Fields of **enum** types that are declared **const** have no default value upon creation of the struct (even if zero is a possible enumerated value). They must be initialized in the ways described in [“Initializing const Fields” on page 4-2](#).

- Fields declared under **when** subtypes with a non-constant determinant cannot be declared **const**.

## Example

```
type packet_kind      : [SPLIT, TOKEN, SOF, DATA, HANDSHAKE, PREAMBLE];
struct packet {
    const %kind : packet_kind;
    %data : list of int;
};

extend driver {
    !pkt : packet;

    send_packets() {
        // pkt = new;
        // pkt.kind = SOF;      // Illegal - will cause compile time error
        pkt = new packet with {
            .kind = SOF;
        };
        send( pkt );
        for i from 0 to 10 {
            gen pkt keeping {
                .kind in [TOKEN,DATA,HANDSHAKE]; // ***
            };
            send( pkt );
        };
    };

    rcv_packet(lob : list of byte)@packet_received {
        var rcvd_pkt : packet;
        unpack(packing.low,lob,rcvd_pkt);
    };
};

extend packet {
    post_generate() {
        if should_be_token() {
            kind = TOKEN;      // Illegal - will cause run time error
                               // because generation in line marked
                               // by *** assigned the value;
        };
    };
};
```

