NAME

ovs−fields − protocol header fields in OpenFlow and Open vSwitch

INTRODUCTION

This document aims to comprehensively document all of the fields, both standard and non-standard, supported by OpenFlow or Open vSwitch, regardless of origin.

Fields

A *field* is a property of a packet. Most familiarly, *data fields* are fields that can be extracted from a packet. Most data fields are copied directly from protocol headers, e.g. at layer 2, the Ethernet source and destination addresses, or the VLAN ID; at layer 3, the IPv4 or IPv6 source and destination; and at layer 4, the TCP or UDP ports. Other data fields are computed, e.g. **ip_frag** describes whether a packet is a fragment but it is not copied directly from the IP header.

Some data fields, called *root fields*, are always present as a consequence of the basic networking technology in use. The Ethernet header fields are root fields in current versions of Open vSwitch, though future versions might support other roots. (Currently, to support LISP tunnels, which do not encapsulate an Ethernet header, Open vSwitch synthesizes one.)

Other data fields are not always present. A packet contains ARP fields, for example, only when its Ethernet header indicates the Ethertype for ARP, 0x0806. In this documentation, we say that a field is *applicable* when it is it present in a packet, and *inapplicable* when it is not. (These are not standard terms.) We refer to the conditions that determine whether a field is applicable as *prerequisites*. Some VLAN-related fields are a special case: these fields are always applicable, but have a designated value or bit that indicates whether a VLAN header is present, with the remaining values or bits indicating the VLAN header's content (if it is present).

An inapplicable field does not have a value, not even a nominal ‘‘value’’ such as all-zero-bits. In many circumstances, OpenFlow and Open vSwitch allow references only to applicable fields. For example, one may match (see *Matching*, below) a given field only if the match includes the field's prerequisite, e.g. matching an ARP field is only allowed if one also matches on Ethertype 0x0806.

Sometimes a packet may contain multiple instances of a header. For example, a packet may contain multiple VLAN or MPLS headers, and tunnels can cause any data field to recur. OpenFlow and Open vSwitch do not address these cases uniformly. For VLAN and MPLS headers, only the outermost header is accessible, so that inner headers may be accessed only by ‘‘popping’’ (removing) the outer header. (Open vSwitch supports only a single VLAN header in any case.) For tunnels, e.g. GRE or VXLAN, the outer header and inner headers are treated as different data fields.

Many network protocols are built in layers as a stack of concatenated headers. Each header typically contains a ‘‘next type’’ field that indicates the type of the protocol header that follows, e.g. Ethernet contains an Ethertype and IPv4 contains a IP protocol type. The exceptional cases, where protocols are layered but an outer layer does not indicate the protocol type for the inner layer, or gives only an ambiguous indication, are troublesome. An MPLS header, for example, only indicates whether another MPLS header or some other protocol follows, and in the latter case the inner protocol must be known from the context. In these exceptional cases, OpenFlow and Open vSwitch cannot provide insight into the inner protocol data fields without additional context, and thus they treat all later data fields as inapplicable until an OpenFlow action explicitly specifies what protocol follows. In the case of MPLS, the OpenFlow ‘‘pop MPLS’’ action that removes the last MPLS header from a packet provides this context, as the Ethertype of the payload. See *Layer 2.5: MPLS* for more information.

OpenFlow and Open vSwitch support some fields other than data fields. *Metadata fields* relate to the origin or treatment of a packet, but they are not extracted from the packet data itself. One example is the physical port on which a packet arrived at the switch. *Register fields* act like variables: they give an OpenFlow switch space for temporary storage while processing a packet. Existing metadata and register fields have no prerequisites.

A field's value consists of an integral number of bytes. For data fields, sometimes those bytes are taken directly from the packet. Other data fields are copied from a packet with padding (usually with zeros and in the most significant positions). The remaining data fields are transformed in other ways as they are copied

from the packets, to make them more useful for matching.

Matching

The most important use of fields in OpenFlow is *matching*, to determine whether particular field values agree with a set of constraints called a *match*. A match consists of zero or more constraints on individual fields, all of which must be met to satisfy the match. (A match that contain no constraints is always satisfied.) OpenFlow and Open vSwitch support a number of forms of matching on individual fields:

Exact match, e.g. **nw_src=10.1.2.3**

Only a particular value of the field is matched; for example, only one particular source IP address. Exact matches are written as *field*=*value*. The forms accepted for *value* depend on the field.

All fields support exact matches.

Bitwise match, e.g. **nw_src=10.1.0.0/255.255.0.0**

Specific bits in the field must have specified values; for example, only source IP addresses in a particular subnet. Bitwise matches are written as *field*=*value*/*mask*, where *value* and *mask* take one of the forms accepted for an exact match on *field*. Some fields accept other forms for bitwise matches; for example, **nw_src=10.1.0.0/255.255.0.0** may also be written **nw_src=10.1.0.0/16**.

Most OpenFlow switches do not allow every bitwise matching on every field (and before OpenFlow 1.2, the protocol did not even provide for the possibility for most fields). Even switches that do allow bitwise matching on a given field may restrict the masks that are allowed, e.g. by allowing matches only on contiguous sets of bits starting from the most significant bit, that is, ‘‘CIDR’’ masks [RFC 4632]. Open vSwitch does not allows bitwise matching on every field, but it allows arbitrary bitwise masks on any field that does support bitwise matching. (Older versions had some restrictions, as documented in the descriptions of individual fields.)

Wildcard, e.g. ‘‘any **ip_src**’’

The value of the field is not constrained. Wildcarded fields may be written as *field*=*****, although it is unusual to mention them at all. (When specifying a wildcard explicitly in a command invocation, be sure to using quoting to protect against shell expansion.)

There is a tiny difference between wildcarding a field and not specifying any match on a field: wildcarding a field requires satisfying the field’s prerequisites.

Some types of matches on individual fields cannot be expressed directly with OpenFlow and Open vSwitch. These can be expressed indirectly:

Set match, e.g. ‘‘**tcp_dst** ∈ {80, 443, 8080}’’

The value of a field is one of a specified set of values; for example, the TCP destination port is 80, 443, or 8080.

For matches used in flows (see *Flows*, below), multiple flows can simulate set matches.

Range match, e.g. ‘‘1000 ≤ **tcp_dst** ≤ 1999’’

The value of the field must lie within a numerical range, for example, TCP destination ports between 1000 and 1999.

Range matches can be expressed as a collection of bitwise matches. For example, suppose that the goal is to match TCP source ports 1000 to 1999, inclusive. The binary representations of 1000 and 1999 are:

```
01111101000
11111001111
```

The following series of bitwise matches will match 1000 and 1999 and all the values in between:

```
01111101xxx
0111111xxxx
10xxxxxxxx
110xxxxxxx
1110xxxxxx
11110xxxxx
1111100xxxx
```

which can be written as the following matches:

tcp,tp_src=0x03e8/0xfff8
tcp,tp_src=0x03f0/0xfff0
tcp,tp_src=0x0400/0xfe00
tcp,tp_src=0x0600/0xff00
tcp,tp_src=0x0700/0xff80
tcp,tp_src=0x0780/0xffc0
tcp,tp_src=0x07c0/0xfff0

Inequality match, e.g. ‘‘**tcp_dst** ≠ 80’’

The value of the field differs from a specified value, for example, all TCP destination ports except 80.

An inequality match on an *n*-bit field can be expressed as a disjunction of *n* 1-bit matches. For example, the inequality match ‘‘**vlan_pcp** ≠ 5’’ can be expressed as ‘‘**vlan_pcp** = 0/4 or **vlan_pcp** = 2/2 or **vlan_pcp** = 0/1.’’ For matches used in flows (see *Flows*, below), sometimes one can more compactly express inequality as a higher-priority flow that matches the exceptional case paired with a lower-priority flow that matches the general case.

Alternatively, an inequality match may be converted to a pair of range matches, e.g. **tcp_src** ≠ **80** may be expressed as ‘‘0 ≤ **tcp_src** < 80 or 80 < **tcp_src** ≤ 65535’’, and then each range match may in turn be converted to a bitwise match.

Conjunctive match, e.g. ‘‘**tcp_src** ∈ {80, 443, 8080} and **tcp_dst** ∈ {80, 443, 8080}’’

As an OpenFlow extension, Open vSwitch supports matching on conditions on conjunctions of the previously mentioned forms of matching. See the documentation for **conj_id** for more information.

All of these supported forms of matching are special cases of bitwise matching. In some cases this influences the design of field values. **ip_frag** is the most prominent example: it is designed to make all of the practically useful checks for IP fragmentation possible as a single bitwise match.

Shorthands

Some matches are very commonly used, so Open vSwitch accepts shorthand notations. In some cases, Open vSwitch also uses shorthand notations when it displays matches. The following shorthands are defined, with their long forms shown on the right side:

ip	**eth_type=0x0800**
ipv6	**eth_type=0x86dd**
icmp	**eth_type=0x0800,ip_proto=1**
icmp6	**eth_type=0x86dd,ip_proto=58**
tcp	**eth_type=0x0800,ip_proto=6**
tcp6	**eth_type=0x86dd,ip_proto=6**
udp	**eth_type=0x0800,ip_proto=17**
udp6	**eth_type=0x86dd,ip_proto=17**

sctp	**eth_type=0x0800,ip_proto=132**
sctp6	**eth_type=0x86dd,ip_proto=132**
arp	**eth_type=0x0806**
rarp	**eth_type=0x8035**
mpls	**eth_type=0x8847**
mplsm	**eth_type=0x8848**

Evolution of OpenFlow Fields

The discussion so far applies to all OpenFlow and Open vSwitch versions. This section starts to draw in specific information by explaining, in broad terms, the treatment of fields and matches in each OpenFlow version.

OpenFlow 1.0

OpenFlow 1.0 defined the OpenFlow protocol format of a match as a fixed-length data structure that could match on the following fields:

- Ingress port.
- Ethernet source and destination MAC.
- Ethertype (with a special value to match frames that lack an Ethertype).
- VLAN ID and priority.
- IPv4 source, destination, protocol, and DSCP.
- TCP source and destination port.
- UDP source and destination port.
- ICMPv4 type and code.
- ARP IPv4 addresses (SPA and TPA) and opcode.

Each supported field corresponded to some member of the data structure. Some members represented multiple fields, in the case of the TCP, UDP, ICMPv4, and ARP fields whose presence is mutually exclusive. This also meant that some members were poor fits for their fields: only the low 8 bits of the 16-bit ARP opcode could be represented, and the ICMPv4 type and code were padded with 8 bits of zeros to fit in the 16-bit members primarily meant for TCP and UDP ports. An additional bitmap member indicated, for each member, whether its field should be an ‘‘exact’’ or ‘‘wildcarded’’ match (see *Matching*), with additional support for CIDR prefix matching on the IPv4 source and destination fields.

Simplicity was recognized early on as the main virtue of this approach. Obviously, any fixed-length data structure cannot support matching new protocols that do not fit. There was no room, for example, for matching IPv6 fields, which was not a priority at the time. Lack of room to support matching the Ethernet addresses inside ARP packets actually caused more of a design problem later, leading to an Open vSwitch extension action specialized for dropping ‘‘spoofed’’ ARP packets in which the frame and ARP Ethernet source addressed differed. (This extension was never standardized. Open vSwitch dropped support for it a few releases after it added support for full ARP matching.)

The design of the OpenFlow fixed-length matches also illustrates compromises, in both directions, between the strengths and weaknesses of software and hardware that have always influenced the design of OpenFlow. Support for matching ARP fields that do fit in the data structure was only added late in the design process (and remained optional in OpenFlow 1.0), for example, because common switch ASICs did not support matching these fields.

The compromises in favor of software occurred for more complicated reasons. The OpenFlow designers did not know how to implement matching in software that was fast, dynamic, and general. (A way was later found [Srinivasan].) Thus, the designers sought to support dynamic, general matching that would be fast in realistic special cases, in particular when all of the matches were *microflows*, that is, matches that specify every field present in a packet, because such matches can be implemented as a single hash table lookup.

Contemporary research supported the feasibility of this approach: the number of microflows in a campus network had been measured to peak at about 10,000 [Casado, section 3.2]. (Calculations show that this can only be true in a lightly loaded network [Pepelnjak].)

As a result, OpenFlow 1.0 required switches to treat microflow matches as the highest possible priority. This let software switches perform the microflow hash table lookup first. Only on failure to match a microflow did the switch need to fall back to checking the more general and presumed slower matches. Also, the OpenFlow 1.0 flow match was minimally flexible, with no support for general bitwise matching, partly on the basis that this seemed more likely amenable to relatively efficient software implementation. (CIDR masking for IPv4 addresses was added relatively late in the OpenFlow 1.0 design process.)

Microflow matching was later discovered to aid some hardware implementations. The TCAM chips used for matching in hardware do not support priority in the same way as OpenFlow but instead tie priority to ordering [Pagiamtzis]. Thus, adding a new match with a priority between the priorities of existing matches can require reordering an arbitrary number of TCAM entries. On the other hand, when microflows are highest priority, they can be managed as a set-aside portion of the TCAM entries.

The emphasis on matching microflows also led designers to carefully consider the bandwidth requirements between switch and controller: to maximize the number of microflow setups per second, one must minimize the size of each flow's description. This favored the fixed-length format in use, because it expressed common TCP and UDP microflows in fewer bytes than more flexible ''type-length-value'' (TLV) formats. (Early versions of OpenFlow also avoided TLVs in general to head off protocol fragmentation.)

Inapplicable Fields

OpenFlow 1.0 does not clearly specify how to treat inapplicable fields. The members for inapplicable fields are always present in the match data structure, as are the bits that indicate whether the fields are matched, and the ''correct'' member and bit values for inapplicable fields is unclear. OpenFlow 1.0 implementations changed their behavior over time as priorities shifted. The early OpenFlow reference implementation, motivated to make every flow a microflow to enable hashing, treated inapplicable fields as exact matches on a value of 0. Initially, this behavior was implemented in the reference controller only.

Later, the reference switch was also changed to actually force any wildcarded inapplicable fields into exact matches on 0. The latter behavior sometimes caused problems, because the modified flow was the one reported back to the controller later when it queried the flow table, and the modifications sometimes meant that the controller could not properly recognize the flow that it had added. In retrospect, perhaps this problem should have alerted the designers to a design error, but the ability to use a single hash table was held to be more important than almost every other consideration at the time.

When more flexible match formats were introduced much later, they disallowed any mention of inapplicable fields as part of a match. This raised the question of how to translate between this new format and the OpenFlow 1.0 fixed format. It seemed somewhat inconsistent and backward to treat fields as exact-match in one format and forbid matching them in the other, so instead the treatment of inapplicable fields in the fixed-length format was changed from exact match on 0 to wildcarding. (A better classifier had by now eliminated software performance problems with wildcards.)

The OpenFlow 1.0.1 errata (released only in 2012) added some additional explanation [OpenFlow 1.0.1, section 3.4], but it did not mandate specific behavior because of variation among implementations.

OpenFlow 1.1

The OpenFlow 1.1 protocol match format was designed as a type/length/value (TLV) format to allow for future flexibility. The specification standardized only a single type **OFPMT_STANDARD** (0) with a fixed-size payload, described here. The additional fields and bitwise masks in OpenFlow 1.1 cause this match structure to be over twice as large as in OpenFlow 1.0, 88 bytes versus 40.

OpenFlow 1.1 added support for the following fields:

- SCTP source and destination port.
- MPLS label and traffic control (TC) fields.

- One 64-bit register (named ‘‘metadata’’).

OpenFlow 1.1 increased the width of the ingress port number field (and all other port numbers in the protocol) from 16 bits to 32 bits.

OpenFlow 1.1 increased matching flexibility by introducing arbitrary bitwise matching on Ethernet and IPv4 address fields and on the new ‘‘metadata’’ register field. Switches were not required to support all possible masks [OpenFlow 1.1, section 4.3].

By a strict reading of the specification, OpenFlow 1.1 removed support for matching ICMPv4 type and code [OpenFlow 1.1, section A.2.3], but this is likely an editing error because ICMP matching is described elsewhere [OpenFlow 1.1, Table 3, Table 4, Figure 4]. Open vSwitch does support ICMPv4 type and code matching with OpenFlow 1.1.

OpenFlow 1.1 avoided the pitfalls of inapplicable fields that OpenFlow 1.0 encountered, by requiring the switch to ignore the specified field values [OpenFlow 1.1, section A.2.3]. It also implied that the switch that should ignore the bits that indicate whether to match inapplicable fields.

Physical Ingress Port

OpenFlow 1.1 introduced a new pseudo-field, the physical ingress port. The physical ingress port is only a pseudo-field because it cannot be used for matching. It appears only one place in the protocol, in the ‘‘packet-in’’ message that passes a packet received at the switch to an OpenFlow controller.

A packet’s ingress port and physical ingress port are identical except for packets processed by a switch feature such as bonding or tunneling that makes a packet appear to arrive on a ‘‘virtual’’ port associated with the bond or the tunnel. For such packets, the ingress port is the virtual port and the physical ingress port is, naturally, the physical port. Open vSwitch implements both bonding and tunneling, but its bonding implementation does not use virtual ports and its tunnels are typically not on the same OpenFlow switch as their physical ingress ports (which need not be part of any switch), so the ingress port and physical ingress port are always the same in Open vSwitch.

OpenFlow 1.2

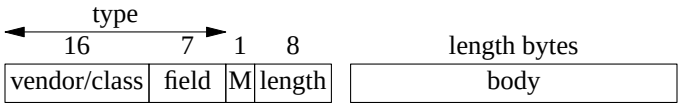
OpenFlow 1.2 abandoned the fixed-length approach to matching. One reason was size, since adding support for IPv6 address matching (now seen as important), with bitwise masks, would have added 64 bytes to the match length, increasing it from 88 bytes in OpenFlow 1.1 to over 150 bytes. Extensibility had also become important as controller writers increasingly wanted support for new fields without having to change messages throughout the OpenFlow protocol. The challenges of carefully defining fixed-length matches to avoid problems with inapplicable fields had also become clear over time.

Therefore, OpenFlow 1.2 adopted a flow format using a flexible type-length-value (TLV) representation, in which each TLV expresses a match on one field. These TLVs were in turn encapsulated inside the outer TLV wrapper introduced in OpenFlow 1.1 with the new identifier **OFPMT_OXM** (1). (This wrapper fulfilled its intended purpose of reducing the amount of churn in the protocol when changing match formats; some messages that included matches remained unchanged from OpenFlow 1.1 to 1.2 and later versions.)

OpenFlow 1.2 added support for the following fields:

- ARP hardware addresses (SHA and THA).
- IPv4 ECN.
- IPv6 source and destination addresses, flow label, DSCP, ECN, and protocol.
- TCP, UDP, and SCTP port numbers when encapsulated inside IPv6.
- ICMPv6 type and code.
- ICMPv6 Neighbor Discovery target address and source and target Ethernet addresses.

The OpenFlow 1.2 format, called *OXM* (*OpenFlow Extensible Match*), was modeled closely on an extension to OpenFlow 1.0 introduced in Open vSwitch 1.1 called *NXM* (*Nicira Extended Match*). Each OXM or NXM TLV has the following format:



The most significant 16 bits of the NXM or OXM header, called **vendor** by NXM and **class** by OXM, identify an organization permitted to allocate identifiers for fields. NXM allocates only two vendors, 0x0000 for fields supported by OpenFlow 1.0 and 0x0001 for fields implemented as an Open vSwitch extension. OXM assigns classes as follows:

0x0000 (**OFPXMC_NXM_0**).
0x0001 (**OFPXMC_NXM_1**).
Reserved for NXM compatibility.

0x0002 to 0x7fff
Reserved for allocation to ONF members, but none yet assigned.

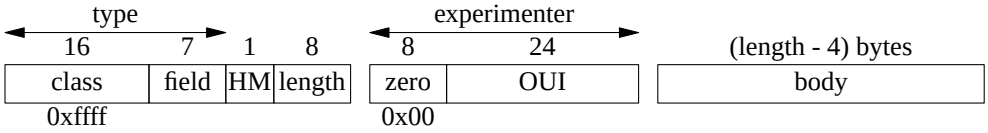
0x8000 (**OFPXMC_OPENFLOW_BASIC**)
Used for most standard OpenFlow fields.

0x8001 (**OFPXMC_PACKET_REGS**)
Used for packet register fields in OpenFlow 1.5 and later.

0x8002 to 0xfffe
Reserved for the OpenFlow specification.

0xffff (**OFPXMC_EXPERIMENTER**)
Experimental use.

When **class** is 0xffff, the OXM header is extended to 64 bits by using the first 32 bits of the body as an **experimenter** field whose most significant byte is zero and whose remaining bytes are an Organizationally Unique Identifier (OUI) assigned by the IEEE [IEEE OUI], as shown below. OpenFlow says that support for experimenter fields is optional. Open vSwitch 2.4 and later does support them, primarily so that it can support the **ONFOXM_ET_*** code points defined by official Open Networking Foundation extensions to OpenFlow 1.3 in e.g. [TCP Flags Match Field Extension].



Taken as a unit, **class** (or **vendor**), **field**, and **experimenter** (when present) uniquely identify a particular field.

When **hasmask** (abbreviated **HM** above) is 0, the OXM is an exact match on an entire field. In this case, the body (excluding the experimenter field, if present) is a single value to be matched.

When **hasmask** is 1, the OXM is a bitwise match. The body (excluding the experimenter field) consists of a value to match, followed by the bitwise mask to apply. A 1-bit in the mask indicates that the corresponding bit in the value should be matched and a 0-bit that it should be ignored. For example, for an IP address field, a value of 192.168.0.0 followed by a mask of 255.255.0.0 would match addresses in the 196.168.0.0/16 subnet.

- Some fields might not support masking at all, and some fields that do support masking might restrict it to certain patterns. For example, fields that have IP address values might be restricted to CIDR masks. The descriptions of individual fields note these restrictions.
- An OXM TLV with a mask that is all zeros is not useful (although it is not forbidden), because it is has the same effect as omitting the TLV entirely.
- It is not meaningful to pair a 0-bit in an OXM mask with a 1-bit in its value, and Open vSwitch rejects such an OXM with the error **OFPBMC_BAD_WILDCARDS**, as

required by OpenFlow 1.3 and later.

The **length** identifies the number of bytes in the body, including the 4-byte **experimenter** header, if it is present. Each OXM TLV has a fixed length; that is, given **class**, **field**, **experimenter** (if present), and **hasmask**, **length** is a constant. The **length** is included explicitly to allow software to minimally parse OXM TLVs of unknown types.

OXM TLVs must be ordered so that a field's prerequisites are satisfied before it is parsed. For example, an OXM TLV that matches on the IPv4 source address field is only allowed following an OXM TLV that matches on the Ethertype for IPv4. Similarly, an OXM TLV that matches on the TCP source port must follow a TLV that matches an Ethertype of IPv4 or IPv6 and one that matches an IP protocol of TCP (in that order). The order of OXM TLVs is not otherwise restricted; no canonical ordering is defined.

A given field may be matched only once in a series of OXM TLVs.

OpenFlow 1.3

OpenFlow 1.3 showed OXM to be largely successful, by adding new fields without making any changes to how flow matches otherwise worked. It added OXMs for the following fields supported by Open vSwitch:

- Tunnel ID for ports associated with e.g. VXLAN or keyed GRE.
- MPLS ''bottom of stack'' (BOS) bit.

OpenFlow 1.3 also added OXMs for the following fields not documented here and not yet implemented by Open vSwitch:

- IPv6 extension header handling.
- PBB I-SID.

OpenFlow 1.4

OpenFlow 1.4 added OXMs for the following fields not documented here and not yet implemented by Open vSwitch:

- PBB UCA.

OpenFlow 1.5

OpenFlow 1.5 added OXMs for the following fields supported by Open vSwitch:

- TCP flags.
- Packet registers.
- The output port in the OpenFlow action set.

OpenFlow 1.5 also added OXMs for the following fields not documented here and not yet implemented by Open vSwitch:

- Packet type.

FIELDS REFERENCE

The following sections document the fields that Open vSwitch supports. Each section provides introductory material on a group of related fields, followed by information on each individual field. In addition to field-specific information, each field begins with a table with entries for the following important properties:

Name The field's name, used for parsing and formatting the field, e.g. in **ovs−ofctl** commands. For historical reasons, some fields have an additional name that is accepted as an alternative in parsing. This name, when there is one, is listed as well, e.g. ''**tun** (aka **tunnel_id**).''

Width The field's width, always a multiple of 8 bits. Some fields don't use all of the bits, so this may be accompanied by an explanation. For example, OpenFlow embeds the 2-bit IP ECN field as as the low bits in an 8-bit byte, and so its width is expressed as ''8 bits (only the least-significant 2 bits may be nonzero).''

Format How a value for the field is formatted or parsed by, e.g., **ovs−ofctl**. Some possibilities are generic:

decimal
: Formats as a decimal number. On input, accepts decimal numbers or hexadecimal numbers prefixed by **0x**.

hexadecimal
: Formats as a hexadecimal number prefixed by **0x**. On input, accepts decimal numbers or hexadecimal numbers prefixed by **0x**. (The default for parsing is **not** hexadecimal: only a **0x** prefix causes input to be treated as hexadecimal.)

Ethernet
: Formats and accepts the common Ethernet address format *xx***:***xx***:***xx***:***xx***:***xx***:***xx*.

IPv4 Formats and accepts the dotted-quad format *a***.***b***.***c***.***d*. For bitwise matches, formats and accepts *address*/*length* CIDR notation in addition to *address*/*mask*.

IPv6 Formats and accepts the common IPv6 address formats, plus CIDR notation for bitwise matches.

OpenFlow 1.0 port
: Accepts 16-bit port numbers in decimal, plus OpenFlow well-known port names (e.g. **IN_PORT**) in uppercase or lowercase.

OpenFlow 1.1+ port
: Same syntax as OpenFlow 1.0 ports but for 32-bit OpenFlow 1.1+ port number fields.

Other, field-specific formats are explained along with their fields.

Masking
: For most fields, this says ‘‘arbitrary bitwise masks,’’ meaning that a flow may match any combination of bits in the field. Some fields instead say ‘‘exact match only,’’ which means that a flow that matches on this field must match on the whole field instead of just certain bits. Either way, this reports masking support for the latest version of Open vSwitch using OXM or NXM (that is, either OpenFlow 1.2+ or OpenFlow 1.0 plus Open vSwitch NXM extensions). In particular, OpenFlow 1.0 (without NXM) and 1.1 don’t always support masking even if Open vSwitch itself does; refer to the **OpenFlow 1.0** and **OpenFlow 1.1** rows to learn about masking with these protocol versions.

Prerequisites
: Requirements that must be met to match on this field. For example, **ip_src** has IPv4 as a prerequisite, meaning that a match must include **eth_type=0x0800** to match on the IPv4 source address. The following prerequisites, with their requirements, are currently in use:

none (no requirements)

VLAN VID
: **vlan_tci=0x1000/0x1000** (i.e. a VLAN header is present)

ARP **eth_type=0x0806** (ARP) or **eth_type=0x8035** (RARP)

IPv4 **eth_type=0x0800**

IPv6 **eth_type=0x86dd**

IPv4/IPv6
: IPv4 or IPv6

MPLS **eth_type=0x8847** or **eth_type=0x8848**

TCP IPv4/IPv6 and **ip_proto=6**

UDP IPv4/IPv6 and **ip_proto=17**

SCTP IPv4/IPv6 and **ip_proto=132**

ICMPv4
: IPv4 and **ip_proto=1**

ICMPv6
: IPv6 and **ip_proto=1**

ND solicit
: ICMPv6 and **icmp_type=135** and **icmp_code=0**

ND advert
: ICMPv6 and **icmp_type=136** and **icmp_code=0**

ND ND solicit or ND advert

The TCP, UDP, and SCTP prerequisites also have the special requirement that **nw_frag** is not being used to select ‘‘later fragments.’’ This is because only the first fragment of a fragmented IPv4 or IPv6 datagram contains the TCP or UDP header.

Access Most fields are ‘‘read/write,’’ which means that common OpenFlow actions like **set_field** can modify them. Fields that are ‘‘read-only’’ cannot be modified in these general-purpose ways, although there may be others ways that actions can modify them.

OpenFlow 1.0
OpenFlow 1.1
: These rows report the level of support that OpenFlow 1.0 or OpenFlow 1.1, respectively, has for a field. For OpenFlow 1.0, supported fields are reported as either ‘‘yes (exact match only)’’ for fields that do not support any bitwise masking or ‘‘yes (CIDR match only)’’ for fields that support CIDR masking. OpenFlow 1.1 supported fields report either ‘‘yes (exact match only)’’ or simply ‘‘yes’’ for fields that do support arbitrary masks. These OpenFlow versions supported a fixed collection of fields that cannot be extended, so many more fields are reported as ‘‘not supported.’’

OXM
NXM These rows report the OXM and NXM code points that correspond to a given field. Either or both may be ‘‘none.’’

A field that has only an OXM code point is usually one that was standardized before it was added to Open vSwitch. A field that has only an NXM code point is usually one that is not yet standardized. When a field has both OXM and NXM code points, it usually indicates that it was introduced as an Open vSwitch extension under the NXM code point, then later standardized under the OXM code point. A field can have more than one OXM code point if it was standardized in OpenFlow 1.4 or later and additionally introduced as an official ONF extension for OpenFlow 1.3. (A field that has neither OXM nor NXM code point is typically an obsolete field that is supported in some other form using OXM or NXM.)

Each code point in these rows is described in the form ‘‘**NAME** (*number*) since OpenFlow *spec* and Open vSwitch *version*,’’ e.g. ‘‘**OXM_OF_ETH_TYPE** (5) since OpenFlow 1.2 and Open vSwitch 1.7.’’ The named OpenFlow *spec*, which is the version of OpenFlow that standardized the code point, is omitted for NXM code points because they are nonstandard. The *version* is the version of Open vSwitch that first supported the code point. The **NAME**, which specifies a name for the code point, starts with a prefix that designates a class and, in some cases, a vendor, as listed in the following table. Refer back to **OpenFlow 1.2** under **Evolution of OpenFlow Fields** for more information on OXM/NXM classes and vendors. Finally, *number* is the field number within the class and vendor.

Prefix	Vendor	Class
NXM_OF	(none)	`0x0000`

NXM_NX	(none)	`0x0001`
OXM_OF	(none)	`0x8000`
OXM_OF_PKT_REG	(none)	`0x8001`
NXOXM_ET	`0x00002320`	`0xffff`
ONFOXM_ET	`0x4f4e4600`	`0xffff`

CONJUNCTIVE MATCH FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
conj_id	4	no	no	none	OVS 2.4+

An individual OpenFlow flow can match only a single value for each field. However, situations often arise where one wants to match one of a set of values within a field or fields. For matching a single field against a set, it is straightforward and efficient to add multiple flows to the flow table, one for each value in the set. For example, one might use the following flows to send packets with IP source address *a*, *b*, *c*, or *d* to the OpenFlow controller:

ip,ip_src=*a* **actions=controller**
ip,ip_src=*b* **actions=controller**
ip,ip_src=*c* **actions=controller**
ip,ip_src=*d* **actions=controller**

Similarly, these flows send packets with IP destination address *e*, *f*, *g*, or *h* to the OpenFlow controller:

ip,ip_dst=*e* **actions=controller**
ip,ip_dst=*f* **actions=controller**
ip,ip_dst=*g* **actions=controller**
ip,ip_dst=*h* **actions=controller**

Installing all of the above flows in a single flow table yields a disjunctive effect: a packet is sent to the controller if **ip_src** ∈ {*a*,*b*,*c*,*d*} or **ip_dst** ∈ {*e*,*f*,*g*,*h*} (or both). (Pedantically, if both of the above sets of flows are present in the flow table, they should have different priorities, because OpenFlow says that the results are undefined when two flows with same priority can both match a single packet.)

Suppose, on the other hand, one wishes to match conjunctively, that is, to send a packet to the controller only if both **ip_src** ∈ {*a*,*b*,*c*,*d*} and **ip_dst** ∈ {*e*,*f*,*g*,*h*}. This requires $4 \times 4 = 16$ flows, one for each possible pairing of **ip_src** and **ip_dst**. That is acceptable for our small example, but it does not gracefully extend to larger sets or greater numbers of dimensions.

The **conjunction** action is a solution for conjunctive matches that is built into Open vSwitch. A **conjunction** action ties groups of individual OpenFlow flows into higher-level ‘‘conjunctive flows’’. Each group corresponds to one dimension, and each flow within the group matches one possible value for the dimension. A packet that matches one flow from each group matches the conjunctive flow.

To implement a conjunctive flow with **conjunction**, assign the conjunctive flow a 32-bit *id*, which must be unique within an OpenFlow table. Assign each of the $n \geq 2$ dimensions a unique number from 1 to *n*; the ordering is unimportant. Add one flow to the OpenFlow flow table for each possible value of each dimension with **conjunction(***id***,** *k*/*n***)** as the flow’s actions, where *k* is the number assigned to the flow’s dimension. Together, these flows specify the conjunctive flow’s match condition. When the conjunctive match condition is met, Open vSwitch looks up one more flow that specifies the conjunctive flow’s actions and receives its statistics. This flow is found by setting **conj_id** to the specified *id* and then again searching the flow table.

The following flows provide an example. Whenever the IP source is one of the values in the flows that match on the IP source (dimension 1 of 2), **and** the IP destination is one of the values in the flows that match on IP destination (dimension 2 of 2), Open vSwitch searches for a flow that matches **conj_id** against the conjunction ID (1234), finding the first flow listed below.

conj_id=1234 actions=controller
ip,ip_src=10.0.0.1 actions=conjunction(1234, 1/2)
ip,ip_src=10.0.0.4 actions=conjunction(1234, 1/2)
ip,ip_src=10.0.0.6 actions=conjunction(1234, 1/2)
ip,ip_src=10.0.0.7 actions=conjunction(1234, 1/2)
ip,ip_dst=10.0.0.2 actions=conjunction(1234, 2/2)
ip,ip_dst=10.0.0.5 actions=conjunction(1234, 2/2)
ip,ip_dst=10.0.0.7 actions=conjunction(1234, 2/2)

ip,ip_dst=10.0.0.8 actions=conjunction(1234, 2/2)

Many subtleties exist:

- In the example above, every flow in a single dimension has the same form, that is, dimension 1 matches on **ip_src** and dimension 2 on **ip_dst**, but this is not a requirement. Different flows within a dimension may match on different bits within a field (e.g. IP network prefixes of different lengths, or TCP/UDP port ranges as bitwise matches), or even on entirely different fields (e.g. to match packets for TCP source port 80 or TCP destination port 80).
- The flows within a dimension can vary their matches across more than one field, e.g. to match only specific pairs of IP source and destination addresses or L4 port numbers.
- A flow may have multiple **conjunction** actions, with different **id** values. This is useful for multiple conjunctive flows with overlapping sets. If one conjunctive flow matches packets with both **ip_src** ∈ {*a*,*b*} and **ip_dst** ∈ {*d*,*e*} and a second conjunctive flow matches **ip_src** ∈ {*b*,*c*} and **ip_dst** ∈ {*f*,*g*}, for example, then the flow that matches **ip_src**=*b* would have two **conjunction** actions, one for each conjunctive flow. The order of **conjunction** actions within a list of actions is not significant.
- A flow with **conjunction** actions may also include **note** actions for annotations, but not any other kind of actions. (They would not be useful because they would never be executed.)
- All of the flows that constitute a conjunctive flow with a given *id* must have the same priority. (Flows with the same *id* but different priorities are currently treated as different conjunctive flows, that is, currently *id* values need only be unique within an OpenFlow table at a given priority. This behavior isn't guaranteed to stay the same in later releases, so please use *id* values unique within an OpenFlow table.)
- Conjunctive flows must not overlap with each other, at a given priority, that is, any given packet must be able to match at most one conjunctive flow at a given priority. Overlapping conjunctive flows yield unpredictable results.
- Following a conjunctive flow match, the search for the flow with **conj_id**=*id* is done in the same general-purpose way as other flow table searches, so one can use flows with **conj_id**=*id* to act differently depending on circumstances. (One exception is that the search for the **conj_id**=*id* flow itself ignores conjunctive flows, to avoid recursion.) If the search with **conj_id**=*id* fails, Open vSwitch acts as if the conjunctive flow had not matched at all, and continues searching the flow table for other matching flows.
- OpenFlow prerequisite checking occurs for the flow with **conj_id**=*id* in the same way as any other flow, e.g. in an OpenFlow 1.1+ context, putting a **mod_nw_src** action into the example above would require adding an **ip** match, like this:

  conj_id=1234,ip actions=mod_nw_src:1.2.3.4,controller
- OpenFlow prerequisite checking also occurs for the individual flows that comprise a conjunctive match in the same way as any other flow.
- The flows that constitute a conjunctive flow do not have useful statistics. They are never updated with byte or packet counts, and so on. (For such a flow, therefore, the idle and hard timeouts work much the same way.)
- Sometimes there is a choice of which flows include a particular match. For example, suppose that we added an extra constraint to our example, to match on **ip_src** ∈ {*a*,*b*,*c*,*d*} and **ip_dst** ∈ {*e*,*f*,*g*,*h*} and **tcp_dst** = *i*. One way to implement this is to add the new constraint to the **conj_id** flow, like this:

  conj_id=1234,tcp,tcp_dst=*i* **actions=mod_nw_src:1.2.3.4,controller**

but **this is not recommended** because of the cost of the extra flow table lookup. Instead, add the constraint to the individual flows, either in one of the dimensions or (slightly better) all of them.

- A conjunctive match must have $n \geq 2$ dimensions (otherwise a conjunctive match is not necessary). Open vSwitch enforces this.
- Each dimension within a conjunctive match should ordinarily have more than one flow. Open vSwitch does not enforce this.

Conjunction ID Field

Name:	**conj_id**
Width:	32 bits
Format:	decimal
Masking:	not maskable
Prerequisites:	none
Access:	read-only
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_CONJ_ID** (37) since Open vSwitch 2.4

Used for conjunctive matching. See above for more information.

TUNNEL FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
tun_id aka **tunnel_id**	8	yes	yes	none	OF1.3+ and OVS 1.1+
tun_src	4	yes	yes	none	OVS 2.0+
tun_dst	4	yes	yes	none	OVS 2.0+
tun_ipv6_src	16	yes	yes	none	OVS 2.5+
tun_ipv6_dst	16	yes	yes	none	OVS 2.5+
tun_gbp_id	2	yes	yes	none	OVS 2.4+
tun_gbp_flags	1	yes	yes	none	OVS 2.4+
tun_metadata0	124	yes	yes	none	OVS 2.5+
tun_metadata1	124	yes	yes	none	OVS 2.5+
tun_metadata2	124	yes	yes	none	OVS 2.5+
tun_metadata3	124	yes	yes	none	OVS 2.5+
tun_metadata4	124	yes	yes	none	OVS 2.5+
tun_metadata5	124	yes	yes	none	OVS 2.5+
tun_metadata6	124	yes	yes	none	OVS 2.5+
tun_metadata7	124	yes	yes	none	OVS 2.5+
tun_metadata8	124	yes	yes	none	OVS 2.5+
tun_metadata9	124	yes	yes	none	OVS 2.5+
tun_metadata10	124	yes	yes	none	OVS 2.5+
tun_metadata11	124	yes	yes	none	OVS 2.5+
tun_metadata12	124	yes	yes	none	OVS 2.5+
tun_metadata13	124	yes	yes	none	OVS 2.5+
tun_metadata14	124	yes	yes	none	OVS 2.5+
tun_metadata15	124	yes	yes	none	OVS 2.5+
tun_metadata16	124	yes	yes	none	OVS 2.5+
tun_metadata17	124	yes	yes	none	OVS 2.5+
tun_metadata18	124	yes	yes	none	OVS 2.5+
tun_metadata19	124	yes	yes	none	OVS 2.5+
tun_metadata20	124	yes	yes	none	OVS 2.5+
tun_metadata21	124	yes	yes	none	OVS 2.5+
tun_metadata22	124	yes	yes	none	OVS 2.5+
tun_metadata23	124	yes	yes	none	OVS 2.5+
tun_metadata24	124	yes	yes	none	OVS 2.5+
tun_metadata25	124	yes	yes	none	OVS 2.5+
tun_metadata26	124	yes	yes	none	OVS 2.5+
tun_metadata27	124	yes	yes	none	OVS 2.5+
tun_metadata28	124	yes	yes	none	OVS 2.5+
tun_metadata29	124	yes	yes	none	OVS 2.5+
tun_metadata30	124	yes	yes	none	OVS 2.5+
tun_metadata31	124	yes	yes	none	OVS 2.5+
tun_metadata32	124	yes	yes	none	OVS 2.5+
tun_metadata33	124	yes	yes	none	OVS 2.5+
tun_metadata34	124	yes	yes	none	OVS 2.5+
tun_metadata35	124	yes	yes	none	OVS 2.5+
tun_metadata36	124	yes	yes	none	OVS 2.5+
tun_metadata37	124	yes	yes	none	OVS 2.5+
tun_metadata38	124	yes	yes	none	OVS 2.5+
tun_metadata39	124	yes	yes	none	OVS 2.5+
tun_metadata40	124	yes	yes	none	OVS 2.5+
tun_metadata41	124	yes	yes	none	OVS 2.5+
tun_metadata42	124	yes	yes	none	OVS 2.5+

tun_metadata43	124	yes	yes	none	OVS 2.5+
tun_metadata44	124	yes	yes	none	OVS 2.5+
tun_metadata45	124	yes	yes	none	OVS 2.5+
tun_metadata46	124	yes	yes	none	OVS 2.5+
tun_metadata47	124	yes	yes	none	OVS 2.5+
tun_metadata48	124	yes	yes	none	OVS 2.5+
tun_metadata49	124	yes	yes	none	OVS 2.5+
tun_metadata50	124	yes	yes	none	OVS 2.5+
tun_metadata51	124	yes	yes	none	OVS 2.5+
tun_metadata52	124	yes	yes	none	OVS 2.5+
tun_metadata53	124	yes	yes	none	OVS 2.5+
tun_metadata54	124	yes	yes	none	OVS 2.5+
tun_metadata55	124	yes	yes	none	OVS 2.5+
tun_metadata56	124	yes	yes	none	OVS 2.5+
tun_metadata57	124	yes	yes	none	OVS 2.5+
tun_metadata58	124	yes	yes	none	OVS 2.5+
tun_metadata59	124	yes	yes	none	OVS 2.5+
tun_metadata60	124	yes	yes	none	OVS 2.5+
tun_metadata61	124	yes	yes	none	OVS 2.5+
tun_metadata62	124	yes	yes	none	OVS 2.5+
tun_metadata63	124	yes	yes	none	OVS 2.5+
tun_flags	2 (low 1 bits)	yes	yes	none	OVS 2.5+

The fields in this group relate to tunnels, which Open vSwitch supports in several forms (GRE, VXLAN, and so on). Most of these fields do appear in the wire format of a packet, so they are data fields from that point of view, but they are metadata from an OpenFlow flow table point of view because they do not appear in packets that are forwarded to the controller or to ordinary (non-tunnel) output ports.

Open vSwitch supports a spectrum of usage models for mapping tunnels to OpenFlow ports:

‘‘Port-based’’ tunnels

In this model, an OpenFlow port represents one tunnel: it matches a particular type of tunnel traffic between two IP endpoints, with a particular tunnel key (if keys are in use). In this situation, **in_port** suffices to distinguish one tunnel from another, so the tunnel header fields have little importance for OpenFlow processing. (They are still populated and may be used if it is convenient.) The tunnel header fields play no role in sending packets out such an OpenFlow port, either, because the OpenFlow port itself fully specifies the tunnel headers.

The following Open vSwitch commands create a bridge **br−int**, add port **tap0** to the bridge as OpenFlow port 1, establish a port-based GRE tunnel between the local host and remote IP 192.168.1.1 using GRE key 5001 as OpenFlow port 2, and arranges to forward all traffic from **tap0** to the tunnel and vice versa:

```
ovs−vsctl add−br br−int
ovs−vsctl add−port br−int tap0 −− set interface tap0 ofport_request=1
ovs−vsctl add−port br−int gre0 −−
  set interface gre0 ofport_request=2 type=gre \
            options:remote_ip=192.168.1.1 options:key=5001
ovs−ofctl add−flow br−int in_port=1,actions=2
ovs−ofctl add−flow br−int in_port=2,actions=1
```

“Flow-based” tunnels

In this model, one OpenFlow port represents all possible tunnels of a given type with an endpoint on the current host, for example, all GRE tunnels. In this situation, **in_port** only indicates that traffic was received on the particular kind of tunnel. This is where the tunnel header fields are most important: they allow the OpenFlow tables to discriminate among tunnels based on their IP endpoints or keys. Tunnel header fields also determine

the IP endpoints and keys of packets sent out such a tunnel port.

The following Open vSwitch commands create a bridge **br−int**, add port **tap0** to the bridge as OpenFlow port 1, establish a flow-based GRE tunnel port 3, and arranges to forward all traffic from **tap0** to remote IP 192.168.1.1 over a GRE tunnel with key 5001 and vice versa:

```
ovs−vsctl add−br br−int
ovs−vsctl add−port br−int tap0 −− set interface tap0 ofport_request=1
ovs−vsctl add−port br−int allgre −−
  set interface gre0 ofport_request=2 type=gre \
                options:remote_ip=flow options:key=flow
ovs−ofctl add−flow br−int \
  'in_port=1 actions=set_tunnel:5001,set_field:192.168.1.1−>tun_dst,3'
ovs−ofctl add−flow br−int 'in_port=3,tun_src=192.168.1.1,tun_id=5001 actions=1'
```

Mixed models.

One may define both flow-based and port-based tunnels at the same time. For example, it is valid and possibly useful to create and configure both **gre0** and **allgre** tunnel ports described above.

Traffic is attributed on ingress to the most specific matching tunnel. For example, **gre0** is more specific than **allgre**. Therefore, if both exist, then **gre0** will be the ingress port for any GRE traffic received from 192.168.1.1 with key 5001.

On egress, traffic may be directed to any appropriate tunnel port. If both **gre0** and **allgre** are configured as already described, then the actions **2** and **set_tunnel:5001,set_field:192.168.1.1−>tun_dst,3** send the same tunnel traffic.

Intermediate models.

Ports may be configured as partially flow-based. For example, one may define an OpenFlow port that represents tunnels between a pair of endpoints but leaves the flow table to discriminate on the flow key.

ovs−vswitchd.conf.db(5) describes all the details of tunnel configuration.

These fields do not have any prerequisites, which means that a flow may match on any or all of them, in any combination.

These fields are zeros for packets that did not arrive on a tunnel.

Tunnel ID Field

Name:	**tun_id** (aka **tunnel_id**)
Width:	64 bits
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_TUNNEL_ID** (38) since OpenFlow 1.3 and Open vSwitch 1.10
NXM:	**NXM_NX_TUN_ID** (16) since Open vSwitch 1.1

Many kinds of tunnels support a tunnel ID:

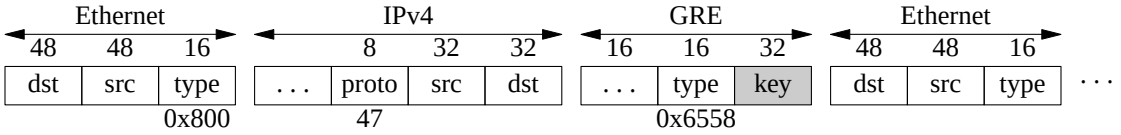
- VXLAN and Geneve have a 24-bit virtual network identifier (VNI).
- LISP has a 24-bit instance ID.
- GRE has an optional 32-bit key.
- STT has a 64-bit key.

When a packet is received from a tunnel, this field holds the tunnel ID in its least significant bits, zero-

extended to fit. This field is zero if the tunnel does not support an ID, or if no ID is in use for a tunnel type that has an optional ID, or if an ID of zero received, or if the packet was not received over a tunnel.

When a packet is output to a tunnel port, the tunnel configuration determines whether the tunnel ID is taken from this field or bound to a fixed value. See the earlier description of ‘‘port-based’’ and ‘‘flow-based’’ tunnels for more information.

The following diagram shows the origin of this field in a typical keyed GRE tunnel:



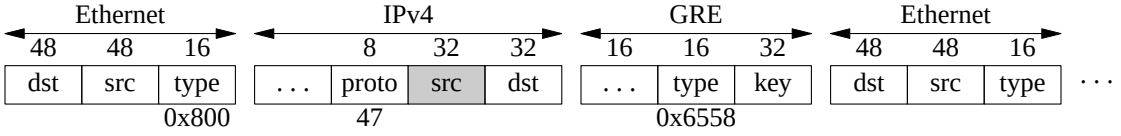
Tunnel IPv4 Source Field

Name:	**tun_src**
Width:	32 bits
Format:	IPv4
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_TUN_IPV4_SRC** (31) since Open vSwitch 2.0

When a packet is received from a tunnel, this field is the source address in the outer IP header of the tunneled packet. This field is zero if the packet was not received over a tunnel.

When a packet is output to a flow-based tunnel port, this field influences the IPv4 source address used to send the packet. If it is zero, then the kernel chooses an appropriate IP address based using the routing table.

The following diagram shows the origin of this field in a typical keyed GRE tunnel:



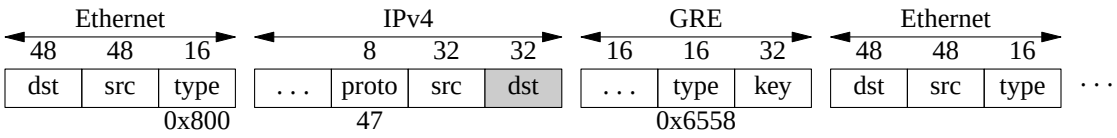
Tunnel IPv4 Destination Field

Name:	**tun_dst**
Width:	32 bits
Format:	IPv4
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_TUN_IPV4_DST** (32) since Open vSwitch 2.0

When a packet is received from a tunnel, this field is the destination address in the outer IP header of the tunneled packet. This field is zero if the packet was not received over a tunnel.

When a packet is output to a flow-based tunnel port, this field specifies the destination to which the tunnel packet is sent.

The following diagram shows the origin of this field in a typical keyed GRE tunnel:



Tunnel IPv6 Source Field

Name:	**tun_ipv6_src**
Width:	128 bits
Format:	IPv6
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_TUN_IPV6_SRC** (109) since Open vSwitch 2.5

Similar to **tun_src**, but for tunnels over IPv6.

Tunnel IPv6 Destination Field

Name:	**tun_ipv6_dst**
Width:	128 bits
Format:	IPv6
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_TUN_IPV6_DST** (110) since Open vSwitch 2.5

Similar to **tun_dst**, but for tunnels over IPv6.

VXLAN Group-Based Policy ID Field

Name:	**tun_gbp_id**
Width:	16 bits
Format:	decimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_TUN_GBP_ID** (38) since Open vSwitch 2.4

For a packet tunneled over VXLAN with the Group-Based Policy (GBP) extension, this field represents the GBP policy ID. Only packets that arrive over a VXLAN tunnel with the GBP extension enabled have this field set.

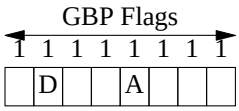
VXLAN Group-Based Policy Flags Field

Name:	**tun_gbp_flags**
Width:	8 bits
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none

Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_TUN_GBP_FLAGS** (39) since Open vSwitch 2.4

For a packet tunneled over VXLAN with the Group-Based Policy (GBP) extension, this field represents the GBP policy flags. Only packets that arrive over a VXLAN tunnel with the GBP extension enabled have this field set.

The field has the format shown below:



Unlabeled bits are reserved and must be transmitted as 0. The following bits have defined meanings:

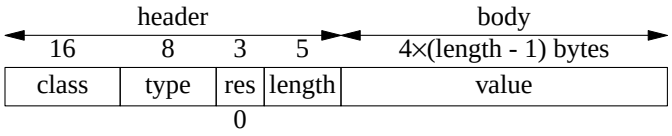
D (Don’t Learn)
: When set, this bit indicates that the egress tunnel endpoint MUST NOT learn the source address of the encapsulated frame.

A (Applied
: When set, indicates that the group policy has already been applied to this packet. Policies MUST NOT be applied by devices when the A bit is set.

Generic Tunnel Option 0 Field

Name:	**tun_metadata0**
Width:	992 bits (124 bytes)
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_TUN_METADATA0** (40) since Open vSwitch 2.5

These fields provide OpenFlow access to the generic type-length-value options defined by the Geneve tunneling protocol or other protocols with options in the same TLV format as Geneve options. Each of these options has the following wire format:



Taken together, the **class** and **type** in the option format mean that there are about 16 million distinct kinds of TLV options, too many to give individual OXM code points. Thus, Open vSwitch requires the user to define the TLV options of interest. Up to 64 TLV options can be bound to generic tunnel option NXM code points. Each option may have up to 124 bytes in its value (the maximum this format allows), and the total set of bound options must total at most 252 bytes.

Open vSwitch extensions to the OpenFlow protocol bind TLV options to NXM code points. The **ovs−ofctl**(8) program offers one way to use these extensions, e.g. to configure a mapping from a TLV option with **class 0xffff**, **type** 0, and a body length of 4 bytes:

ovs−ofctl add−tlv−map br0 "{class=0xffff,type=0,len=4}−>tun_metadata0"

Once a TLV option is properly bound, it can be accessed and modified like any other field, e.g. to send packets that have value 1234 for the option described above to the controller:

ovs−ofctl add−flow br0 tun_metadata0=1234,actions=controller

Tunnel Flags Field

Name:	**tun_flags**
Width:	16 bits (only the least-significant 1 bits may be nonzero)
Format:	tunnel flags
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_TUN_FLAGS** (104) since Open vSwitch 2.5

Flags indicating various aspects of the tunnel encapsulation.

Matches on this field are most conveniently written in terms of symbolic names (given in the diagram below), each preceded by either **+** for a flag that must be set, or **−** for a flag that must be unset, without any other delimiters between the flags. Flags not mentioned are wildcarded. For example, **tun_flags=+oam** matches only OAM packets. Matches can also be written as *flags*/*mask*, where *flags* and *mask* are 16-bit numbers in decimal or in hexadecimal prefixed by **0x**.

Currently, there is only one flag defined:

oam The tunnel protocol indicated that this is an OAM (Operations and Management) control packet.

The switch may reject matches against unknown flags.

Newer versions of Open vSwitch may introduce additional flags with new meanings. It is therefore not recommended to use an exact match on this field since the behavior of these new flags is unknown and should be ignored.

For non-tunneled packets, the value is 0.

METADATA FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
in_port	2	no	yes	none	OVS 1.1+
in_port_oxm	4	no	yes	none	OF1.2+ and OVS 1.7+
skb_priority	4	no	no	none	
pkt_mark	4	yes	yes	none	OVS 2.0+
actset_output	4	no	no	none	OF1.3+ and OVS 2.4+

These fields relate to the origin or treatment of a packet, but they are not extracted from the packet data itself.

Ingress Port Field

Name:	**in_port**
Width:	16 bits
Format:	OpenFlow 1.0 port
Masking:	not maskable
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	none
NXM:	**NXM_OF_IN_PORT** (0) since Open vSwitch 1.1

The OpenFlow port on which the packet being processed arrived. This is a 16-bit field that holds an OpenFlow 1.0 port number. For receiving a packet, the only values that appear in this field are:

1 through **0xfeff** (65,279), inclusive.
: Conventional OpenFlow port numbers.

OFPP_LOCAL (**0xfffe** or 65,534).
: The ‘‘local’’ port, which in Open vSwitch is always named the same as the bridge itself. This represents a connection between the switch and the local TCP/IP stack. This port is where an IP address is most commonly configured on an Open vSwitch switch.

  OpenFlow does not require a switch to have a local port, but all existing versions of Open vSwitch have always included a local port. **Future Directions:** Future versions of Open vSwitch might be able to optionally omit the local port, if someone submits code to implement such a feature.

OFPP_NONE (**0xffff** or 65,535).
OFPP_CONTROLLER (**0xfffd** or 65,533).
: When a controller injects a packet into an OpenFlow switch with a ‘‘packet-out’’ request, it can specify one of these ingress ports to indicate that the packet was generated internally rather than having been received on some port.

  OpenFlow 1.0 specified **OFPP_NONE** for this purpose. Despite that, some controllers used **OFPP_CONTROLLER**, and some switches only accepted **OFPP_CONTROLLER**, so OpenFlow 1.0.2 required support for both ports. OpenFlow 1.1 and later were more clearly drafted to allow only **OFPP_CONTROLLER**. For maximum compatibility, Open vSwitch allows both ports with all OpenFlow versions.

Values not mentioned above will never appear when receiving a packet, including the following notable values:

0
: Zero is not a valid OpenFlow port number.

OFPP_MAX (**0xff00** or 65,280).
: This value has only been clearly specified as a valid port number as of OpenFlow 1.3.3. Before that, its status was unclear, and so Open vSwitch has never allowed **OFPP_MAX** to be used as a port number, so packets will never be received on this port. (Other

OpenFlow switches, of course, might use it.)

OFPP_UNSET (**0xfff7** or 65,527)
OFPP_IN_PORT (**0xfff8** or 65,528)
OFPP_TABLE (**0xfff9** or 65,529)
OFPP_NORMAL (**0xfffa** or 65,530)
OFPP_FLOOD (**0xfffb** or 65,531)
OFPP_ALL (**0xfffc** or 65,532)

These port numbers are used only in output actions and never appear as ingress ports.

Most of these port numbers were defined in OpenFlow 1.0, but **OFPP_UNSET** was only introduced in OpenFlow 1.5.

Values that will never appear when receiving a packet may still be matched against in the flow table. There are still circumstances in which those flows can be matched:

- The **resubmit** Open vSwitch extension action allows a flow table lookup with an arbitrary ingress port.
- An action that modifies the ingress port field (see below), such as e.g. **load** or **set_field**, followed by an action or instruction that performs another flow table lookup, such as **resubmit** or **goto_table**.

This field is heavily used for matching in OpenFlow tables, but for packet egress, it has only very limited roles:

- OpenFlow requires suppressing output actions to **in_port**. That is, the following two flows both drop all packets that arrive on port 1:

  in_port=1,actions=1
  in_port=1,actions=drop

  (This behavior is occasionally useful for flooding to a subset of ports. Specifying **actions=1,2,3,4**, for example, outputs to ports 1, 2, 3, and 4, omitting the ingress port.)
- OpenFlow has a special port **OFPP_IN_PORT** (with value 0xfff8) that outputs to the ingress port. For example, in a switch that has four ports numbered 1 through 4, **actions=1,2,3,4,in_port** outputs to ports 1, 2, 3, and 4, including the ingress port.

Because the ingress port field has so little influence on packet processing, it does not ordinarily make sense to modify the ingress port field. The field is writable only to support the occasional use case where the ingress port's roles in packet egress, described above, become troublesome. For example, **actions=load:0−>NXM_OF_IN_PORT[],output:123** will output to port 123 regardless of whether it is in the ingress port. If the ingress port is important, then one may save and restore it on the stack:

actions=push:NXM_OF_IN_PORT[],load:0−>NXM_OF_IN_PORT[],output:123,pop:NXM_OF_IN_PORT[]

The ability to modify the ingress port is an Open vSwitch extension to OpenFlow.

Modifying the ingress port does not prevent or frustrate specifying an ingress port in the **resubmit** action, because **resubmit** only (optionally) changes the **in_port** used for **resubmit**'s flow table lookup. It does not otherwise affect the ingress port.

OXM Ingress Port Field

Name:	**in_port_oxm**
Width:	32 bits
Format:	OpenFlow 1.1+ port
Masking:	not maskable
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	yes (exact match only)

OXM: **OXM_OF_IN_PORT** (0) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: none

OpenFlow 1.1 and later use a 32-bit port number, so this field supplies a 32-bit view of the ingress port. Current versions of Open vSwitch support only a 16-bit range of ports:

- OpenFlow 1.0 ports **0x0000** to **0xfeff**, inclusive, map to OpenFlow 1.1 port numbers with the same values.
- OpenFlow 1.0 ports **0xff00** to **0xffff**, inclusive, map to OpenFlow 1.1 port numbers **0xffffff00** to **0xffffffff**.
- OpenFlow 1.1 ports **0x0000ff00** to **0xffffeff** are not mapped and not supported.

in_port and **in_port_oxm** are two views of the same information, so all of the comments on **in_port** apply to **in_port_oxm** too. Modifying **in_port** changes **in_port_oxm**, and vice versa.

Setting **in_port_oxm** to an unsupported value yields unspecified behavior.

Output Queue Field

Name: **skb_priority**
Width: 32 bits
Format: hexadecimal
Masking: not maskable
Prerequisites: none
Access: read-only
OpenFlow 1.0: not supported
OpenFlow 1.1: not supported
OXM: none
NXM: none

This field influences how packets in the flow will be queued, for quality of service (QoS) purposes, when they egress the switch. Its range of meaningful values, and their meanings, varies greatly from one OpenFlow implementation to another. Even within a single implementation, there is no guarantee that all OpenFlow ports have the same queues configured or that all OpenFlow ports in an implementation can be configured the same way queue-wise.

Configuring queues on OpenFlow is not well standardized. On Linux, Open vSwitch supports queue configuration via OVSDB, specifically the **QoS** and **Queue** tables (see **ovs−vswitchd.conf.db(5)** for details). Ports of Open vSwitch to other platforms might require queue configuration through some separate protocol (such as a CLI). Even on Linux, Open vSwitch exposes only a fraction of the kernel's queuing features through OVSDB, so advanced or unusual uses might require use of separate utilities (e.g. **tc**). OpenFlow switches other than Open vSwitch might use OF-CONFIG or any of the configuration methods mentioned above. Finally, some OpenFlow switches have a fixed number of fixed-function queues (e.g. eight queues with strictly defined priorities) and others do not support any control over queuing.

The only output queue that all OpenFlow implementations must support is zero, to identify a default queue, whose properties are implementation-defined. Outputting a packet to a queue that does not exist on the output port yields unpredictable behavior: among the possibilities are that the packet might be dropped or transmitted with a very high or very low priority.

OpenFlow 1.0 only allowed output queues to be specified as part of an ''enqueue'' action that specified both a queue and an output port. That is, OpenFlow 1.0 treats the queue as an argument to an action, not as a field.

To increase flexibility, OpenFlow 1.1 added an action to set the output queue. This model was carried forward, without change, through OpenFlow 1.5.

Open vSwitch implements the native queuing model of each OpenFlow version it supports. Open vSwitch also includes an extension for setting the output queue as an action in OpenFlow 1.0.

When a packet ingresses into an OpenFlow switch, the output queue is ordinarily set to 0, indicating the

default queue. However, Open vSwitch supports various ways to forward a packet from one OpenFlow switch to another within a single host. In these cases, Open vSwitch maintains the output queue across the forwarding step. For example:

- A hop across a Open vSwitch ‘‘patch port’’ (which does not actually involve queuing) preserves the output queue.
- When a flow sets the output queue then outputs to an OpenFlow tunnel port, the encapsulation preserves the output queue. If the kernel TCP/IP stack routes the encapsulated packet directly to a physical interface, then that output honors the output queue. Alternatively, if the kernel routes the encapsulated packet to another Open vSwitch bridge, then the output queue set previously becomes the initial output queue on ingress to the second bridge and will thus be used for further output actions (unless overridden by a new ‘‘set queue’’ action).

  (This description reflects the current behavior of Open vSwitch on Linux. This behavior relies on details of the Linux TCP/IP stack. It could be difficult to make ports to other operating systems behave the same way.)

Future Directions: Open vSwitch implements the output queue as a field, but does not currently expose it through OXM or NXM for matching purposes. If this turns out to be a useful feature, it could be implemented in future versions.

Packet Mark Field

Name:	**pkt_mark**
Width:	32 bits
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_PKT_MARK** (33) since Open vSwitch 2.0

Packet mark comes to Open vSwitch from the Linux kernel, in which the **sk_buff** data structure that represents a packet contains a 32-bit member named **skb_mark**. The value of **skb_mark** propagates along with the packet it accompanies wherever the packet goes in the kernel. It has no predefined semantics but various kernel-user interfaces can set and match on it, which makes it suitable for ‘‘marking’’ packets at one point in their handling and then acting on the mark later. With **iptables**, for example, one can mark some traffic specially at ingress and then handle that traffic differently at egress based on the marked value.

Packet mark is an attempt at a generalization of the **skb_mark** concept beyond Linux, at least through more generic naming. Like **skb_priority**, packet mark is preserved across forwarding steps within a machine. Unlike **skb_priority**, packet mark has no direct effect on packet forwarding: the value set in packet mark does not matter unless some later OpenFlow table or switch matches on packet mark, or unless the packet passes through some other kernel subsystem that has been configured to interpret packet mark in specific ways, e.g. through **iptables** configuration mentioned above.

Preserving packet mark across kernel forwarding steps relies heavily on kernel support, which ports to non-Linux operating systems may not have. Regardless of operating system support, Open vSwitch supports packet mark within a single bridge and across patch ports.

The value of packet mark when a packet ingresses into the first Open vSwich bridge is typically zero, but it could be nonzero if its value was previously set by some kernel subsystem.

Action Set Output Port Field

Name:	**actset_output**
Width:	32 bits

Format:	OpenFlow 1.1+ port
Masking:	not maskable
Prerequisites:	none
Access:	read-only
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**ONFOXM_ET_ACTSET_OUTPUT** (43) since OpenFlow 1.3 and Open vSwitch 2.4; **OXM_OF_ACTSET_OUTPUT** (43) since OpenFlow 1.5 and Open vSwitch 2.4
NXM:	none

Holds the output port currently in the OpenFlow action set (i.e. from an **output** action within a **write_actions** instruction). Its value is an OpenFlow port number. If there is no output port in the OpenFlow action set, or if the output port will be ignored (e.g. because there is an output group in the OpenFlow action set), then the value will be **OFPP_UNSET**.

Open vSwitch allows any table to match this field. OpenFlow, however, only requires this field to be matchable from within an OpenFlow egress table (a feature that Open vSwitch does not yet implement).

CONNECTION TRACKING FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
ct_state	4	yes	no	none	OVS 2.5+
ct_zone	2	no	no	none	OVS 2.5+
ct_mark	4	yes	yes	none	OVS 2.5+
ct_label	16	yes	yes	none	OVS 2.5+

Open vSwitch 2.5 and later support ‘‘connection tracking,’’ which allows bidirectional streams of packets to be statefully grouped into connections. Open vSwitch connection tracking, for example, identifies the patterns of TCP packets that indicates a successfully initiated connection, as well as those that indicate that a connection has been torn down. Open vSwitch connection tracking can also identify related connections, such as FTP data connections spawned from FTP control connections.

An individual packet passing through the pipeline may be in one of two states: ‘‘untracked’’ or ‘‘tracked.’’ A packet is *untracked* at the beginning of the Open vSwitch pipeline and continues to be untracked until the pipeline invokes the **ct** action. The connection tracking fields are all zeroes in an untracked packet. When a flow in the Open vSwitch pipeline invokes the **ct** action, the action initializes the connection tracking fields and the packet becomes *tracked* for the remainder of its processing.

The connection tracker stores connection state in an internal table, but it only adds a new entry to this table when a **ct** action for a new connection invokes **ct** with the **commit** parameter. For a given connection, when a pipeline has executed **ct**, but not yet with **commit**, the connection is said to be *uncommitted*. State for an uncommitted connection is ephemeral and does not persist past the end of the pipeline, so some features are only available to committed connections. A connection would typically be left uncommitted as a way to drop its packets.

Connection tracking is an Open vSwitch extension to OpenFlow.

Connection Tracking State Field

Name:	**ct_state**
Width:	32 bits
Format:	ct state
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read-only
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_CT_STATE** (105) since Open vSwitch 2.5

This field holds several flags that can be used to determine the state of the connection to which the packet belongs.

Matches on this field are most conveniently written in terms of symbolic names (listed below), each preceded by either **+** for a flag that must be set, or **−** for a flag that must be unset, without any other delimiters between the flags. Flags not mentioned are wildcarded. For example, **tcp,ct_state=+trk−new** matches TCP packets that have been run through the connection tracker and do not establish a new connection. Matches can also be written as *flags*/*mask*, where *flags* and *mask* are 32-bit numbers in decimal or in hexadecimal prefixed by **0x**.

The following flags are defined:

new (0x01)
A new connection. Set to 1 if this is an uncommitted connection.

est (0x02)
Part of an existing connection. Set to 1 if this is a committed connection.

rel (0x04)
Related to an existing connection, e.g. an ICMP ‘‘destination unreachable’’ message or an FTP data connections. This flag will only be 1 if the connection to which this one is related is committed.

Connections identified as **rel** are separate from the originating connection and must be committed separately. All packets for a related connection will have the **rel** flag set, not just the initial packet.

rpl (0x08)
This packet is in the reply direction, meaning that it is in the opposite direction from the packet that initiated the connection. This flag will only be 1 if the connection is committed.

inv (0x10)
The state is invalid, meaning that the connection tracker couldn’t identify the connection. This flag is a catch-all for problems in the connection or the connection tracker, such as:

- L3/L4 protocol handler is not loaded/unavailable. With the Linux kernel datapath, this may mean that the **nf_conntrack_ipv4** or **nf_conntrack_ipv6** modules are not loaded.
- L3/L4 protocol handler determines that the packet is malformed.
- Packets are unexpected length for protocol.

trk (0x20)
This packet is tracked, meaning that it has previously traversed the connection tracker. If this flag is not set, then no other flags will be set. If this flag is set, then the packet is tracked and other flags may also be set.

snat (0x40)
This packet was transformed by source address/port translation by a preceding **ct** action. Open vSwitch 2.6 added this flag.

dnat (0x80)
This packet was transformed by destination address/port translation by a preceding **ct** action. Open vSwitch 2.6 added this flag.

There are additional constraints on these flags, listed in decreasing order of precedence below:

1. If **trk** is unset, no other flags are set.
2. If **trk** is set, one or more other flags may be set.
3. If **inv** is set, only the **trk** flag is also set.
4. **new** and **est** are mutually exclusive.
5. **new** and **rpy** are mutually exclusive.
6. **rel** may be set in conjunction with any other flags.

Future versions of Open vSwitch may define new flags.

Connection Tracking Zone Field

Name:	**ct_zone**
Width:	16 bits
Format:	hexadecimal
Masking:	not maskable
Prerequisites:	none
Access:	read-only
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported

OXM: none
NXM: **NXM_NX_CT_ZONE** (106) since Open vSwitch 2.5

A connection tracking zone, the zone value passed to the most recent **ct** action. Each zone is an independent connection tracking context, so tracking the same packet in multiple contexts requires using the **ct** action multiple times.

Connection Tracking Mark Field

Name: **ct_mark**
Width: 32 bits
Format: hexadecimal
Masking: arbitrary bitwise masks
Prerequisites: none
Access: read/write
OpenFlow 1.0: not supported
OpenFlow 1.1: not supported
OXM: none
NXM: **NXM_NX_CT_MARK** (107) since Open vSwitch 2.5

The metadata committed, by an action within the **exec** parameter to the **ct** action, to the connection to which the current packet belongs.

Connection Tracking Label Field

Name: **ct_label**
Width: 128 bits
Format: hexadecimal
Masking: arbitrary bitwise masks
Prerequisites: none
Access: read/write
OpenFlow 1.0: not supported
OpenFlow 1.1: not supported
OXM: none
NXM: **NXM_NX_CT_LABEL** (108) since Open vSwitch 2.5

The label committed, by an action within the **exec** parameter to the **ct** action, to the connection to which the current packet belongs.

REGISTER FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
metadata	8	yes	yes	none	OF1.2+ and OVS 1.8+
reg0	4	yes	yes	none	OVS 1.1+
reg1	4	yes	yes	none	OVS 1.1+
reg2	4	yes	yes	none	OVS 1.1+
reg3	4	yes	yes	none	OVS 1.1+
reg4	4	yes	yes	none	OVS 1.3+
reg5	4	yes	yes	none	OVS 1.7+
reg6	4	yes	yes	none	OVS 1.7+
reg7	4	yes	yes	none	OVS 1.7+
reg8	4	yes	yes	none	OVS 2.6+
reg9	4	yes	yes	none	OVS 2.6+
reg10	4	yes	yes	none	OVS 2.6+
reg11	4	yes	yes	none	OVS 2.6+
reg12	4	yes	yes	none	OVS 2.6+
reg13	4	yes	yes	none	OVS 2.6+
reg14	4	yes	yes	none	OVS 2.6+
reg15	4	yes	yes	none	OVS 2.6+
xreg0	8	yes	yes	none	OF1.3+ and OVS 2.4+
xreg1	8	yes	yes	none	OF1.3+ and OVS 2.4+
xreg2	8	yes	yes	none	OF1.3+ and OVS 2.4+
xreg3	8	yes	yes	none	OF1.3+ and OVS 2.4+
xreg4	8	yes	yes	none	OF1.3+ and OVS 2.4+
xreg5	8	yes	yes	none	OF1.3+ and OVS 2.4+
xreg6	8	yes	yes	none	OF1.3+ and OVS 2.4+
xreg7	8	yes	yes	none	OF1.3+ and OVS 2.4+
xxreg0	16	yes	yes	none	OVS 2.6+
xxreg1	16	yes	yes	none	OVS 2.6+
xxreg2	16	yes	yes	none	OVS 2.6+
xxreg3	16	yes	yes	none	OVS 2.6+

These fields give an OpenFlow switch space for temporary storage while the pipeline is running. Whereas metadata fields can have a meaningful initial value and can persist across some hops across OpenFlow switches, registers are always initially 0 and their values never persist across inter-switch hops (not even across patch ports).

OpenFlow Metadata Field

Name:	**metadata**
Width:	64 bits
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	yes
OXM:	**OXM_OF_METADATA** (2) since OpenFlow 1.2 and Open vSwitch 1.8
NXM:	none

This field is the oldest standardized OpenFlow register field, introduced in OpenFlow 1.1. It was introduced to model the limited number of user-defined bits that some ASIC-based switches can carry through their pipelines. Because of hardware limitations, OpenFlow allows switches to support writing and masking only an implementation-defined subset of bits, even no bits at all. The Open vSwitch software switch always supports all 64 bits, but of course an Open vSwitch port to an ASIC would have the same restriction as the ASIC itself.

This field has an OXM code point, but OpenFlow 1.4 and earlier allow it to be modified only with a specialized instruction, not with a ‘‘set-field’’ action. OpenFlow 1.5 removes this restriction. Open vSwitch does not enforce this restriction, regardless of OpenFlow version.

Register 0 Field

Name:	**reg0**
Width:	32 bits
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_REG0** (0) since Open vSwitch 1.1

This is the first of several Open vSwitch registers, all of which have the same properties. Open vSwitch 1.1 introduced registers 0, 1, 2, and 3, version 1.3 added register 4, version 1.7 added registers 5, 6, and 7, and version 2.6 added registers 8 through 15.

Extended Register 0 Field

Name:	**xreg0**
Width:	64 bits
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_PKT_REG0** (0) since OpenFlow 1.3 and Open vSwitch 2.4
NXM:	none

This is the first of the registers introduced in OpenFlow 1.5. OpenFlow 1.5 calls these fields just the ‘‘packet registers,’’ but Open vSwitch already had 32-bit registers by that name, so Open vSwitch uses the name ‘‘extended registers’’ in an attempt to reduce confusion. The standard allows for up to 128 registers, each 64 bits wide, but Open vSwitch only implements 4 (in versions 2.4 and 2.5) or 8 (in version 2.6 and later).

Each of the 64-bit extended registers overlays two of the 32-bit registers: **xreg0** overlays **reg0** and **reg1**, with **reg0** supplying the most-significant bits of **xreg0** and **reg1** the least-significant. Similarly, **xreg1** overlays **reg2** and **reg3**, and so on.

The OpenFlow specification says that, ‘‘In most cases, the packet registers can not be matched in tables, i.e. they usually can not be used in the flow entry match structure’’ [OpenFlow 1.5, section 7.2.3.10], but there is no reason for a software switch to impose such a restriction, and Open vSwitch does not.

Double-Extended Register 0 Field

Name:	**xxreg0**
Width:	128 bits
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none

NXM: **NXM_NX_XXREG0** (111) since Open vSwitch 2.6

This is the first of the double-extended registers introduce in Open vSwitch 2.6. Each of the 128-bit extended registers overlays four of the 32-bit registers: **xxreg0** overlays **reg0** through **reg3**, with **reg0** supplying the most-significant bits of **xxreg0** and **reg3** the least-significant. **xxreg1** similarly overlays **reg4** through **reg7**, and so on.

LAYER 2 (ETHERNET) FIELDS

Summary:

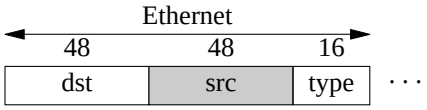
Name	Bytes	Mask	RW?	Prereqs	Support
eth_src aka **dl_src**	6	yes	yes	none	OF1.2+ and OVS 1.1+
eth_dst aka **dl_dst**	6	yes	yes	none	OF1.2+ and OVS 1.1+
eth_type aka **dl_type**	2	no	no	none	OF1.2+ and OVS 1.1+

Ethernet is the only layer−2 protocol that Open vSwitch supports. As with most software, Open vSwitch and OpenFlow regard an Ethernet frame to begin with the 14-byte header and end with the final byte of the payload; that is, the frame check sequence is not considered part of the frame.

Ethernet Source Field

Name:	**eth_src** (aka **dl_src**)
Width:	48 bits
Format:	Ethernet
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes
OXM:	**OXM_OF_ETH_SRC** (4) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_ETH_SRC** (2) since Open vSwitch 1.1

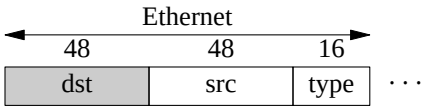
The Ethernet source address:



Ethernet Destination Field

Name:	**eth_dst** (aka **dl_dst**)
Width:	48 bits
Format:	Ethernet
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes
OXM:	**OXM_OF_ETH_DST** (3) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_ETH_DST** (1) since Open vSwitch 1.1

The Ethernet destination address:



Open vSwitch 1.8 and later support arbitrary masks for source and/or destination. Earlier versions only support masking the destination with the following masks:

01:00:00:00:00:00

Match only the multicast bit. Thus, **dl_dst=01:00:00:00:00:00/01:00:00:00:00:00** matches all multicast (including broadcast) Ethernet packets, and **dl_dst=00:00:00:00:00:00/01:00:00:00:00:00** matches all unicast Ethernet packets.

fe:ff:ff:ff:ff:ff
Match all bits except the multicast bit. This is probably not useful.

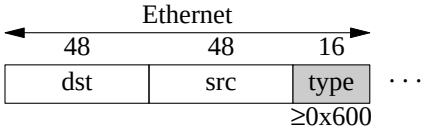
ff:ff:ff:ff:ff:ff
Exact match (equivalent to omitting the mask).

00:00:00:00:00:00
Wildcard all bits (equivalent to **dl_dst=***.)

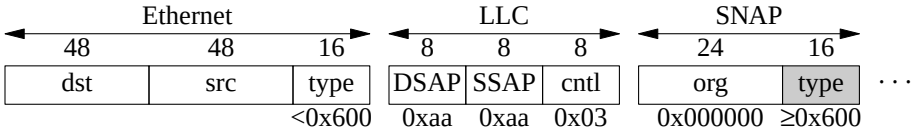
Ethernet Type Field

Name:	**eth_type** (aka **dl_type**)
Width:	16 bits
Format:	hexadecimal
Masking:	not maskable
Prerequisites:	none
Access:	read-only
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_ETH_TYPE** (5) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_ETH_TYPE** (3) since Open vSwitch 1.1

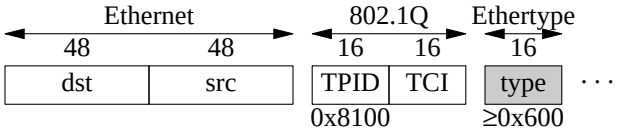
The most commonly seen Ethernet frames today use a format called ‘‘Ethernet II,’’ in which the last two bytes of the Ethernet header specify the Ethertype. For such a frame, this field is copied from those bytes of the header, like so:



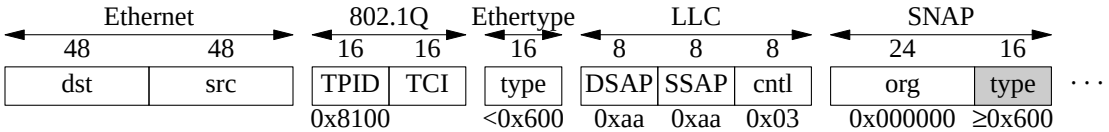
Every Ethernet type has a value 0x600 (1,536) or greater. When the last two bytes of the Ethernet header have a value too small to be an Ethernet type, then the value found there is the total length of the frame in bytes, excluding the Ethernet header. An 802.2 LLC header typically follows the Ethernet header. OpenFlow and Open vSwitch only support LLC headers with DSAP and SSAP **0xaa** and control byte **0x03**, which indicate that a SNAP header follows the LLC header. In turn, OpenFlow and Open vSwitch only support a SNAP header with organization **0x000000**. In such a case, this field is copied from the type field in the SNAP header, like this:



When an 802.1Q header is inserted after the Ethernet source and destination, this field is populated with the encapsulated Ethertype, not the 802.1Q Ethertype. With an Ethernet II inner frame, the result looks like this:



LLC and SNAP encapsulation look like this with an 802.1Q header:



When a packet doesn’t match any of the header formats described above, Open vSwitch and OpenFlow set this field to **0x5ff** (**OFP_DL_TYPE_NOT_ETH_TYPE**).

VLAN FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
dl_vlan	2 (low 12 bits)	no	yes	none	
dl_vlan_pcp	1 (low 3 bits)	no	yes	none	
vlan_vid	2 (low 12 bits)	yes	yes	none	OF1.2+ and OVS 1.7+
vlan_pcp	1 (low 3 bits)	no	yes	VLAN VID	OF1.2+ and OVS 1.7+
vlan_tci	2	yes	yes	none	OVS 1.1+

The 802.1Q VLAN header causes more trouble than any other 4 bytes in networking. OpenFlow 1.0, 1.1, and 1.2+ all treat VLANs differently. Open vSwitch extensions add another variant to the mix. Open vSwitch reconciles all four treatments as best it can.

VLAN Header Format

An 802.1Q VLAN header consists of two 16-bit fields:

The first 16 bits of the VLAN header, the *TPID* (Tag Protocol IDentifier), is an Ethertype. When the VLAN header is inserted just after the source and destination MAC addresses in a Ethertype frame, the TPID serves to identify the presence of the VLAN. The standard TPID, the only one that Open vSwitch supports, is **0x8100**. OpenFlow 1.0 explicitly supports only TPID **0x8100**. OpenFlow 1.1, but not earlier or later versions, also requires support for TPID **0x88a8** (Open vSwitch does not support this). OpenFlow 1.2 through 1.5 do not require support for specific TPIDs (the ‘‘push vlan header’’ action does say that only **0x8100** and **0x88a8** should be pushed). No version of OpenFlow provides a way to distinguish or match on the TPID.

The remaining 16 bits of the VLAN header, the *TCI* (Tag Control Information), is subdivided into three subfields:

- *PCP* (Priority Control Point), is a 3-bit 802.1p *priority*. The lowest priority is value 1, the second-lowest is value 0, and priority increases from 2 up to highest priority 7.
- *CFI* (Canonical Format Indicator), is a 1-bit field. On an Ethernet network, its value is always 0. This led to it later being repurposed under the name *DEI* (Drop Eligibility Indicator). By either name, OpenFlow and Open vSwitch don’t provide any way to match or set this bit.
- *VID* (VLAN IDentifier), is a 12-bit VLAN. If the VID is 0, then the frame is not part of a VLAN. In that case, the VLAN header is called a *priority tag* because it is only meaningful for assigning the frame a priority. VID **0xfff** (4,095) is reserved.

See **eth_type** for illustrations of a complete Ethernet frame with 802.1Q tag included.

Multiple VLANs

Open vSwitch can match only a single VLAN header. If more than one VLAN header is present, then **eth_type** holds the TPID of the inner VLAN header. Open vSwitch stops parsing the packet after the inner TPID, so matching further into the packet (e.g. on the inner TCI or L3 fields) is not possible.

OpenFlow only directly supports matching a single VLAN header. In OpenFlow 1.1 or later, one OpenFlow table can match on the outermost VLAN header and pop it off, and a later OpenFlow table can match on the next outermost header. Open vSwitch does not support this.

VLAN Field Details

The four variants have three different levels of expressiveness: OpenFlow 1.0 and 1.1 VLAN matching are less powerful than OpenFlow 1.2+ VLAN matching, which is less powerful than Open vSwitch extension VLAN matching.

OpenFlow 1.0 VLAN Fields

OpenFlow 1.0 uses two fields, called **dl_vlan** and **dl_vlan_pcp**, each of which can be either exact-matched or wildcarded, to specify VLAN matches:

- When both **dl_vlan** and **dl_vlan_pcp** are wildcarded, the flow matches packets without an 802.1Q header or with any 802.1Q header.
- The match **dl_vlan=0xffff** causes a flow to match only packets without an 802.1Q header. Such a flow should also wildcard **dl_vlan_pcp**, since a packet without an 802.1Q header does not have a PCP. OpenFlow does not specify what to do if a match on PCP is actually present, but Open vSwitch ignores it.
- Otherwise, the flow matches only packets with an 802.1Q header. If **dl_vlan** is not wildcarded, then the flow only matches packets with the VLAN ID specified in **dl_vlan**'s low 12 bits. If **dl_vlan_pcp** is not wildcarded, then the flow only matches packets with the priority specified in **dl_vlan_pcp**'s low 3 bits.

  OpenFlow does not specify how to interpret the high 4 bits of **dl_vlan** or the high 5 bits of **dl_vlan_pcp**. Open vSwitch ignores them.

OpenFlow 1.1 VLAN Fields

VLAN matching in OpenFlow 1.1 is similar to OpenFlow 1.0. The one refinement is that when **dl_vlan** matches on **0xfffe** (**OFVPID_ANY**), the flow matches only packets with an 802.1Q header, with any VLAN ID. If **dl_vlan_pcp** is wildcarded, the flow matches any packet with an 802.1Q header, regardless of VLAN ID or priority. If **dl_vlan_pcp** is not wildcarded, then the flow only matches packets with the priority specified in **dl_vlan_pcp**'s low 3 bits.

OpenFlow 1.1 uses the name **OFPVID_NONE**, instead of **OFP_VLAN_NONE**, for a **dl_vlan** of **0xffff**, but it has the same meaning.

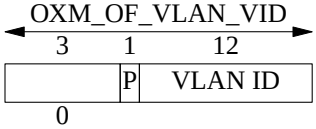
In OpenFlow 1.1, Open vSwitch reports error **OFPBMC_BAD_VALUE** for an attempt to match on **dl_vlan** between 4,096 and **0xfffd**, inclusive, or **dl_vlan_pcp** greater than 7.

OpenFlow 1.2 VLAN Fields

OpenFlow 1.2+ VLAN ID Field

Name:	**vlan_vid**
Width:	16 bits (only the least-significant 12 bits may be nonzero)
Format:	decimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_VLAN_VID** (6) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	none

The OpenFlow standard describes this field as consisting of ''12+1'' bits. On ingress, its value is 0 if no 802.1Q header is present, and otherwise it holds the VLAN VID in its least significant 12 bits, with bit 12 (**0x1000** aka **OFPVID_PRESENT**) also set to 1. The three most significant bits are always zero:



As a consequence of this field's format, one may use it to match the VLAN ID in all of the ways available with the OpenFlow 1.0 and 1.1 formats, and a few new ways:

Fully wildcarded
: Matches any packet, that is, one without an 802.1Q header or with an 802.1Q header with any TCI value.

Value **0x0000** (**OFPVID_NONE**), mask **0xffff** (or no mask)
Matches only packets without an 802.1Q header.

Value **0x1000**, mask **0x1000**
Matches any packet with an 802.1Q header, regardless of VLAN ID.

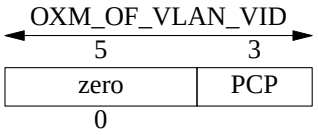
Value **0x1009**, mask **0xffff** (or no mask)
Match only packets with an 802.1Q header with VLAN ID 9.

Value **0x1001**, mask **0x1001**
Matches only packets that have an 802.1Q header with an odd-numbered VLAN ID. (This is just an example; one can match on any desired VLAN ID bit pattern.)

OpenFlow 1.2+ VLAN Priority Field

Name:	**vlan_pcp**
Width:	8 bits (only the least-significant 3 bits may be nonzero)
Format:	decimal
Masking:	not maskable
Prerequisites:	VLAN VID
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_VLAN_PCP** (7) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	none

The 3 least significant bits may be used to match the PCP bits in an 802.1Q header. Other bits are always zero:



May only be used when **vlan_vid** is not wildcarded and does not exact match on 0 (which only matches when there is no 802.1Q header).

See *VLAN Comparison Chart*, below, for some examples.

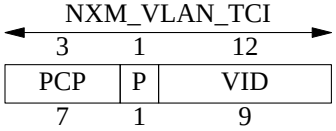
Open vSwitch Extension VLAN Field

This extension **vlan_tci** can describe more kinds of VLAN matches than the other variants. It is also simpler than the other variants.

VLAN TCI Field

Name:	**vlan_tci**
Width:	16 bits
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	none
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	none
NXM:	**NXM_OF_VLAN_TCI** (4) since Open vSwitch 1.1

For a packet without an 802.1Q header, this field is zero. For a packet with an 802.1Q header, this field is the TCI with the bit in CFI's position (marked **P** for ''present'' below) forced to 1. Thus, for a packet in VLAN 9 with priority 7, it has the value **0xf009**:



Usage examples:

vlan_tci=0
Match packets without an 802.1Q header.

vlan_tci=0x1000/0x1000
Match packets with an 802.1Q header, regardless of VLAN and priority values.

vlan_tci=0xf123
Match packets tagged with priority 7 in VLAN 0x123.

vlan_tci=0x1123/0x1fff
Match packets tagged with VLAN 0x123 (and any priority).

vlan_tci=0x5000/0xf000
Match packets tagged with priority 2 (in any VLAN).

vlan_tci=0/0xfff
Match packets with no 802.1Q header or tagged with VLAN 0 (and any priority).

vlan_tci=0x5000/0xe000
Match packets with no 802.1Q header or tagged with prior- ity 2 (in any VLAN).

vlan_tci=0/0xefff
Match packets with no 802.1Q header or tagged with VLAN 0 and priority 0.

See *VLAN Comparison Chart*, below, for more examples.

VLAN Comparison Chart

The following table describes each of several possible matching criteria on 802.1Q header may be expressed with each variation of the VLAN matching fields:

Criteria	OpenFlow 1.0	OpenFlow 1.1	OpenFlow 1.2+	NXM
[1]	`????/1,??/?`	`????/1,??/?`	`0000/0000,--`	`0000/0000`
[2]	`ffff/0,??/?`	`ffff/0,??/?`	`0000/ffff,--`	`0000/ffff`
[3]	`0xxx/0,??/1`	`0xxx/0,??/1`	`1xxx/ffff,--`	`1xxx/1fff`
[4]	`????/1,0y/0`	`fffe/0,0y/0`	`1000/1000,0y`	`z000/f000`
[5]	`0xxx/0,0y/0`	`0xxx/0,0y/0`	`1xxx/ffff,0y`	`zxxx/ffff`
[6]	(none)	(none)	`1001/1001,--`	`1001/1001`
[7]	(none)	(none)	(none)	`3000/3000`
[8]	(none)	(none)	(none)	`0000/0fff`
[9]	(none)	(none)	(none)	`0000/f000`
[10]	(none)	(none)	(none)	`0000/efff`

All numbers in the table are expressed in hexadecimal. The columns in the table are interpreted as follows:

Criteria See the list below.

OpenFlow 1.0
OpenFlow 1.1
`wwww/x,yy/z` means VLAN ID match value `wwww` with wildcard bit `x` and VLAN PCP match value `yy` with wildcard bit `z`. `?` means that the given bits are ignored (and conventionally `0` for `wwww` or `yy`, conventionally `1` for `x` or `z`). ‘‘(none)’’ means that OpenFlow 1.0 (or 1.1) cannot match with these criteria.

OF1.2 `xxxx/yyyy,zz` means **vlan_vid** with value `xxxx` and mask `yyyy`, and **vlan_pcp** (which is not maskable) with value `zz`. `--` means that **vlan_pcp** is omitted. ‘‘(none)’’ means that OpenFlow 1.2 cannot match with these criteria.

NXM `xxxx/yyyy` means **vlan_tci** with value `xxxx` and mask `yyyy`.

The matching criteria described by the table are:

[1] Matches any packet, that is, one without an 802.1Q header or with an 802.1Q header with any TCI value.

[2] Matches only packets without an 802.1Q header.

OpenFlow 1.0 doesn't define the behavior if **dl_vlan** is set to **0xffff** and **dl_vlan_pcp** is not wildcarded. (Open vSwitch always ignores **dl_vlan_pcp** when **dl_vlan** is set to **0xffff**.)

OpenFlow 1.1 says explicitly to ignore **dl_vlan_pcp** when **dl_vlan** is set to **0xffff**.

OpenFlow 1.2 doesn't say how to interpret a match with **vlan_vid** value 0 and a mask with **OFPVID_PRESENT** (**0x1000**) set to 1 and some other bits in the mask set to 1 also. Open vSwitch interprets it the same way as a mask of **0x1000**.

Any NXM match with **vlan_tci** value 0 and the CFI bit set to 1 in the mask is equivalent to the one listed in the table.

[3] Matches only packets that have an 802.1Q header with VID xxx (and any PCP).

[4] Matches only packets that have an 802.1Q header with PCP y (and any VID).

OpenFlow 1.0 doesn't clearly define the behavior for this case. Open vSwitch implements it this way.

In the NXM value, z equals (y << 1) | 1.

[5] Matches only packets that have an 802.1Q header with VID xxx and PCP y.

In the NXM value, z equals (y << 1) | 1.

[6] Matches only packets that have an 802.1Q header with an odd-numbered VID (and any PCP). Only possible with OpenFlow 1.2 and NXM. (This is just an example; one can match on any desired VID bit pattern.)

[7] Matches only packets that have an 802.1Q header with an odd-numbered PCP (and any VID). Only possible with NXM. (This is just an example; one can match on any desired VID bit pattern.)

[8] Matches packets with no 802.1Q header or with an 802.1Q header with a VID of 0. Only possible with NXM.

[9] Matches packets with no 802.1Q header or with an 802.1Q header with a PCP of 0. Only possible with NXM.

[10] Matches packets with no 802.1Q header or with an 802.1Q header with both VID and PCP of 0. Only possible with NXM.

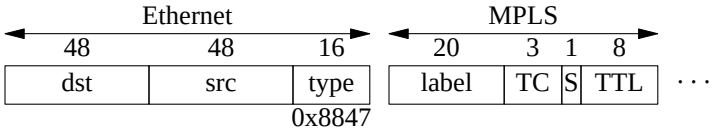
LAYER 2.5: MPLS FIELDS

Summary:

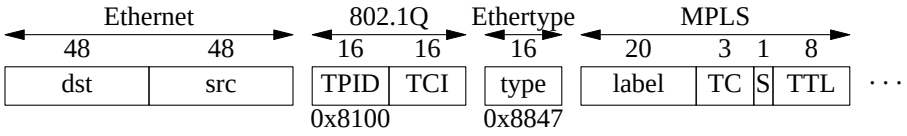
Name	Bytes	Mask	RW?	Prereqs	Support
mpls_label	4 (low 20 bits)	no	yes	MPLS	OF1.2+ and OVS 1.11+
mpls_tc	1 (low 3 bits)	no	yes	MPLS	OF1.2+ and OVS 1.11+
mpls_bos	1 (low 1 bits)	no	no	MPLS	OF1.3+ and OVS 1.11+
mpls_ttl	1	no	yes	MPLS	OVS 2.6+

One or more MPLS headers (more commonly called *MPLS labels*) follow an Ethernet type field that specifies an MPLS Ethernet type [RFC 3032]. Ethertype **0x8847** is used for all unicast. Multicast MPLS is divided into two specific classes, one of which uses Ethertype **0x8847** and the other **0x8848** [RFC 5332].

The most common overall packet format is Ethernet II, shown below (SNAP encapsulation may be used but is not ordinarily seen in Ethernet networks):



MPLS can be encapsulated inside an 802.1Q header, in which case the combination looks like this:



The fields within an MPLS label are:

Label, 20 bits.
: An identifier.

Traffic control (TC), 3 bits.
: Used for quality of service.

Bottom of stack (BOS), 1 bit (labeled just ‘‘S’’ above).
: 0 indicates that another MPLS label follows this one.

  1 indicates that this MPLS label is the last one in the stack, so that some other protocol follows this one.

Time to live (TTL), 8 bits.
: Each hop across an MPLS network decrements the TTL by 1. If it reaches 0, the packet is discarded.

  OpenFlow does not make the MPLS TTL available as a match field, but actions are available to set and decrement the TTL. Open vSwitch 2.6 and later makes the MPLS TTL available as an extension.

MPLS Label Stacks

Unlike the other encapsulations supported by OpenFlow and Open vSwitch, MPLS labels are routinely used in ‘‘stacks’’ two or three deep and sometimes even deeper. Open vSwitch currently supports up to three labels.

OpenFlow only supports matching on the outermost MPLS label at any given time. To match on the second label, one must first ‘‘pop’’ the outer label and advance to another OpenFlow table, where the inner label may be matched. To match on the third label, one must pop the two outer labels, and so on. The Open Networking Foundation is considering support for directly matching on multiple MPLS labels for OpenFlow 1.6.

MPLS Inner Protocol

Unlike all other forms of encapsulation that Open vSwitch and OpenFlow support, an MPLS label does not indicate what inner protocol it encapsulates. Different deployments determine the inner protocol in different ways [RFC 3032]:

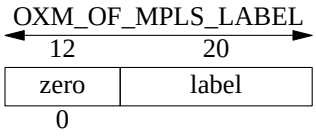
- A few reserved label values do indicate an inner protocol. Label 0, the ‘‘IPv4 Explicit NULL Label,’’ indicates inner IPv4. Label 2, the ‘‘IPv6 Explicit NULL Label,’’ indicates inner IPv6.
- Some deployments use a single inner protocol consistently.
- In some deployments, the inner protocol must be inferred from the innermost label.
- In some deployments, the inner protocol must be inferred from the innermost label and the encapsulated data, e.g. to distinguish between inner IPv4 and IPv6 based on whether the first nibble of the inner protocol data are **4** or **6**. OpenFlow and Open vSwitch do not currently support these cases.

Field Details

MPLS Label Field

Name:	**mpls_label**
Width:	32 bits (only the least-significant 20 bits may be nonzero)
Format:	decimal
Masking:	not maskable
Prerequisites:	MPLS
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_MPLS_LABEL** (34) since OpenFlow 1.2 and Open vSwitch 1.11
NXM:	none

The least significant 20 bits hold the ‘‘label’’ field from the MPLS label. Other bits are zero:

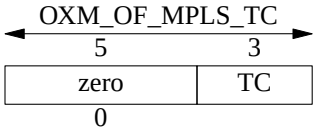


Most label values are available for any use by deployments. Values under 16 are reserved.

MPLS Traffic Class Field

Name:	**mpls_tc**
Width:	8 bits (only the least-significant 3 bits may be nonzero)
Format:	decimal
Masking:	not maskable
Prerequisites:	MPLS
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_MPLS_TC** (35) since OpenFlow 1.2 and Open vSwitch 1.11
NXM:	none

The least significant 3 bits hold the TC field from the MPLS label. Other bits are zero:



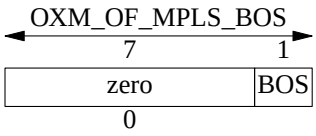
This field is intended for use for Quality of Service (QoS) and Explicit Congestion Notification purposes, but its particular interpretation is deployment specific.

Before 2009, this field was named EXP and reserved for experimental use [RFC 5462].

MPLS Bottom of Stack Field

Name:	**mpls_bos**
Width:	8 bits (only the least-significant 1 bits may be nonzero)
Format:	decimal
Masking:	not maskable
Prerequisites:	MPLS
Access:	read-only
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_MPLS_BOS** (36) since OpenFlow 1.3 and Open vSwitch 1.11
NXM:	none

The least significant bit holds the BOS field from the MPLS label. Other bits are zero:



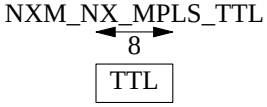
This field is useful as part of processing a series of incoming MPLS labels. A flow that includes a **pop_mpls** action should generally match on **mpls_bos**:

- When **mpls_bos** is 1, there is another MPLS label following this one, so the Ethertype passed to **pop_mpls** should be an MPLS Ethertype. For example: **table=0, dl_type=0x8847, mpls_bos=1, actions=pop_mpls:0x8847, goto_table:1**
- When **mpls_bos** is 0, this MPLS label is the last one, so the Ethertype passed to **pop_mpls** should be a non-MPLS Ethertype such as IPv4. For example: **table=1, dl_type=0x8847, mpls_bos=0, actions=pop_mpls:0x0800, goto_table:2**

MPLS Time-to-Live Field

Name:	**mpls_ttl**
Width:	8 bits
Format:	decimal
Masking:	not maskable
Prerequisites:	MPLS
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_MPLS_TTL** (30) since Open vSwitch 2.6

Holds the 8-bit time-to-live field from the MPLS label:



LAYER 3: IPV4 AND IPV6 FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
ip_src aka **nw_src**	4	yes	yes	IPv4	OF1.2+ and OVS 1.1+
ip_dst aka **nw_dst**	4	yes	yes	IPv4	OF1.2+ and OVS 1.1+
ipv6_src	16	yes	yes	IPv6	OF1.2+ and OVS 1.1+
ipv6_dst	16	yes	yes	IPv6	OF1.2+ and OVS 1.1+
ipv6_label	4 (low 20 bits)	yes	yes	IPv6	OF1.2+ and OVS 1.4+
nw_proto aka **ip_proto**	1	no	no	IPv4/IPv6	OF1.2+ and OVS 1.1+
nw_ttl	1	no	yes	IPv4/IPv6	OVS 1.4+
ip_frag	1 (low 2 bits)	yes	no	IPv4/IPv6	OVS 1.3+
nw_tos	1	no	yes	IPv4/IPv6	OVS 1.1+
ip_dscp	1 (low 6 bits)	no	yes	IPv4/IPv6	OF1.2+ and OVS 1.7+
nw_ecn aka **ip_ecn**	1 (low 2 bits)	no	yes	IPv4/IPv6	OF1.2+ and OVS 1.4+

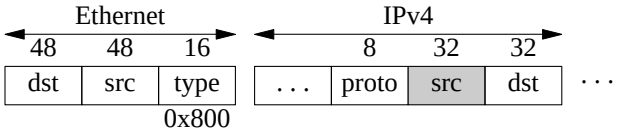
IPv4 Specific Fields

These fields are applicable only to IPv4 flows, that is, flows that match on the IPv4 Ethertype **0x0800**.

IPv4 Source Address Field

Name:	**ip_src** (aka **nw_src**)
Width:	32 bits
Format:	IPv4
Masking:	arbitrary bitwise masks
Prerequisites:	IPv4
Access:	read/write
OpenFlow 1.0:	yes (CIDR match only)
OpenFlow 1.1:	yes
OXM:	**OXM_OF_IPV4_SRC** (11) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_IP_SRC** (7) since Open vSwitch 1.1

The source address from the IPv4 header:

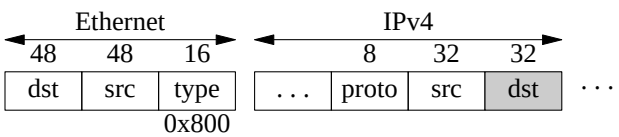


For historical reasons, in an ARP or RARP flow, Open vSwitch interprets matches on **nw_src** as actually referring to the ARP SPA.

IPv4 Destination Address Field

Name:	**ip_dst** (aka **nw_dst**)
Width:	32 bits
Format:	IPv4
Masking:	arbitrary bitwise masks
Prerequisites:	IPv4
Access:	read/write
OpenFlow 1.0:	yes (CIDR match only)
OpenFlow 1.1:	yes
OXM:	**OXM_OF_IPV4_DST** (12) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_IP_DST** (8) since Open vSwitch 1.1

The destination address from the IPv4 header:



For historical reasons, in an ARP or RARP flow, Open vSwitch interprets matches on **nw_dst** as actually referring to the ARP TPA.

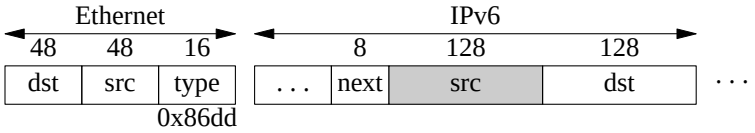
IPv6 Specific Fields

These fields apply only to IPv6 flows, that is, flows that match on the IPv6 Ethertype **0x86dd**.

IPv6 Source Address Field

Name:	**ipv6_src**
Width:	128 bits
Format:	IPv6
Masking:	arbitrary bitwise masks
Prerequisites:	IPv6
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_IPV6_SRC** (26) since OpenFlow 1.2 and Open vSwitch 1.1
NXM:	**NXM_NX_IPV6_SRC** (19) since Open vSwitch 1.1

The source address from the IPv6 header:

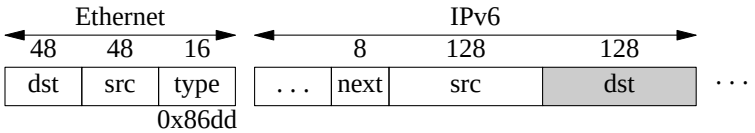


Open vSwitch 1.8 added support for bitwise matching; earlier versions supported only CIDR masks.

IPv6 Destination Address Field

Name:	**ipv6_dst**
Width:	128 bits
Format:	IPv6
Masking:	arbitrary bitwise masks
Prerequisites:	IPv6
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_IPV6_DST** (27) since OpenFlow 1.2 and Open vSwitch 1.1
NXM:	**NXM_NX_IPV6_DST** (20) since Open vSwitch 1.1

The destination address from the IPv6 header:



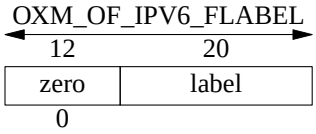
Open vSwitch 1.8 added support for bitwise matching; earlier versions supported only CIDR masks.

IPv6 Flow Label Field

Name:	**ipv6_label**
Width:	32 bits (only the least-significant 20 bits may be nonzero)
Format:	hexadecimal
Masking:	arbitrary bitwise masks
Prerequisites:	IPv6
Access:	read/write

OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_IPV6_FLABEL** (28) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_NX_IPV6_LABEL** (27) since Open vSwitch 1.4

The least significant 20 bits hold the flow label field from the IPv6 header. Other bits are zero:



IPv4/IPv6 Fields

These fields exist with at least approximately the same meaning in both IPv4 and IPv6, so they are treated as a single field for matching purposes. Any flow that matches on the IPv4 Ethertype **0x0800** or the IPv6 Ethertype **0x86dd** may match on these fields.

IPv4/v6 Protocol Field

Name:	**nw_proto** (aka **ip_proto**)
Width:	8 bits
Format:	decimal
Masking:	not maskable
Prerequisites:	IPv4/IPv6
Access:	read-only
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_IP_PROTO** (10) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_IP_PROTO** (6) since Open vSwitch 1.1

Matches the IPv4 or IPv6 protocol type.

For historical reasons, in an ARP or RARP flow, Open vSwitch interprets matches on **nw_proto** as actually referring to the ARP opcode. The ARP opcode is a 16-bit field, so for matching purposes ARP opcodes greater than 255 are treated as 0; this works adequately because in practice ARP and RARP only use opcodes 1 through 4.

IPv4/v6 TTL/Hop Limit Field

Name:	**nw_ttl**
Width:	8 bits
Format:	decimal
Masking:	not maskable
Prerequisites:	IPv4/IPv6
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	none
NXM:	**NXM_NX_IP_TTL** (29) since Open vSwitch 1.4

IPv4/v6 Fragment Bitmask Field

Name:	**ip_frag**
Width:	8 bits (only the least-significant 2 bits may be nonzero)
Format:	frag
Masking:	arbitrary bitwise masks
Prerequisites:	IPv4/IPv6
Access:	read-only
OpenFlow 1.0:	not supported

OpenFlow 1.1: not supported
OXM: none
NXM: **NXM_NX_IP_FRAG** (26) since Open vSwitch 1.3

Specifies what kinds of IP fragments or non-fragments to match. The value for this field is most conveniently specified as one of the following:

no Match only non-fragmented packets.

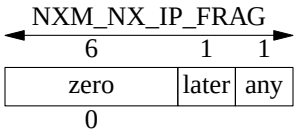
yes Matches all fragments.

first Matches only fragments with offset 0.

later Matches only fragments with nonzero offset.

not_later
Matches non-fragmented packets and fragments with zero offset.

The field is internally formatted as 2 bits: bit 0 is 1 for an IP fragment with any offset (and otherwise 0), and bit 1 is 1 for an IP fragment with nonzero offset (and otherwise 0), like so:



Even though 2 bits have 4 possible values, this field only uses 3 of them:

- A packet that is not an IP fragment has value 0.
- A packet that is an IP fragment with offset 0 (the first fragment) has bit 0 set and thus value 1.
- A packet that is an IP fragment with nonzero offset has bits 0 and 1 set and thus value 3.

The switch may reject matches against values that can never appear.

It is important to understand how this field interacts with the OpenFlow fragment handling mode:

- In **OFPC_FRAG_DROP** mode, the OpenFlow switch drops all IP fragments before they reach the flow table, so every packet that is available for matching will have value 0 in this field.
- Open vSwitch does not implement **OFPC_FRAG_REASM** mode, but if it did then IP fragments would be reassembled before they reached the flow table and again every packet available for matching would always have value 0.
- In **OFPC_FRAG_NORMAL** mode, all three values are possible, but OpenFlow 1.0 says that fragments' transport ports are always 0, even for the first fragment, so this does not provide much extra information.
- In **OFPC_FRAG_NX_MATCH** mode, all three values are possible. For fragments with offset 0, Open vSwitch makes L4 header information available.

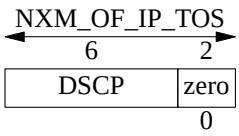
Thus, this field is likely to be most useful for an Open vSwitch switch configured in **OFPC_FRAG_NX_MATCH** mode. See the description of the **set−frags** command in **ovs−ofctl**(8), for more details.

IPv4/IPv6 TOS Fields

IPv4/v6 DSCP (Bits 2-7) Field

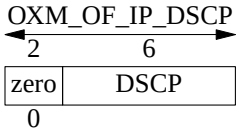
Name: **nw_tos**
Width: 8 bits
Format: decimal
Masking: not maskable

Prerequisites: IPv4/IPv6
Access: read/write
OpenFlow 1.0: yes (exact match only)
OpenFlow 1.1: yes (exact match only)
OXM: none
NXM: **NXM_OF_IP_TOS** (5) since Open vSwitch 1.1



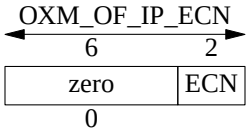
IPv4/v6 DSCP (Bits 0-5) Field

Name: **ip_dscp**
Width: 8 bits (only the least-significant 6 bits may be nonzero)
Format: decimal
Masking: not maskable
Prerequisites: IPv4/IPv6
Access: read/write
OpenFlow 1.0: yes (exact match only)
OpenFlow 1.1: yes (exact match only)
OXM: **OXM_OF_IP_DSCP** (8) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: none



IPv4/v6 ECN Field

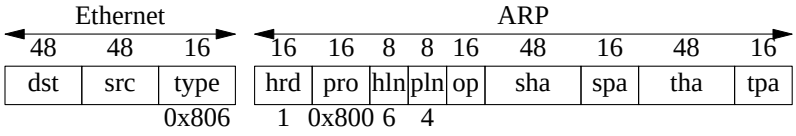
Name: **nw_ecn** (aka **ip_ecn**)
Width: 8 bits (only the least-significant 2 bits may be nonzero)
Format: decimal
Masking: not maskable
Prerequisites: IPv4/IPv6
Access: read/write
OpenFlow 1.0: not supported
OpenFlow 1.1: not supported
OXM: **OXM_OF_IP_ECN** (9) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_NX_IP_ECN** (28) since Open vSwitch 1.4



LAYER 3: ARP FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
arp_op	2	no	yes	ARP	OF1.2+ and OVS 1.1+
arp_spa	4	yes	yes	ARP	OF1.2+ and OVS 1.1+
arp_tpa	4	yes	yes	ARP	OF1.2+ and OVS 1.1+
arp_sha	6	yes	yes	ARP	OF1.2+ and OVS 1.1+
arp_tha	6	yes	yes	ARP	OF1.2+ and OVS 1.1+



ARP Opcode Field

Name: **arp_op**
Width: 16 bits
Format: decimal
Masking: not maskable
Prerequisites: ARP
Access: read/write
OpenFlow 1.0: yes (exact match only)
OpenFlow 1.1: yes (exact match only)
OXM: **OXM_OF_ARP_OP** (21) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_OF_ARP_OP** (15) since Open vSwitch 1.1

ARP Source IPv4 Address Field

Name: **arp_spa**
Width: 32 bits
Format: IPv4
Masking: arbitrary bitwise masks
Prerequisites: ARP
Access: read/write
OpenFlow 1.0: yes (CIDR match only)
OpenFlow 1.1: yes
OXM: **OXM_OF_ARP_SPA** (22) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_OF_ARP_SPA** (16) since Open vSwitch 1.1

ARP Target IPv4 Address Field

Name: **arp_tpa**
Width: 32 bits
Format: IPv4
Masking: arbitrary bitwise masks
Prerequisites: ARP
Access: read/write
OpenFlow 1.0: yes (CIDR match only)
OpenFlow 1.1: yes
OXM: **OXM_OF_ARP_TPA** (23) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_OF_ARP_TPA** (17) since Open vSwitch 1.1

ARP Source Ethernet Address Field

Name: **arp_sha**
Width: 48 bits
Format: Ethernet

Masking:	arbitrary bitwise masks
Prerequisites:	ARP
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_ARP_SHA** (24) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_NX_ARP_SHA** (17) since Open vSwitch 1.1

ARP Target Ethernet Address Field

Name:	**arp_tha**
Width:	48 bits
Format:	Ethernet
Masking:	arbitrary bitwise masks
Prerequisites:	ARP
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_ARP_THA** (25) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_NX_ARP_THA** (18) since Open vSwitch 1.1

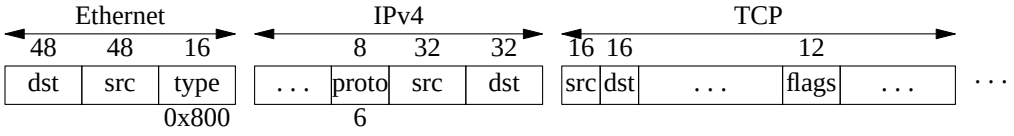
LAYER 4: TCP, UDP, AND SCTP FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
tcp_src aka **tp_src**	2	yes	yes	TCP	OF1.2+ and OVS 1.1+
tcp_dst aka **tp_dst**	2	yes	yes	TCP	OF1.2+ and OVS 1.1+
tcp_flags	2 (low 12 bits)	yes	no	TCP	OF1.3+ and OVS 2.1+
udp_src	2	yes	yes	UDP	OF1.2+ and OVS 1.1+
udp_dst	2	yes	yes	UDP	OF1.2+ and OVS 1.1+
sctp_src	2	yes	yes	SCTP	OF1.2+ and OVS 2.0+
sctp_dst	2	yes	yes	SCTP	OF1.2+ and OVS 2.0+

For matching purposes, no distinction is made whether these protocols are encapsulated within IPv4 or IPv6.

TCP



TCP Source Port Field

Name:	**tcp_src** (aka **tp_src**)
Width:	16 bits
Format:	decimal
Masking:	arbitrary bitwise masks
Prerequisites:	TCP
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_TCP_SRC** (13) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_TCP_SRC** (9) since Open vSwitch 1.1

Open vSwitch 1.6 added support for bitwise matching.

TCP Destination Port Field

Name:	**tcp_dst** (aka **tp_dst**)
Width:	16 bits
Format:	decimal
Masking:	arbitrary bitwise masks
Prerequisites:	TCP
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_TCP_DST** (14) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_TCP_DST** (10) since Open vSwitch 1.1

Open vSwitch 1.6 added support for bitwise matching.

TCP Flags Field

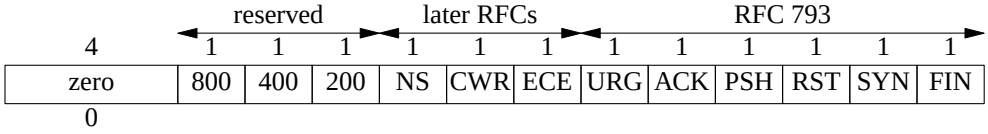
Name:	**tcp_flags**
Width:	16 bits (only the least-significant 12 bits may be nonzero)
Format:	TCP flags
Masking:	arbitrary bitwise masks
Prerequisites:	TCP
Access:	read-only
OpenFlow 1.0:	not supported

OpenFlow 1.1: not supported
OXM: **ONFOXM_ET_TCP_FLAGS** (42) since OpenFlow 1.3 and Open vSwitch 2.4; **OXM_OF_TCP_FLAGS** (42) since OpenFlow 1.5 and Open vSwitch 2.3
NXM: **NXM_NX_TCP_FLAGS** (34) since Open vSwitch 2.1

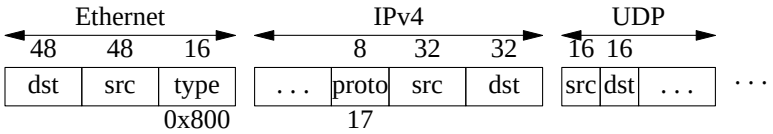
This field holds the TCP flags. TCP currently defines 9 flag bits. An additional 3 bits are reserved. For more information, see [RFC 793], [RFC 3168], and [RFC 3540].

Matches on this field are most conveniently written in terms of symbolic names (given in the diagram below), each preceded by either **+** for a flag that must be set, or **−** for a flag that must be unset, without any other delimiters between the flags. Flags not mentioned are wildcarded. For example, **tcp,tcp_flags=+syn−ack** matches TCP SYNs that are not ACKs. Matches can also be written as *flags*/*mask*, where *flags* and *mask* are 16-bit numbers in decimal or in hexadecimal prefixed by **0x**.

The flag bits are:



UDP



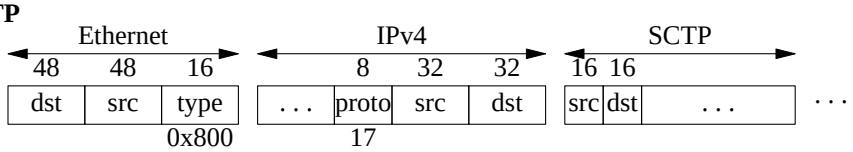
UDP Source Port Field

Name: **udp_src**
Width: 16 bits
Format: decimal
Masking: arbitrary bitwise masks
Prerequisites: UDP
Access: read/write
OpenFlow 1.0: yes (exact match only)
OpenFlow 1.1: yes (exact match only)
OXM: **OXM_OF_UDP_SRC** (15) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_OF_UDP_SRC** (11) since Open vSwitch 1.1

UDP Destination Port Field

Name: **udp_dst**
Width: 16 bits
Format: decimal
Masking: arbitrary bitwise masks
Prerequisites: UDP
Access: read/write
OpenFlow 1.0: yes (exact match only)
OpenFlow 1.1: yes (exact match only)
OXM: **OXM_OF_UDP_DST** (16) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_OF_UDP_DST** (12) since Open vSwitch 1.1

SCTP



SCTP Source Port Field

Name:	**sctp_src**
Width:	16 bits
Format:	decimal
Masking:	arbitrary bitwise masks
Prerequisites:	SCTP
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_SCTP_SRC** (17) since OpenFlow 1.2 and Open vSwitch 2.0
NXM:	none

SCTP Destination Port Field

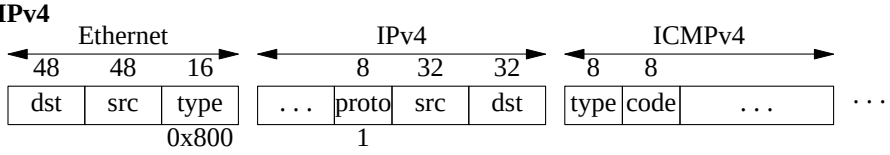
Name:	**sctp_dst**
Width:	16 bits
Format:	decimal
Masking:	arbitrary bitwise masks
Prerequisites:	SCTP
Access:	read/write
OpenFlow 1.0:	not supported
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_SCTP_DST** (18) since OpenFlow 1.2 and Open vSwitch 2.0
NXM:	none

LAYER 4: ICMPV4 AND ICMPV6 FIELDS

Summary:

Name	Bytes	Mask	RW?	Prereqs	Support
icmp_type	1	no	yes	ICMPv4	OF1.2+ and OVS 1.1+
icmp_code	1	no	yes	ICMPv4	OF1.2+ and OVS 1.1+
icmpv6_type	1	no	yes	ICMPv6	OF1.2+ and OVS 1.1+
icmpv6_code	1	no	yes	ICMPv6	OF1.2+ and OVS 1.1+
nd_target	16	yes	yes	ND	OF1.2+ and OVS 1.1+
nd_sll	6	yes	yes	ND solicit	OF1.2+ and OVS 1.1+
nd_tll	6	yes	yes	ND advert	OF1.2+ and OVS 1.1+

ICMPv4



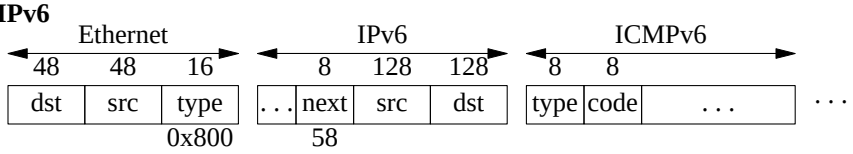
ICMPv4 Type Field

Name:	**icmp_type**
Width:	8 bits
Format:	decimal
Masking:	not maskable
Prerequisites:	ICMPv4
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_ICMPV4_TYPE** (19) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_ICMP_TYPE** (13) since Open vSwitch 1.1

ICMPv4 Code Field

Name:	**icmp_code**
Width:	8 bits
Format:	decimal
Masking:	not maskable
Prerequisites:	ICMPv4
Access:	read/write
OpenFlow 1.0:	yes (exact match only)
OpenFlow 1.1:	yes (exact match only)
OXM:	**OXM_OF_ICMPV4_CODE** (20) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_OF_ICMP_CODE** (14) since Open vSwitch 1.1

ICMPv6



ICMPv6 Type Field

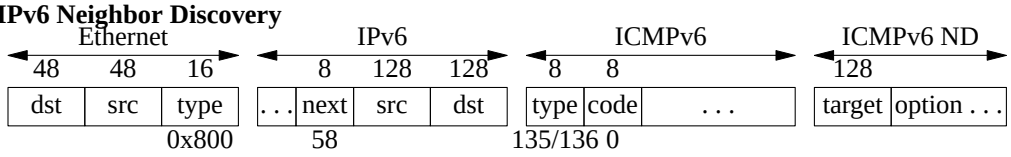
Name:	**icmpv6_type**
Width:	8 bits
Format:	decimal
Masking:	not maskable
Prerequisites:	ICMPv6
Access:	read/write

OpenFlow 1.0: not supported
OpenFlow 1.1: not supported
OXM: **OXM_OF_ICMPV6_TYPE** (29) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_NX_ICMPV6_TYPE** (21) since Open vSwitch 1.1

ICMPv6 Code Field

Name: **icmpv6_code**
Width: 8 bits
Format: decimal
Masking: not maskable
Prerequisites: ICMPv6
Access: read/write
OpenFlow 1.0: not supported
OpenFlow 1.1: not supported
OXM: **OXM_OF_ICMPV6_CODE** (30) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_NX_ICMPV6_CODE** (22) since Open vSwitch 1.1

ICMPv6 Neighbor Discovery



ICMPv6 Neighbor Discovery Target IPv6 Field

Name: **nd_target**
Width: 128 bits
Format: IPv6
Masking: arbitrary bitwise masks
Prerequisites: ND
Access: read/write
OpenFlow 1.0: not supported
OpenFlow 1.1: not supported
OXM: **OXM_OF_IPV6_ND_TARGET** (31) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_NX_ND_TARGET** (23) since Open vSwitch 1.1

ICMPv6 Neighbor Discovery Source Ethernet Address Field

Name: **nd_sll**
Width: 48 bits
Format: Ethernet
Masking: arbitrary bitwise masks
Prerequisites: ND solicit
Access: read/write
OpenFlow 1.0: not supported
OpenFlow 1.1: not supported
OXM: **OXM_OF_IPV6_ND_SLL** (32) since OpenFlow 1.2 and Open vSwitch 1.7
NXM: **NXM_NX_ND_SLL** (24) since Open vSwitch 1.1

ICMPv6 Neighbor Discovery Target Ethernet Address Field

Name: **nd_tll**
Width: 48 bits
Format: Ethernet
Masking: arbitrary bitwise masks
Prerequisites: ND advert
Access: read/write

OpenFlow 1.0:	not supported
OpenFlow 1.1:	not supported
OXM:	**OXM_OF_IPV6_ND_TLL** (33) since OpenFlow 1.2 and Open vSwitch 1.7
NXM:	**NXM_NX_ND_TLL** (25) since Open vSwitch 1.1

REFERENCES

Casado M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, ‘‘Ethane: Taking Control of the Enterprise,’’ Computer Communications Review, October 2007.

EXT−56
J. Tonsing, ‘‘Permit one of a set of prerequisites to apply, e.g. don’t preclude non-Ethernet media,’’ ⟨https://rs.opennetworking.org/bugs/browse/EXT−56⟩ (ONF members only).

EXT−112
J. Tourrilhes, ‘‘Support non-Ethernet packets throughout the pipeline,’’ ⟨https://rs.opennetworking.org/bugs/browse/EXT−112⟩ (ONF members only).

EXT−134
J. Tourrilhes, ‘‘Match first nibble of the MPLS payload,’’ ⟨https://rs.opennetworking.org/bugs/browse/EXT−134⟩ (ONF members only).

IEEE OUI
IEEE Standards Association, ‘‘MAC Address Block Large (MA-L),’’ ⟨https://standards.ieee.org/develop/regauth/oui/index.html⟩.

OpenFlow 1.0.1
Open Networking Foundation, ‘‘OpenFlow Switch Errata, Version 1.0.1,’’ June 2012.

OpenFlow 1.1
OpenFlow Consortium, ‘‘OpenFlow Switch Specification Version 1.1.0 Implemented (Wire Protocol 0x02),’’ February 2011.

OpenFlow 1.5
Open Networking Foundation, ‘‘OpenFlow Switch Specification Version 1.5.0 (Protocol version 0x06),’’ December 2014.

OpenFlow Extensions 1.3.x Package 2
Open Networking Foundation, ‘‘OpenFlow Extensions 1.3.x Package 2,’’ December 2013.

TCP Flags Match Field Extension
Open Networking Foundation, ‘‘TCP flags match field Extension,’’ December 2014. In [OpenFlow Extensions 1.3.x Package 2].

Pepelnjak
I. Pepelnjak, ‘‘OpenFlow and Fermi Estimates,’’ ⟨http://blog.ipspace.net/2013/09/openflow-and-fermi-estimates.html⟩.

RFC 793
‘‘Transmission Control Protocol,’’ ⟨http://www.ietf.org/rfc/rfc793.txt⟩.

RFC 3032
E. Rosen, D. Tappan, G. Fedorkow, Y. Rekhter, D. Farinacci, T. Li, and A. Conta, ‘‘MPLS Label Stack Encoding,’’ ⟨http://www.ietf.org/rfc/rfc3032.txt⟩.

RFC 3168
K. Ramakrishnan, S. Floyd, and D. Black, ‘‘The Addition of Explicit Congestion Notification (ECN) to IP,’’ ⟨https://tools.ietf.org/html/rfc3168⟩.

RFC 3540
N. Spring, D. Wetherall, and D. Ely, ‘‘Robust Explicit Congestion Notification (ECN) Signaling with Nonces,’’ ⟨https://tools.ietf.org/html/rfc3540⟩.

RFC 4632
V. Fuller and T. Li, ‘‘Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan,’’ ⟨https://tools.ietf.org/html/rfc4632⟩ .

RFC 5462
L. Andersson and R. Asati, ‘‘Multiprotocol Label Switching (MPLS) Label Stack Entry: ‘‘EXP’’ Field Renamed to ‘‘Traffic Class’’ Field,’’ ⟨http://www.ietf.org/rfc/rfc5462.txt⟩ .

RFC 6830
D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, ‘‘The Locator/ID Separation Protocol (LISP),’’ ⟨http://www.ietf.org/rfc/rfc6830.txt⟩ .

Srinivasan
V. Srinivasan, S. Suriy, and G. Varghese, ‘‘Packet Classification using Tuple Space Search,’’ SIGCOMM 1999.

Pagiamtzis
K. Pagiamtzis and A. Sheikholeslami, ‘‘Content-addressable memory (CAM) circuits and architectures: A tutorial and survey,’’ IEEE Journal of Solid-State Circuits, vol. 41, no. 3, pp. 712−727, March 2006.

VXLAN Group Policy Option
M. Smith and L. Kreeger, ‘‘ VXLAN Group Policy Option.’’ Internet-Draft. ⟨https://tools.ietf.org/html/draft-smith-vxlan-group-policy⟩ .

AUTHORS

Ben Pfaff, with advice from Justin Pettit and Jean Tourrilhes.