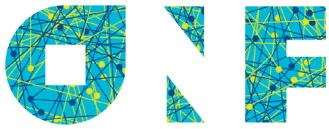


OPEN NETWORKING
FOUNDATION

OpenFlow Switch Specification

Version 1.3.2 (Wire Protocol 0x04)
April 25, 2013

Disclaimer

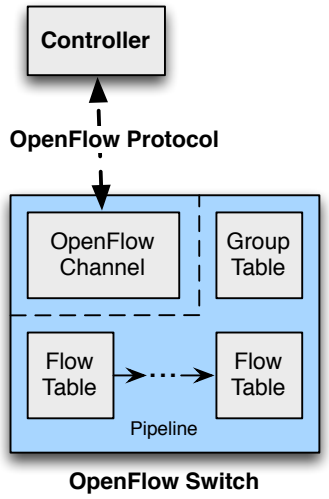
Contents

List of Tables

List of Figures

1 Introduction

This document describes the requirements of an OpenFlow Switch. We recommend that you read the latest version of the OpenFlow whitepaper before reading this specification. The whitepaper is available on the Open Networking Foundation website (`https://www.opennetworking.org/standards/intro-to-openflow`). This specification covers the components and the basic functions of the switch, and the OpenFlow protocol to manage an OpenFlow switch from a remote controller.



Figure˜1: Main components of an OpenFlow switch.

2 Switch Components

An OpenFlow Switch consists of one or more *flow tables* and a *group table*, which perform packet lookups and forwarding, and an *OpenFlow channel* to an external controller (Figure 1). The switch communicates with the controller and the controller manages the switch via the OpenFlow protocol.

Using the OpenFlow protocol, the controller can add, update, and delete *flow entries* in flow tables, both reactively (in response to packets) and proactively. Each flow table in the switch contains a set of flow entries; each flow entry consists of *match fields*, *counters*, and a set of *instructions* to apply to matching packets (see 5.2).

Matching starts at the first flow table and may continue to additional flow tables (see 5.1). Flow entries match packets in priority order, with the first matching entry in each table being used (see 5.3). If a matching entry is found, the instructions associated with the specific flow entry are executed. If no match is found in a flow table, the outcome depends on configuration of the table-miss flow entry: for example, the packet may be forwarded to the controller over the OpenFlow channel, dropped, or may continue to the next flow table (see 5.4).

Instructions associated with each flow entry either contain actions or modify pipeline processing (see 5.9). Actions included in instructions describe packet forwarding, packet modification and group table

processing. Pipeline processing instructions allow packets to be sent to subsequent tables for further processing and allow information, in the form of metadata, to be communicated between tables. Table pipeline processing stops when the instruction set associated with a matching flow entry does not specify a next table; at this point the packet is usually modified and forwarded (see 5.10).

Flow entries may forward to a *port*. This is usually a physical port, but it may also be a logical port defined by the switch or a reserved port defined by this specification (see 4.1). Reserved ports may specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as "normal" switch processing (see 4.5), while switch-defined logical ports may specify link aggregation groups, tunnels or loopback interfaces (see 4.4).

Actions associated with flow entries may also direct packets to a group, which specifies additional processing (see 5.6). Groups represent sets of actions for flooding, as well as more complex forwarding semantics (e.g. multipath, fast reroute, and link aggregation). As a general layer of indirection, groups also enable multiple flow entries to forward to a single identifier (e.g. IP forwarding to a common next hop). This abstraction allows common output actions across flow entries to be changed efficiently.

The group table contains group entries; each group entry contains a list of *action buckets* with specific semantics dependent on group type (see 5.6.1). The actions in one or more action buckets are applied to packets sent to the group.

Switch designers are free to implement the internals in any way convenient, provided that correct match and instruction semantics are preserved. For example, while a flow entry may use an all group to forward to multiple ports, a switch designer may choose to implement this as a single bitmask within the hardware forwarding table. Another example is matching; the pipeline exposed by an OpenFlow switch may be physically implemented with a different number of hardware tables.

3 Glossary

This section describes key OpenFlow specification terms:

- **Byte**: an 8-bit octet.
- **Packet**: an Ethernet frame, including header and payload.
- **Port**: where packets enter and exit the OpenFlow pipeline (see 4.1). May be a physical port, a logical port defined by the switch, or a reserved port defined by the OpenFlow protocol.
- **Pipeline**: the set of linked flow tables that provide matching, forwarding, and packet modification in an OpenFlow switch.
- **Flow Table**: a stage of the pipeline. It contains flow entries.
- **Flow Entry**: an element in a flow table used to match and process packets. It contains a set of match fields for matching packets, a priority for matching precedence, a set of counters to track packets, and a set of instructions to apply.
- **Match Field**: a field against which a packet is matched, including packet headers, the ingress port, and the metadata value. A match field may be wildcarded (match any value) and in some cases bitmasked.

- **Metadata**: a maskable register value that is used to carry information from one table to the next.
- **Instruction**: instructions are attached to a flow entry and describe the OpenFlow processing that happens when a packet matches the flow entry. An instruction either modifies pipeline processing, such as directing the packet to another flow table, or contains a *set* of actions to add to the action set, or contains a *list* of actions to apply immediately to the packet.
- **Action**: an operation that forwards the packet to a port or modifies the packet, such as decrementing the TTL field. Actions may be specified as part of the instruction set associated with a flow entry or in an action bucket associated with a group entry. Actions may be accumulated in the Action Set of the packet or applied immediately to the packet.
- **Action Set**: a set of actions associated with the packet that are accumulated while the packet is processed by each table and that are executed when the instruction set instructs the packet to exit the processing pipeline.
- **Group**: a list of action buckets and some means of choosing one or more of those buckets to apply on a per-packet basis.
- **Action Bucket**: a set of actions and associated parameters, defined for groups.
- **Tag**: a header that can be inserted or removed from a packet via push and pop actions.
- **Outermost Tag**: the tag that appears closest to the beginning of a packet.
- **Controller**: an entity interacting with the OpenFlow switch using the OpenFlow protocol.
- **Meter**: a switch element that can measure and control the rate of packets. The meter triggers a meter band if the packet rate or byte rate passing through the meter exceeds a predefined threshold. If the meter band drops the packet, it is called a **Rate Limiter**.

4 OpenFlow Ports

This section describes the OpenFlow port abstraction and the various types of OpenFlow ports supported by OpenFlow.

4.1 OpenFlow Ports

OpenFlow ports are the network interfaces for passing packets between OpenFlow processing and the rest of the network. OpenFlow switches connect logically to each other via their OpenFlow ports.

An OpenFlow switch makes a number of OpenFlow ports available for OpenFlow processing. The set of OpenFlow ports may not be identical to the set of network interfaces provided by the switch hardware, some network interfaces may be disabled for OpenFlow, and the OpenFlow switch may define additional OpenFlow ports.

OpenFlow packets are received on an **ingress port** and processed by the OpenFlow pipeline (see 5.1) which may forward them to an **output port**. The packet ingress port is a property of the packet throughout the OpenFlow pipeline and represents the OpenFlow port on which the packet was received

into the OpenFlow switch. The ingress port can be used when matching packets (see 5.3). The OpenFlow pipeline can decide to send the packet on an output port using the output action (see 5.12), which defines how the packet goes back to the network.

An OpenFlow switch must support three types of OpenFlow ports: *physical ports*, *logical ports* and *reserved ports*.

4.2 Standard Ports

The OpenFlow **standard ports** are defined as physical ports, logical ports, and the `LOCAL` reserved port if supported (excluding other reserved ports).

Standard ports can be used as ingress and output ports, they can be used in groups (see 5.6), and they have port counters (see 5.8).

4.3 Physical Ports

The OpenFlow **physical ports** are switch defined ports that correspond to a hardware interface of the switch. For example, on an Ethernet switch, physical ports map one-to-one to the Ethernet interfaces.

In some deployments, the OpenFlow switch may be virtualised over the switch hardware. In those cases, an OpenFlow physical port may represent a virtual slice of the corresponding hardware interface of the switch.

4.4 Logical Ports

The OpenFlow **logical ports** are switch defined ports that don't correspond directly to a hardware interface of the switch. Logical ports are higher level abstractions that may be defined in the switch using non-OpenFlow methods (e.g. link aggregation groups, tunnels, loopback interfaces).

Logical ports may include packet encapsulation and may map to various physical ports. The processing done by the logical port must be transparent to OpenFlow processing and those ports must interact with OpenFlow processing like OpenFlow physical ports.

The only differences between *physical ports* and *logical ports* is that a packet associated with a logical port may have an extra metadata field called *Tunnel-ID* associated with it (see 7.2.3.7) and when a packet received on a logical port is sent to the controller, both its logical port and its underlying physical port are reported to the controller (see 7.4.1).

4.5 Reserved Ports

The OpenFlow **reserved ports** are defined by this specification. They specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as "normal" switch processing.

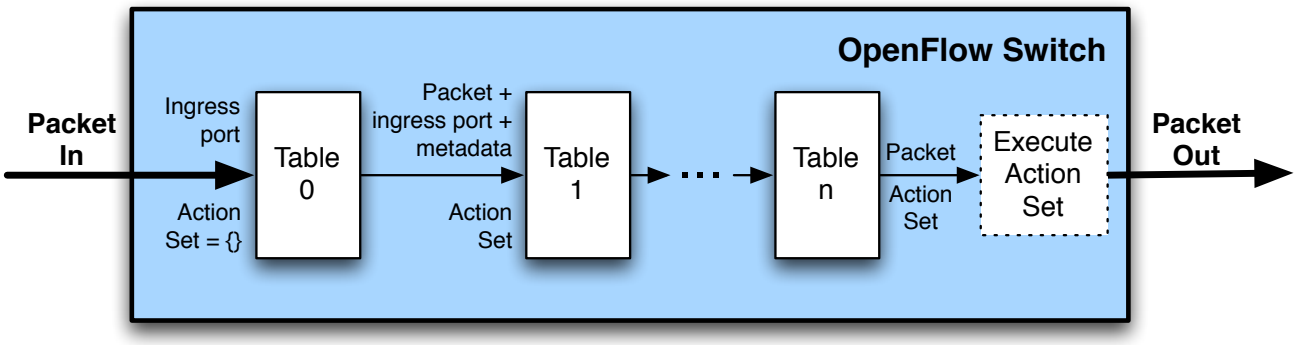
A switch is not required to support all reserved ports, just those marked *"Required"* below.

- *Required:* **ALL:** Represents all ports the switch can use for forwarding a specific packet. Can be used only as an output port. In that case a copy of the packet is sent to all standard ports, excluding the packet ingress port and ports that are configured `OFPPC_NO_FWD`.
- *Required:* **CONTROLLER:** Represents the control channel with the OpenFlow controller. Can be used as an ingress port or as an output port. When used as an output port, encapsulate the packet in a *packet-in* message and send it using the OpenFlow protocol (see 7.4.1). When used as an ingress port, this identifies a packet originating from the controller.
- *Required:* **TABLE:** Represents the start of the OpenFlow pipeline (see 5.1). This port is only valid in an output action in the action list of a *packet-out* message (see 7.3.7), and submits the packet to the first flow table so that the packet can be processed through the regular OpenFlow pipeline.
- *Required:* **IN_PORT:** Represents the packet ingress port. Can be used only as an output port, send the packet out through its ingress port.
- *Required:* **ANY:** Special value used in some OpenFlow commands when no port is specified (i.e. port is wildcarded). Can neither be used as an ingress port nor as an output port.
- *Optional:* **LOCAL:** Represents the switch's local networking stack and its management stack. Can be used as an ingress port or as an output port. The local port enables remote entities to interact with the switch and its network services via the OpenFlow network, rather than via a separate control network. With a suitable set of default flow entries it can be used to implement an in-band controller connection.
- *Optional:* **NORMAL:** Represents the traditional non-OpenFlow pipeline of the switch (see 5.1). Can be used only as an output port and processes the packet using the normal pipeline. If the switch cannot forward packets from the OpenFlow pipeline to the normal pipeline, it must indicate that it does not support this action.
- *Optional:* **FLOOD:** Represents flooding using the normal pipeline of the switch (see 5.1). Can be used only as an output port, in general will send the packet out all standard ports, but not to the ingress port, nor ports that are in `OFPPS_BLOCKED` state. The switch may also use the packet VLAN ID to select which ports to flood.

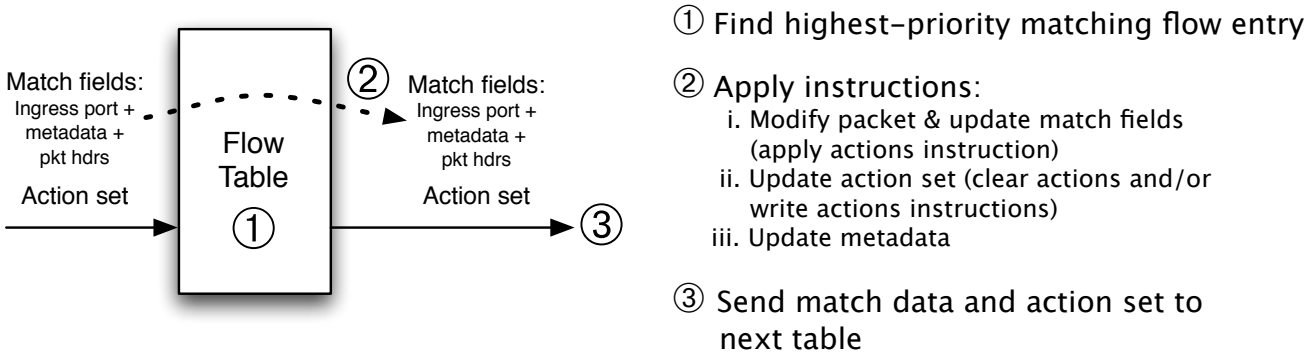
OpenFlow-only switches do not support the **NORMAL** port and **FLOOD** port, while *OpenFlow-hybrid* switches may support them (see 5.1). Forwarding packets to the **FLOOD** port depends on the switch implementation and configuration, while forwarding using a **group** of type *all* enables the controller to more flexibly implement flooding (see 5.6.1).

5 OpenFlow Tables

This section describes the components of flow tables and group tables, along with the mechanics of matching and action handling.



(a) Packets are matched against multiple tables in the pipeline



(b) Per-table packet processing

Figure˜2: Packet flow through the processing pipeline.

5.1 Pipeline Processing

OpenFlow-compliant switches come in two types: *OpenFlow-only*, and *OpenFlow-hybrid*. **OpenFlow-only** switches support only OpenFlow operation, in those switches all packets are processed by the OpenFlow pipeline, and can not be processed otherwise.

OpenFlow-hybrid switches support both OpenFlow operation and *normal* Ethernet switching operation, i.e. traditional L2 Ethernet switching, VLAN isolation, L3 routing (IPv4 routing, IPv6 routing...), ACL and QoS processing. Those switches must provide a classification mechanism outside of OpenFlow that routes traffic to *either* the OpenFlow pipeline *or* the normal pipeline. For example, a switch may use the VLAN tag or input port of the packet to decide whether to process the packet using one pipeline or the other, or it may direct all packets to the OpenFlow pipeline. This classification mechanism is outside the scope of this specification. An OpenFlow-hybrid switch may also allow a packet to go from the OpenFlow pipeline to the normal pipeline through the *NORMAL* and *FLOOD* reserved ports (see 4.5).

The **OpenFlow pipeline** of every OpenFlow switch contains multiple flow tables, each flow table containing multiple flow entries. The OpenFlow pipeline processing defines how packets interact with those flow tables (see Figure 2). An OpenFlow switch is required to have at least one flow table, and can optionally have more flow tables. An OpenFlow switch with only a single flow table is valid, in this case pipeline processing is greatly simplified.

The flow tables of an OpenFlow switch are sequentially numbered, starting at 0. Pipeline processing always starts at the first flow table: the packet is first matched against flow entries of flow table 0. Other flow tables may be used depending on the outcome of the match in the first table.

When processed by a flow table, the packet is matched against the flow entries of the flow table to select a flow entry (see 5.3). If a flow entry is found, the instruction set included in that flow entry is executed. These instructions may explicitly direct the packet to another flow table (using the Goto Instruction, see 5.9), where the same process is repeated again. A flow entry can only direct a packet to a flow table number which is greater than its own flow table number, in other words pipeline processing can only go forward and not backward. Obviously, the flow entries of the last table of the pipeline can not include the Goto instruction. If the matching flow entry does not direct packets to another flow table, pipeline processing stops at this table. When pipeline processing stops, the packet is processed with its associated action set and usually forwarded (see 5.10).

If a packet does not match a flow entry in a flow table, this is a table miss. The behavior on a table miss depends on the table configuration (see 5.4). A table-miss flow entry in the flow table can specify how to process unmatched packets: options include dropping them, passing them to another table or sending them to the controller over the control channel via packet-in messages (see 6.1.2).

The OpenFlow pipeline and various OpenFlow operations process packets of a specific type in conformance with the specifications defined for that packet type, unless the present specification or the OpenFlow configuration specify otherwise. For example, the Ethernet header definition used by OpenFlow must conform to IEEE specifications, and the TCP/IP header definition used by OpenFlow must conform to RFC specifications. Additionally, packet reordering in an OpenFlow switch must conform to the requirements of IEEE specifications, provided that the packets are processed by the same flow entries, group bucket and meter band.

5.2 Flow Table

A flow table consists of flow entries.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

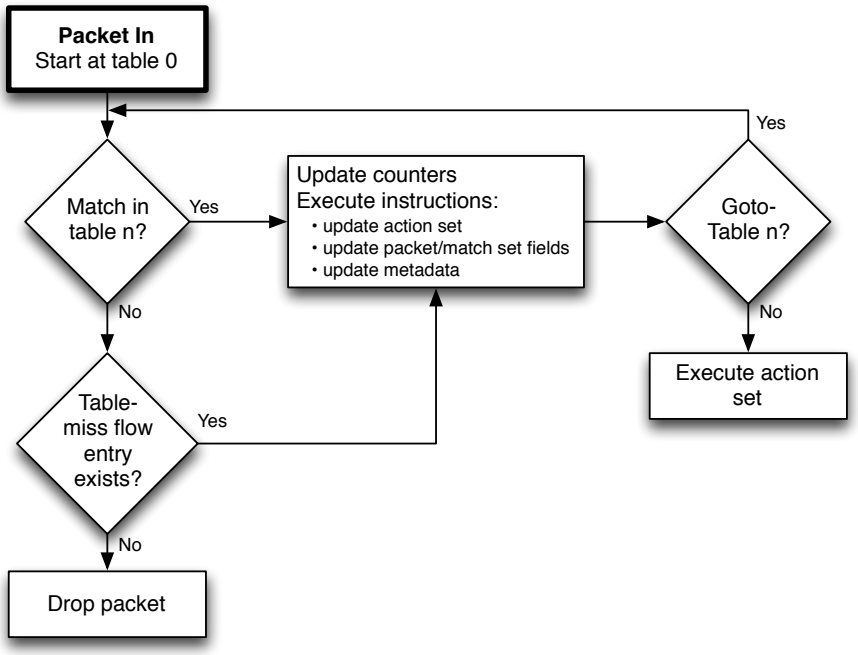
Table˜1: Main components of a flow entry in a flow table.

Each flow table entry (see Table 1) contains:

- **match fields**: to match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.
- **priority**: matching precedence of the flow entry.
- **counters**: updated when packets are matched.
- **instructions**: to modify the action set or pipeline processing.
- **timeouts**: maximum amount of time or idle time before flow is expired by the switch.
- **cookie**: opaque data value chosen by the controller. May be used by the controller to filter flow statistics, flow modification and flow deletion. Not used when processing packets.

A flow table entry is identified by its match fields and priority: the match fields and priority taken together identify a unique flow entry in the flow table. The flow entry that wildcards all fields (all fields omitted) and has priority equal to 0 is called the table-miss flow entry (see 5.4).

5.3 Matching



Figure˜3: Flowchart detailing packet flow through an OpenFlow switch.

On receipt of a packet, an OpenFlow Switch performs the functions shown in Figure 3. The switch starts by performing a table lookup in the first flow table, and based on pipeline processing, may perform table lookups in other flow tables (see 5.1).

Packet match fields are extracted from the packet. Packet match fields used for table lookups depend on the packet type, and typically include various packet header fields, such as Ethernet source address or IPv4 destination address (see 7.2.3). In addition to packet headers, matches can also be performed against the ingress port and metadata fields. Metadata may be used to pass information between tables in a switch. The packet match fields represent the packet in its current state, if actions applied in a previous table using the *Apply-Actions* changed the packet headers, those changes are reflected in the packet match fields.

A packet matches a flow table entry if the values in the packet match fields used for the lookup match those defined in the flow table entry. If a flow table entry field has a value of ANY (field omitted), it matches all possible values in the header. If the switch supports arbitrary bitmasks on specific match fields, these masks can more precisely specify matches.

The packet is matched against the table and *only* the highest priority flow entry that matches the packet must be selected. The counters associated with the selected flow entry must be updated and the instruction set included in the selected flow entry must be applied. If there are multiple matching flow

entries with the same highest priority, the selected flow entry is explicitly undefined. This case can only arise when a controller writer never sets the `OFPFF_CHECK_OVERLAP` bit on flow mod messages and adds overlapping entries.

IP fragments must be reassembled before pipeline processing if the switch configuration contains the `OFPC_FRAG_REASM` flag (see 7.3.2).

This version of the specification does *not* define the expected behavior when a switch receives a malformed or corrupted packet.

5.4 Table-miss

Every flow table must support a table-miss flow entry to process table misses. The table-miss flow entry specifies how to process packets unmatched by other flow entries in the flow table (see 5.1), and may, for example, send packets to the controller, drop packets or direct packets to a subsequent table.

The table-miss flow entry is identified by its match and its priority (see 5.2), it wildcards all match fields (all fields omitted) and has the lowest priority (0). The match of the table-miss flow entry may fall outside the normal range of matches supported by a flow table, for example an exact match table would not support wildcards for other flow entries but must support the table-miss flow entry wildcarding all fields. The table-miss flow entry may not have the same capability as regular flow entry (see 7.3.5.5). The table-miss flow entry must support at least sending packets to the controller using the CONTROLLER reserved port (see 4.5) and dropping packets using the Clear-Actions instruction (see 5.9). Implementations are encouraged to support directing packets to a subsequent table when possible for compatibility with earlier versions of this specification.

The table-miss flow entry behaves in most ways like any other flow entry: it does not exist by default in a flow table, the controller may add it or remove it at any time (see 6.4), and it may expire (see 5.5). The table-miss flow entry matches packets in the table as expected from its set of match fields and priority (see 5.3): it matches packets unmatched by other flow entries in the flow table. The table-miss flow entry instructions are applied to packets matching the table-miss flow entry (see 5.9). If the table-miss flow entry directly sends packets to the controller using the CONTROLLER reserved port (see 4.5), the packet-in reason must identify a table-miss (see 7.4.1).

If the table-miss flow entry does not exist, by default packets unmatched by flow entries are dropped (discarded). A switch configuration, for example using the OpenFlow Configuration Protocol, may override this default and specify another behaviour.

5.5 Flow Removal

Flow entries are removed from flow tables in two ways, either at the request of the controller or via the switch flow expiry mechanism.

The switch **flow expiry** mechanism is run by the switch independently of the controller and is based on the state and configuration of flow entries. Each flow entry has an `idle_timeout` and a `hard_timeout` associated with it. If the `hard_timeout` field is non-zero, the switch must note the flow entry's arrival time, as it may need to evict the entry later. A non-zero `hard_timeout` field causes the flow entry to be removed after the given number of seconds, regardless of how many packets it has matched. If the

idle_timeout field is non-zero, the switch must note the arrival time of the last packet associated with the flow, as it may need to evict the entry later. A non-zero idle_timeout field causes the flow entry to be removed when it has matched no packets in the given number of seconds. The switch must implement flow expiry and remove flow entries from the flow table when one of their timeouts is exceeded.

The controller may actively remove flow entries from flow tables by sending **delete** flow table modification messages (OFPFC_DELETE or OFPFC_DELETE_STRICT - see 6.4).

When a flow entry is removed, either by the controller or the flow expiry mechanism, the switch must check the flow entry's OFPFF_SEND_FLOW_REM flag. If this flag is set, the switch must send a flow removed message to the controller. Each flow removed message contains a complete description of the flow entry, the reason for removal (expiry or delete), the flow entry duration at the time of removal, and the flow statistics at the time of removal.

5.6 Group Table

A group table consists of group entries. The ability for a flow entry to point to a *group* enables OpenFlow to represent additional methods of forwarding (e.g. select and all).

Group Identifier	Group Type	Counters	Action Buckets

Table˜2: Main components of a group entry in the group table.

Each group entry (see Table 2) is identified by its group identifier and contains:

- **group identifier**: a 32 bit unsigned integer uniquely identifying the group
- **group type**: to determine group semantics (see Section 5.6.1)
- **counters**: updated when packets are processed by a group
- **action buckets**: an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters

5.6.1 Group Types

A switch is not required to support all group types, just those marked *"Required"* below. The controller can also query the switch about which of the *"Optional"* group types it supports.

- *Required:* **all**: Execute all buckets in the group. This group is used for multicast or broadcast forwarding. The packet is effectively cloned for each bucket; one packet is processed for each bucket of the group. If a bucket directs a packet explicitly out the ingress port, this packet clone is dropped. If the controller writer wants to forward out the ingress port, the group must include an extra bucket which includes an output action to the OFPP_IN_PORT reserved port.

- *Optional:* **select**: Execute one bucket in the group. Packets are processed by a single bucket in the group, based on a switch-computed selection algorithm (e.g. hash on some user-configured tuple or simple round robin). All configuration and state for the selection algorithm is external to OpenFlow. The selection algorithm should implement equal load sharing and can optionally be based on bucket weights. When a port specified in a bucket in a select group goes down, the switch may restrict bucket selection to the remaining set (those with forwarding actions to live ports) instead of dropping packets destined to that port. This behavior may reduce the disruption of a downed link or switch.
- *Required:* **indirect**: Execute the one defined bucket in this group. This group supports only a single bucket. Allows multiple flow entries or groups to point to a common group identifier, supporting faster, more efficient convergence (e.g. next hops for IP forwarding). This group type is effectively identical to an all group with one bucket.
- *Optional:* **fast failover**: Execute the first live bucket. Each action bucket is associated with a specific port and/or group that controls its liveness. The buckets are evaluated in the order defined by the group, and the first bucket which is associated with a live port/group is selected. This group type enables the switch to change forwarding without requiring a round trip to the controller. If no buckets are live, packets are dropped. This group type must implement a *liveness mechanism* (see 6.5).

5.7 Meter Table

A meter table consists of meter entries, defining per-flow meters. Per-flow meters enable OpenFlow to implement various simple QoS operations, such as rate-limiting, and can be combined with per-port queues (see 5.12) to implement complex QoS frameworks, such as DiffServ.

A meter measures the rate of packets assigned to it and enables controlling the rate of those packets. Meters are attached directly to flow entries (as opposed to queues which are attached to ports). Any flow entry can specify a meter in its instruction set (see 5.9): the meter measures and controls the rate of the aggregate of all flow entries to which it is attached. Multiple meters can be used in the same table, but in an exclusive way (disjoint set of flow entries). Multiple meters can be used on the same set of packets by using them in successive flow tables.

Meter Identifier	Meter Bands	Counters

Table˜3: Main components of a meter entry in the meter table.

Each meter entry (see Table 3) is identified by its meter identifier and contains:

- **meter identifier**: a 32 bit unsigned integer uniquely identifying the meter
- **meter bands**: an unordered list of meter bands, where each meter band specifies the rate of the band and the way to process the packet
- **counters**: updated when packets are processed by a meter

5.7.1 Meter Bands

Each meter may have one or more meter bands. Each band specifies the rate at which the band applies and the way packets should be processed. Packets are processed by a single meter band based on the current measured meter rate. The meter applies the meter band with the highest configured rate that is lower than the current measured rate. If the current rate is lower than any specified meter band rate, no meter band is applied.

Band Type	Rate	Counters	Type specific arguments

Table˜4: Main components of a meter band in a meter entry.

Each meter band (see Table 4) is identified by its rate and contains:

- **band type**: defines how packet are processed
- **rate**: used by the meter to select the meter band, defines the lowest rate at which the band can apply
- **counters**: updated when packets are processed by a meter band
- **type specific arguments**: some band types have optional arguments

There is no band type *"Required"* by this specification. The controller can query the switch about which of the *"Optional"* meter band types it supports.

- *Optional:* **drop**: drop (discard) the packet. Can be used to define a rate limiter band.
- *Optional:* **dscp remark**: increase the drop precedence of the DSCP field in the IP header of the packet. Can be used to define a simple DiffServ policer.

5.8 Counters

Counters are maintained for each flow table, flow entry, port, queue, group, group bucket, meter and meter band. OpenFlow-compliant counters may be implemented in software and maintained by polling hardware counters with more limited ranges. Table 5 contains the set of counters defined by the OpenFlow specification. A switch is not required to support all counters, just those marked *"Required"* in Table 5.

Duration refers to the amount of time the flow entry, a port, a group, a queue or a meter has been installed in the switch, and must be tracked with second precision. The Receive Errors field is the total of all receive and collision errors defined in Table 5, as well as any others not called out in the table.

Counters are unsigned and wrap around with no overflow indicator. If a specific numeric counter is not available in the switch, its value must be set to the maximum field value (the unsigned equivalent of -1).

Counter	**Bits**	
Per Flow Table		
Reference Count (active entries)	32	*Required*
Packet Lookups	64	*Optional*
Packet Matches	64	*Optional*
Per Flow Entry		
Received Packets	64	*Optional*
Received Bytes	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Port		
Received Packets	64	*Required*
Transmitted Packets	64	*Required*
Received Bytes	64	*Optional*
Transmitted Bytes	64	*Optional*
Receive Drops	64	*Optional*
Transmit Drops	64	*Optional*
Receive Errors	64	*Optional*
Transmit Errors	64	*Optional*
Receive Frame Alignment Errors	64	*Optional*
Receive Overrun Errors	64	*Optional*
Receive CRC Errors	64	*Optional*
Collisions	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Queue		
Transmit Packets	64	*Required*
Transmit Bytes	64	*Optional*
Transmit Overrun Errors	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group		
Reference Count (flow entries)	32	*Optional*
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group Bucket		
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Per Meter		
Flow Count	32	*Optional*
Input Packet Count	64	*Optional*
Input Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Meter Band		
In Band Packet Count	64	*Optional*
In Band Byte Count	64	*Optional*

Table˜5: List of counters.

5.9 Instructions

Each flow entry contains a set of instructions that are executed when a packet matches the entry. These instructions result in changes to the packet, action set and/or pipeline processing.

A switch is not required to support all instruction types, just those marked *"Required Instruction"* below. The controller can also query the switch about which of the *"Optional Instruction"* types it supports.

- *Optional Instruction:* **Meter *meter_id***: Direct packet to the specified meter. As the result of the metering, the packet may be dropped (depending on meter configuration and state).
- *Optional Instruction:* **Apply-Actions *action(s)***: Applies the specific action(s) immediately, without any change to the Action Set. This instruction may be used to modify the packet between two tables or to execute multiple actions of the same type. The actions are specified as an action list (see 5.11).
- *Optional Instruction:* **Clear-Actions**: Clears all the actions in the action set immediately.
- *Required Instruction:* **Write-Actions *action(s)***: Merges the specified action(s) into the current action set (see 5.10). If an action of the given type exists in the current set, overwrite it, otherwise add it.
- *Optional Instruction:* **Write-Metadata *metadata / mask***: Writes the masked metadata value into the metadata field. The mask specifies which bits of the metadata register should be modified (i.e. new_metadata = old_metadata & ˜mask | value & mask).
- *Required Instruction:* **Goto-Table *next-table-id***: Indicates the next table in the processing pipeline. The table-id must be greater than the current table-id. The flow entries of the last table of the pipeline can not include this instruction (see 5.1). OpenFlow switches with only a single flow table are not required to implement this instruction.

The instruction set associated with a flow entry contains a maximum of one instruction of each type. The instructions of the set execute in the order specified by this above list. In practice, the only constraints are that the *Meter* instruction is executed before the *Apply-Actions* instruction, that the *Clear-Actions* instruction is executed before the *Write-Actions* instruction, and that *Goto-Table* is executed last.

A switch must reject a flow entry if it is unable to execute the instructions associated with the flow entry. In this case, the switch must return an unsupported flow error (see 6.4). Flow tables may not support every match, every instruction or every action.

5.10 Action Set

An action set is associated with each packet. This set is empty by default. A flow entry can modify the action set using a *Write-Action* instruction or a *Clear-Action* instruction associated with a particular match. The action set is carried between flow tables. When the instruction set of a flow entry does not contain a *Goto-Table* instruction, pipeline processing stops and the actions in the action set of the packet are executed.

An action set contains a maximum of one action of each type. The *set-field* actions are identified by their field types, therefore the action set contains a maximum of one *set-field* action for each field type (i.e. multiple fields can be set). When multiple actions of the same type are required, e.g. pushing multiple MPLS labels or popping multiple MPLS labels, the *Apply-Actions* instruction should be used (see 5.11).

The actions in an action set are applied in the order specified below, regardless of the order that they were added to the set. If an action set contains a group action, the actions in the appropriate action bucket of the group are also applied in the order specified below. The switch may support arbitrary action execution order through the action list of the *Apply-Actions* instruction.

1. **copy TTL inwards**: apply copy TTL inward actions to the packet
2. **pop**: apply all tag pop actions to the packet
3. **push-MPLS**: apply MPLS tag push action to the packet
4. **push-PBB**: apply PBB tag push action to the packet
5. **push-VLAN**: apply VLAN tag push action to the packet
6. **copy TTL outwards**: apply copy TTL outwards action to the packet
7. **decrement TTL**: apply decrement TTL action to the packet
8. **set**: apply all set-field actions to the packet
9. **qos**: apply all QoS actions, such as set_queue to the packet
10. **group**: if a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list
11. **output**: if no group action is specified, forward the packet on the port specified by the output action

The output action in the action set is executed last. If both an output action and a group action are specified in an action set, the output action is ignored and the group action takes precedence. If no output action and no group action were specified in an action set, the packet is dropped. The execution of groups is recursive if the switch supports it; a group bucket may specify another group, in which case the execution of actions traverses all the groups specified by the group configuration.

5.11 Action List

The *Apply-Actions* instruction and the *Packet-out* message include an action list. The semantics of the action list is identical to the OpenFlow 1.0 specification. The actions of an action list are executed in the order specified by the list, and are applied immediately to the packet.

The execution of an action list starts with the first action in the list and each action is executed on the packet in sequence. The effect of those actions is cumulative, if the action list contains two Push VLAN actions, two VLAN headers are added to the packet. If the action list contains an output action, a copy of the packet is forwarded in its current state to the desired port. If the list contains group actions, a copy of the packet in its current state is processed by the relevant group buckets.

After the execution of the action list in an *Apply-Actions* instruction, pipeline execution continues on the modified packet (see 5.1). The action set of the packet is unchanged by the execution of the action list.

5.12 Actions

A switch is not required to support all action types, just those marked *"Required Action"* below. The controller can also query the switch about which of the *"Optional Action"* it supports.

Required Action: **Output**. The Output action forwards a packet to a specified OpenFlow port (see 4.1). OpenFlow switches must support forwarding to physical ports, switch-defined logical ports and the required reserved ports (see 4.5).

Optional Action: **Set-Queue**. The set-queue action sets the queue id for a packet. When the packet is forwarded to a port using the output action, the queue id determines which queue attached to this port is used for scheduling and forwarding the packet. Forwarding behavior is dictated by the configuration of the queue and is used to provide basic Quality-of-Service (QoS) support (see section 7.2.2).

Required Action: **Drop**. There is no explicit action to represent drops. Instead, packets whose action sets have no output actions should be dropped. This result could come from empty instruction sets or empty action buckets in the processing pipeline, or after executing a Clear-Actions instruction.

Required Action: **Group**. Process the packet through the specified group. The exact interpretation depends on group type.

Optional Action: **Push-Tag/Pop-Tag**. Switches may support the ability to push/pop tags as shown in Table 6. To aid integration with existing networks, we suggest that the ability to push/pop VLAN tags be supported.

Newly pushed tags should *always* be inserted as the outermost tag in the outermost valid location for that tag. When a new VLAN tag is pushed, it should be the outermost tag inserted, immediately after the Ethernet header and before other tags. Likewise, when a new MPLS tag is pushed, it should be the outermost tag inserted, immediately after the Ethernet header and before other tags.

When multiple push actions are added to the action set of the packet, they apply to the packet in the order defined by the action set rules, first MPLS, then PBB, than VLAN (se 5.10). When multiple push actions are included in an action list, they apply to the packet in the list order (see 5.11).

Note: Refer to section 5.12.1 for information on default field values.

Action	Associated Data	Description
Push VLAN header	Ethertype	Push a new VLAN header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8100 and 0x88a8 should be used.
Pop VLAN header	-	Pop the outer-most VLAN header from the packet.
Push MPLS header	Ethertype	Push a new MPLS shim header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8847 and 0x8848 should be used.
Table 6 – Continued on next page		

Table 6 – concluded from previous page

Action	Associated Data	Description
Pop MPLS header	Ethertype	Pop the outer-most MPLS tag or shim header from the packet. The Ethertype is used as the Ethertype for the resulting packet (Ethertype for the MPLS payload).
Push PBB header	Ethertype	Push a new PBB service instance header (I-TAG TCI) onto the packet (see 7.2.5). The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x88E7 should be used.
Pop PBB header	-	Pop the outer-most PBB service instance header (I-TAG TCI) from the packet (see 7.2.5).

Table˜6: Push/pop tag actions.

Optional Action: **Set-Field**. The various Set-Field actions are identified by their field type and modify the values of respective header fields in the packet. While not strictly required, the support of rewriting various header fields using Set-Field actions greatly increase the usefulness of an OpenFlow implementation. To aid integration with existing networks, we suggest that VLAN modification actions be supported. Set-Field actions should *always* be applied to the outermost-possible header (e.g. a "Set VLAN ID" action always sets the ID of the outermost VLAN tag), unless the field type specifies otherwise.

Optional Action: **Change-TTL**. The various Change-TTL actions modify the values of the IPv4 TTL, IPv6 Hop Limit or MPLS TTL in the packet. While not strictly required, the actions shown in Table 7 greatly increase the usefulness of an OpenFlow implementation for implementing routing functions. Change-TTL actions should *always* be applied to the outermost-possible header.

Action	Associated Data	Description
Set MPLS TTL	8 bits: New MPLS TTL	Replace the existing MPLS TTL. Only applies to packets with an existing MPLS shim header.
Decrement MPLS TTL	-	Decrement the MPLS TTL. Only applies to packets with an existing MPLS shim header.
Set IP TTL	8 bits: New IP TTL	Replace the existing IPv4 TTL or IPv6 Hop Limit and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Decrement IP TTL	-	Decrement the IPv4 TTL or IPv6 Hop Limit field and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Copy TTL outwards	-	Copy the TTL from next-to-outermost to outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or IP-to-MPLS.
Copy TTL inwards	-	Copy the TTL from outermost to next-to-outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or MPLS-to-IP.

Table˜7: Change-TTL actions.

The OpenFlow switch checks for packets with invalid IP TTL or MPLS TTL and rejects them. Checking for invalid TTL does not need to be done for every packet, but it must be done at a minimum every time a *decrement TTL* action is applied to a packet. The asynchronous configuration of the switch may

be changed (see 6.1.1) to send packets with invalid TTL to the controller over the control channel via a packet-in message (see 6.1.2).

5.12.1 Default values for fields on push

Field values for all fields specified in Table 8 should be copied from existing outer headers to new outer headers when executing a push action. New fields listed in Table 8 without corresponding existing fields should be set to zero. Fields that cannot be modified via OpenFlow set-field actions should be initialized to appropriate protocol values.

New Fields		**Existing Field(s)**
VLAN ID	←	VLAN ID
VLAN priority	←	VLAN priority
MPLS label	←	MPLS label
MPLS traffic class	←	MPLS traffic class
MPLS TTL	←	{ MPLS TTL IP TTL
PBB I-SID	←	PBB I-SID
PBB I-PCP	←	VLAN PCP
PBB C-DA	←	ETH DST
PBB C-SA	←	ETH SRC

Table˜8: Existing fields that may be copied into new fields on a push action.

Fields in new headers may be overridden by specifying a "set" action for the appropriate field(s) after the push operation.

6 OpenFlow Channel

The OpenFlow channel is the interface that connects each OpenFlow switch to a controller. Through this interface, the controller configures and manages the switch, receives events from the switch, and sends packets out the switch.

Between the datapath and the OpenFlow channel, the interface is implementation-specific, however all OpenFlow channel messages must be formatted according to the OpenFlow protocol. The OpenFlow channel is usually encrypted using TLS, but may be run directly over TCP.

6.1 OpenFlow Protocol Overview

The OpenFlow protocol supports three message types, *controller-to-switch*, *asynchronous*, and *symmetric*, each with multiple sub-types. Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch. Asynchronous messages are initiated by the switch and used to update the controller of network events and changes to the switch state. Symmetric messages are initiated by either the switch or the controller and sent without solicitation. The message types used by OpenFlow are described below.

6.1.1 Controller-to-Switch

Controller/switch messages are initiated by the controller and may or may not require a response from the switch.

Features: The controller may request the identity and the basic capabilities of a switch by sending a features request; the switch must respond with a features reply that specifies the identity and basic capabilities of the switch. This is commonly performed upon establishment of the OpenFlow channel.

Configuration: The controller is able to set and query configuration parameters in the switch. The switch only responds to a query from the controller.

Modify-State: Modify-State messages are sent by the controller to manage state on the switches. Their primary purpose is to add, delete and modify flow/group entries in the OpenFlow tables and to set switch port properties.

Read-State: Read-State messages are used by the controller to collect various information from the switch, such as current configuration, statistics and capabilities.

Packet-out: These are used by the controller to send packets out of a specified port on the switch, and to forward packets received via Packet-in messages. Packet-out messages must contain a full packet or a buffer ID referencing a packet stored in the switch. The message must also contain a list of actions to be applied in the order they are specified; an empty action list drops the packet.

Barrier: Barrier request/reply messages are used by the controller to ensure message dependencies have been met or to receive notifications for completed operations.

Role-Request: Role-Request messages are used by the controller to set the role of its OpenFlow channel, or query that role. This is mostly useful when the switch connects to multiple controllers (see 6.3.4).

Asynchronous-Configuration: The Asynchronous-Configuration message are used by the controller to set an additional filter on the asynchronous messages that it wants to receive on its OpenFlow channel, or to query that filter. This is mostly useful when the switch connects to multiple controllers (see 6.3.4) and commonly performed upon establishment of the OpenFlow channel.

6.1.2 Asynchronous

Asynchronous messages are sent without a controller soliciting them from a switch. Switches send asynchronous messages to controllers to denote a packet arrival, switch state change, or error. The four main asynchronous message types are described below.

Packet-in: Transfer the control of a packet to the controller. For all packets forwarded to the **CONTROLLER** reserved port using a flow entry or the table-miss flow entry, a packet-in event is always sent to controllers (see 5.12). Other processing, such as TTL checking, may also generate packet-in events to send packets to the controller.

Packet-in events can be configured to buffer packets. For packet-in generated by an output action in a flow entries or group bucket, it can be specified individually in the output action itself (see 7.2.5), for other packet-in it can be configured in the switch configuration (see 7.3.2). If the packet-in event is configured to buffer packets and the switch has sufficient memory to buffer them, the packet-in events

contain only some fraction of the packet header and a buffer ID to be used by a controller when it is ready for the switch to forward the packet. Switches that do not support internal buffering, are configured to not buffer packets for the packet-in event, or have run out of internal buffering, must send the full packet to controllers as part of the event. Buffered packets will usually be processed via a **Packet-out** message from a controller, or automatically expired after some time.

If the packet is buffered, the number of bytes of the original packet to include in the packet-in can be configured. By default, it is 128 bytes. For packet-in generated by an output action in a flow entries or group bucket, it can be specified individually in the output action itself (see 7.2.5), for other packet-in it can be configured in the switch configuration (see 7.3.2).

Flow-Removed: Inform the controller about the removal of a flow entry from a flow table. Flow-Removed messages are only sent for flow entries with the `OFPFF_SEND_FLOW_REM` flag set. They are generated as the result of a controller flow delete requests or the switch flow expiry process when one of the flow timeout is exceeded (see 5.5).

Port-status: Inform the controller of a change on a port. The switch is expected to send port-status messages to controllers as port configuration or port state changes. These events include change in port configuration events, for example if it was brought down directly by a user, and port state change events, for example if the link went down.

Error: The switch is able to notify controllers of problems using error messages.

6.1.3 Symmetric

Symmetric messages are sent without solicitation, in either direction.

Hello: Hello messages are exchanged between the switch and controller upon connection startup.

Echo: Echo request/reply messages can be sent from either the switch or the controller, and must return an echo reply. They are mainly used to verify the liveness of a controller-switch connection, and may as well be used to measure its latency or bandwidth.

Experimenter: Experimenter messages provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space. This is a staging area for features meant for future OpenFlow revisions.

6.2 Message Handling

The OpenFlow protocol provides reliable message delivery and processing, but does *not* automatically provide acknowledgements or ensure ordered message processing. The OpenFlow message handling behaviour described in this section is provided on the main connection and auxiliary connections using reliable transport, however it is not supported on auxiliary connections using unreliable transport (see 6.3.5).

Message Delivery: Messages are guaranteed delivery, unless the OpenFlow channel fails entirely, in which case the controller should not assume anything about the switch state (e.g., the switch may have gone into “fail standalone mode”).

Message Processing: Switches must process every message received from a controller in full, possibly generating a reply. If a switch cannot completely process a message received from a controller, it must send back an error message. For packet-out messages, fully processing the message does not guarantee that the included packet actually exits the switch. The included packet may be silently dropped after OpenFlow processing due to congestion at the switch, QoS policy, or if sent to a blocked or invalid port.

In addition, switches must send to the controller all asynchronous messages generated by OpenFlow state changes, such as flow-removed, port-status or packet-in messages, so that the controller view of the switch is consistent with its actual state. Those messages may get filtered out based on the *Asynchronous Configuration* (see 6.1.1). Moreover, conditions that would trigger an OpenFlow state change may get filtered prior to causing such change. For example, packets received on data ports that should be forwarded to the controller may get dropped due to congestion or QoS policy within the switch and generate no packet-in messages. These drops may occur for packets with an explicit output action to the controller. These drops may also occur when a packet fails to match any entries in a table and that table's default action is to send to the controller. The policing of packets destined to the controller using QoS actions or rate limiting is advised, to prevent denial of service of the controller connection, and is outside the scope of the present specification.

Controllers are free to ignore messages they receive, but must respond to echo messages to prevent the switch from terminating the connection.

Message Ordering: Ordering can be ensured through the use of *barrier* messages. In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance; hence, controllers should not depend on a specific processing order. In particular, flow entries may be inserted in tables in an order different than that of flow mod messages received by the switch. Messages must not be reordered across a barrier message and the barrier message must be processed only when all prior messages have been processed. More precisely:

1. messages before a barrier must be fully processed before the barrier, including sending any resulting replies or errors
2. the barrier must then be processed and a barrier reply sent
3. messages after the barrier may then begin processing

If two messages from the controller depend on each other, they must be separated by a barrier message. Examples of such message dependancies include a group mod add with a flow mod add referencing the group, a port mod with a packet-out forwarding to the port, or a flow mod add with a following packet-out to `OFPP_TABLE`.

6.3 OpenFlow Channel Connections

The OpenFlow channel is used to exchange OpenFlow message between an OpenFlow switch and an OpenFlow controller. A typical OpenFlow controller manages multiple OpenFlow channels, each one to a different OpenFlow switch. An OpenFlow switch may have one OpenFlow channel to a single controller, or multiple channels for reliability, each to a different controller (see 6.3.4).

An OpenFlow controller typically manages an OpenFlow switch remotely over one or more networks. The specification of the networks used for the OpenFlow channels is outside the scope of the present

specification. It may be a separate dedicated network, or the OpenFlow channel may use the network managed by the OpenFlow switch (in-band controller connection). The only requirement is that it should provide TCP/IP connectivity.

The OpenFlow channel is usually instantiated as a single network connection between the switch and the controller, using TLS or plain TCP (see 6.3.3). Alternatively, the OpenFlow channel may be composed of multiple network connections to exploit parallelism (see 6.3.5). The OpenFlow switch must be able to create an OpenFlow channel by initiating a connection to an OpenFlow controller (see 6.3.1). Some switch implementations may optionally allow an OpenFlow controller to connect to the OpenFlow switch, in this case the switch usually should restrict itself to secured connections (see 6.3.3) to prevent unauthorised connections.

6.3.1 Connection Setup

The switch must be able to establish communication with a controller at a user-configurable (but otherwise fixed) IP address, using either a user-specified transport port or the default transport port (see 6.3.3). If the switch is configured with the IP address of the controller to connect to, the switch initiates a standard TLS or TCP connection to the controller. Traffic to and from the OpenFlow channel is not run through the OpenFlow pipeline. Therefore, the switch must identify incoming traffic as local before checking it against the flow tables.

When an OpenFlow connection is first established, each side of the connection must immediately send an `OFPT_HELLO` message with the `version` field set to the highest OpenFlow protocol version supported by the sender (see 7.1). This Hello message may optionally include some OpenFlow elements to help connection setup (see 7.5.1). Upon receipt of this message, the recipient must calculate the OpenFlow protocol version to be used. If both the Hello message sent and the Hello message received contained a `OFPHET_VERSIONBITMAP` hello element, and if those bitmaps have some common bits set, the negotiated version must be the highest version set in both bitmaps. Otherwise, the negotiated version must be the smaller of the version number that was sent and the one that was received in the `version` fields.

If the negotiated version is supported by the recipient, then the connection proceeds. Otherwise, the recipient must reply with an `OFPT_ERROR` message with a `type` field of `OFPET_HELLO_FAILED`, a `code` field of `OFPHFC_INCOMPATIBLE`, and optionally an ASCII string explaining the situation in `data`, and then terminate the connection.

After the switch and the controller have exchanged `OFPT_HELLO` messages and successfully negotiated a common version number, the connection setup is done and standard OpenFlow messages can be exchanged over the connection. One of the first thing that the controller should do is to send a `OFPT_FEATURES_REQUEST` message to get the *Datapath ID* of the switch (see 7.3.1).

6.3.2 Connection Interruption

In the case that a switch loses contact with all controllers, as a result of echo request timeouts, TLS session timeouts, or other disconnections, the switch must *immediately* enter either "fail secure mode" or "fail standalone mode", depending upon the switch implementation and configuration. In "fail secure mode", the only change to switch behavior is that packets and messages destined to the controllers are dropped. Flow entries should continue to expire according to their timeouts in "fail secure mode". In

"fail standalone mode", the switch processes all packets using the `OFPP_NORMAL` reserved port; in other words, the switch acts as a legacy Ethernet switch or router. The "fail standalone mode" is usually only available on Hybrid switches (see 5.1).

Upon connecting to a controller again, the existing flow entries remain. The controller then has the option of deleting all flow entries, if desired.

The first time a switch starts up, it will operate in either "fail secure mode" or "fail standalone mode" mode, until it successfully connects to a controller. Configuration of the default set of flow entries to be used at startup is outside the scope of the OpenFlow protocol.

6.3.3 Encryption

The switch and controller may communicate through a TLS connection. The TLS connection is initiated by the switch on startup to the controller, which is located by default on TCP port 6633 . The switch and controller mutually authenticate by exchanging certificates signed by a site-specific private key. Each switch must be user-configurable with one certificate for authenticating the controller (controller certificate) and the other for authenticating to the controller (switch certificate).

The switch and controller may optionally communicate using plain TCP. The TCP connection is initiated by the switch on startup to the controller, which is located by default on TCP port 6633 . When using plain TCP, it is recommended to use alternative security measures to prevent eavesdropping, controller impersonation or other attacks on the OpenFlow channel.

6.3.4 Multiple Controllers

The switch may establish communication with a single controller, or may establish communication with multiple controllers. Having multiple controllers improves reliability, as the switch can continue to operate in OpenFlow mode if one controller or controller connection fails. The hand-over between controllers is entirely managed by the controllers themselves, which enables fast recovery from failure and also controller load balancing. The controllers coordinate the management of the switch amongst themselves via mechanisms outside the scope of the present specification, and the goal of the multiple controller functionality is only to help synchronise controller handoffs performed by the controllers. The multiple controller functionality only addresses controller fail-over and load balancing, and doesn't address virtualisation which can be done outside the OpenFlow protocol.

When OpenFlow operation is initiated, the switch must connect to all controllers it is configured with, and try to maintain connectivity with all of them concurrently. Many controllers may send controller-to-switch commands to the switch, the reply or error messages related to those commands must only be sent on the controller connection associated with that command. Asynchronous messages may need to be sent to multiple controllers, the message is duplicated for each eligible OpenFlow channel and each message sent when the respective controller connection allows it.

The default role of a controller is `OFPCR_ROLE_EQUAL`. In this role, the controller has full access to the switch and is equal to other controllers in the same role. By default, the controller receives all the switch asynchronous messages (such as packet-in, flow-removed). The controller can send controller-to-switch commands to modify the state of the switch. The switch does not do any arbitration or resource sharing between controllers.

A controller can request its role to be changed to OFPCR_ROLE_SLAVE. In this role, the controller has read-only access to the switch. By default, the controller does not receive switch asynchronous messages, apart from Port-status messages. The controller is denied the ability to execute all controller-to-switch commands that send packets or modify the state of the switch. For example, OFPT_PACKET_OUT, OFPT_FLOW_MOD, OFPT_GROUP_MOD, OFPT_PORT_MOD, OFPT_TABLE_MOD requests, and OFPMP_TABLE_FEATURES multiparts requests with a non-empty body must be rejected. If the controller sends one of those commands, the switch must reply with an OFPT_ERROR message with a type field of OFPET_BAD_REQUEST, a code field of OFPBRC_IS_SLAVE. Other controller-to-switch messages, such as OFPT_ROLE_REQUEST, OFPT_SET_ASYNC and OFPT_MULTIPART_REQUEST that only query data, should be processed normally.

A controller can request its role to be changed to OFPCR_ROLE_MASTER. This role is similar to OFPCR_ROLE_EQUAL and has full access to the switch, the difference is that the switch ensures it is the only controller in this role. When a controller changes its role to OFPCR_ROLE_MASTER, the switch changes all other controllers with the role OFPCR_ROLE_MASTER to have the role OFPCR_ROLE_SLAVE, but does not affect controllers with role OFPCR_ROLE_EQUAL. When the switch performs such role changes, no message is generated to the controller whose role is changed (in most cases that controller is no longer reachable).

Each controller may send a OFPT_ROLE_REQUEST message to communicate its role to the switch (see 7.3.9), and the switch must remember the role of each controller connection. A controller may change role at any time, provided the generation_id in the message is current (see below).

The role request message offers a lightweight mechanism to help the controller master election process, the controllers configure their role and usually still need to coordinate among themselve. The switch can not change the state of a controller on its own, controller state is always changed as a result of a request from one of the controllers. Any Slave controller or Equal controller can elect itself Master. A switch may be *simultaneously* connected to multiple controllers in Equal state, multiple controllers in Slave state, and at most one controller in Master state. The controller in Master state (if any) and all the controllers in Equal state can fully change the switch state, there is no mechanism to enforce partitioning of the switch between those controllers. If the controller in Master role need to be the only controller able to make changes on the switch, then no controllers should be in Equal state and all other controllers should be in Slave state.

A controller can also control which types of switch asynchronous messages are sent over its OpenFlow channel, and change the defaults described above. This is done via a *Asynchronous Configuration* message (see 6.1.1), listing all reasons for each message type that need to be enabled or filtered out (see 7.3.10) for the specific OpenFlow channel. Using this feature, different controllers can receive different notifications, a controller in master mode can selectively disable notifications it does not care about, and a controller in slave mode can enable notifications it wants to monitor.

To detect out-of-order messages during a master/slave transition, the OFPT_ROLE_REQUEST message contains a 64-bit sequence number field, generation_id, that identifies a given mastership view. As a part of the master election mechanism, controllers (or a third party on their behalf) coordinate the assignment of generation_id. generation_id is a monotonically increasing counter: a new (larger) generation_id is assigned each time the mastership view changes, e.g. when a new master is designated. generation_id can wrap around.

On receiving a OFPT_ROLE_REQUEST with role equal to OFPCR_ROLE_MASTER or OFPCR_ROLE_SLAVE the switch must compare the generation_id in the message against the largest generation id seen so far.

A message with a generation_id smaller than a previously seen generation id must be considered stale and discarded. The switch must respond to stale messages with an error message with type OFPET_ROLE_REQUEST_FAILED and code OFPRRFC_STALE.

The following pseudo-code describes the behavior of the switch in dealing with generation_id.

On switch startup:

```
generation_is_defined = false;
```

On receiving OFPT_ROLE_REQUEST with role equal to OFPCR_ROLE_MASTER or OFPCR_ROLE_SLAVE and with a given generation_id, say GEN_ID_X:

```
if (generation_is_defined AND
    distance(GEN_ID_X, cached_generation_id) < 0) {
  <discard OFPT_ROLE_REQUEST message>;
  <send an error message with code OFPRRFC_STALE>;
} else {
  cached_generation_id = GEN_ID_X;
  generation_is_defined = true;
  <process the message normally>;
}
```

where distance() is the *Wrapping Sequence Number Distance* operator defined as following:

```
distance(a, b) := (int64_t)(a - b)
```

I.e. distance() is the unsigned difference between the sequence numbers, interpreted as a two's complement signed value. This results in a positive distance if a is greater than b (in a circular sense) but less than "half the sequence number space" away from it. It results in a negative distance otherwise (a < b).

The switch must ignore generation_id if the role in the OFPT_ROLE_REQUEST is OFPCR_ROLE_EQUAL, as generation_id is specifically intended for the disambiguation of race condition in master/slave transition.

6.3.5 Auxiliary Connections

By default, the *OpenFlow channel* between an OpenFlow switch and an OpenFlow controller is a single network connection. The OpenFlow channel may also be composed of a **main connection** and multiple **auxiliary connections**. Auxiliary connections are created by the OpenFlow switch and are helpful to improve the switch processing performance and exploit the parallelism of most switch implementations.

Each connection from the switch to the controller is identified by the switch *Datapath ID* and a *Auxiliary ID* (see 7.3.1). The main connection must have its Auxiliary ID set to zero, whereas auxiliary connection must have a non-zero Auxiliary ID and the same Datapath ID. Auxiliary connections must use the same source IP address as the main connection, but can use a different transport, for example TLS, TCP,

DTLS or UDP, depending on the switch configuration. The auxiliary connection should have the same destination IP address and same transport destination port as the main connection, unless the switch configuration specifies otherwise. The controller must recognise incoming connections with non-zero Auxiliary ID as auxiliary connections and bind them to the main connection with the same Datapath ID.

The switch must not initiate auxiliary connection before having completed the connection setup over the main connection (see 6.3.1), it must setup and maintain auxiliary connections with the controller only while the corresponding main connection is alive. The connection setup for auxiliary connections is the same as for the main connection (see 6.3.1). If the switch detects that the main connection to a controller is broken, it must immediately close all its auxiliary connections to that controller, to enable the controller to properly resolve Datapath ID conflicts.

Both the OpenFlow switch and the OpenFlow controller must accept any OpenFlow message types and sub-types on all connections : the main connection or an auxiliary connection can not be restricted to a specific message type or sub-type. However, the processing performance of different message types or sub-types on different connections may be different. The switch may service auxiliary connections with different priorities, for example one auxiliary connection may be dedicated to high priority requests and always processed by the switch before other auxiliary connections. A switch configuration, for example using the OpenFlow Configuration Protocol, may optionally configure the priority of auxiliary connections.

A reply to an OpenFlow request must be made on the same connection it came in. There is no synchronisation between connections, and messages sent on different connections may be processed in any order. A barrier message applies only to the connection where it is used (see 6.2). Auxiliary connections using DTLS or UDP may lose or reorder messages, OpenFlow does not provide ordering or delivery guarantees on those connections (see 6.2). If messages must be processed in sequence, they must be sent over the same connection, use a connection that does not reorder packets, and use barrier messages.

The controller is free to use the various switch connections for sending OpenFlow messages at its entire discretion, however to maximise performance on most switches the following guidelines are suggested:

- All OpenFlow controller messages which are not Packet-out (flow-mod, statistic request...) should be sent over the main connection.
- All Packet-Out messages containing a packet from a Packet-In message should be sent on the connection where the Packet-In came from.
- All other Packet-Out messages should be spread across the various auxiliary connections using a mechanism keeping the packets of a same flow mapped to the same connection.
- If the desired auxiliary connection is not available, the controller should use the main connection.

The switch is free to use the various controller connections for sending OpenFlow messages as it wishes, however the following guidelines are suggested :

- All OpenFlow messages which are not Packet-in should be sent over the main connection.
- All Packet-In messages spread across the various auxiliary connection using a mechanism keeping the packets of a same flow mapped to the same connection.

Auxiliary connection on **unreliable transports** (UDP, DTLS) have additional restrictions and rules that don't apply to auxiliary connection on other transport (TCP, TLS). The only message types supported on unreliable auxiliary connections are `OFPT_HELLO`, `OFPT_ERROR`, `OFPT_ECHO_REQUEST`, `OFPT_ECHO_REPLY`, `OFPT_FEATURES_REQUEST`, `OFPT_FEATURES_REPLY`, `OFPT_PACKET_IN`, `OFPT_PACKET_OUT` and `OFPT_EXPERIMENTER`, other messages types are not supported by the specification.

On unreliable auxiliary connection, *Hello messages* are sent at connection initiation to setup the connection (see 6.3.1). If an OpenFlow device receives another message on an unreliable auxiliary connection prior to receiving a *Hello message*, the device must either assume the connection is setup properly and use the version number from that message, or it must return an Error message with `OFPET_BAD_REQUEST` type and `OFPBRC_BAD_VERSION` code. If a OpenFlow device receives a error message with `OFPET_BAD_REQUEST` type and `OFPBRC_BAD_VERSION` code on unreliable auxiliary connection, it must either send a new *Hello message* or terminate the unreliable auxiliary connection (the connection may be retried at a later time). If no message was ever received on an auxiliary connection after some implementation chosen amount of time lower than 5 seconds, the device must either send a new *Hello* message or terminate the unreliable auxiliary connection. If after sending a *Feature Request* message, the controller does not receives a *Feature Reply* message after some implementation chosen amount of time lower than 5 seconds, the device must either send a new *Feature Request* message or terminate the unreliable auxiliary connection. If after receiving a message, a device does not receives any other message after some implementation chosen amount of time lower than 30 seconds, the device must terminate the unreliable auxiliary connection. If a device receives a message for a unreliable auxiliary connection already terminated, it must assume it is a new connection.

OpenFlow devices using unreliable auxiliary connection should follow recommendations in RFC 5405 when possible.

6.4 Flow Table Modification Messages

Flow table modification messages can have the following types:

```
enum ofp_flow_mod_command {
    OFPFC_ADD           = 0, /* New flow. */
    OFPFC_MODIFY        = 1, /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT = 2, /* Modify entry strictly matching wildcards and
                                priority. */
    OFPFC_DELETE        = 3, /* Delete all matching flows. */
    OFPFC_DELETE_STRICT = 4, /* Delete entry strictly matching wildcards and
                                priority. */
};
```

For **add** requests (`OFPFC_ADD`) with the `OFPFF_CHECK_OVERLAP` flag set, the switch must first check for any overlapping flow entries in the requested table. Two flow entries overlap if a single packet may match both, and both entries have the same priority. If an overlap conflict exists between an existing flow entry and the **add** request, the switch must refuse the addition and respond with an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_OVERLAP` code.

For non-overlapping **add** requests, or those with no overlap checking, the switch must insert the flow entry in the requested table. If a flow entry with identical match fields and priority already resides in

the requested table, then that entry, including its duration, must be cleared from the table, and the new flow entry added. If the OFPFF_RESET_COUNTS flag is set, the flow entry counters must be cleared, otherwise they should be copied from the replaced flow entry. No flow-removed message is generated for the flow entry eliminated as part of an **add** request; if the controller wants a flow-removed message it should explicitly send a **delete** request for the old flow entry prior to adding the new one.

For **modify** requests (OFPFC_MODIFY or OFPFC_MODIFY_STRICT), if a matching entry exists in the table, the instructions field of this entry is updated with the value from the request, whereas its cookie, idle_timeout, hard_timeout, flags, counters and duration fields are left unchanged. If the OFPFF_RESET_COUNTS flag is set, the flow entry counters must be cleared. For **modify** requests, if no flow entry currently residing in the requested table matches the request, no error is recorded, and no flow table modification occurs.

For **delete** requests (OFPFC_DELETE or OFPFC_DELETE_STRICT), if a matching entry exists in the table, it must be deleted, and if the entry has the OFPFF_SEND_FLOW_REM flag set, it should generate a flow removed message. For **delete** requests, if no flow entry currently residing in the requested table matches the request, no error is recorded, and no flow table modification occurs.

Modify and **delete** flow_mod commands have *non-strict* versions (OFPFC_MODIFY and OFPFC_DELETE) and *strict* versions (OFPFC_MODIFY_STRICT or OFPFC_DELETE_STRICT). In the *strict* versions, the set of match fields, all match fields, including their masks, and the priority, are strictly matched against the entry, and only an identical flow entry is modified or removed. For example, if a message to remove entries is sent that has no match fields included, the OFPFC_DELETE command would delete all flow entries from the tables, while the OFPFC_DELETE_STRICT command would only delete a flow entry that applies to all packets at the specified priority.

For *non-strict* **modify** and **delete** commands, all flow entries that match the flow_mod description are modified or removed. In the *non-strict* versions, a match will occur when a flow entry exactly matches or is more specific than the description in the flow_mod command; in the flow_mod the missing match fields are wildcarded, field masks are active, and other flow_mod fields such as priority are ignored. For example, if a OFPFC_DELETE command says to delete all flow entries with a destination port of 80, then a flow entry that wildcards all match fields will not be deleted. However, a OFPFC_DELETE command that wildcards all match fields will delete an entry that matches all port 80 traffic. This same interpretation of mixed wildcard and exact match fields also applies to individual and aggregate flows stats requests.

Delete commands can be optionally filtered by destination group or output port. If the out_port field contains a value other than OFPP_ANY, it introduces a constraint when matching. This constraint is that each matching flow entry must contain an *output* action directed at the specified port in the actions associated with that flow entry. This constraint is limited to only the actions directly associated with the flow entry. In other words, the switch must not recurse through the action sets of pointed-to groups, which may have matching *output* actions. The out_group, if different from OFPG_ANY, introduce a similar constraint on the *group* action. These fields are ignored by OFPFC_ADD, OFPFC_MODIFY and OFPFC_MODIFY_STRICT messages.

Modify and **delete** commands can also be filtered by cookie value, if the cookie_mask field contains a value other than 0. This constraint is that the bits specified by the cookie_mask in both the cookie field of the flow mod and a flow entry's cookie value must be equal. In other words, $(flow\_entry.cookie \,\&\, flow\_mod.cookie\_mask) == (flow\_mod.cookie \,\&\, flow\_mod.cookie\_mask)$.

Delete commands can use the OFPTT_ALL value for table-id to indicate that matching flow entries are to be deleted from all flow tables.

If the flow modification message specifies an invalid table-id, the switch must send an ofp_error_msg with OFPET_FLOW_MOD_FAILED type and OFPFMFC_BAD_TABLE_ID code. If the flow modification message specifies OFPTT_ALL for table-id in a **add** or **modify** request, the switch must send the same error message.

If a switch cannot find any space in the requested table in which to add the incoming flow entry, the switch must send an ofp_error_msg with OFPET_FLOW_MOD_FAILED type and OFPFMFC_TABLE_FULL code.

If the instructions requested in a flow mod message are unknown the switch must return an ofp_error_msg with OFPET_BAD_INSTRUCTION type and OFPBIC_UNKNOWN_INST code. If the instructions requested in a flow mod message are unsupported the switch must return an ofp_error_msg with OFPET_BAD_INSTRUCTION type and OFPBIC_UNSUP_INST code.

If the instructions requested contain a Goto-Table and the next-table-id refers to an invalid table the switch must return an ofp_error_msg with OFPET_BAD_INSTRUCTION type and OFPBIC_BAD_TABLE_ID code.

If the instructions requested contain a Write-Metadata and the metadata value or metadata mask value is unsupported then the switch must return an ofp_error_msg with OFPET_BAD_INSTRUCTION type and OFPBIC_UNSUP_METADATA or OFPBIC_UNSUP_METADATA_MASK code.

If the match in a flow mod message specifies a field that is unsupported in the table, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_FIELD code. If the match in a flow mod message specifies a field more than once, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_DUP_FIELD code. If the match in a flow mod message specifies a field but fail to specify its associated prerequisites, for example specifies an IPv4 address without matching the EtherType to 0x800, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_PREREQ code.

If the match in a flow mod specifies an arbitrary bitmask for either the datalink or network addresses which the switch cannot support, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and either OFPBMC_BAD_DL_ADDR_MASK or OFPBMC_BAD_NW_ADDR_MASK. If the bitmasks specified in *both* the datalink and network addresses are not supported then OFPBMC_BAD_DL_ADDR_MASK should be used. If the match in a flow mod specifies an arbitrary bitmask for another field which the switch cannot support, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_MASK code.

If the match in a flow mod specifies values that cannot be matched, for example, a VLAN ID greater than 4095 and not one of the reserved values, or a DSCP value with one of the two higher bits set, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_VALUE code.

If any action references a port that will never be valid on a switch, the switch must return an ofp_error_msg with OFPET_BAD_ACTION type and OFPBAC_BAD_OUT_PORT code. If the referenced port may be valid in the future, e.g. when a linecard is added to a chassis switch, or a port is dynamically added to a software switch, the switch must either silently drop packets sent to the referenced port, or immediately return an OFPBAC_BAD_OUT_PORT error and refuse the flow mod.

If an action in a flow mod message references a group that is not currently defined on the switch, or is a reserved group, such as OFPG_ALL, the switch must return an ofp_error_msg with OFPET_BAD_ACTION type and OFPBAC_BAD_OUT_GROUP code.

If an action in a flow mod message has a value that is invalid, for example a Set VLAN ID action with value greater than 4095, or a Push action with an invalid Ethertype, the switch must return an ofp_error_msg with OFPET_BAD_ACTION type and OFPBAC_BAD_ARGUMENT code.

If an action in a flow mod message performs an operation which is inconsistent with the match, for example, a pop VLAN action with a match specifying no VLAN, or a set IPv4 address action with a match wildcarding the Ethertype, the switch may optionally reject the flow mod and immediately return an ofp_error_msg with OFPET_BAD_ACTION type and OFPBAC_MATCH_INCONSISTENT code. The effect of any inconsistent actions on matched packets is undefined. Controllers are strongly encouraged to avoid generating combinations of table entries that may yield inconsistent actions.

If an action list contain a sequence of actions that the switch can not support in the specified order, the switch must return an ofp_error_msg with OFPET_BAD_ACTION type and OFPBAC_UNSUPPORTED_ORDER code.

If any other errors occur during the processing of the flow mod message, the switch may return an ofp_error_msg with OFPET_FLOW_MOD_FAILED type and OFPFMC_UNKNOWN code.

6.5 Group Table Modification Messages

Group table modification messages can have the following types:

```
/* Group commands */
enum ofp_group_mod_command {
    OFPGC_ADD    = 0,       /* New group. */
    OFPGC_MODIFY = 1,       /* Modify all matching groups. */
    OFPGC_DELETE = 2,       /* Delete all matching groups. */
};
```

Groups may consist of zero or more buckets. A group with no buckets will not alter the action set associated with a packet. A group may also include buckets which themselves forward to other groups if the switch supports it.

The action set for each bucket must be validated using the same rules as those for flow mods (Section 6.4), with additional group-specific checks. If an action in one of the buckets is invalid or unsupported, the switch should return an ofp_error_msg with OFPET_BAD_ACTION type and code corresponding to the error (see 6.4).

For **add** requests (OFPGC_ADD), if a group entry with the specified group identifier already resides in the group table, then the switch must refuse to add the group entry and must send an ofp_error_msg with OFPET_GROUP_MOD_FAILED type and OFPGMFC_GROUP_EXISTS code.

For **modify** requests (OFPGC_MODIFY), if a group entry with the specified group identifier already resides in the group table, then that entry, including its type and action buckets, must be removed, and the new group entry added. If a group entry with the specified group identifier does not already exist then the switch must refuse the group mod and send an ofp_error_msg with OFPET_GROUP_MOD_FAILED type and OFPGMFC_UNKNOWN_GROUP code.

If a specified group type is invalid (ie: includes fields such as `weight` that are undefined for the specified group type) then the switch must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_INVALID_GROUP` code.

If a switch does not support unequal load sharing with select groups (buckets with weight different than 1), it must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_WEIGHT_UNSUPPORTED` code.

If a switch cannot add the incoming group entry due to lack of space, the switch must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_OUT_OF_GROUPS` code.

If a switch cannot add the incoming group entry due to restrictions (hardware or otherwise) limiting the number of group buckets, it must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_OUT_OF_BUCKETS` code.

If a switch cannot add the incoming group because it does not support the proposed liveliness configuration, the switch must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_WATCH_UNSUPPORTED` code. This includes specifying `watch_port` or `watch_group` for a group that does not support liveness, or specifying a port that does not support liveness in `watch_port`, or specifying a group that does not support liveness in `watch_group`.

For **delete** requests (`OFPGC_DELETE`), if no group entry with the specified group identifier currently exists in the group table, no error is recorded, and no group table modification occurs. Otherwise, the group is removed, and all flow entries containing this group in a Group action are also removed. The group type need not be specified for the **delete** request. **Delete** also differs from an **add** or **modify** with no buckets specified in that future attempts to **add** the group identifier will not result in a group exists error. If one wishes to effectively delete a group yet leave in flow entries using it, that group can be cleared by sending a **modify** with no buckets specified.

To delete all groups with a single message, specify `OFPG_ALL` as the group value.

Groups may be *chained* if the switch supports it, when at least one group forward to another group, or in more complex configuration. For example, a fast reroute group may have two buckets, where each points to a select group. If a switch does not support groups of groups, it must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_CHAINING_UNSUPPORTED` code.

A switch may support checking that no loop is created while chaining groups : if a group mod is sent such that a forwarding loop would be created, the switch must reject the group mod and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_LOOP` code. If the switch does not support such checking, the forwarding behavior is undefined.

A switch may support checking that groups forwarded to by other groups are not removed : If a switch cannot delete a group because it is referenced by another group, it must refuse to delete the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_CHAINED_GROUP` code. If the switch does not support such checking, the forwarding behavior is undefined.

Fast failover group support requires *liveness monitoring*, to determine the specific bucket to execute. Other group types are not required to implement liveness monitoring, but may optionally implement it. If a switch cannot implement liveness checking for any bucket in a group, it must refuse the group mod and return an error. The rules for determining liveness include:

- A port is considered live if it has the OFPPS_LIVE flag set in its port state. Port liveness may be managed by code outside of the OpenFlow portion of a switch, defined outside of the OpenFlow specification, such as Spanning Tree or a KeepAlive mechanism. The port must not be considered live (and the OFPPS_LIVE flag must be unset) if one of the port liveness mechanisms enabled on the switch consider the port not live, or if the port config bit OFPPC_PORT_DOWN indicates the port is down, or if the port state bit OFPPS_LINK_DOWN indicates the link is down.
- A bucket is considered live if either watch_port is not OFPP_ANY and the port watched is live, or if watch_group is not OFPG_ANY and the group watched is live.
- A group is considered live if a least one of its buckets is live.

The controller can infer the liveness state of the group by monitoring the states of the various ports.

6.6 Meter Modification Messages

Meter modification messages can have the following types:

```
/* Meter commands */
enum ofp_meter_mod_command {
    OFPMC_ADD,              /* New meter. */
    OFPMC_MODIFY,           /* Modify specified meter. */
    OFPMC_DELETE,           /* Delete specified meter. */
};
```

For **add** requests (OFPMC_ADD), if a meter entry with the specified meter identifier already exist, then the switch must refuse to add the meter entry and must send an ofp_error_msg with OFPET_METER_MOD_FAILED type and OFPMMFC_METER_EXISTS code.

For **modify** requests (OFPMC_MODIFY), if a meter entry with the specified meter identifier already exists, then that entry, including its bands, must be removed, and the new meter entry added. If a meter entry with the specified meter identifier does not already exists then the switch must refuse the meter mod and send an ofp_error_msg with OFPET_METER_MOD_FAILED type and OFPMMFC_UNKNOWN_METER code.

If a switch cannot add the incoming meter entry due to lack of space, the switch must send an ofp_error_msg with OFPET_METER_MOD_FAILED type and OFPMMFC_OUT_OF_METERS code.

If a switch cannot add the incoming meter entry due to restrictions (hardware or otherwise) limiting the number of bands, it must refuse to add the meter entry and must send an ofp_error_msg with OFPET_METER_MOD_FAILED type and OFPMMFC_OUT_OF_BANDS code.

For **delete** requests (OFPMC_DELETE), if no meter entry with the specified meter identifier currently exists, no error is recorded, and no meter modification occurs. Otherwise, the meter is removed, and all flows that include the meter in their instruction set are also removed. Only the meter identifier need to be specified for the **delete** request, other fields such as bands can be omitted.

To delete all meters with a single message, specify OFPM_ALL as the meter value. Virtual meters can never be deleted and are not removed when deleting all meters.

7 The OpenFlow Protocol

The heart of the OpenFlow switch specification is the set of structures used for OpenFlow Protocol messages.

The structures, defines, and enumerations described below are derived from the file `include/openflow/openflow.h`, which is part of the standard OpenFlow specification distribution. All structures are packed with padding and 8-byte aligned, as checked by the assertion statements. All OpenFlow messages are sent in big-endian format.

7.1 OpenFlow Header

Each OpenFlow message begins with the OpenFlow header:

```
/* Header on all OpenFlow packets. */
struct ofp_header {
    uint8_t version;    /* OFP_VERSION. */
    uint8_t type;       /* One of the OFPT_ constants. */
    uint16_t length;    /* Length including this ofp_header. */
    uint32_t xid;       /* Transaction id associated with this packet.
                           Replies use the same id as was in the request
                           to facilitate pairing. */
};
OFP_ASSERT(sizeof(struct ofp_header) == 8);
```

The `version` specifies the OpenFlow protocol version being used. During the earlier draft phase of the OpenFlow Protocol, the most significant bit was set to indicate an experimental version. The lower bits indicate the revision number of the protocol. The version of the protocol described by the current specification is 1.3.2, and its *ofp_version* is 0x04 .

The `length` field indicates the total length of the message, so no additional framing is used to distinguish one frame from the next. The `type` can have the following values:

```
enum ofp_type {
    /* Immutable messages. */
    OFPT_HELLO              = 0,  /* Symmetric message */
    OFPT_ERROR              = 1,  /* Symmetric message */
    OFPT_ECHO_REQUEST       = 2,  /* Symmetric message */
    OFPT_ECHO_REPLY         = 3,  /* Symmetric message */
    OFPT_EXPERIMENTER       = 4,  /* Symmetric message */

    /* Switch configuration messages. */
    OFPT_FEATURES_REQUEST   = 5,  /* Controller/switch message */
    OFPT_FEATURES_REPLY     = 6,  /* Controller/switch message */
    OFPT_GET_CONFIG_REQUEST = 7,  /* Controller/switch message */
    OFPT_GET_CONFIG_REPLY   = 8,  /* Controller/switch message */
    OFPT_SET_CONFIG         = 9,  /* Controller/switch message */

    /* Asynchronous messages. */
    OFPT_PACKET_IN          = 10, /* Async message */
    OFPT_FLOW_REMOVED       = 11, /* Async message */
```



```
    OFPT_PORT_STATUS        = 12, /* Async message */

    /* Controller command messages. */
    OFPT_PACKET_OUT         = 13, /* Controller/switch message */
    OFPT_FLOW_MOD           = 14, /* Controller/switch message */
    OFPT_GROUP_MOD          = 15, /* Controller/switch message */
    OFPT_PORT_MOD           = 16, /* Controller/switch message */
    OFPT_TABLE_MOD          = 17, /* Controller/switch message */

    /* Multipart messages. */
    OFPT_MULTIPART_REQUEST      = 18, /* Controller/switch message */
    OFPT_MULTIPART_REPLY        = 19, /* Controller/switch message */

    /* Barrier messages. */
    OFPT_BARRIER_REQUEST    = 20, /* Controller/switch message */
    OFPT_BARRIER_REPLY      = 21, /* Controller/switch message */

    /* Queue Configuration messages. */
    OFPT_QUEUE_GET_CONFIG_REQUEST = 22,  /* Controller/switch message */
    OFPT_QUEUE_GET_CONFIG_REPLY   = 23,  /* Controller/switch message */

    /* Controller role change request messages. */
    OFPT_ROLE_REQUEST       = 24, /* Controller/switch message */
    OFPT_ROLE_REPLY         = 25, /* Controller/switch message */

    /* Asynchronous message configuration. */
    OFPT_GET_ASYNC_REQUEST  = 26, /* Controller/switch message */
    OFPT_GET_ASYNC_REPLY    = 27, /* Controller/switch message */
    OFPT_SET_ASYNC          = 28, /* Controller/switch message */

    /* Meters and rate limiters configuration messages. */
    OFPT_METER_MOD          = 29, /* Controller/switch message */
};
```

7.1.1 Padding

Most OpenFlow messages contain padding fields. Those are included in the various message types and in various common structures. Most of those padding fields can be identified by the fact that their name start with `pad`. The goal of padding fields is to align multi-byte entities on natural processor boundaries.

All common structures included in messages are aligned on 64 bit boundaries. Various other types are aligned as needed, for example 32 bits integer are aligned on 32 bit boundaries. An exception to the padding rules are OXM match fields which are never padded (7.2.3.2). In general, the end of OpenFlow messages is not padded, unless explicitly specified. On the other hand, common structures are almost always padded at the end.

The padding fields should be set to zero. An OpenFlow implementation must accept any values set in padding fields, and must just ignore the content of padding fields.

7.2 Common Structures

This section describes structures used by multiple message types.

7.2.1 Port Structures

The OpenFlow pipeline receives and sends packets on ports. The switch may define physical and logical ports, and the OpenFlow specification defines some reserved ports (see 4.1).

The physical ports, switch-defined logical ports, and the `OFPP_LOCAL` reserved port are described with the following structure:

```
/* Description of a port */
struct ofp_port {
    uint32_t port_no;
    uint8_t pad[4];
    uint8_t hw_addr[OFP_ETH_ALEN];
    uint8_t pad2[2];                  /* Align to 64 bits. */
    char name[OFP_MAX_PORT_NAME_LEN]; /* Null-terminated */

    uint32_t config;        /* Bitmap of OFPPC_* flags. */
    uint32_t state;         /* Bitmap of OFPPS_* flags. */

    /* Bitmaps of OFPPF_* that describe features.  All bits zeroed if
     * unsupported or unavailable. */
    uint32_t curr;          /* Current features. */
    uint32_t advertised;    /* Features being advertised by the port. */
    uint32_t supported;     /* Features supported by the port. */
    uint32_t peer;          /* Features advertised by peer. */

    uint32_t curr_speed;    /* Current port bitrate in kbps. */
    uint32_t max_speed;     /* Max port bitrate in kbps */
};
OFP_ASSERT(sizeof(struct ofp_port) == 64);
```

The `port_no` field uniquely identifies a port within a switch. The `hw_addr` field typically is the MAC address for the port; `OFP_ETH_ALEN` is 6. The name field is a null-terminated string containing a human-readable name for the interface. The value of `OFP_MAX_PORT_NAME_LEN` is 16.

The `config` field describes port administrative settings, and has the following structure:

```
/* Flags to indicate behavior of the physical port.  These flags are
 * used in ofp_port to describe the current configuration.  They are
 * used in the ofp_port_mod message to configure the port's behavior.
 */
enum ofp_port_config {
    OFPPC_PORT_DOWN    = 1 << 0,  /* Port is administratively down. */

    OFPPC_NO_RECV      = 1 << 2,  /* Drop all packets received by port. */
    OFPPC_NO_FWD       = 1 << 5,  /* Drop packets forwarded to port. */
    OFPPC_NO_PACKET_IN = 1 << 6   /* Do not send packet-in msgs for port. */
};
```


The OFPPC_PORT_DOWN bit indicates that the port has been administratively brought down and should not be used by OpenFlow. The OFPPC_NO_RECV bit indicates that packets received on that port should be ignored. The OFPPC_NO_FWD bit indicates that OpenFlow should not send packets to that port. The OFPPFL_NO_PACKET_IN bit indicates that packets on that port that generate a table miss should never trigger a packet-in message to the controller.

In general, the port config bits are set by the controller and not changed by the switch. Those bits may be useful for the controller to implement protocols such as STP or BFD. If the port config bits are changed by the switch through another administrative interface, the switch sends an OFPT_PORT_STATUS message to notify the controller of the change.

The state field describes the port internal state, and has the following structure:

```
/* Current state of the physical port.  These are not configurable from
 * the controller.
 */
enum ofp_port_state {
    OFPPS_LINK_DOWN    = 1 << 0,  /* No physical link present. */
    OFPPS_BLOCKED      = 1 << 1,  /* Port is blocked */
    OFPPS_LIVE         = 1 << 2,  /* Live for Fast Failover Group. */
};
```

The port state bits represent the state of the physical link or switch protocols outside of OpenFlow. The OFPPS_LINK_DOWN bit indicates the the physical link is not present. The OFPPS_BLOCKED bit indicates that a switch protocol outside of OpenFlow, such as 802.1D Spanning Tree, is preventing the use of that port with OFPP_FLOOD.

All port state bits are read-only and cannot be changed by the controller. When the port flags are changed, the switch sends an OFPT_PORT_STATUS message to notify the controller of the change.

The port numbers use the following conventions:

```
/* Port numbering. Ports are numbered starting from 1. */
enum ofp_port_no {
    /* Maximum number of physical and logical switch ports. */
    OFPP_MAX        = 0xffffff00,

    /* Reserved OpenFlow Port (fake output "ports"). */
    OFPP_IN_PORT    = 0xfffffff8,  /* Send the packet out the input port.  This
                                      reserved port must be explicitly used
                                      in order to send back out of the input
                                      port. */
    OFPP_TABLE      = 0xfffffff9,  /* Submit the packet to the first flow table
                                      NB: This destination port can only be
                                      used in packet-out messages. */
    OFPP_NORMAL     = 0xfffffffa,  /* Process with normal L2/L3 switching. */
    OFPP_FLOOD      = 0xfffffffb,  /* All physical ports in VLAN, except input
                                      port and those blocked or link down. */
    OFPP_ALL        = 0xfffffffc,  /* All physical ports except input port. */
    OFPP_CONTROLLER = 0xfffffffd,  /* Send to controller. */
    OFPP_LOCAL      = 0xfffffffe,  /* Local openflow "port". */
    OFPP_ANY        = 0xffffffff   /* Wildcard port used only for flow mod
                                      (delete) and flow stats requests. Selects
```



```
                                    all flows regardless of output port
                                    (including flows with no output port). */
};
```

The `curr`, `advertised`, `supported`, and `peer` fields indicate link modes (speed and duplexity), link type (copper/fiber) and link features (autonegotiation and pause). Port features are represented by the following structure:

```
/* Features of ports available in a datapath. */
enum ofp_port_features {
    OFPPF_10MB_HD    = 1 << 0,  /* 10 Mb half-duplex rate support. */
    OFPPF_10MB_FD    = 1 << 1,  /* 10 Mb full-duplex rate support. */
    OFPPF_100MB_HD   = 1 << 2,  /* 100 Mb half-duplex rate support. */
    OFPPF_100MB_FD   = 1 << 3,  /* 100 Mb full-duplex rate support. */
    OFPPF_1GB_HD     = 1 << 4,  /* 1 Gb half-duplex rate support. */
    OFPPF_1GB_FD     = 1 << 5,  /* 1 Gb full-duplex rate support. */
    OFPPF_10GB_FD    = 1 << 6,  /* 10 Gb full-duplex rate support. */
    OFPPF_40GB_FD    = 1 << 7,  /* 40 Gb full-duplex rate support. */
    OFPPF_100GB_FD   = 1 << 8,  /* 100 Gb full-duplex rate support. */
    OFPPF_1TB_FD     = 1 << 9,  /* 1 Tb full-duplex rate support. */
    OFPPF_OTHER      = 1 << 10, /* Other rate, not in the list. */

    OFPPF_COPPER     = 1 << 11, /* Copper medium. */
    OFPPF_FIBER      = 1 << 12, /* Fiber medium. */
    OFPPF_AUTONEG    = 1 << 13, /* Auto-negotiation. */
    OFPPF_PAUSE      = 1 << 14, /* Pause. */
    OFPPF_PAUSE_ASYM = 1 << 15  /* Asymmetric pause. */
};
```

Multiple of these flags may be set simultaneously. If none of the port speed flags are set, the `max_speed` or `curr_speed` are used.

The `curr_speed` and `max_speed` fields indicate the current and maximum bit rate (raw transmission speed) of the link in kbps. The number should be rounded to match common usage. For example, an optical 10 Gb Ethernet port should have this field set to 10000000 (instead of 10312500), and an OC-192 port should have this field set to 10000000 (instead of 9953280).

The `max_speed` fields indicate the maximum configured capacity of the link, whereas the `curr_speed` indicates the current capacity. If the port is a LAG with 3 links of 1Gb/s capacity, with one of the ports of the LAG being down, one port auto-negotiated at 1Gb/s and 1 port auto-negotiated at 100Mb/s, the `max_speed` is 3 Gb/s and the `curr_speed` is 1.1 Gb/s.

7.2.2 Queue Structures

An OpenFlow switch provides limited Quality-of-Service support (QoS) through a simple queuing mechanism. One (or more) queues can attach to a port and be used to map flow entries on it. Flow entries mapped to a specific queue will be treated according to that queue's configuration (e.g. min rate).

A queue is described by the `ofp_packet_queue` structure:


```
/* Full description for a queue. */
struct ofp_packet_queue {
    uint32_t queue_id;     /* id for the specific queue. */
    uint32_t port;         /* Port this queue is attached to. */
    uint16_t len;          /* Length in bytes of this queue desc. */
    uint8_t pad[6];        /* 64-bit alignment. */
    struct ofp_queue_prop_header properties[0]; /* List of properties. */
};
OFP_ASSERT(sizeof(struct ofp_packet_queue) == 16);
```

Each queue is further described by a set of properties, each of a specific type and configuration.

```
enum ofp_queue_properties {
    OFPQT_MIN_RATE      = 1,      /* Minimum datarate guaranteed. */
    OFPQT_MAX_RATE      = 2,      /* Maximum datarate. */
    OFPQT_EXPERIMENTER  = 0xffff  /* Experimenter defined property. */
};
```

Each queue property description starts with a common header:

```
/* Common description for a queue. */
struct ofp_queue_prop_header {
    uint16_t property;    /* One of OFPQT_. */
    uint16_t len;         /* Length of property, including this header. */
    uint8_t pad[4];       /* 64-bit alignemnt. */
};
OFP_ASSERT(sizeof(struct ofp_queue_prop_header) == 8);
```

A *minimum-rate* queue property uses the following structure and fields:

```
/* Min-Rate queue property description. */
struct ofp_queue_prop_min_rate {
    struct ofp_queue_prop_header prop_header; /* prop: OFPQT_MIN, len: 16. */
    uint16_t rate;        /* In 1/10 of a percent; >1000 -> disabled. */
    uint8_t pad[6];       /* 64-bit alignment */
};
OFP_ASSERT(sizeof(struct ofp_queue_prop_min_rate) == 16);
```

If `rate` is not configured, it is set to `OFPQ_MIN_RATE_UNCFG`, which is 0xffff.

A *maximum-rate* queue property uses the following structure and fields:

```
/* Max-Rate queue property description. */
struct ofp_queue_prop_max_rate {
    struct ofp_queue_prop_header prop_header; /* prop: OFPQT_MAX, len: 16. */
    uint16_t rate;        /* In 1/10 of a percent; >1000 -> disabled. */
    uint8_t pad[6];       /* 64-bit alignment */
};
OFP_ASSERT(sizeof(struct ofp_queue_prop_max_rate) == 16);
```

If `rate` is not configured, it is set to `OFPQ_MAX_RATE_UNCFG`, which is 0xffff.

A *experimenter* queue property uses the following structure and fields:


```
/* Experimenter queue property description. */
struct ofp_queue_prop_experimenter {
    struct ofp_queue_prop_header prop_header; /* prop: OFPQT_EXPERIMENTER, len: 16. */
    uint32_t experimenter;          /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint8_t pad[4];       /* 64-bit alignment */
    uint8_t data[0];      /* Experimenter defined data. */
};
OFP_ASSERT(sizeof(struct ofp_queue_prop_experimenter) == 16);
```

The rest of the experimenter queue property body is uninterpreted by standard OpenFlow processing and is arbitrarily defined by the corresponding experimenter.

7.2.3 Flow Match Structures

An OpenFlow match is composed of a flow match header and a sequence of zero or more flow match fields.

7.2.3.1 Flow Match Header

The flow match header is described by the `ofp_match` structure:

```
/* Fields to match against flows */
struct ofp_match {
    uint16_t type;             /* One of OFPMT_* */
    uint16_t length;           /* Length of ofp_match (excluding padding) */
    /* Followed by:
     *   - Exactly (length - 4) (possibly 0) bytes containing OXM TLVs, then
     *   - Exactly ((length + 7)/8*8 - length) (between 0 and 7) bytes of
     *     all-zero bytes
     * In summary, ofp_match is padded as needed, to make its overall size
     * a multiple of 8, to preserve alignement in structures using it.
     */
    uint8_t oxm_fields[0];     /* 0 or more OXM match fields */
    uint8_t pad[4];            /* Zero bytes - see above for sizing */
};
OFP_ASSERT(sizeof(struct ofp_match) == 8);
```

The `type` field is set to `OFPMT_OXM` and `length` field is set to the actual length of `ofp_match` structure including all match fields. The payload of the OpenFlow match is a set of OXM Flow match fields.

```
/* The match type indicates the match structure (set of fields that compose the
 * match) in use. The match type is placed in the type field at the beginning
 * of all match structures. The "OpenFlow Extensible Match" type corresponds
 * to OXM TLV format described below and must be supported by all OpenFlow
 * switches. Extensions that define other match types may be published on the
 * ONF wiki. Support for extensions is optional.
 */
enum ofp_match_type {
```



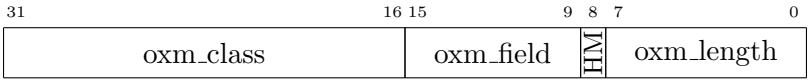
```
    OFPMT_STANDARD = 0,       /* Deprecated. */
    OFPMT_OXM      = 1,       /* OpenFlow Extensible Match */
};
```

The only valid match type in this specification is `OFPMT_OXM`, the OpenFlow 1.1 match type `OFPMT_STANDARD` is deprecated. If an alternate match type is used, the match fields and payload may be set differently, but this is outside the scope of this specification.

7.2.3.2 Flow Match Field Structures

The flow match fields are described using the OpenFlow Extensible Match (OXM) format, which is a compact type-length-value (TLV) format. Each OXM TLV is 5 to 259 (inclusive) bytes long. OXM TLVs are not aligned on or padded to any multibyte boundary. The first 4 bytes of an OXM TLV are its header, followed by the entry's body.

An OXM TLV's header is interpreted as a 32-bit word in network byte order (see figure 4).



Figure˜4: OXM TLV header layout.

The OXM TLV's header fields are defined in Table 9

Name		Width	Usage
oxm_type	`oxm_class`	16	Match class: member class or reserved class
	`oxm_field`	7	Match field within the class
	`oxm_hasmask`	1	Set if OXM include a bitmask in payload
	`oxm_length`	8	Length of OXM payload

Table˜9: OXM TLV header fields.

The `oxm_class` is a OXM match class that contains related match types, and is described in section 7.2.3.3. `oxm_field` is an class-specific value, identifying one of the match types within the match class. The combination of `oxm_class` and `oxm_field` (the most-significant 23 bits of the header) are collectively `oxm_type`. The `oxm_type` normally designates a protocol header field, such as the Ethernet type, but it can also refer to packet metadata, such as the switch port on which a packet arrived.

`oxm_hasmask` defines if the OXM TLV contains a bitmask, more details is explained in section 7.2.3.5.

`oxm_length` is a positive integer describing the length of the OXM TLV payload in bytes. The length of the OXM TLV, including the header, is exactly 4 + `oxm_length` bytes.

For a given `oxm_class`, `oxm_field`, and `oxm_hasmask` value, `oxm_length` is a constant. It is included only to allow software to minimally parse OXM TLVs of unknown types. (Similarly, for a given `oxm_class`, `oxm_field`, and `oxm_length`, `oxm_hasmask` is a constant.)

7.2.3.3 OXM classes

The match types are structured using OXM match classes. The OpenFlow specification distinguishes two types of OXM match classes, ONF member classes and ONF reserved classes, differentiated by their high order bit. Classes with the high order bit set to 1 are ONF reserved classes, they are used for the OpenFlow specification itself. Classes with the high order bit set to zero are ONF member classes, they are allocated by the ONF on an as needed basis, they uniquely identify an ONF member and can be used arbitrarily by that member. Support for ONF member classes is optional.

The following OXM classes are defined:

```
/* OXM Class IDs.
 * The high order bit differentiate reserved classes from member classes.
 * Classes 0x0000 to 0x7FFF are member classes, allocated by ONF.
 * Classes 0x8000 to 0xFFFE are reserved classes, reserved for standardisation.
 */
enum ofp_oxm_class {
    OFPXMC_NXM_0          = 0x0000,    /* Backward compatibility with NXM */
    OFPXMC_NXM_1          = 0x0001,    /* Backward compatibility with NXM */
    OFPXMC_OPENFLOW_BASIC = 0x8000,    /* Basic class for OpenFlow */
    OFPXMC_EXPERIMENTER   = 0xFFFF,    /* Experimenter class */
};
```

The class `OFPXMC_OPENFLOW_BASIC` contains the basic set of OpenFlow match fields (see 7.2.3.7). The optional class `OFPXMC_EXPERIMENTER` is used for experimenter matches (see 7.2.3.8). Other ONF reserved classes are reserved for future uses such as modularisation of the specification. The first two ONF member classes `OFPXMC_NXM_0` and `OFPXMC_NXM_1` are reserved for backward compatibility with the Nicira Extensible Match (NXM) specification.

7.2.3.4 Flow Matching

A zero-length OpenFlow match (one with no OXM TLVs) matches every packet. Match fields that should be wildcarded are omitted from the OpenFlow match.

An OXM TLV places a constraint on the packets matched by the OpenFlow match:

- If `oxm_hasmask` is 0, the OXM TLV's body contains a value for the field, called `oxm_value`. The OXM TLV match matches only packets in which the corresponding field equals `oxm_value`.
- If `oxm_hasmask` is 1, then the `oxm_entry`'s body contains a value for the field (`oxm_value`), followed by a bitmask of the same length as the value, called `oxm_mask`. Each 1-bit in `oxm_mask` constrains the OXM TLV to match only packets in which the corresponding bit of the field equals the the corresponding bit in `oxm_value`. Each 0-bit in `oxm_mask` places no constraint on the corresponding bit in the field.

When using masking, it is an error for a 0-bit in `oxm_mask` to have a corresponding 1-bit in `oxm_value`. The switch must report an error message of type `OFPET_BAD_MATCH` and code `OFPBMC_BAD_WILDCARDS` in such a case.

The following table summarizes the constraint that a pair of corresponding `oxm_mask` and `oxm_value` bits place upon the corresponding field bit when using masking. Omitting `oxm_mask` is equivalent to supplying an `oxm_mask` that is all 1-bits.

	`oxm_value`	
`oxm_mask`	0	1
0	no constraint	error
1	must be 0	must be 1

Table˜10: OXM mask and value.

When there are multiple OXM TLVs, all of the constraints must be met: the packet fields must match all OXM TLVs part of the OpenFlow match. The fields for which OXM TLVs that are not present are wildcarded to ANY, omitted OXM TLVs are effectively fully masked to zero.

7.2.3.5 Flow Match Field Masking

When `oxm_hasmask` is 1, the OXM TLV contains a bitmask and its length is effectively doubled, so `oxm_length` is always even. The bitmask follows the field value and is encoded in the same way. The masks are defined such that a 0 in a given bit position indicates a "don't care" match for the same bit in the corresponding field, whereas a 1 means match the bit exactly.

An all-zero-bits `oxm_mask` is equivalent to omitting the OXM TLV entirely. An all-one-bits `oxm_mask` is equivalent to specifying 0 for `oxm_hasmask` and omitting `oxm_mask`.

Some `oxm_types` may not support masked wildcards, that is, `oxm_hasmask` must always be 0 when these fields are specified. For example, the field that identifies the ingress port on which a packet was received may not be masked.

Some `oxm_types` that do support masked wildcards may only support certain `oxm_mask` patterns. For example, some fields that have IPv4 address values may be restricted to CIDR masks (subnet masks).

These restrictions are detailed in specifications for individual fields. A switch may accept an `oxm_hasmask` or `oxm_mask` value that the specification disallows, but only if the switch correctly implements support for that `oxm_hasmask` or `oxm_mask` value. A switch must reject an attempt to set up a flow entry that contains a `oxm_hasmask` or `oxm_mask` value that it does not support (see 6.4).

7.2.3.6 Flow Match Field Prerequisite

The presence of an OXM TLV with a given `oxm_type` may be restricted based on the presence or values of other OXM TLVs. In general, matching header fields of a protocol can only be done if the OpenFlow match explitly matches the corresponding protocol.

For example:

- An OXM TLV for `oxm_type`=OXM_OF_IPV4_SRC is allowed only if it is preceded by another entry with `oxm_type=OXM_OF_ETH_TYPE`, `oxm_hasmask`=0, and `oxm_value`=0x0800. That is, matching on the IPv4 source address is allowed only if the Ethernet type is explicitly set to IPv4.

- An OXM TLV for `oxm_type`=OXM_OF_TCP_SRC is allowed only if it is preceded by an entry with `oxm_type`=OXM_OF_ETH_TYPE, `oxm_hasmask`=0, `oxm_value`=0x0800 or 0x86dd, and another with `oxm_type`=OXM_OF_IP_PROTO, `oxm_hasmask`=0, `oxm_value`=6, in that order. That is, matching on the TCP source port is allowed only if the Ethernet type is IP and the IP protocol is TCP.
- An OXM TLV for `oxm_type`=OXM_OF_MPLS_LABEL is allowed only if it is preceded by an entry with `oxm_type`=OXM_OF_ETH_TYPE, `oxm_hasmask`=0, `oxm_value`=0x8847 or 0x8848.
- An OXM TLV for `oxm_type`=OXM_OF_VLAN_PCP is allowed only if it is preceded by an entry with `oxm_type`=OXM_OF_VLAN_VID, `oxm_value`!=OFPVID_NONE.

These restrictions are noted in specifications for individual fields (see 7.2.3.7). A switch may implement relaxed versions of these restrictions. For example, a switch may accept no prerequisite at all. A switch must reject an attempt to set up a flow entry that violates its restrictions (see 6.4), and must deal with inconsistent matches created by the lack of prerequisite (for example matching both a TCP source port and a UDP destination port).

New match fields defined by members (in member classes or as experimenter fields) may provide alternate prerequisites to already specified match fields. For example, this could be used to reuse existing IP match fields over an alternate link technology (such as PPP) by substituting the `ETH_TYPE` prerequisite as needed (for PPP, that could be an hypothetical `PPP_PROTOCOL` field).

An OXM TLV that has prerequisite restrictions must appear after the OXM TLVs for its prerequisites. Ordering of OXM TLVs within an OpenFlow match is not otherwise constrained.

Any given `oxm_type` may appear in an OpenFlow match at most once, otherwise the switch must generate an error (see 6.4). A switch may implement a relaxed version of this rule and may allow in some cases a `oxm_type` to appear multiple time in an OpenFlow match, however the behaviour of matching is then implementation-defined.

If a flow table implements a specific OXM TLV, this flow table must accept valid matches containing the prerequisites of this OXM TLV, even if the flow table does not support matching all possible values for the match fields specified by those prerequisites. For example, if a flow table matches the IPv4 source address, this flow table must accept matching the Ethertype exactly to IPv4, however this flow table does not need to support matching Ethertype to any other value.

7.2.3.7 Flow Match Fields

The specification defines a default set of match fields with `oxm_class`=`OFPXMC_OPENFLOW_BASIC` which can have the following values:

```
/* OXM Flow match field types for OpenFlow basic class. */
enum oxm_ofb_match_fields {
    OFPXMT_OFB_IN_PORT        = 0,  /* Switch input port. */
    OFPXMT_OFB_IN_PHY_PORT    = 1,  /* Switch physical input port. */
    OFPXMT_OFB_METADATA       = 2,  /* Metadata passed between tables. */
    OFPXMT_OFB_ETH_DST        = 3,  /* Ethernet destination address. */
    OFPXMT_OFB_ETH_SRC        = 4,  /* Ethernet source address. */
    OFPXMT_OFB_ETH_TYPE       = 5,  /* Ethernet frame type. */
    OFPXMT_OFB_VLAN_VID       = 6,  /* VLAN id. */
```



```
    OFPXMT_OFB_VLAN_PCP       = 7,  /* VLAN priority. */
    OFPXMT_OFB_IP_DSCP        = 8,  /* IP DSCP (6 bits in ToS field). */
    OFPXMT_OFB_IP_ECN         = 9,  /* IP ECN (2 bits in ToS field). */
    OFPXMT_OFB_IP_PROTO       = 10, /* IP protocol. */
    OFPXMT_OFB_IPV4_SRC       = 11, /* IPv4 source address. */
    OFPXMT_OFB_IPV4_DST       = 12, /* IPv4 destination address. */
    OFPXMT_OFB_TCP_SRC        = 13, /* TCP source port. */
    OFPXMT_OFB_TCP_DST        = 14, /* TCP destination port. */
    OFPXMT_OFB_UDP_SRC        = 15, /* UDP source port. */
    OFPXMT_OFB_UDP_DST        = 16, /* UDP destination port. */
    OFPXMT_OFB_SCTP_SRC       = 17, /* SCTP source port. */
    OFPXMT_OFB_SCTP_DST       = 18, /* SCTP destination port. */
    OFPXMT_OFB_ICMPV4_TYPE    = 19, /* ICMP type. */
    OFPXMT_OFB_ICMPV4_CODE    = 20, /* ICMP code. */
    OFPXMT_OFB_ARP_OP         = 21, /* ARP opcode. */
    OFPXMT_OFB_ARP_SPA        = 22, /* ARP source IPv4 address. */
    OFPXMT_OFB_ARP_TPA        = 23, /* ARP target IPv4 address. */
    OFPXMT_OFB_ARP_SHA        = 24, /* ARP source hardware address. */
    OFPXMT_OFB_ARP_THA        = 25, /* ARP target hardware address. */
    OFPXMT_OFB_IPV6_SRC       = 26, /* IPv6 source address. */
    OFPXMT_OFB_IPV6_DST       = 27, /* IPv6 destination address. */
    OFPXMT_OFB_IPV6_FLABEL    = 28, /* IPv6 Flow Label */
    OFPXMT_OFB_ICMPV6_TYPE    = 29, /* ICMPv6 type. */
    OFPXMT_OFB_ICMPV6_CODE    = 30, /* ICMPv6 code. */
    OFPXMT_OFB_IPV6_ND_TARGET = 31, /* Target address for ND. */
    OFPXMT_OFB_IPV6_ND_SLL    = 32, /* Source link-layer for ND. */
    OFPXMT_OFB_IPV6_ND_TLL    = 33, /* Target link-layer for ND. */
    OFPXMT_OFB_MPLS_LABEL     = 34, /* MPLS label. */
    OFPXMT_OFB_MPLS_TC        = 35, /* MPLS TC. */
    OFPXMT_OFP_MPLS_BOS       = 36, /* MPLS BoS bit. */
    OFPXMT_OFB_PBB_ISID       = 37, /* PBB I-SID. */
    OFPXMT_OFB_TUNNEL_ID      = 38, /* Logical Port Metadata. */
    OFPXMT_OFB_IPV6_EXTHDR    = 39, /* IPv6 Extension Header pseudo-field */
};
```

A switch must support the required match fields listed in Table 11 in its pipeline. Each required match field must be supported in at least one flow table of the switch : that flow table must enable matching that field and the match field prerequisites must be met in that table (see 7.2.3.6). The required fields don't need to be implemented in all flow tables, and don't need to be implemented in the same flow table. Flow tables can support non-required and experimenter match fields. The controller can query the switch about which match fields are supported in each flow table (see 7.3.5.5).

Field		Description
OXM_OF_IN_PORT	*Required*	Ingress port. This may be a physical or switch-defined logical port.
OXM_OF_ETH_DST	*Required*	Ethernet destination address. Can use arbitrary bitmask
OXM_OF_ETH_SRC	*Required*	Ethernet source address. Can use arbitrary bitmask
OXM_OF_ETH_TYPE	*Required*	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_IP_PROTO	*Required*	IPv4 or IPv6 protocol number
OXM_OF_IPV4_SRC	*Required*	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	*Required*	IPv4 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_SRC	*Required*	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	*Required*	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_TCP_SRC	*Required*	TCP source port
OXM_OF_TCP_DST	*Required*	TCP destination port
OXM_OF_UDP_SRC	*Required*	UDP source port
OXM_OF_UDP_DST	*Required*	UDP destination port

Table˜11: Required match fields.

All match fields have different size, prerequisites and masking capability, as specified in Table 12. If not explictely specified in the field description, each field type refer the the outermost occurrence of the field in the packet headers.

Field	Bits	Mask	Pre-requisite	Description
OXM_OF_IN_PORT	32	No	None	Ingress port. Numerical representation of incoming port, starting at 1. This may be a physical or switch-defined logical port.
OXM_OF_IN_PHY_PORT	32	No	IN_PORT present	Physical port. In `ofp_packet_in` messages, underlying physical port when packet received on a logical port.
OXM_OF_METADATA	64	Yes	None	Table metadata. Used to pass information between tables.
OXM_OF_ETH_DST	48	Yes	None	Ethernet destination MAC address.
OXM_OF_ETH_SRC	48	Yes	None	Ethernet source MAC address.
OXM_OF_ETH_TYPE	16	No	None	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_VLAN_VID	12+1	Yes	None	VLAN-ID from 802.1Q header. The CFI bit indicate the presence of a valid VLAN-ID, see below.
OXM_OF_VLAN_PCP	3	No	VLAN_VID!=NONE	VLAN-PCP from 802.1Q header.
OXM_OF_IP_DSCP	6	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	Diff Serv Code Point (DSCP). Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_ECN	2	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	ECN bits of the IP header. Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_PROTO	8	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	IPv4 or IPv6 protocol number.
OXM_OF_IPV4_SRC	32	Yes	ETH_TYPE=0x0800	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	32	Yes	ETH_TYPE=0x0800	IPv4 destination address. Can use subnet mask or arbitrary bitmask

Table 12 – Continued on next page

Table 12 – concluded from previous page

Field	Bits	Mask	Pre-requisite	Description
OXM_OF_TCP_SRC	16	No	IP_PROTO=6	TCP source port
OXM_OF_TCP_DST	16	No	IP_PROTO=6	TCP destination port
OXM_OF_UDP_SRC	16	No	IP_PROTO=17	UDP source port
OXM_OF_UDP_DST	16	No	IP_PROTO=17	UDP destination port
OXM_OF_SCTP_SRC	16	No	IP_PROTO=132	SCTP source port
OXM_OF_SCTP_DST	16	No	IP_PROTO=132	SCTP destination port
OXM_OF_ICMPV4_TYPE	8	No	IP_PROTO=1	ICMP type
OXM_OF_ICMPV4_CODE	8	No	IP_PROTO=1	ICMP code
OXM_OF_ARP_OP	16	No	ETH_TYPE=0x0806	ARP opcode
OXM_OF_ARP_SPA	32	Yes	ETH_TYPE=0x0806	Source IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_TPA	32	Yes	ETH_TYPE=0x0806	Target IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_SHA	48	Yes	ETH_TYPE=0x0806	Source Ethernet address in the ARP payload.
OXM_OF_ARP_THA	48	Yes	ETH_TYPE=0x0806	Target Ethernet address in the ARP payload.
OXM_OF_IPV6_SRC	128	Yes	ETH_TYPE=0x86dd	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	128	Yes	ETH_TYPE=0x86dd	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_FLABEL	20	Yes	ETH_TYPE=0x86dd	IPv6 flow label.
OXM_OF_ICMPV6_TYPE	8	No	IP_PROTO=58	ICMPv6 type
OXM_OF_ICMPV6_CODE	8	No	IP_PROTO=58	ICMPv6 code
OXM_OF_IPV6_ND_TARGET	128	No	ICMPV6_TYPE=135 or ICMPV6_TYPE=136	The target address in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_SLL	48	No	ICMPV6_TYPE=135	The source link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_TLL	48	No	ICMPV6_TYPE=136	The target link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_MPLS_LABEL	20	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The LABEL in the first MPLS shim header.
OXM_OF_MPLS_TC	3	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The TC in the first MPLS shim header.
OXM_OF_MPLS_BOS	1	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The BoS bit (Bottom of Stack bit) in the first MPLS shim header.
OXM_OF_PBB_ISID	24	Yes	ETH_TYPE=0x88E7	The I-SID in the first PBB service instance tag.
OXM_OF_TUNNEL_ID	64	Yes	None	Metadata associated with a logical port.
OXM_OF_IPV6_EXTHDR	9	Yes	ETH_TYPE=0x86dd	IPv6 Extension Header pseudo-field.

Table˜12: Match fields details.

The ingress port OXM_OF_IN_PORT is a valid standard OpenFlow port, either a physical, a logical port, the OFPP_LOCAL reserved port or the OFPP_CONTROLLER reserved port. The physical port OXM_OF_IN_PHY_PORT is used in *Packet-in* messages to identify a physical port underneath a logical port (see 7.4.1).

The metadata field OXM_OF_METADATA is used to pass information between lookups across multiple tables. This value can be arbitrarily masked.

The Tunnel ID field OXM_OF_TUNNEL_ID carries optional metadata associated with a logical port. The mapping of this metadata is defined by the logical port implementation. If the logical port does not provide such data or if the packet was received on a physical port, its value is zero. For example, for a packet received via GRE tunnel including a (32-bit) key, the key is stored in the low 32-bits and the

high bits are zeroed. For a MPLS logical port, the low 20 bits represent the MPLS Label. For a VxLAN logical port, the low 24 bits represent the VNI.

Omitting the OFPXMT_OFB_VLAN_VID field specifies that a flow entry should match packets regardless of whether they contain the corresponding tag. Special values are defined below for the VLAN tag to allow matching of packets with any tag, independent of the tag's value, and to supports matching packets without a VLAN tag. The special values defined for OFPXMT_OFB_VLAN_VID are:

```
/* The VLAN id is 12-bits, so we can use the entire 16 bits to indicate
 * special conditions.
 */
enum ofp_vlan_id {
    OFPVID_PRESENT = 0x1000, /* Bit that indicate that a VLAN id is set */
    OFPVID_NONE    = 0x0000, /* No VLAN id was set. */
};
```

The OFPXMT_OFB_VLAN_PCP field must be rejected when the OFPXMT_OFB_VLAN_VID field is wildcarded (not present) or when the value of OFPXMT_OFB_VLAN_VID is set to OFPVID_NONE. Table 13 summarizes the combinations of wildcard bits and field values for particular VLAN tag matches.

OXM field	oxm_value	oxm_mask	Matching packets
absent	-	-	Packets *with* and *without* a VLAN tag
present	OFPVID_NONE	absent	Only packets *without* a VLAN tag
present	OFPVID_PRESENT	OFPVID_PRESENT	Only packets *with* a VLAN tag regardless of its value
present	*value* | OFPVID_PRESENT	absent	Only packets with VLAN tag and VID equal *value*

Table˜13: Match combinations for VLAN tags.

The field OXM_OF_IPV6_EXTHDR is a pseudo field that indicates the presence of various IPv6 extension headers in the packet header. The IPv6 extension header bits are combined together in the fields OXM_OF_IPV6_EXTHDR, and those bits can have the following values:

```
/* Bit definitions for IPv6 Extension Header pseudo-field. */
enum ofp_ipv6exthdr_flags {
    OFPIEH_NONEXT = 1 << 0,     /* "No next header" encountered. */
    OFPIEH_ESP    = 1 << 1,     /* Encrypted Sec Payload header present. */
    OFPIEH_AUTH   = 1 << 2,     /* Authentication header present. */
    OFPIEH_DEST   = 1 << 3,     /* 1 or 2 dest headers present. */
    OFPIEH_FRAG   = 1 << 4,     /* Fragment header present. */
    OFPIEH_ROUTER = 1 << 5,     /* Router header present. */
    OFPIEH_HOP    = 1 << 6,     /* Hop-by-hop header present. */
    OFPIEH_UNREP  = 1 << 7,     /* Unexpected repeats encountered. */
    OFPIEH_UNSEQ  = 1 << 8,     /* Unexpected sequencing encountered. */
};
```

- OFPIEH_HOP is set to 1 if a hop-by-hop IPv6 extension header is present as the first extension header in the packet.
- OFPIEH_ROUTER is set to 1 if a router IPv6 extension header is present.

- OFPIEH_FRAG is set to 1 if a fragmentation IPv6 extension header is present.
- OFPIEH_DEST is set to 1 if one or more Destination options IPv6 extension headers are present. It is normal to have either one or two of these in one IPv6 packet (see RFC 2460).
- OFPIEH_AUTH is set to 1 if an Authentication IPv6 extension header is present.
- OFPIEH_ESP is set to 1 if an Encrypted Security Payload IPv6 extension header is present.
- OFPIEH_NONEXT is set to 1 if a No Next Header IPv6 extension header is present.
- OFPIEH_UNSEQ is set to 1 if IPv6 extension headers were not in the order preferred (but not required) by RFC 2460.
- OFPIEH_UNREP is set to 1 if more than one of a given IPv6 extension header is unexpectedly encountered. (Two destination options headers may be expected and would not cause this bit to be set.)

7.2.3.8 Experimenter Flow Match Fields

Support for experimenter-specific flow match fields is optional. Experimenter-specific flow match fields may defined using the oxm_class=OFPXMC_EXPERIMENTER. The first four bytes of the OXM TLV's body contains the experimenter identifier, which takes the same form as in struct ofp_experimenter (see 7.5.4). Both oxm_field and the rest of the OXM TLV is experimenter-defined and does not need to be padded or aligned.

```
/* Header for OXM experimenter match fields. */
struct ofp_oxm_experimenter_header {
    uint32_t oxm_header;        /* oxm_class = OFPXMC_EXPERIMENTER */
    uint32_t experimenter;      /* Experimenter ID which takes the same
                                   form as in struct ofp_experimenter_header. */
};
OFP_ASSERT(sizeof(struct ofp_oxm_experimenter_header) == 8);
```

7.2.4 Flow Instruction Structures

Flow instructions associated with a flow table entry are executed when a flow matches the entry. The list of instructions that are currently defined are:

```
enum ofp_instruction_type {
    OFPIT_GOTO_TABLE = 1,       /* Setup the next table in the lookup
                                   pipeline */
    OFPIT_WRITE_METADATA = 2,   /* Setup the metadata field for use later in
                                   pipeline */
    OFPIT_WRITE_ACTIONS = 3,    /* Write the action(s) onto the datapath action
                                   set */
    OFPIT_APPLY_ACTIONS = 4,    /* Applies the action(s) immediately */
    OFPIT_CLEAR_ACTIONS = 5,    /* Clears all actions from the datapath
                                   action set */
    OFPIT_METER = 6,            /* Apply meter (rate limiter) */
```



```
    OFPIT_EXPERIMENTER = 0xFFFF  /* Experimenter instruction */
};
```

The instruction set is described in section 5.9. Flow tables may support a subset of instruction types. An instruction definition contains the instruction type, length, and any associated data:

```
/* Instruction header that is common to all instructions.  The length includes
 * the header and any padding used to make the instruction 64-bit aligned.
 * NB: The length of an instruction *must* always be a multiple of eight. */
struct ofp_instruction {
    uint16_t type;                /* Instruction type */
    uint16_t len;                 /* Length of this struct in bytes. */
};
OFP_ASSERT(sizeof(struct ofp_instruction) == 4);
```

The `OFPIT_GOTO_TABLE` instruction uses the following structure and fields:

```
/* Instruction structure for OFPIT_GOTO_TABLE */
struct ofp_instruction_goto_table {
    uint16_t type;                /* OFPIT_GOTO_TABLE */
    uint16_t len;                 /* Length of this struct in bytes. */
    uint8_t table_id;             /* Set next table in the lookup pipeline */
    uint8_t pad[3];               /* Pad to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_goto_table) == 8);
```

`table_id` indicates the next table in the packet processing pipeline.

The `OFPIT_WRITE_METADATA` instruction uses the following structure and fields:

```
/* Instruction structure for OFPIT_WRITE_METADATA */
struct ofp_instruction_write_metadata {
    uint16_t type;                /* OFPIT_WRITE_METADATA */
    uint16_t len;                 /* Length of this struct in bytes. */
    uint8_t pad[4];               /* Align to 64-bits */
    uint64_t metadata;            /* Metadata value to write */
    uint64_t metadata_mask;       /* Metadata write bitmask */
};
OFP_ASSERT(sizeof(struct ofp_instruction_write_metadata) == 24);
```

Metadata for the next table lookup can be written using the `metadata` and the `metadata_mask` in order to set specific bits on the match field. If this instruction is not specified, the metadata is passed, unchanged.

The `OFPIT_WRITE_ACTIONS`, `OFPIT_APPLY_ACTIONS`, and `OFPIT_CLEAR_ACTIONS` instructions use the following structure and fields:


```
/* Instruction structure for OFPIT_WRITE/APPLY/CLEAR_ACTIONS */
struct ofp_instruction_actions {
    uint16_t type;              /* One of OFPIT_*_ACTIONS */
    uint16_t len;               /* Length of this struct in bytes. */
    uint8_t pad[4];             /* Align to 64-bits */
    struct ofp_action_header actions[0];  /* 0 or more actions associated with
                                             OFPIT_WRITE_ACTIONS and
                                             OFPIT_APPLY_ACTIONS */
};
OFP_ASSERT(sizeof(struct ofp_instruction_actions) == 8);
```

For the Apply-Actions instruction, the `actions` field is treated as a list and the actions are applied to the packet *in-order*. For the Write-Actions instruction, the `actions` field is treated as a set and the actions are merged into the current action set.

For the Clear-Actions instruction, the structure does not contain any actions.

The `OFPIT_METER` instruction uses the following structure and fields:

```
/* Instruction structure for OFPIT_METER */
struct ofp_instruction_meter {
    uint16_t type;                /* OFPIT_METER */
    uint16_t len;                 /* Length is 8. */
    uint32_t meter_id;            /* Meter instance. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_meter) == 8);
```

`meter_id` indicates which meter to apply on the packet.

An `OFPIT_EXPERIMENTER` instruction uses the following structure and fields:

```
/* Instruction structure for experimental instructions */
struct ofp_instruction_experimenter {
    uint16_t type; /* OFPIT_EXPERIMENTER */
    uint16_t len;               /* Length of this struct in bytes */
    uint32_t experimenter;      /* Experimenter ID which takes the same form
                                   as in struct ofp_experimenter_header. */
    /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_experimenter) == 8);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in struct `ofp_experimenter` (see 7.5.4).

7.2.5 Action Structures

A number of actions may be associated with flow entries, groups or packets. The currently defined action types are:


```
enum ofp_action_type {
    OFPAT_OUTPUT       = 0,  /* Output to switch port. */
    OFPAT_COPY_TTL_OUT = 11, /* Copy TTL "outwards" -- from next-to-outermost
                                to outermost */
    OFPAT_COPY_TTL_IN  = 12, /* Copy TTL "inwards" -- from outermost to
                                next-to-outermost */
    OFPAT_SET_MPLS_TTL = 15, /* MPLS TTL */
    OFPAT_DEC_MPLS_TTL = 16, /* Decrement MPLS TTL */

    OFPAT_PUSH_VLAN    = 17, /* Push a new VLAN tag */
    OFPAT_POP_VLAN     = 18, /* Pop the outer VLAN tag */
    OFPAT_PUSH_MPLS    = 19, /* Push a new MPLS tag */
    OFPAT_POP_MPLS     = 20, /* Pop the outer MPLS tag */
    OFPAT_SET_QUEUE    = 21, /* Set queue id when outputting to a port */
    OFPAT_GROUP        = 22, /* Apply group. */
    OFPAT_SET_NW_TTL   = 23, /* IP TTL. */
    OFPAT_DEC_NW_TTL   = 24, /* Decrement IP TTL. */
    OFPAT_SET_FIELD    = 25, /* Set a header field using OXM TLV format. */
    OFPAT_PUSH_PBB     = 26, /* Push a new PBB service tag (I-TAG) */
    OFPAT_POP_PBB      = 27, /* Pop the outer PBB service tag (I-TAG) */
    OFPAT_EXPERIMENTER = 0xffff
};
```

Output, group, and set-queue actions are described in Section 5.12, tag push/pop actions are described in Table 6, and Set-Field actions are described from their OXM types in Table 12. An action definition contains the action type, length, and any associated data:

```
/* Action header that is common to all actions.  The length includes the
 * header and any padding used to make the action 64-bit aligned.
 * NB: The length of an action *must* always be a multiple of eight. */
struct ofp_action_header {
    uint16_t type;                  /* One of OFPAT_*. */
    uint16_t len;                   /* Length of action, including this
                                       header.  This is the length of action,
                                       including any padding to make it
                                       64-bit aligned. */
    uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_action_header) == 8);
```

An *Output* action uses the following structure and fields:

```
/* Action structure for OFPAT_OUTPUT, which sends packets out 'port'.
 * When the 'port' is the OFPP_CONTROLLER, 'max_len' indicates the max
 * number of bytes to send.  A 'max_len' of zero means no bytes of the
 * packet should be sent. A 'max_len' of OFPCML_NO_BUFFER means that
 * the packet is not buffered and the complete packet is to be sent to
 * the controller. */
struct ofp_action_output {
    uint16_t type;                  /* OFPAT_OUTPUT. */
    uint16_t len;                   /* Length is 16. */
    uint32_t port;                  /* Output port. */
    uint16_t max_len;               /* Max length to send to controller. */
    uint8_t pad[6];                 /* Pad to 64 bits. */
```



```
};
OFP_ASSERT(sizeof(struct ofp_action_output) == 16);
```

The `port` specifies the port through which the packet should be sent. The `max_len` indicates the maximum amount of data from a packet that should be sent when the port is `OFPP_CONTROLLER`. If `max_len` is zero, the switch must send zero bytes of the packet. A `max_len` of `OFPCML_NO_BUFFER` means that the complete packet should be sent, and it should not be buffered.

```
enum ofp_controller_max_len {
OFPCML_MAX       = 0xffe5, /* maximum max_len value which can be used
                              to request a specific byte length. */
OFPCML_NO_BUFFER = 0xffff  /* indicates that no buffering should be
                              applied and the whole packet is to be
                              sent to the controller. */
};
```

A *Group* action uses the following structure and fields:

```
/* Action structure for OFPAT_GROUP. */
struct ofp_action_group {
    uint16_t type;                  /* OFPAT_GROUP. */
    uint16_t len;                   /* Length is 8. */
    uint32_t group_id;              /* Group identifier. */
};
OFP_ASSERT(sizeof(struct ofp_action_group) == 8);
```

The `group_id` indicates the group used to process this packet. The set of buckets to apply depends on the group type.

The *Set-Queue* action sets the queue id that will be used to map a flow entry to an already-configured queue on a port, regardless of the ToS and VLAN PCP bits. The packet should not change as a result of a Set-Queue action. If the switch needs to set the ToS/PCP bits for internal handling, the original values should be restored before sending the packet out.

A switch may support only queues that are tied to specific PCP/ToS bits. In that case, we cannot map an arbitrary flow entry to a specific queue, therefore the Set-Queue action is not supported. The user can still use these queues and map flow entries to them by setting the relevant fields (ToS, VLAN PCP).

A *Set Queue* action uses the following structure and fields:

```
/* OFPAT_SET_QUEUE action struct: send packets to given queue on port. */
struct ofp_action_set_queue {
    uint16_t type;            /* OFPAT_SET_QUEUE. */
    uint16_t len;             /* Len is 8. */
    uint32_t queue_id;        /* Queue id for the packets. */
};
OFP_ASSERT(sizeof(struct ofp_action_set_queue) == 8);
```

A *Set MPLS TTL* action uses the following structure and fields:


```
/* Action structure for OFPAT_SET_MPLS_TTL. */
struct ofp_action_mpls_ttl {
    uint16_t type;                  /* OFPAT_SET_MPLS_TTL. */
    uint16_t len;                   /* Length is 8. */
    uint8_t mpls_ttl;               /* MPLS TTL */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_mpls_ttl) == 8);
```

The `mpls_ttl` field is the MPLS TTL to set.

A *Decrement MPLS TTL* action takes no arguments and consists only of a generic `ofp_action_header`. The action decrements the MPLS TTL.

A *Set IPv4 TTL* action uses the following structure and fields:

```
/* Action structure for OFPAT_SET_NW_TTL. */
struct ofp_action_nw_ttl {
    uint16_t type;                  /* OFPAT_SET_NW_TTL. */
    uint16_t len;                   /* Length is 8. */
    uint8_t nw_ttl;                 /* IP TTL */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_nw_ttl) == 8);
```

The `nw_ttl` field is the TTL address to set in the IP header.

An *Decrement IPv4 TTL* action takes no arguments and consists only of a generic `ofp_action_header`. This action decrement the TTL in the IP header if one is present.

A *Copy TTL outwards* action takes no arguments and consists only of a generic `ofp_action_header`. The action copies the TTL from the next-to-outermost header with TTL to the outermost header with TTL.

A *Copy TTL inwards* action takes no arguments and consists only of a generic `ofp_action_header`. The action copies the TTL from the outermost header with TTL to the next-to-outermost header with TTL.

A *Push VLAN header* action, *Push MPLS header* and *Push PBB header* action use the following structure and fields:

```
/* Action structure for OFPAT_PUSH_VLAN/MPLS/PBB. */
struct ofp_action_push {
    uint16_t type;                  /* OFPAT_PUSH_VLAN/MPLS/PBB. */
    uint16_t len;                   /* Length is 8. */
    uint16_t ethertype;             /* Ethertype */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_push) == 8);
```


The `ethertype` indicates the Ethertype of the new tag. It is used when pushing a new VLAN tag, new MPLS header or PBB service header.

The *Push PBB header* action logically pushes a new PBB service instance header onto the packet (I-TAG TCI), and copy the original Ethernet addresses of the packet into the customer addresses (C-DA and C-SA) of the tag. The customer addresses of the I-TAG are in the location of the original Ethernet addresses of the encapsulated packet, therefore this operations can be seen as adding both the backbone MAC-in-MAC header and the I-SID field to the front of the packet. The *Push PBB header* action does not add a backbone VLAN header (B-TAG) to the packet, it can be added via the *Push VLAN header* action after the push PBB header operation. After this operation, regular set-field actions can be used to modify the outer Ethernet addresses (B-DA and B-SA).

A *Pop VLAN header* action takes no arguments and consists only of a generic `ofp_action_header`. The action pops the outermost VLAN tag from the packet.

A *Pop PBB header* action takes no arguments and consists only of a generic `ofp_action_header`. The action logically pops the outer-most PBB service instance header from the packet (I-TAG TCI) and copy the customer addresses (C-DA and C-SA) in the Ethernet addresses of the packet. This operation can be seen as removing the backbone MAC-in-MAC header and the I-SID field from the front of the packet. The *Pop PBB header* action does not remove the backbone VLAN header (B-TAG) from the packet, it should be removed prior to this operation via the *Pop VLAN header* action.

A *Pop MPLS header* action uses the following structure and fields:

```
/* Action structure for OFPAT_POP_MPLS. */
struct ofp_action_pop_mpls {
    uint16_t type;                  /* OFPAT_POP_MPLS. */
    uint16_t len;                   /* Length is 8. */
    uint16_t ethertype;             /* Ethertype */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_pop_mpls) == 8);
```

The `ethertype` indicates the Ethertype of the MPLS payload. The `ethertype` is used as the Ethertype for the resulting packet regardless of whether the "bottom of stack (BoS)" bit was set in the removed MPLS shim. It is recommended that flow entries using this action match both the MPLS label and the MPLS BoS fields to avoid applying the wrong Ethertype to the MPLS payload.

The MPLS specification does not allow to set arbitrary Ethertype to MPLS payload when BoS is not equal to 1, and the controller is responsible in complying with this requirement and only set 0x8847 or 0x8848 as the Ethertype for those MPLS payloads. The switch can optionally enforce this MPLS requirement: in this case the switch should reject any flow entry matching a wildcard BoS and any flow entry matching BoS to 0 with the wrong `ethertype` in the Pop MPLS header action, and in both cases should return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_MATCH_INCONSISTENT` code.

Set Field actions uses the following structure and fields:

```
/* Action structure for OFPAT_SET_FIELD. */
struct ofp_action_set_field {
    uint16_t type;                  /* OFPAT_SET_FIELD. */
```



```
    uint16_t len;                   /* Length is padded to 64 bits. */
    /* Followed by:
     *   - Exactly oxm_len bytes containing a single OXM TLV, then
     *   - Exactly ((oxm_len + 4) + 7)/8*8 - (oxm_len + 4) (between 0 and 7)
     *     bytes of all-zero bytes
     */
    uint8_t field[4];               /* OXM TLV - Make compiler happy */
};
OFP_ASSERT(sizeof(struct ofp_action_set_field) == 8);
```

The `field` contains a header field described using a single OXM TLV structure (see 7.2.3). Set-Field actions are defined by `oxm_type`, the type of the OXM TLV, and modify the corresponding header field in the packet with the value of `oxm_value`, the payload of the OXM TLV. The value of `oxm_hasmask` must be zero and no `oxm_mask` is included. The match of the flow entry must contain the OXM prerequisite curresponding to the field to be set (see 7.2.3.6), otherwise an error must be generated (see 6.4).

The type of a set-field action can be any valid OXM header type, the list of possible OXM types are described in Section 7.2.3.7 and Table 12. Set-Field actions for OXM types `OFPXMT_OFB_IN_PORT`, `OXM_OF_IN_PHY_PORT` and `OFPXMT_OFB_METADATA` are not supported, because those are not header fields. The Set-Field action overwrite the header field specified by the OXM type, and perform the necessary CRC recalculation based on the header field. The OXM fields refers to the outermost-possible occurence in the header, unless the field type explictely specifies otherwise, and therefore in general the set-field actions applies to the outermost-possible header (e.g. a "Set VLAN ID" set-field action always sets the ID of the outermost VLAN tag).

An *Experimenter* action uses the following structure and fields:

```
/* Action header for OFPAT_EXPERIMENTER.
 * The rest of the body is experimenter-defined. */
struct ofp_action_experimenter_header {
    uint16_t type;                  /* OFPAT_EXPERIMENTER. */
    uint16_t len;                   /* Length is a multiple of 8. */
    uint32_t experimenter;          /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
};
OFP_ASSERT(sizeof(struct ofp_action_experimenter_header) == 8);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in struct `ofp_experimenter` (see 7.5.4).

7.3 Controller-to-Switch Messages

7.3.1 Handshake

The `OFPT_FEATURES_REQUEST` message is used by the controller to identify the switch and read its basic capabilities. Upon session establishment (see 6.3.1), the controller should send an `OFPT_FEATURES_REQUEST` message. This message does not contain a body beyond the OpenFlow header. The switch must respond with an `OFPT_FEATURES_REPLY` message:


```
/* Switch features. */
struct ofp_switch_features {
    struct ofp_header header;
    uint64_t datapath_id;   /* Datapath unique ID.  The lower 48-bits are for
                               a MAC address, while the upper 16-bits are
                               implementer-defined. */

    uint32_t n_buffers;     /* Max packets buffered at once. */

    uint8_t n_tables;       /* Number of tables supported by datapath. */
    uint8_t auxiliary_id;   /* Identify auxiliary connections */
    uint8_t pad[2];         /* Align to 64-bits. */

    /* Features. */
    uint32_t capabilities;  /* Bitmap of support "ofp_capabilities". */
    uint32_t reserved;
};
OFP_ASSERT(sizeof(struct ofp_switch_features) == 32);
```

The `datapath_id` field uniquely identifies a datapath. The lower 48 bits are intended for the switch MAC address, while the top 16 bits are up to the implementer. An example use of the top 16 bits would be a VLAN ID to distinguish multiple virtual switch instances on a single physical switch. This field should be treated as an opaque bit string by controllers.

The `n_buffers` field specifies the maximum number of packets the switch can buffer when sending packets to the controller using *packet-in* messages (see 6.1.2).

The `n_tables` field describes the number of tables supported by the switch, each of which can have a different set of supported match fields, actions and number of entries. When the controller and switch first communicate, the controller will find out how many tables the switch supports from the Features Reply. If it wishes to understand the size, types, and order in which tables are consulted, the controller sends a `OFPMP_TABLE_FEATURES` multipart request (see 7.3.5.5). A switch must return these tables in the order the packets traverse the tables.

The `auxiliary_id` field identify the type of connection from the switch to the controller, the main connection has this field set to zero, an auxiliary connection has this field set to a non-zero value (see 6.3.5).

The `capabilities` field uses a combination of the following flags:

```
/* Capabilities supported by the datapath. */
enum ofp_capabilities {
    OFPC_FLOW_STATS     = 1 << 0,  /* Flow statistics. */
    OFPC_TABLE_STATS    = 1 << 1,  /* Table statistics. */
    OFPC_PORT_STATS     = 1 << 2,  /* Port statistics. */
    OFPC_GROUP_STATS    = 1 << 3,  /* Group statistics. */
    OFPC_IP_REASM       = 1 << 5,  /* Can reassemble IP fragments. */
    OFPC_QUEUE_STATS    = 1 << 6,  /* Queue statistics. */
    OFPC_PORT_BLOCKED   = 1 << 8   /* Switch will block looping ports. */
};
```

The `OFPC_PORT_BLOCKED` bit indicates that a switch protocol outside of OpenFlow, such as 802.1D Spanning Tree, will detect topology loops and block ports to prevent packet loops. If this bit is not set, in most cases the controller should implement a mechanism to prevent packet loops.

7.3.2 Switch Configuration

The controller is able to set and query configuration parameters in the switch with the OFPT_SET_CONFIG and OFPT_GET_CONFIG_REQUEST messages, respectively. The switch responds to a configuration request with an OFPT_GET_CONFIG_REPLY message; it does not reply to a request to set the configuration.

There is no body for OFPT_GET_CONFIG_REQUEST beyond the OpenFlow header. The OFPT_SET_CONFIG and OFPT_GET_CONFIG_REPLY use the following:

```
/* Switch configuration. */
struct ofp_switch_config {
    struct ofp_header header;
    uint16_t flags;             /* Bitmap of OFPC_* flags. */
    uint16_t miss_send_len;     /* Max bytes of packet that datapath
                                   should send to the controller. See
                                   ofp_controller_max_len for valid values.
                                   */
};
OFP_ASSERT(sizeof(struct ofp_switch_config) == 12);
```

The configuration flags include the following:

```
enum ofp_config_flags {
    /* Handling of IP fragments. */
    OFPC_FRAG_NORMAL   = 0,       /* No special handling for fragments. */
    OFPC_FRAG_DROP     = 1 << 0,  /* Drop fragments. */
    OFPC_FRAG_REASM    = 1 << 1,  /* Reassemble (only if OFPC_IP_REASM set). */
    OFPC_FRAG_MASK     = 3,
};
```

The OFPC_FRAG_* flags indicate whether IP fragments should be treated normally, dropped, or reassembled. "Normal" handling of fragments means that an attempt should be made to pass the fragments through the OpenFlow tables. If any field is not present (e.g., the TCP/UDP ports didn't fit), then the packet should not match any entry that has that field set.

The miss_send_len field defines the number of bytes of each packet sent to the controller by the OpenFlow pipeline when not using an output action to the OFPP_CONTROLLER logical port, for example sending packets with invalid TTL if this message reason is enabled. If this field equals 0, the switch must send zero bytes of the packet in the ofp_packet_in message. If the value is set to OFPCML_NO_BUFFER the complete packet must be included in the message, and should not be buffered.

7.3.3 Flow Table Configuration

Flow tables are numbered from 0 and can take any number until OFPTT_MAX. OFPTT_ALL is a reserved value.


```
/* Table numbering. Tables can use any number up to OFPT_MAX. */
enum ofp_table {
    /* Last usable table number. */
    OFPTT_MAX        = 0xfe,

    /* Fake tables. */
    OFPTT_ALL        = 0xff   /* Wildcard table used for table config,
                                 flow stats and flow deletes. */
};
```

The controller can configure and query table state in the switch with the OFP_TABLE_MOD and OFPMP_TABLE requests, respectively. The switch responds to a table multipart request with a OFPT_MULTIPART_REPLY message. The OFP_TABLE_MOD use the following structure and fields:

```
/* Configure/Modify behavior of a flow table */
struct ofp_table_mod {
    struct ofp_header header;
    uint8_t table_id;       /* ID of the table, OFPTT_ALL indicates all tables */
    uint8_t pad[3];         /* Pad to 32 bits */
    uint32_t config;        /* Bitmap of OFPTC_* flags */
};
OFP_ASSERT(sizeof(struct ofp_table_mod) == 16);
```

The table_id chooses the table to which the configuration change should be applied. If the table_id is OFPTT_ALL, the configuration is applied to all tables in the switch.

The config field is a bitmap that is provided for backward compatibility with earlier version of the specification, it is reserved for future use. There is no flags that are currently defined for that field. The following values are defined for that field :

```
/* Flags to configure the table. Reserved for future use. */
enum ofp_table_config {
    OFPTC_DEPRECATED_MASK       = 3,  /* Deprecated bits */
};
```

7.3.4 Modify State Messages

7.3.4.1 Modify Flow Entry Message

Modifications to a flow table from the controller are done with the OFPT_FLOW_MOD message:

```
/* Flow setup and teardown (controller -> datapath). */
struct ofp_flow_mod {
    struct ofp_header header;
    uint64_t cookie;            /* Opaque controller-issued identifier. */
    uint64_t cookie_mask;       /* Mask used to restrict the cookie bits
                                   that must match when the command is
                                   OFPFC_MODIFY* or OFPFC_DELETE*. A value
                                   of 0 indicates no restriction. */

    /* Flow actions. */
```



```
    uint8_t table_id;            /* ID of the table to put the flow in.
                                    For OFPFC_DELETE_* commands, OFPTT_ALL
                                    can also be used to delete matching
                                    flows from all tables. */
    uint8_t command;             /* One of OFPFC_*. */
    uint16_t idle_timeout;       /* Idle time before discarding (seconds). */
    uint16_t hard_timeout;       /* Max time before discarding (seconds). */
    uint16_t priority;           /* Priority level of flow entry. */
    uint32_t buffer_id;          /* Buffered packet to apply to, or
                                    OFP_NO_BUFFER.
                                    Not meaningful for OFPFC_DELETE*. */
    uint32_t out_port;           /* For OFPFC_DELETE* commands, require
                                    matching entries to include this as an
                                    output port.  A value of OFPP_ANY
                                    indicates no restriction. */
    uint32_t out_group;          /* For OFPFC_DELETE* commands, require
                                    matching entries to include this as an
                                    output group.  A value of OFPG_ANY
                                    indicates no restriction. */
    uint16_t flags;              /* Bitmap of OFPFF_* flags. */
    uint8_t pad[2];
    struct ofp_match match;      /* Fields to match. Variable size. */
    /* The variable size and padded match is always followed by instructions. */
    //struct ofp_instruction instructions[0]; /* Instruction set - 0 or more.
                                                 The length of the instruction
                                                 set is inferred from the
                                                 length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_flow_mod) == 56);
```

The `cookie` field is an opaque data value chosen by the controller. This value appears in flow removed messages and flow statistics, and can also be used to filter flow statistics, flow modification and flow deletion (see 6.4). It is not used by the packet processing pipeline, and thus does not need to reside in hardware. The value -1 (0xffffffffffffffff) is reserved and must not be used. When a flow entry is inserted in a table through an `OFPFC_ADD` message, its `cookie` field is set to the provided value. When a flow entry is modified (`OFPFC_MODIFY` or `OFPFC_MODIFY_STRICT` messages), its `cookie` field is unchanged.

If the `cookie_mask` field is non-zero, it is used with the `cookie` field to restrict flow matching while modifying or deleting flow entries. This field is ignored by `OFPFC_ADD` messages. The `cookie_mask` field's behavior is explained in Section 6.4.

The `table_id` field specifies the table into which the flow entry should be inserted, modified or deleted. Table 0 signifies the first table in the pipeline. The use of `OFPTT_ALL` is only valid for delete requests.

The `command` field must be one of the following:

```
enum ofp_flow_mod_command {
    OFPFC_ADD           = 0, /* New flow. */
    OFPFC_MODIFY        = 1, /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT = 2, /* Modify entry strictly matching wildcards and
                                priority. */
    OFPFC_DELETE        = 3, /* Delete all matching flows. */
    OFPFC_DELETE_STRICT = 4, /* Delete entry strictly matching wildcards and
                                priority. */
};
```


The differences between OFPFC_MODIFY and OFPFC_MODIFY_STRICT are explained in Section 6.4 and differences between OFPFC_DELETE and OFPFC_DELETE_STRICT are explained in Section 6.4.

The idle_timeout and hard_timeout fields control how quickly flow entries expire (see 5.5). When a flow entry is inserted in a table, its idle_timeout and hard_timeout fields are set with the values from the message. When a flow entry is modified (OFPFC_MODIFY or OFPFC_MODIFY_STRICT messages), the idle_timeout and hard_timeout fields are ignored.

If the idle_timeout is set and the hard_timeout is zero, the entry must expire after idle_timeout seconds with no received traffic. If the idle_timeout is zero and the hard_timeout is set, the entry must expire in hard_timeout seconds regardless of whether or not packets are hitting the entry. If both idle_timeout and hard_timeout are set, the flow entry will timeout after idle_timeout seconds with no traffic, or hard_timeout seconds, whichever comes first. If both idle_timeout and hard_timeout are zero, the entry is considered permanent and will never time out. It can still be removed with a flow_mod message of type OFPFC_DELETE.

The priority indicates priority within the specified flow table table. Higher numbers indicate higher priorities. This field is used only for OFPFC_ADD messages when matching and adding flow entries, and for OFPFC_MODIFY_STRICT or OFPFC_DELETE_STRICT messages when matching flow entries.

The buffer_id refers to a packet buffered at the switch and sent to the controller by a *packet-in* message. If no buffered packet is associated with the flow mod, it must be set to OFP_NO_BUFFER. A flow mod that includes a valid buffer_id removes the corresponding packet from the buffer and processes it through the entire OpenFlow pipeline after the flow is inserted, starting at the first flow table. This is effectively equivalent to sending a two-message sequence of a flow mod and a packet-out forwarding to the OFPP_TABLE logical port (see 7.3.7), with the requirement that the switch must fully process the flow mod before the packet out. These semantics apply regardless of the table to which the flow mod refers, or the instructions contained in the flow mod. This field is ignored by OFPFC_DELETE and OFPFC_DELETE_STRICT flow mod messages.

The out_port and out_group fields optionally filter the scope of OFPFC_DELETE and OFPFC_DELETE_STRICT messages by output port and group. If either out_port or out_group contains a value other than OFPP_ANY or OFPG_ANY respectively, it introduces a constraint when matching. This constraint is that the flow entry must contain an output action directed at that port or group. Other constraints such as ofp_match structs and priorities are still used; this is purely an *additional* constraint. Note that to disable output filtering, both out_port and out_group must be set to OFPP_ANY and OFPG_ANY respectively. These fields are ignored by OFPFC_ADD, OFPFC_MODIFY or OFPFC_MODIFY_STRICT messages.

The flags field may include a combination of the following flags:

```
enum ofp_flow_mod_flags {
    OFPFF_SEND_FLOW_REM = 1 << 0,  /* Send flow removed message when flow
                                    * expires or is deleted. */
    OFPFF_CHECK_OVERLAP = 1 << 1,  /* Check for overlapping entries first. */
    OFPFF_RESET_COUNTS  = 1 << 2,  /* Reset flow packet and byte counts. */
    OFPFF_NO_PKT_COUNTS = 1 << 3,  /* Don't keep track of packet count. */
    OFPFF_NO_BYT_COUNTS = 1 << 4,  /* Don't keep track of byte count. */
};
```


When the OFPFF_SEND_FLOW_REM flag is set, the switch must send a flow removed message when the flow entry expires or is deleted.

When the OFPFF_CHECK_OVERLAP flag is set, the switch must check that there are no conflicting entries with the same priority prior to inserting it in the flow table. If there is one, the flow mod fails and an error message is returned (see 6.4).

When the OFPFF_NO_PKT_COUNTS flag is set, the switch does not need to keep track of the flow packet count. When the OFPFF_NO_BYT_COUNTS flag is set, the switch does not need to keep track of the flow byte count. Setting those flags may decrease the processing load on some OpenFlow switches, however those counters may not be available in flow statistics and flow removed messages for this flow entry. A switch is not required to honor those flags and may keep track of a flow count and return it despite the corresponding flag being set. If a switch does not keep track of a flow count, the corresponding counter is not available and must be set to the maximum field value (see 5.8).

When a flow entry is inserted in a table, its flags field is set with the values from the message. When a flow entry is matched and modified (OFPFC_MODIFY or OFPFC_MODIFY_STRICT messages), the flags field is ignored.

The instructions field contains the instruction set for the flow entry when adding or modifying entries. If the instruction set is not valid or supported, the switch must generate an error (see 6.4).

7.3.4.2 Modify Group Entry Message

Modifications to the group table from the controller are done with the OFPT_GROUP_MOD message:

```
/* Group setup and teardown (controller -> datapath). */
struct ofp_group_mod {
    struct ofp_header header;
    uint16_t command;             /* One of OFPGC_*. */
    uint8_t type;                 /* One of OFPGT_*. */
    uint8_t pad;                  /* Pad to 64 bits. */
    uint32_t group_id;            /* Group identifier. */
    struct ofp_bucket buckets[0]; /* The length of the bucket array is inferred
                                     from the length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_group_mod) == 16);
```

The semantics of the type and group fields are explained in Section 6.5.

The command field must be one of the following:

```
/* Group commands */
enum ofp_group_mod_command {
    OFPGC_ADD    = 0,       /* New group. */
    OFPGC_MODIFY = 1,       /* Modify all matching groups. */
    OFPGC_DELETE = 2,       /* Delete all matching groups. */
};
```

The type field must be one of the following:


```
/* Group types.  Values in the range [128, 255] are reserved for experimental
 * use. */
enum ofp_group_type {
    OFPGT_ALL      = 0, /* All (multicast/broadcast) group.  */
    OFPGT_SELECT   = 1, /* Select group. */
    OFPGT_INDIRECT = 2, /* Indirect group. */
    OFPGT_FF       = 3, /* Fast failover group. */
};
```

The `group_id` field uniquely identifies a group within a switch. The following special group identifiers are defined:

```
/* Group numbering. Groups can use any number up to OFPG_MAX. */
enum ofp_group {
    /* Last usable group number. */
    OFPG_MAX        = 0xffffff00,

    /* Fake groups. */
    OFPG_ALL        = 0xfffffffc,  /* Represents all groups for group delete
                                      commands. */
    OFPG_ANY        = 0xffffffff   /* Wildcard group used only for flow stats
                                      requests. Selects all flows regardless of
                                      group (including flows with no group).
                                      */
};
```

The `buckets` field is an array of buckets. For *Indirect* group, the arrays must contain exactly one bucket (see 5.6.1), other group types may have multiple buckets in the array. For *Fast Failover* group, the bucket order does define the bucket priorities (see 5.6.1), and the bucket order can be changed by modifying the group (for example using a `OFPT_GROUP_MOD` message with command `OFPGC_MODIFY`).

Buckets in the array use the following struct:

```
/* Bucket for use in groups. */
struct ofp_bucket {
    uint16_t len;                   /* Length the bucket in bytes, including
                                       this header and any padding to make it
                                       64-bit aligned. */
    uint16_t weight;                /* Relative weight of bucket.  Only
                                       defined for select groups. */
    uint32_t watch_port;            /* Port whose state affects whether this
                                       bucket is live.  Only required for fast
                                       failover groups. */
    uint32_t watch_group;           /* Group whose state affects whether this
                                       bucket is live.  Only required for fast
                                       failover groups. */
    uint8_t pad[4];
    struct ofp_action_header actions[0]; /* 0 or more actions associated with
                                            the bucket - The action list length
                                            is inferred from the length
                                            of the bucket. */
};
OFP_ASSERT(sizeof(struct ofp_bucket) == 16);
```


The weight field is only defined for select groups, and its support is optional. The bucket's share of the traffic processed by the group is defined by the individual bucket's weight divided by the sum of the bucket weights in the group. When a port goes down, the change in traffic distribution is undefined. The precision by which a switch's packet distribution should match bucket weights is undefined.

The watch_port and watch_group fields are only required for fast failover groups, and may be optionally implemented for other group types. These fields indicate the port and/or group whose liveness controls whether this bucket is a candidate for forwarding. For fast failover groups, the first bucket defined is the highest-priority bucket, and only the highest-priority live bucket is used (see 5.6.1).

The actions field is the action set associated with the bucket. When the bucket is selected for a packet, its action set is applied to the packet (see 5.10).

7.3.4.3 Port Modification Message

The controller uses the OFPT_PORT_MOD message to modify the behavior of the port:

```
/* Modify behavior of the physical port */
struct ofp_port_mod {
    struct ofp_header header;
    uint32_t port_no;
    uint8_t pad[4];
    uint8_t hw_addr[OFP_ETH_ALEN]; /* The hardware address is not
                                      configurable.  This is used to
                                      sanity-check the request, so it must
                                      be the same as returned in an
                                      ofp_port struct. */
    uint8_t pad2[2];        /* Pad to 64 bits. */
    uint32_t config;        /* Bitmap of OFPPC_* flags. */
    uint32_t mask;          /* Bitmap of OFPPC_* flags to be changed. */

    uint32_t advertise;     /* Bitmap of OFPPF_*.  Zero all bits to prevent
                               any action taking place. */
    uint8_t pad3[4];        /* Pad to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_port_mod) == 40);
```

The mask field is used to select bits in the config field to change. The advertise field has no mask; all port features change together.

7.3.4.4 Meter Modification Message

Modifications to a meter from the controller are done with the OFPT_METER_MOD message:

```
/* Meter configuration. OFPT_METER_MOD. */
struct ofp_meter_mod {
    struct ofp_header header;
    uint16_t            command;        /* One of OFPMC_*. */
    uint16_t            flags;          /* Bitmap of OFPMF_* flags. */
    uint32_t            meter_id;       /* Meter instance. */
```



```
    struct ofp_meter_band_header bands[0]; /* The band list length is
                                              inferred from the length field
                                              in the header. */
};
OFP_ASSERT(sizeof(struct ofp_meter_mod) == 16);
```

The `meter_id` field uniquely identifies a meter within a switch. Meter are defined starting with `meter_id`=1 up to the maximum number of meters that the switch can support. The OpenFlow protocol also defines some additional virtual meters that can not be associated with flows:

```
/* Meter numbering. Flow meters can use any number up to OFPM_MAX. */
enum ofp_meter {
    /* Last usable meter. */
    OFPM_MAX        = 0xffff0000,

    /* Virtual meters. */
    OFPM_SLOWPATH   = 0xfffffffd,  /* Meter for slow datapath. */
    OFPM_CONTROLLER = 0xfffffffe,  /* Meter for controller connection. */
    OFPM_ALL        = 0xffffffff,  /* Represents all meters for stat requests
                                      commands. */
};
```

Virtual meters are provided to support existing implementations of OpenFlow. New implementations are encouraged to use regular per-flow meters or priority queues instead.

- `OFPM_CONTROLLER`: Virtual meter controlling all packets sent to the controller via *Packet-in* messages, either using the **CONTROLLER** reserved port or in other processing (see 6.1.2). Can be used to limit the amount of traffic sent to the controller.
- `OFPM_SLOWPATH`: Virtual meter controlling all packets processed by the slow datapath of the switch. Many switch implementations have a fast and slow datapath, for example a hardware switch may have a slow software datapath, or a software switch may have a slow userspace datapath.

The `command` field must be one of the following:

```
/* Meter commands */
enum ofp_meter_mod_command {
    OFPMC_ADD,              /* New meter. */
    OFPMC_MODIFY,           /* Modify specified meter. */
    OFPMC_DELETE,           /* Delete specified meter. */
};
```

The `flags` field may include a combination of following flags:

```
/* Meter configuration flags */
enum ofp_meter_flags {
    OFPMF_KBPS    = 1 << 0,     /* Rate value in kb/s (kilo-bit per second). */
    OFPMF_PKTPS   = 1 << 1,     /* Rate value in packet/sec. */
    OFPMF_BURST   = 1 << 2,     /* Do burst size. */
    OFPMF_STATS   = 1 << 3,     /* Collect statistics. */
};
```


The bands field is a list of rate bands. It can contain any number of bands, and each band type can be repeated when it make sense. Only a single band is used at a time, if the current rate of packets exceed the rate of multiple bands, the band with the highest configured rate is used.

All the rate bands are defined using the same common header:

```
/* Common header for all meter bands */
struct ofp_meter_band_header {
    uint16_t        type;    /* One of OFPMBT_*. */
    uint16_t        len;     /* Length in bytes of this band. */
    uint32_t        rate;    /* Rate for this band. */
    uint32_t        burst_size; /* Size of bursts. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_header) == 12);
```

The rate field indicates the rate value above which the corresponding band may apply to packets (see 5.7.1). The rate value is in kilobit per seconds, unless the flags field includes OFPMF_PKTPS, in which case the rate is in packets per seconds.

The burst_size field is used only if the flags field includes OFPMF_BURST. It defines the granularity of the meter, for all packet or byte burst which length is greater than burst value, the meter rate will always be strictly enforced. The burst value is in kilobits, unless the flags field includes OFPMF_PKTPS, in which case the burst value is in packets.

The type field must be one of the following:

```
/* Meter band types */
enum ofp_meter_band_type {
    OFPMBT_DROP            = 1,      /* Drop packet. */
    OFPMBT_DSCP_REMARK     = 2,      /* Remark DSCP in the IP header. */
    OFPMBT_EXPERIMENTER    = 0xFFFF  /* Experimenter meter band. */
};
```

An OpenFlow switch may not support all band types, and may not allow to use all its supported bands on all meters, i.e. some meters may be specialised.

The band OFPMBT_DROP defines a simple rate limiter that drop packets that exceed the band rate value, and uses the following structure:

```
/* OFPMBT_DROP band - drop packets */
struct ofp_meter_band_drop {
    uint16_t        type;    /* OFPMBT_DROP. */
    uint16_t        len;     /* Length in bytes of this band. */
    uint32_t        rate;    /* Rate for dropping packets. */
    uint32_t        burst_size; /* Size of bursts. */
    uint8_t         pad[4];
};
OFP_ASSERT(sizeof(struct ofp_meter_band_drop) == 16);
```

The band OFPMBT_DSCP_REMARK defines a simple DiffServ policer that remark the drop precedence of the DSCP field in the IP header of the packets that exceed the band rate value, and uses the following structure:


```
/* OFPMBT_DSCP_REMARK band - Remark DSCP in the IP header */
struct ofp_meter_band_dscp_remark {
    uint16_t        type;    /* OFPMBT_DSCP_REMARK. */
    uint16_t        len;     /* Length in bytes of this band. */
    uint32_t        rate;    /* Rate for remarking packets. */
    uint32_t        burst_size; /* Size of bursts. */
    uint8_t         prec_level; /* Number of drop precedence level to add. */
    uint8_t         pad[3];
};
OFP_ASSERT(sizeof(struct ofp_meter_band_dscp_remark) == 16);
```

The `prec_level` field indicates by which amount the drop precedence of the packet should be increased if the band is exceeded.

The band `OFPMBT_EXPERIMENTER` is experimenter defined and uses the following structure:

```
/* OFPMBT_EXPERIMENTER band - Write actions in action set */
struct ofp_meter_band_experimenter {
    uint16_t        type;    /* One of OFPMBT_*. */
    uint16_t        len;     /* Length in bytes of this band. */
    uint32_t        rate;    /* Rate for this band. */
    uint32_t        burst_size;   /* Size of bursts. */
    uint32_t        experimenter; /* Experimenter ID which takes the same
                                     form as in struct
                                     ofp_experimenter_header. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_experimenter) == 16);
```

7.3.5 Multipart Messages

Multipart messages are used to encode requests or replies that potentially carry a large amount of data and would not always fit in a single OpenFlow message, which is limited to 64KB. The request or reply is encoded as a sequence of multipart messages with a specific multipart type, and re-assembled by the receiver. Multipart messages are primarily used to request statistics or state information from the switch.

The request is carried in one or more `OFPT_MULTIPART_REQUEST` messages:

```
struct ofp_multipart_request {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPMP_* constants. */
    uint16_t flags;             /* OFPMPF_REQ_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the request. 0 or more bytes. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_request) == 16);


enum ofp_multipart_request_flags {
    OFPMPF_REQ_MORE  = 1 << 0  /* More requests to follow. */
};
```

The switch responds with one or more `OFPT_MULTIPART_REPLY` messages:


```
struct ofp_multipart_reply {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPMP_* constants. */
    uint16_t flags;             /* OFPMPF_REPLY_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the reply. 0 or more bytes. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_reply) == 16);


enum ofp_multipart_reply_flags {
    OFPMPF_REPLY_MORE  = 1 << 0  /* More replies to follow. */
};
```

The only value defined for `flags` in a request and reply is whether more requests/replies will follow this one - this has the value `0x0001`. To ease implementation, the controller is allowed to send requests and the switch is allowed to send replies with no additional entries (i.e. an empty `body`). However, another message must always follow a message with the *more* flag set. A request or reply that spans multiple messages (has one or more messages with the *more* flag set), must use the same multipart `type` and transaction id (xid) for all messages in the message sequence. Messages from a multipart request or reply may be interleaved with other OpenFlow message types, including other multipart requests or replies, but must have distinct transaction ids if multiple unanswered multipart requests or replies are in flight simultaneously. Transaction ids of replies must always match the request that prompted them.

In both the request and response, the `type` field specifies the kind of information being passed and determines how the `body` field is interpreted:

```
enum ofp_multipart_type {
    /* Description of this OpenFlow switch.
     * The request body is empty.
     * The reply body is struct ofp_desc. */
    OFPMP_DESC = 0,

    /* Individual flow statistics.
     * The request body is struct ofp_flow_stats_request.
     * The reply body is an array of struct ofp_flow_stats. */
    OFPMP_FLOW = 1,

    /* Aggregate flow statistics.
     * The request body is struct ofp_aggregate_stats_request.
     * The reply body is struct ofp_aggregate_stats_reply. */
    OFPMP_AGGREGATE = 2,

    /* Flow table statistics.
     * The request body is empty.
     * The reply body is an array of struct ofp_table_stats. */
    OFPMP_TABLE = 3,

    /* Port statistics.
     * The request body is struct ofp_port_stats_request.
     * The reply body is an array of struct ofp_port_stats. */
    OFPMP_PORT_STATS = 4,
```



```
    /* Queue statistics for a port
     * The request body is struct ofp_queue_stats_request.
     * The reply body is an array of struct ofp_queue_stats */
    OFPMP_QUEUE = 5,

    /* Group counter statistics.
     * The request body is struct ofp_group_stats_request.
     * The reply is an array of struct ofp_group_stats. */
    OFPMP_GROUP = 6,

    /* Group description.
     * The request body is empty.
     * The reply body is an array of struct ofp_group_desc. */
    OFPMP_GROUP_DESC = 7,

    /* Group features.
     * The request body is empty.
     * The reply body is struct ofp_group_features. */
    OFPMP_GROUP_FEATURES = 8,

    /* Meter statistics.
     * The request body is struct ofp_meter_multipart_requests.
     * The reply body is an array of struct ofp_meter_stats. */
    OFPMP_METER = 9,

    /* Meter configuration.
     * The request body is struct ofp_meter_multipart_requests.
     * The reply body is an array of struct ofp_meter_config. */
    OFPMP_METER_CONFIG = 10,

    /* Meter features.
     * The request body is empty.
     * The reply body is struct ofp_meter_features. */
    OFPMP_METER_FEATURES = 11,

    /* Table features.
     * The request body is either empty or contains an array of
     * struct ofp_table_features containing the controller's
     * desired view of the switch. If the switch is unable to
     * set the specified view an error is returned.
     * The reply body is an array of struct ofp_table_features. */
    OFPMP_TABLE_FEATURES = 12,

    /* Port description.
     * The request body is empty.
     * The reply body is an array of struct ofp_port. */
    OFPMP_PORT_DESC = 13,

    /* Experimenter extension.
     * The request and reply bodies begin with
     * struct ofp_experimenter_multipart_header.
     * The request and reply bodies are otherwise experimenter-defined. */
    OFPMP_EXPERIMENTER = 0xffff
};
```

If a multipart request spans multiple messages and grows to a size that the switch is unable to buffer, the switch must respond with an error messsage of type `OFPET_BAD_REQUEST` and code

OFPBRC_MULTIPART_BUFFER_OVERFLOW. If a multipart request contains a type that is not supported, the switch must respond with an error messsage of type OFPET_BAD_REQUEST and code OFPBRC_BAD_MULTIPART.

In all types of multipart reply containing statistics, if a specific numeric counter is not available in the switch, its value must be set to the maximum field value (the unsigned equivalent of -1). Counters are unsigned and wrap around with no overflow indicator.

7.3.5.1 Description

Information about the switch manufacturer, hardware revision, software revision, serial number, and a description field is available from the OFPMP_DESC multipart request type:

```
/* Body of reply to OFPMP_DESC request.  Each entry is a NULL-terminated
 * ASCII string. */
struct ofp_desc {
    char mfr_desc[DESC_STR_LEN];       /* Manufacturer description. */
    char hw_desc[DESC_STR_LEN];        /* Hardware description. */
    char sw_desc[DESC_STR_LEN];        /* Software description. */
    char serial_num[SERIAL_NUM_LEN];   /* Serial number. */
    char dp_desc[DESC_STR_LEN];        /* Human readable description of datapath. */
};
OFP_ASSERT(sizeof(struct ofp_desc) == 1056);
```

Each entry is ASCII formatted and padded on the right with null bytes (\0). DESC_STR_LEN is 256 and SERIAL_NUM_LEN is 32. The dp_desc field is a free-form string to describe the datapath for debugging purposes, e.g., "switch3 in room 3120". As such, it is not guaranteed to be unique and should not be used as the primary identifier for the datapath—use the datapath_id field from the switch features instead (see 7.3.1).

7.3.5.2 Individual Flow Statistics

Information about individual flow entries is requested with the OFPMP_FLOW multipart request type:

```
/* Body for ofp_multipart_request of type OFPMP_FLOW. */
struct ofp_flow_stats_request {
    uint8_t table_id;         /* ID of table to read (from ofp_table_stats),
                                 OFPTT_ALL for all tables. */
    uint8_t pad[3];           /* Align to 32 bits. */
    uint32_t out_port;        /* Require matching entries to include this
                                 as an output port.  A value of OFPP_ANY
                                 indicates no restriction. */
    uint32_t out_group;       /* Require matching entries to include this
                                 as an output group.  A value of OFPG_ANY
                                 indicates no restriction. */
    uint8_t pad2[4];          /* Align to 64 bits. */
    uint64_t cookie;          /* Require matching entries to contain this
                                 cookie value */
    uint64_t cookie_mask;     /* Mask used to restrict the cookie bits that
                                 must match. A value of 0 indicates
```



```
                                no restriction. */
    struct ofp_match match;   /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 40);
```

The `match` field contains a description of the flow entries that should be matched and may contain wildcarded and masked fields. This field's matching behavior is described in Section 6.4.

The `table_id` field indicates the index of a single table to read, or `OFPTT_ALL` for all tables.

The `out_port` and `out_group` fields optionally filter by output port and group. If either `out_port` or `out_group` contain a value other than `OFPP_ANY` and `OFPG_ANY` respectively, it introduces a constraint when matching. This constraint is that the flow entry must contain an output action directed at that port or group. Other constraints such as `ofp_match` structs are still used; this is purely an *additional* constraint. Note that to disable output filtering, both `out_port` and `out_group` must be set to `OFPP_ANY` and `OFPG_ANY` respectively.

The usage of the `cookie` and `cookie_mask` fields is defined in Section 6.4.

The `body` of the reply to a `OFPMP_FLOW` multipart request consists of an array of the following:

```
/* Body of reply to OFPMP_FLOW request. */
struct ofp_flow_stats {
    uint16_t length;          /* Length of this entry. */
    uint8_t table_id;         /* ID of table flow came from. */
    uint8_t pad;
    uint32_t duration_sec;    /* Time flow has been alive in seconds. */
    uint32_t duration_nsec;   /* Time flow has been alive in nanoseconds beyond
                                 duration_sec. */
    uint16_t priority;        /* Priority of the entry. */
    uint16_t idle_timeout;    /* Number of seconds idle before expiration. */
    uint16_t hard_timeout;    /* Number of seconds before expiration. */
    uint16_t flags;           /* Bitmap of OFPFF_* flags. */
    uint8_t pad2[4];          /* Align to 64-bits. */
    uint64_t cookie;          /* Opaque controller-issued identifier. */
    uint64_t packet_count;    /* Number of packets in flow. */
    uint64_t byte_count;      /* Number of bytes in flow. */
    struct ofp_match match;   /* Description of fields. Variable size. */
    /* The variable size and padded match is always followed by instructions. */
    //struct ofp_instruction instructions[0]; /* Instruction set - 0 or more. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 56);
```

The fields consist of those provided in the `flow_mod` that created the flow entry (see 7.3.4.1), plus the `table_id` into which the entry was inserted, the `packet_count`, and the `byte_count` counting all packets processed by the flow entry.

The `duration_sec` and `duration_nsec` fields indicate the elapsed time the flow entry has been installed in the switch. The total duration in nanoseconds can be computed as $duration\_sec \times 10^9 + duration\_nsec$. Implementations are required to provide second precision; higher precision is encouraged where available.

7.3.5.3 Aggregate Flow Statistics

Aggregate information about multiple flow entries is requested with the OFPMP_AGGREGATE multipart request type:

```
/* Body for ofp_multipart_request of type OFPMP_AGGREGATE. */
struct ofp_aggregate_stats_request {
    uint8_t table_id;         /* ID of table to read (from ofp_table_stats)
                                 OFPTT_ALL for all tables. */
    uint8_t pad[3];           /* Align to 32 bits. */
    uint32_t out_port;        /* Require matching entries to include this
                                 as an output port.  A value of OFPP_ANY
                                 indicates no restriction. */
    uint32_t out_group;       /* Require matching entries to include this
                                 as an output group.  A value of OFPG_ANY
                                 indicates no restriction. */
    uint8_t pad2[4];          /* Align to 64 bits. */
    uint64_t cookie;          /* Require matching entries to contain this
                                 cookie value */
    uint64_t cookie_mask;     /* Mask used to restrict the cookie bits that
                                 must match. A value of 0 indicates
                                 no restriction. */
    struct ofp_match match;   /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_request) == 40);
```

The fields in this message have the same meanings as in the individual flow stats request type (OFPMP_FLOW).

The body of the reply consists of the following:

```
/* Body of reply to OFPMP_AGGREGATE request. */
struct ofp_aggregate_stats_reply {
    uint64_t packet_count;    /* Number of packets in flows. */
    uint64_t byte_count;      /* Number of bytes in flows. */
    uint32_t flow_count;      /* Number of flows. */
    uint8_t pad[4];           /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_reply) == 24);
```

7.3.5.4 Table Statistics

Information about tables is requested with the OFPMP_TABLE multipart request type. The request does not contain any data in the body.

The body of the reply consists of an array of the following:

```
/* Body of reply to OFPMP_TABLE request. */
struct ofp_table_stats {
    uint8_t table_id;        /* Identifier of table.  Lower numbered tables
                                are consulted first. */
    uint8_t pad[3];          /* Align to 32-bits. */
```



```
    uint32_t active_count;   /* Number of active entries. */
    uint64_t lookup_count;   /* Number of packets looked up in table. */
    uint64_t matched_count;  /* Number of packets that hit table. */
};
OFP_ASSERT(sizeof(struct ofp_table_stats) == 24);
```

The array has one structure for each table supported by the switch. The entries are returned in the order that packets traverse the tables.

7.3.5.5 Table Features

The `OFPMP_TABLE_FEATURES` multipart type allows a controller to both query for the capabilities of existing tables, and to optionally ask the switch to reconfigure its tables to match a supplied configuration. In general, the table feature capabilities represents all possible features of a table, however some features may be mutually exclusive and the current capabilities structures do not allow us to represent such exclusions.

7.3.5.5.1 Table Features request and reply

If the `OFPMP_TABLE_FEATURES` request body is empty the switch will return an array of struct `ofp_table_features` containing the capabilities of the currently configured flow tables.

If the request body contains an array of one or more `ofp_table_features` structs, the switch will attempt to change its flow tables to match the requested flow table configuration. This operation configures the entire pipeline, and the set of flow tables in the pipeline must match the set in the request, or an error must be returned. In particular, if the requested configuration does not contain an `ofp_table_features` struct for one or more flow tables that the switch supports, these flow tables are to be removed from the pipeline if the configuration is successfully set. A successful configuration change will modify the features for all flow tables in the request, that is, either all the flow tables specified in the request are modified or none, and the new capabilities for each flow table must be either a superset of, or equal to the requested capabilities. If the flow table configuration is successful, flow entries from flow tables that have been removed or flow tables that had their capabilities change between the prior and new configuration are removed from the flow table, however no `ofp_flow_removed` messages are sent. The switch then replies with the new configuration. If the switch is unable to set the requested configuration, an error of type `OFPET_TABLE_FEATURES_FAILED` is returned with the appropriate error code.

Requests and replies containing `ofp_table_features` are expected to meet the following minimum requirements:

- Each `ofp_table_features` struct's `table_id` field value should be unique amongst all `ofp_table_features` structs in the message
- Each `ofp_table_features` struct's `properties` field must contain exactly one of each of the `ofp_table_feature_prop_type` properties, with two exceptions. First, properties with the `_MISS` suffix may be ommited if it is the same as the corresponding property for regular flow entries. Second, properties of type `OFPTFPT_EXPERIMENTER` and `OFPTFPT_EXPERIMENTER_MISS` may be ommited or included many times. Ordering is unspecified, but implementers are encouraged to use the ordering listed in the specification (see 7.3.5.5.2).

A switch receiving a request that does not meet these requirements should return an error of type OFPET_TABLE_FEATURES_FAILED with the appropriate error code.

The following structure describes the body of the table features request and reply:

```
/* Body for ofp_multipart_request of type OFPMP_TABLE_FEATURES./
 * Body of reply to OFPMP_TABLE_FEATURES request. */
struct ofp_table_features {
    uint16_t length;         /* Length is padded to 64 bits. */
    uint8_t table_id;        /* Identifier of table.  Lower numbered tables
                                are consulted first. */
    uint8_t pad[5];          /* Align to 64-bits. */
    char name[OFP_MAX_TABLE_NAME_LEN];
    uint64_t metadata_match; /* Bits of metadata table can match. */
    uint64_t metadata_write; /* Bits of metadata table can write. */
    uint32_t config;         /* Bitmap of OFPTC_* values */
    uint32_t max_entries;    /* Max number of entries supported. */

    /* Table Feature Property list */
    struct ofp_table_feature_prop_header properties[0]; /* List of properties */
};
OFP_ASSERT(sizeof(struct ofp_table_features) == 64);
```

The array has one structure for each flow table supported by the switch. The entries are always returned in the order that packets traverse the flow tables. OFP_MAX_TABLE_NAME_LEN is 32 .

The metadata_match field indicates the bits of the metadata field that the table can match on, when using the metadata field of struct ofp_match. A value of 0xFFFFFFFFFFFFFFFF indicates that the table can match the full metadata field.

The metadata_write field indicates the bits of the metadata field that the table can write using the OFPIT_WRITE_METADATA instruction. A value of 0xFFFFFFFFFFFFFFFF indicates that the table can write the full metadata field.

The config field is the table configuration that was set on the table via a table configuration message (see 7.3.3).

The max_entries field describes the maximum number of flow entries that can be inserted into that table. Due to limitations imposed by modern hardware, the max_entries value should be considered advisory and a best effort approximation of the capacity of the table. Despite the high-level abstraction of a table, in practice the resource consumed by a single flow table entry is not constant. For example, a flow table entry might consume more than one entry, depending on its match parameters (e.g., IPv4 vs. IPv6). Also, tables that appear distinct at an OpenFlow-level might in fact share the same underlying physical resources. Further, on OpenFlow hybrid switches, those table may be shared with non-OpenFlow functions. The result is that switch implementers should report an approximation of the total flow entries supported and controller writers should not treat this value as a fixed, physical constant.

The properties field is a list of table feature properties, describing various capabilities of the table.

7.3.5.5.2 Table Features properties

The list of table feature property types that are currently defined are:

```
/* Table Feature property types.
 * Low order bit cleared indicates a property for a regular Flow Entry.
 * Low order bit set indicates a property for the Table-Miss Flow Entry.
 */
enum ofp_table_feature_prop_type {
    OFPTFPT_INSTRUCTIONS           = 0,  /* Instructions property. */
    OFPTFPT_INSTRUCTIONS_MISS      = 1,  /* Instructions for table-miss. */
    OFPTFPT_NEXT_TABLES            = 2,  /* Next Table property. */
    OFPTFPT_NEXT_TABLES_MISS       = 3,  /* Next Table for table-miss. */
    OFPTFPT_WRITE_ACTIONS          = 4,  /* Write Actions property. */
    OFPTFPT_WRITE_ACTIONS_MISS     = 5,  /* Write Actions for table-miss. */
    OFPTFPT_APPLY_ACTIONS          = 6,  /* Apply Actions property. */
    OFPTFPT_APPLY_ACTIONS_MISS     = 7,  /* Apply Actions for table-miss. */
    OFPTFPT_MATCH                  = 8,  /* Match property. */
    OFPTFPT_WILDCARDS              = 10, /* Wildcards property. */
    OFPTFPT_WRITE_SETFIELD         = 12, /* Write Set-Field property. */
    OFPTFPT_WRITE_SETFIELD_MISS    = 13, /* Write Set-Field for table-miss. */
    OFPTFPT_APPLY_SETFIELD         = 14, /* Apply Set-Field property. */
    OFPTFPT_APPLY_SETFIELD_MISS    = 15, /* Apply Set-Field for table-miss. */
    OFPTFPT_EXPERIMENTER           = 0xFFFE, /* Experimenter property. */
    OFPTFPT_EXPERIMENTER_MISS      = 0xFFFF, /* Experimenter for table-miss. */
};
```

The properties with the `_MISS` suffix describe the capabilities for the table-miss flow entry (see 5.4), whereas other properties describe the capabilities for regular flow entry. If a specific property does not have any capability (for example no Set-Field support), a property with an empty list must be included in the property list. When a property of the table-miss flow entry is the same as the corresponding property for regular flow entries (i.e. both properties have the same list of capabilities), this table-miss property can be omited from the property list.

A property definition contains the property type, length, and any associated data:

```
/* Common header for all Table Feature Properties */
struct ofp_table_feature_prop_header {
    uint16_t         type;    /* One of OFPTFPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_header) == 4);
```

The `OFPTFPT_INSTRUCTIONS` and `OFPTFPT_INSTRUCTIONS_MISS` properties use the following structure and fields:

```
/* Instructions property */
struct ofp_table_feature_prop_instructions {
    uint16_t         type;    /* One of OFPTFPT_INSTRUCTIONS,
                                 OFPTFPT_INSTRUCTIONS_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the instruction ids, then
```



```
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    struct ofp_instruction   instruction_ids[0];   /* List of instructions */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_instructions) == 4);
```

The instruction_ids is the list of instructions supported by this table (see 5.9). The elements of that list are variable size to enable expressing experimenter instructions, non-experimenter instructions are 4 bytes.

The OFPTFPT_NEXT_TABLES and OFPTFPT_NEXT_TABLES_MISS properties use the following structure and fields:

```
/* Next Tables property */
struct ofp_table_feature_prop_next_tables {
    uint16_t         type;    /* One of OFPTFPT_NEXT_TABLES,
                                 OFPTFPT_NEXT_TABLES_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the table_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint8_t          next_table_ids[0];        /* List of table ids. */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_next_tables) == 4);
```

The next_table_ids is the array of tables that can be directly reached from the present table using the OFPIT_GOTO_TABLE instruction (see 5.1).

The OFPTFPT_WRITE_ACTIONS, OFPTFPT_WRITE_ACTIONS_MISS, OFPTFPT_APPLY_ACTIONS and OFPTFPT_APPLY_ACTIONS_MISS properties use the following structure and fields:

```
/* Actions property */
struct ofp_table_feature_prop_actions {
    uint16_t         type;    /* One of OFPTFPT_WRITE_ACTIONS,
                                 OFPTFPT_WRITE_ACTIONS_MISS,
                                 OFPTFPT_APPLY_ACTIONS,
                                 OFPTFPT_APPLY_ACTIONS_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the action_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    struct ofp_action_header  action_ids[0];      /* List of actions */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_actions) == 4);
```

The action_ids is the list of actions for the feature (see 5.12). The elements of that list are variable size to enable expressing experimenter actions, non-experimenter actions are 4 bytes. The OFPTFPT_WRITE_ACTIONS and OFPTFPT_WRITE_ACTIONS_MISS properties describe actions supported by the table using the OFPIT_WRITE_ACTIONS instruction, whereas the OFPTFPT_APPLY_ACTIONS

and OFPTFPT_APPLY_ACTIONS_MISS properties describe actions supported by the table using the OFPIT_APPLY_ACTIONS instruction.

The OFPTFPT_MATCH, OFPTFPT_WILDCARDS, OFPTFPT_WRITE_SETFIELD, OFPTFPT_WRITE_SETFIELD_MISS, OFPTFPT_APPLY_SETFIELD and OFPTFPT_APPLY_SETFIELD_MISS properties use the following structure and fields:

```
/* Match, Wildcard or Set-Field property */
struct ofp_table_feature_prop_oxm {
    uint16_t         type;    /* One of OFPTFPT_MATCH,
                                 OFPTFPT_WILDCARDS,
                                 OFPTFPT_WRITE_SETFIELD,
                                 OFPTFPT_WRITE_SETFIELD_MISS,
                                 OFPTFPT_APPLY_SETFIELD,
                                 OFPTFPT_APPLY_SETFIELD_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the oxm_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         oxm_ids[0];   /* Array of OXM headers */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_oxm) == 4);
```

The oxm_ids is the list of OXM types for the feature (see 7.2.3.2). The elements of that list are 32-bit OXM headers or 64-bit OXM headers for experimenter OXM fields.

The OFPTFPT_MATCH property indicates the fields for which that particular table supports matching on (see 7.2.3.7). For example, if the table can match the ingress port, an OXM header with the type OXM_OF_IN_PORT should be included in the list. If the HASMASK bit is set on the OXM header then the switch must support masking for the given type. The OFPTFPT_WILDCARDS property indicates the fields for which that particular table supports wildcarding (omiting). For example, a direct look-up hash table would have that list empty, while a TCAM or sequentially searched table would have it set to the same value as the OFPTFPT_MATCH property.

The OFPTFPT_WRITE_SETFIELD and OFPTFPT_WRITE_SETFIELD_MISS properties describe Set-Field action types supported by the table using the OFPIT_WRITE_ACTIONS instruction, whereas the OFPTFPT_APPLY_SETFIELD and OFPTFPT_APPLY_SETFIELD_MISS properties describe Set-Field action types supported by the table using the OFPIT_APPLY_ACTIONS instruction.

All fields in ofp_table_features may be requested to be changed by the controller with the exception of the max_entries field, this is read only and returned by the switch.

The OFPTFPT_APPLY_ACTIONS, OFPTFPT_APPLY_ACTIONS_MISS, OFPTFPT_APPLY_SETFIELD, and OFPTFPT_APPLY_SETFIELD_MISS properties contain actions and fields the table is capable of applying. For each of these lists, if an element is present it means the table is at least capable of applying the element in isolation one time. There is currently no way to indicate which elements can be applied together, in which order, and how many time an element can be applied in a single flow entry.

The OFPTFPT_EXPERIMENTER and OFPTFPT_EXPERIMENTER_MISS properties uses the following structure and fields:


```
/* Experimenter table feature property */
struct ofp_table_feature_prop_experimenter {
    uint16_t         type;    /* One of OFPTFPT_EXPERIMENTER,
                                 OFPTFPT_EXPERIMENTER_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_experimenter) == 12);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in struct `ofp_experimenter` (see 7.5.4).

7.3.5.6 Port Statistics

Information about ports statistics is requested with the `OFPMP_PORT_STATS` multipart request type:

```
/* Body for ofp_multipart_request of type OFPMP_PORT. */
struct ofp_port_stats_request {
    uint32_t port_no;        /* OFPMP_PORT message must request statistics
                              * either for a single port (specified in
                              * port_no) or for all ports (if port_no ==
                              * OFPP_ANY). */
    uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_port_stats_request) == 8);
```

The `port_no` field optionally filters the stats request to the given port. To request all port statistics, `port_no` must be set to `OFPP_ANY`.

The `body` of the reply consists of an array of the following:

```
/* Body of reply to OFPMP_PORT request. If a counter is unsupported, set
 * the field to all ones. */
struct ofp_port_stats {
    uint32_t port_no;
    uint8_t pad[4];          /* Align to 64-bits. */
    uint64_t rx_packets;     /* Number of received packets. */
    uint64_t tx_packets;     /* Number of transmitted packets. */
    uint64_t rx_bytes;       /* Number of received bytes. */
    uint64_t tx_bytes;       /* Number of transmitted bytes. */
    uint64_t rx_dropped;     /* Number of packets dropped by RX. */
    uint64_t tx_dropped;     /* Number of packets dropped by TX. */
    uint64_t rx_errors;      /* Number of receive errors.  This is a super-set
                                of more specific receive errors and should be
```



```
                                greater than or equal to the sum of all
                                rx_*_err values. */
    uint64_t tx_errors;      /* Number of transmit errors.  This is a super-set
                                of more specific transmit errors and should be
                                greater than or equal to the sum of all
                                tx_*_err values (none currently defined.) */
    uint64_t rx_frame_err;   /* Number of frame alignment errors. */
    uint64_t rx_over_err;    /* Number of packets with RX overrun. */
    uint64_t rx_crc_err;     /* Number of CRC errors. */
    uint64_t collisions;     /* Number of collisions. */
    uint32_t duration_sec;   /* Time port has been alive in seconds. */
    uint32_t duration_nsec;  /* Time port has been alive in nanoseconds beyond
                                duration_sec. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats) == 112);
```

The `duration_sec` and `duration_nsec` fields indicate the elapsed time the port has been configured into the OpenFlow pipeline. The total duration in nanoseconds can be computed as $duration\_sec \times 10^9 + duration\_nsec$. Implementations are required to provide second precision; higher precision is encouraged where available.

7.3.5.7 Port Description

The port description request `OFPMP_PORT_DESCRIPTION` enables the controller to get a description of all the ports in the system that support OpenFlow. The request body is empty. The reply body consists of an array of the following:

```
/* Description of a port */
struct ofp_port {
    uint32_t port_no;
    uint8_t pad[4];
    uint8_t hw_addr[OFP_ETH_ALEN];
    uint8_t pad2[2];                  /* Align to 64 bits. */
    char name[OFP_MAX_PORT_NAME_LEN]; /* Null-terminated */

    uint32_t config;        /* Bitmap of OFPPC_* flags. */
    uint32_t state;         /* Bitmap of OFPPS_* flags. */

    /* Bitmaps of OFPPF_* that describe features.  All bits zeroed if
     * unsupported or unavailable. */
    uint32_t curr;          /* Current features. */
    uint32_t advertised;    /* Features being advertised by the port. */
    uint32_t supported;     /* Features supported by the port. */
    uint32_t peer;          /* Features advertised by peer. */

    uint32_t curr_speed;    /* Current port bitrate in kbps. */
    uint32_t max_speed;     /* Max port bitrate in kbps */
};
OFP_ASSERT(sizeof(struct ofp_port) == 64);
```

This structure is described in section 7.2.1.

7.3.5.8 Queue Statistics

The OFPMP_QUEUE multipart request message provides queue statistics for one or more ports and one or more queues. The request body contains a port_no field identifying the OpenFlow port for which statistics are requested, or OFPP_ANY to refer to all ports. The queue_id field identifies one of the priority queues, or OFPQ_ALL to refer to all queues configured at the specified port. OFPQ_ALL is 0xffffffff.

```
struct ofp_queue_stats_request {
    uint32_t port_no;        /* All ports if OFPP_ANY. */
    uint32_t queue_id;       /* All queues if OFPQ_ALL. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats_request) == 8);
```

The body of the reply consists of an array of the following structure:

```
struct ofp_queue_stats {
    uint32_t port_no;
    uint32_t queue_id;       /* Queue i.d */
    uint64_t tx_bytes;       /* Number of transmitted bytes. */
    uint64_t tx_packets;     /* Number of transmitted packets. */
    uint64_t tx_errors;      /* Number of packets dropped due to overrun. */
    uint32_t duration_sec;   /* Time queue has been alive in seconds. */
    uint32_t duration_nsec;  /* Time queue has been alive in nanoseconds beyond
                                duration_sec. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats) == 40);
```

The duration_sec and duration_nsec fields indicate the elapsed time the queue has been installed in the switch. The total duration in nanoseconds can be computed as $duration\_sec \times 10^9 + duration\_nsec$. Implementations are required to provide second precision; higher precision is encouraged where available.

7.3.5.9 Group Statistics

The OFPMP_GROUP multipart request message provides statistics for one or more groups. The request body consists of a group_id field, which can be set to OFPG_ALL to refer to all groups on the switch.

```
/* Body of OFPMP_GROUP request. */
struct ofp_group_stats_request {
    uint32_t group_id;       /* All groups if OFPG_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_group_stats_request) == 8);
```

The body of the reply consists of an array of the following structure:


```
/* Body of reply to OFPMP_GROUP request. */
struct ofp_group_stats {
    uint16_t length;         /* Length of this entry. */
    uint8_t pad[2];          /* Align to 64 bits. */
    uint32_t group_id;       /* Group identifier. */
    uint32_t ref_count;      /* Number of flows or groups that directly forward
                                to this group. */
    uint8_t pad2[4];         /* Align to 64 bits. */
    uint64_t packet_count;   /* Number of packets processed by group. */
    uint64_t byte_count;     /* Number of bytes processed by group. */
    uint32_t duration_sec;   /* Time group has been alive in seconds. */
    uint32_t duration_nsec;  /* Time group has been alive in nanoseconds beyond
                                duration_sec. */
    struct ofp_bucket_counter bucket_stats[0]; /* One counter set per bucket. */
};
OFP_ASSERT(sizeof(struct ofp_group_stats) == 40);
```

The fields consist of those provided in the `group_mod` that created the group, plus the `ref_count` counting the number of flows referencing directly the group, the `packet_count`, and the `byte_count` counting all packets processed by the group.

The `duration_sec` and `duration_nsec` fields indicate the elapsed time the group has been installed in the switch. The total duration in nanoseconds can be computed as $duration\_sec \times 10^9 + duration\_nsec$. Implementations are required to provide second precision; higher precision is encouraged where available.

The `bucket_stats` field consists of an array of `ofp_bucket_counter` structs:

```
/* Used in group stats replies. */
struct ofp_bucket_counter {
    uint64_t packet_count;   /* Number of packets processed by bucket. */
    uint64_t byte_count;     /* Number of bytes processed by bucket. */
};
OFP_ASSERT(sizeof(struct ofp_bucket_counter) == 16);
```

7.3.5.10 Group Description

The `OFPMP_GROUP_DESC` multipart request message provides a way to list the set of groups on a switch, along with their corresponding bucket actions. The request body is empty, while the reply body is an array of the following structure:

```
/* Body of reply to OFPMP_GROUP_DESC request. */
struct ofp_group_desc {
    uint16_t length;              /* Length of this entry. */
    uint8_t type;                 /* One of OFPGT_*. */
    uint8_t pad;                  /* Pad to 64 bits. */
    uint32_t group_id;            /* Group identifier. */
    struct ofp_bucket buckets[0];   /* List of buckets - 0 or more. */
};
OFP_ASSERT(sizeof(struct ofp_group_desc) == 8);
```

Fields for group description are the same as those used with the `ofp_group_mod` struct (see 7.3.4.2).

7.3.5.11 Group Features

The OFPMP_GROUP_FEATURES multipart request message provides a way to list the capabilities of groups on a switch. The request body is empty, while the reply body is the following structure:

```
/* Body of reply to OFPMP_GROUP_FEATURES request. Group features. */
struct ofp_group_features {
    uint32_t  types;           /* Bitmap of OFPGT_* values supported. */
    uint32_t  capabilities;    /* Bitmap of OFPGFC_* capability supported. */
    uint32_t  max_groups[4];   /* Maximum number of groups for each type. */
    uint32_t  actions[4];      /* Bitmaps of OFPAT_* that are supported. */
};
OFP_ASSERT(sizeof(struct ofp_group_features) == 40);
```

The max_groups field is the maximum number of groups for each type of groups. The actions is the supported actions for each type of groups. The capabilities uses a combination of the following flags:

```
/* Group configuration flags */
enum ofp_group_capabilities {
    OFPGFC_SELECT_WEIGHT   = 1 << 0,  /* Support weight for select groups */
    OFPGFC_SELECT_LIVENESS = 1 << 1,  /* Support liveness for select groups */
    OFPGFC_CHAINING        = 1 << 2,  /* Support chaining groups */
    OFPGFC_CHAINING_CHECKS = 1 << 3,  /* Check chaining for loops and delete */
};
```

7.3.5.12 Meter Statistics

The OFPMT_METER stats request message provides statistics for one or more meters. The request body consists of a meter_id field, which can be set to OFPM_ALL to refer to all meters on the switch.

```
/* Body of OFPMP_METER and OFPMP_METER_CONFIG requests. */
struct ofp_meter_multipart_request {
    uint32_t meter_id;       /* Meter instance, or OFPM_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_meter_multipart_request) == 8);
```

The body of the reply consists of an array of the following structure:

```
/* Body of reply to OFPMP_METER request. Meter statistics. */
struct ofp_meter_stats {
    uint32_t        meter_id;         /* Meter instance. */
    uint16_t        len;              /* Length in bytes of this stats. */
    uint8_t         pad[6];
    uint32_t        flow_count;       /* Number of flows bound to meter. */
    uint64_t        packet_in_count;  /* Number of packets in input. */
    uint64_t        byte_in_count;    /* Number of bytes in input. */
    uint32_t   duration_sec;  /* Time meter has been alive in seconds. */
    uint32_t   duration_nsec; /* Time meter has been alive in nanoseconds beyond
```



```
                                duration_sec. */
    struct ofp_meter_band_stats band_stats[0]; /* The band_stats length is
                                          inferred from the length field. */
};
OFP_ASSERT(sizeof(struct ofp_meter_stats) == 40);
```

The `packet_in_count`, and the `byte_in_count` count all packets processed by the meter.

The `duration_sec` and `duration_nsec` fields indicate the elapsed time the meter has been installed in the switch. The total duration in nanoseconds can be computed as $duration\_sec \times 10^9 + duration\_nsec$. Implementations are required to provide second precision; higher precision is encouraged where available.

The `band_stats` field consists of an array of `ofp_meter_band_stats` structs:

```
/* Statistics for each meter band */
struct ofp_meter_band_stats {
    uint64_t        packet_band_count;   /* Number of packets in band. */
    uint64_t        byte_band_count;     /* Number of bytes in band. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_stats) == 16);
```

The `packet_band_count`, and the `byte_band_count` count all packets processed by the band.

The order of the band statistics must be the same as in the `OFPMT_METER_CONFIG` stats reply.

7.3.5.13 Meter Configuration Statistics

The `OFPMT_METER_CONFIG` stats request message provides configuration for one or more meter. The request body consists of a `meter_id` field, which can be set to `OFPM_ALL` to refer to all meters on the switch.

```
/* Body of OFPMP_METER and OFPMP_METER_CONFIG requests. */
struct ofp_meter_multipart_request {
    uint32_t meter_id;       /* Meter instance, or OFPM_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_meter_multipart_request) == 8);
```

The body of the reply consists of an array of the following structure:

```
/* Body of reply to OFPMP_METER_CONFIG request. Meter configuration. */
struct ofp_meter_config {
    uint16_t        length;           /* Length of this entry. */
    uint16_t        flags;            /* All OFPMC_* that apply. */
    uint32_t        meter_id;         /* Meter instance. */
    struct ofp_meter_band_header bands[0]; /* The bands length is
                                          inferred from the length field. */
};
OFP_ASSERT(sizeof(struct ofp_meter_config) == 8);
```

The fields are the same fields used for configuring the meter (see 7.3.4.4).

7.3.5.14 Meter Features Statistics

The OFPMT_METER_FEATURES stats request message provides the set of features of the metering subsystem. The request body is empty, and the body of the reply consists of the following structure:

```
/* Body of reply to OFPMP_METER_FEATURES request. Meter features. */
struct ofp_meter_features {
    uint32_t    max_meter;    /* Maximum number of meters. */
    uint32_t    band_types;   /* Bitmaps of OFPMBT_* values supported. */
    uint32_t    capabilities; /* Bitmaps of "ofp_meter_flags". */
    uint8_t     max_bands;    /* Maximum bands per meters */
    uint8_t     max_color;    /* Maximum color value */
    uint8_t     pad[2];
};
OFP_ASSERT(sizeof(struct ofp_meter_features) == 16);
```

7.3.5.15 Experimenter Multipart

Experimenter-specific multipart messages are requested with the OFPMP_EXPERIMENTER multipart type. The first bytes of the request and reply bodies are the following structure:

```
/* Body for ofp_multipart_request/reply of type OFPMP_EXPERIMENTER. */
struct ofp_experimenter_multipart_header {
    uint32_t experimenter;    /* Experimenter ID which takes the same form
                                 as in struct ofp_experimenter_header. */
    uint32_t exp_type;        /* Experimenter defined. */
    /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_multipart_header) == 8);
```

The rest of the request and reply bodies are experimenter-defined.

The experimenter field is the Experimenter ID, which takes the same form as in struct ofp_experimenter (see 7.5.4).

7.3.6 Queue Configuration Messages

Queue configuration takes place outside the OpenFlow protocol, either through a command line tool or through an external dedicated configuration protocol.

The controller can query the switch for configured queues on a port using the following structure:

```
/* Query for port queue configuration. */
struct ofp_queue_get_config_request {
    struct ofp_header header;
    uint32_t port;         /* Port to be queried. Should refer
                              to a valid physical port (i.e. < OFPP_MAX),
                              or OFPP_ANY to request all configured
                              queues.*/
    uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_queue_get_config_request) == 16);
```


The switch replies back with an ofp_queue_get_config_reply command, containing a list of configured queues.

```
/* Queue configuration for a given port. */
struct ofp_queue_get_config_reply {
    struct ofp_header header;
    uint32_t port;
    uint8_t pad[4];
    struct ofp_packet_queue queues[0]; /* List of configured queues. */
};
OFP_ASSERT(sizeof(struct ofp_queue_get_config_reply) == 16);
```

7.3.7 Packet-Out Message

When the controller wishes to send a packet out through the datapath, it uses the OFPT_PACKET_OUT message:

```
/* Send packet (controller -> datapath). */
struct ofp_packet_out {
    struct ofp_header header;
    uint32_t buffer_id;           /* ID assigned by datapath (OFP_NO_BUFFER
                                     if none). */
    uint32_t in_port;             /* Packet's input port or OFPP_CONTROLLER. */
    uint16_t actions_len;         /* Size of action array in bytes. */
    uint8_t pad[6];
    struct ofp_action_header actions[0]; /* Action list - 0 or more. */
    /* The variable size action list is optionally followed by packet data.
     * This data is only present and meaningful if buffer_id == -1.
    /* uint8_t data[0]; */        /* Packet data.  The length is inferred
                                     from the length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_packet_out) == 24);
```

The buffer_id is the same given in the ofp_packet_in message. If the buffer_id is OFP_NO_BUFFER, then the packet data is included in the data array, and the packet encapsulated in the message is processed by the actions of the message. OFP_NO_BUFFER is 0xffffffff. If buffer_id is valid, the corresponding packet is removed from the buffer and processed by the actions of the message.

The in_port field is the ingress port that must be associated with the packet for OpenFlow processing. It must be set to either a valid standard switch port (see 4.2) or OFPP_CONTROLLER. For example, this field is used when processing the packet using OFPP_TABLE, OFPP_IN_PORT, OFPP_ALL and groups.

The action field is an action list defining how the packet should be processed by the switch. It may include packet modification, group processing and an output port. The action list of an OFPT_PACKET_OUT message can also specify the OFPP_TABLE reserved port as an output action to process the packet through the OpenFlow pipeline, starting at the first flow table (see 4.5). If OFPP_TABLE is specified, the in_port field is used as the ingress port in the flow table lookup.

In some cases, packets sent to OFPP_TABLE may be forwarded back to the controller as the result of a flow entry action or table miss. Detecting and taking action for such controller-to-switch loops is outside the scope of this specification. In general, OpenFlow messages are not guaranteed to be processed in order,

therefore if a OFPT_PACKET_OUT message using OFPP_TABLE depends on a flow that was recently sent to the switch (with a OFPT_FLOW_MOD message), a OFPT_BARRIER_REQUEST message may be required prior to the OFPT_PACKET_OUT message to make sure the flow entry was committed to the flow table prior to execution of OFPP_TABLE.

7.3.8 Barrier Message

When the controller wants to ensure message dependencies have been met or wants to receive notifications for completed operations, it may use an OFPT_BARRIER_REQUEST message. This message has no body. Upon receipt, the switch must finish processing all previously-received messages, including sending corresponding reply or error messages, before executing any messages beyond the Barrier Request. When such processing is complete, the switch must send an OFPT_BARRIER_REPLY message with the xid of the original request.

7.3.9 Role Request Message

When the controller wants to change its role, it uses the OFPT_ROLE_REQUEST message with the following structure:

```
/* Role request and reply message. */
struct ofp_role_request {
    struct ofp_header header;   /* Type OFPT_ROLE_REQUEST/OFPT_ROLE_REPLY. */
    uint32_t role;              /* One of OFPCR_ROLE_*. */
    uint8_t pad[4];             /* Align to 64 bits. */
    uint64_t generation_id;     /* Master Election Generation Id */
};
OFP_ASSERT(sizeof(struct ofp_role_request) == 24);
```

The field role is the new role that the controller wants to assume, and can have the following values:

```
/* Controller roles. */
enum ofp_controller_role {
    OFPCR_ROLE_NOCHANGE = 0,    /* Don't change current role. */
    OFPCR_ROLE_EQUAL    = 1,    /* Default role, full access. */
    OFPCR_ROLE_MASTER   = 2,    /* Full access, at most one master. */
    OFPCR_ROLE_SLAVE    = 3,    /* Read-only access. */
};
```

If the role value in the message is OFPCR_ROLE_MASTER or OFPCR_ROLE_SLAVE, the switch must validate generation_id to check for stale messages (see 6.3.4). If the validation fails, the switch must discard the role request and return an error message with type OFPET_ROLE_REQUEST_FAILED and code OFPRRFC_STALE.

If the role value is OFPCR_ROLE_MASTER, all other controllers whose role was OFPCR_ROLE_MASTER are changed to OFPCR_ROLE_SLAVE (see 6.3.4). If the role value is OFPCR_ROLE_NOCHANGE, the current role of the controller is not changed ; this enables a controller to query its current role without changing it.

Upon receipt of a OFPT_ROLE_REQUEST message, if there is no error, the switch must return a OFPT_ROLE_REPLY message. The structure of this message is exactly the same as the OFPT_ROLE_REQUEST message, and the field role is the current role of the controller. The field generation_id is set to the current generation_id (the generation_id associated with the last successful role request with role OFPCR_ROLE_MASTER or OFPCR_ROLE_SLAVE), if the current generation_id was never set by a controller, the field generation_id in the reply must be set to the maximum field value (the unsigned equivalent of -1).

7.3.10 Set Asynchronous Configuration Message

The controller is able to set and query the asynchronous messages that it wants to receive (other than error messages) on a given OpenFlow channel with the OFPT_SET_ASYNC and OFPT_GET_ASYNC_REQUEST messages, respectively. The switch responds to a OFPT_GET_ASYNC_REQUEST message with an OFPT_GET_ASYNC_REPLY message; it does not reply to a request to set the configuration.

There is no body for OFPT_GET_ASYNC_REQUEST beyond the OpenFlow header. The OFPT_SET_ASYNC and OFPT_GET_ASYNC_REPLY messages have the following format:

```
/* Asynchronous message configuration. */
struct ofp_async_config {
    struct ofp_header header;     /* OFPT_GET_ASYNC_REPLY or OFPT_SET_ASYNC. */
    uint32_t packet_in_mask[2];   /* Bitmasks of OFPR_* values. */
    uint32_t port_status_mask[2]; /* Bitmasks of OFPPR_* values. */
    uint32_t flow_removed_mask[2];/* Bitmasks of OFPRR_* values. */
};
OFP_ASSERT(sizeof(struct ofp_async_config) == 32);
```

struct ofp_async_config contains three 2-element arrays. Each array controls whether the controller receives asynchronous messages with a specific enum ofp_type. Within each array, element 0 specifies messages of interest when the controller has a OFPCR_ROLE_EQUAL or OFPCR_ROLE_MASTER role; element 1, when the controller has a OFPCR_ROLE_SLAVE role. Each array element is a bit-mask in which a 0-bit disables receiving a message sent with the reason code corresponding to the bit index and a 1-bit enables receiving it. For example, the bit with value $2^2 = 4$ in port_status_mask[1] determines whether the controller will receive OFPT_PORT_STATUS messages with reason OFPPR_MODIFY (value 2) when the controller has role OFPCR_ROLE_SLAVE.

OFPT_SET_ASYNC sets whether a controller should receive a given asynchronous message that is generated by the switch. Other OpenFlow features control whether a given message is generated; for example, the OFPFF_SEND_FLOW_REM flag controls whether the switch generates OFPT_FLOW_REMOVED a message when a flow entry is removed.

A switch configuration, for example using the OpenFlow Configuration Protocol, may set the initial configuration of asynchronous messages when an OpenFlow connection is established. In the absence of such switch configuration, the initial configuration shall be:

- In the "master" or "equal" role, enable all OFPT_PACKET_IN messages, except those with reason OFPR_INVALID_TTL, and enable all OFPT_PORT_STATUS and OFPT_FLOW_REMOVED messages.

- In the "slave" role, enable all OFPT_PORT_STATUS messages and disable all OFPT_PACKET_IN and OFPT_FLOW_REMOVED messages.

The configuration set with OFPT_SET_ASYNC is specific to a particular OpenFlow channel. It does not affect any other OpenFlow channel, whether currently established or to be established in the future.

The configuration set with OFPT_SET_ASYNC does not filter or otherwise affect error messages.

7.4 Asynchronous Messages

7.4.1 Packet-In Message

When packets are received by the datapath and sent to the controller, they use the OFPT_PACKET_IN message:

```
/* Packet received on port (datapath -> controller). */
struct ofp_packet_in {
    struct ofp_header header;
    uint32_t buffer_id;     /* ID assigned by datapath. */
    uint16_t total_len;     /* Full length of frame. */
    uint8_t reason;         /* Reason packet is being sent (one of OFPR_*) */
    uint8_t table_id;       /* ID of the table that was looked up */
    uint64_t cookie;        /* Cookie of the flow entry that was looked up. */
    struct ofp_match match; /* Packet metadata. Variable size. */
    /* The variable size and padded match is always followed by:
     *   - Exactly 2 all-zero padding bytes, then
     *   - An Ethernet frame whose length is inferred from header.length.
     * The padding bytes preceding the Ethernet frame ensure that the IP
     * header (if any) following the Ethernet header is 32-bit aligned.
     */
    //uint8_t pad[2];       /* Align to 64 bit + 16 bit */
    //uint8_t data[0];      /* Ethernet frame */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 32);
```

The buffer_id is an opaque value used by the datapath to identify a buffered packet. When a packet is buffered, some number of bytes from the message will be included in the data portion of the message. If the packet is sent because of a "send to controller" action, then max_len bytes from the ofp_action_output of the flow setup request are sent. If the packet is sent for other reasons, such as an invalid TTL, then at least miss_send_len bytes from the OFPT_SET_CONFIG message are sent. The default miss_send_len is 128 bytes. If the packet is not buffered - either because of no available buffers, or because of explicitly requested via OFPCML_NO_BUFFER - the entire packet is included in the data portion, and the buffer_id is OFP_NO_BUFFER.

Switches that implement buffering are expected to expose, through documentation, both the amount of available buffering, and the length of time before buffers may be reused. A switch must gracefully handle the case where a buffered packet_in message yields no response from the controller. A switch should prevent a buffer from being reused until it has been handled by the controller, or some amount of time (indicated in documentation) has passed.

The data field contains the packet itself, or a fraction of the packet if the packet is buffered. The packet header reflect any changes applied to the packet in previous processing.

The reason field can be any of these values:

```
/* Why is this packet being sent to the controller? */
enum ofp_packet_in_reason {
    OFPR_NO_MATCH    = 0,   /* No matching flow (table-miss flow entry). */
    OFPR_ACTION      = 1,   /* Action explicitly output to controller. */
    OFPR_INVALID_TTL = 2,   /* Packet has invalid TTL */
};
```

OFPR_INVALID_TTL indicates that a packets with an invalid IP TTL or MPLS TTL was rejected by the OpenFlow pipeline and passed to the controller. Checking for invalid TTL does not need to be done for every packet, but it must be done at a minimum every time a OFPAT_DEC_MPLS_TTL or OFPAT_DEC_NW_TTL action is applied to a packet.

The cookie field contains the cookie of the flow entry that caused the packet to be sent to the controller. This field must be set to -1 (0xffffffffffffffff) if a cookie cannot be associated with a particular flow. For example, if the packet-in was generated in a group bucket or from the action set.

The match field reflect the packet's headers and context when the event that triggers the packet-in message occurred and contains a set of OXM TLVs. This context includes any changes applied to the packet in previous processing, including actions already executed, if any, but not any changes in the action set. The OXM TLVs must include context fields, that is, fields whose values cannot be determined from the packet data. The standard context fields are OFPXMT_OFB_IN_PORT, OFPXMT_OFB_IN_PHY_PORT, OFPXMT_OFB_METADATA and OFPXMT_OFB_TUNNEL_ID. Fields whose values are all-bits-zero should be omitted. Optionally, the OXM TLVs may also include packet header fields that were previously extracted from the packet, including any modifications of those in the course of the processing.

When a packet is received directly on a physical port and not processed by a logical port, OFPXMT_OFB_IN_PORT and OFPXMT_OFB_IN_PHY_PORT have the same value, the OpenFlow port_no of this physical port. OFPXMT_OFB_IN_PHY_PORT should be omitted if it has the same value as OFPXMT_OFB_IN_PORT.

When a packet is received on a logical port by way of a physical port, OFPXMT_OFB_IN_PORT is the logical port's port_no and OFPXMT_OFB_IN_PHY_PORT is the physical port's port_no. For example, consider a packet received on a tunnel interface defined over a link aggregation group (LAG) with two physical port members. If the tunnel interface is the logical port bound to OpenFlow, then OFPXMT_OFB_IN_PORT is the tunnel port_no and OFPXMT_OFB_IN_PHY_PORT is the physical port_no member of the LAG on which the tunnel is configured.

The port referenced by the OFPXMT_OFB_IN_PORT TLV must be the port used for matching flow entries (see 5.3) and must be available to OpenFlow processing (i.e. OpenFlow can forward packet to this port, depending on port flags). OFPXMT_OFB_IN_PHY_PORT need not be available for matching or OpenFlow processing.

7.4.2 Flow Removed Message

If the controller has requested to be notified when flow entries time out or are deleted from tables (see 5.5), the datapath does this with the `OFPT_FLOW_REMOVED` message:

```
/* Flow removed (datapath -> controller). */
struct ofp_flow_removed {
    struct ofp_header header;
    uint64_t cookie;          /* Opaque controller-issued identifier. */

    uint16_t priority;        /* Priority level of flow entry. */
    uint8_t reason;           /* One of OFPRR_*. */
    uint8_t table_id;         /* ID of the table */

    uint32_t duration_sec;    /* Time flow was alive in seconds. */
    uint32_t duration_nsec;   /* Time flow was alive in nanoseconds beyond
                                 duration_sec. */
    uint16_t idle_timeout;    /* Idle timeout from original flow mod. */
    uint16_t hard_timeout;    /* Hard timeout from original flow mod. */
    uint64_t packet_count;
    uint64_t byte_count;
    struct ofp_match match;   /* Description of fields. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_removed) == 56);
```

The `match`, `cookie`, and `priority` fields are the same as those used in the flow mod request.

The `reason` field is one of the following:

```
/* Why was this flow removed? */
enum ofp_flow_removed_reason {
    OFPRR_IDLE_TIMEOUT = 0,     /* Flow idle time exceeded idle_timeout. */
    OFPRR_HARD_TIMEOUT = 1,     /* Time exceeded hard_timeout. */
    OFPRR_DELETE       = 2,     /* Evicted by a DELETE flow mod. */
    OFPRR_GROUP_DELETE = 3,     /* Group was removed. */
};
```

The `duration_sec` and `duration_nsec` fields are described in Section 7.3.5.2.

The `idle_timeout` and `hard_timeout` fields are directly copied from the flow mod that created this entry.

With the above three fields, one can find both the amount of time the flow entry was active, as well as the amount of time the flow entry received traffic.

The `packet_count` and `byte_count` indicate the number of packets and bytes that were associated with this flow entry, respectively. Those counters should behave like other statistics counters (see 7.3.5) ; they are unsigned and should be set to the maximum field value if not available.

7.4.3 Port Status Message

As ports are added, modified, and removed from the datapath, the controller needs to be informed with the OFPT_PORT_STATUS message:

```
/* A physical port has changed in the datapath */
struct ofp_port_status {
    struct ofp_header header;
    uint8_t reason;          /* One of OFPPR_*. */
    uint8_t pad[7];          /* Align to 64-bits. */
    struct ofp_port desc;
};
OFP_ASSERT(sizeof(struct ofp_port_status) == 80);
```

The reason can be one of the following values:

```
/* What changed about the physical port */
enum ofp_port_reason {
    OFPPR_ADD     = 0,         /* The port was added. */
    OFPPR_DELETE  = 1,         /* The port was removed. */
    OFPPR_MODIFY  = 2,         /* Some attribute of the port has changed. */
};
```

7.4.4 Error Message

There are times that the switch needs to notify the controller of a problem. This is done with the OFPT_ERROR_MSG message:

```
/* OFPT_ERROR: Error message (datapath -> controller). */
struct ofp_error_msg {
    struct ofp_header header;

    uint16_t type;
    uint16_t code;
    uint8_t data[0];          /* Variable-length data.  Interpreted based
                                 on the type and code.  No padding. */
};
OFP_ASSERT(sizeof(struct ofp_error_msg) == 12);
```

The type value indicates the high-level type of error. The code value is interpreted based on the type. The data is variable length and interpreted based on the type and code. Unless specified otherwise, the data field contains at least 64 bytes of the failed request that caused the error message to be generated, if the failed request is shorter than 64 bytes it should be the full request without any padding.

If the error message is in response to a specific message from the controller, e.g., OFPET_BAD_REQUEST, OFPET_BAD_ACTION, OFPET_BAD_INSTRUCTION, OFPET_BAD_MATCH, or OFPET_FLOW_MOD_FAILED, then the xid field of the header must match that of the offending message.

Error codes ending in _EPERM correspond to a permissions error generated by, for example, an OpenFlow hypervisor interposing between a controller and switch.

Currently defined error types are:


```
/* Values for 'type' in ofp_error_message.  These values are immutable: they
 * will not change in future versions of the protocol (although new values may
 * be added). */
enum ofp_error_type {
    OFPET_HELLO_FAILED         = 0,  /* Hello protocol failed. */
    OFPET_BAD_REQUEST          = 1,  /* Request was not understood. */
    OFPET_BAD_ACTION           = 2,  /* Error in action description. */
    OFPET_BAD_INSTRUCTION      = 3,  /* Error in instruction list. */
    OFPET_BAD_MATCH            = 4,  /* Error in match. */
    OFPET_FLOW_MOD_FAILED      = 5,  /* Problem modifying flow entry. */
    OFPET_GROUP_MOD_FAILED     = 6,  /* Problem modifying group entry. */
    OFPET_PORT_MOD_FAILED      = 7,  /* Port mod request failed. */
    OFPET_TABLE_MOD_FAILED     = 8,  /* Table mod request failed. */
    OFPET_QUEUE_OP_FAILED      = 9,  /* Queue operation failed. */
    OFPET_SWITCH_CONFIG_FAILED = 10, /* Switch config request failed. */
    OFPET_ROLE_REQUEST_FAILED  = 11, /* Controller Role request failed. */
    OFPET_METER_MOD_FAILED     = 12, /* Error in meter. */
    OFPET_TABLE_FEATURES_FAILED = 13, /* Setting table features failed. */
    OFPET_EXPERIMENTER = 0xffff      /* Experimenter error messages. */
};
```

For the OFPET_HELLO_FAILED error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_HELLO_FAILED.  'data' contains an
 * ASCII text string that may give failure details. */
enum ofp_hello_failed_code {
    OFPHFC_INCOMPATIBLE = 0,    /* No compatible version. */
    OFPHFC_EPERM        = 1,    /* Permissions error. */
};
```

The data field contains an ASCII text string that adds detail on why the error occurred.

For the OFPET_BAD_REQUEST error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_BAD_REQUEST.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_request_code {
    OFPBRC_BAD_VERSION      = 0,  /* ofp_header.version not supported. */
    OFPBRC_BAD_TYPE         = 1,  /* ofp_header.type not supported. */
    OFPBRC_BAD_MULTIPART    = 2,  /* ofp_multipart_request.type not supported. */
    OFPBRC_BAD_EXPERIMENTER = 3,  /* Experimenter id not supported
                                   * (in ofp_experimenter_header or
                                   * ofp_multipart_request or
                                   * ofp_multipart_reply). */
    OFPBRC_BAD_EXP_TYPE     = 4,  /* Experimenter type not supported. */
    OFPBRC_EPERM            = 5,  /* Permissions error. */
    OFPBRC_BAD_LEN          = 6,  /* Wrong request length for type. */
    OFPBRC_BUFFER_EMPTY     = 7,  /* Specified buffer has already been used. */
    OFPBRC_BUFFER_UNKNOWN   = 8,  /* Specified buffer does not exist. */
    OFPBRC_BAD_TABLE_ID     = 9,  /* Specified table-id invalid or does not
                                   * exist. */
    OFPBRC_IS_SLAVE         = 10, /* Denied because controller is slave. */
    OFPBRC_BAD_PORT         = 11, /* Invalid port. */
    OFPBRC_BAD_PACKET       = 12, /* Invalid packet in packet-out. */
    OFPBRC_MULTIPART_BUFFER_OVERFLOW    = 13, /* ofp_multipart_request
```



```
                                    overflowed the assigned buffer. */
};
```

For the OFPET_BAD_ACTION error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_BAD_ACTION.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_action_code {
    OFPBAC_BAD_TYPE           = 0,  /* Unknown action type. */
    OFPBAC_BAD_LEN            = 1,  /* Length problem in actions. */
    OFPBAC_BAD_EXPERIMENTER   = 2,  /* Unknown experimenter id specified. */
    OFPBAC_BAD_EXP_TYPE       = 3,  /* Unknown action for experimenter id. */
    OFPBAC_BAD_OUT_PORT       = 4,  /* Problem validating output port. */
    OFPBAC_BAD_ARGUMENT       = 5,  /* Bad action argument. */
    OFPBAC_EPERM              = 6,  /* Permissions error. */
    OFPBAC_TOO_MANY           = 7,  /* Can't handle this many actions. */
    OFPBAC_BAD_QUEUE          = 8,  /* Problem validating output queue. */
    OFPBAC_BAD_OUT_GROUP      = 9,  /* Invalid group id in forward action. */
    OFPBAC_MATCH_INCONSISTENT = 10, /* Action can't apply for this match,
                                       or Set-Field missing prerequisite. */
    OFPBAC_UNSUPPORTED_ORDER  = 11, /* Action order is unsupported for the
                                     action list in an Apply-Actions instruction */
    OFPBAC_BAD_TAG            = 12, /* Actions uses an unsupported
                                       tag/encap. */
    OFPBAC_BAD_SET_TYPE       = 13, /* Unsupported type in SET_FIELD action. */
    OFPBAC_BAD_SET_LEN        = 14, /* Length problem in SET_FIELD action. */
    OFPBAC_BAD_SET_ARGUMENT   = 15, /* Bad argument in SET_FIELD action. */
};
```

For the OFPET_BAD_INSTRUCTION error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_BAD_INSTRUCTION.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_instruction_code {
    OFPBIC_UNKNOWN_INST     = 0, /* Unknown instruction. */
    OFPBIC_UNSUP_INST       = 1, /* Switch or table does not support the
                                    instruction. */
    OFPBIC_BAD_TABLE_ID     = 2, /* Invalid Table-ID specified. */
    OFPBIC_UNSUP_METADATA   = 3, /* Metadata value unsupported by datapath. */
    OFPBIC_UNSUP_METADATA_MASK = 4, /* Metadata mask value unsupported by
                                       datapath. */
    OFPBIC_BAD_EXPERIMENTER = 5, /* Unknown experimenter id specified. */
    OFPBIC_BAD_EXP_TYPE     = 6, /* Unknown instruction for experimenter id. */
    OFPBIC_BAD_LEN          = 7, /* Length problem in instructions. */
    OFPBIC_EPERM            = 8, /* Permissions error. */
};
```

For the OFPET_BAD_MATCH error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_BAD_MATCH.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_match_code {
    OFPBMC_BAD_TYPE         = 0,  /* Unsupported match type specified by the
```



```
                                   match */
    OFPBMC_BAD_LEN          = 1,  /* Length problem in match. */
    OFPBMC_BAD_TAG          = 2,  /* Match uses an unsupported tag/encap. */
    OFPBMC_BAD_DL_ADDR_MASK = 3,  /* Unsupported datalink addr mask - switch
                                     does not support arbitrary datalink
                                     address mask. */
    OFPBMC_BAD_NW_ADDR_MASK = 4,  /* Unsupported network addr mask - switch
                                     does not support arbitrary network
                                     address mask. */
    OFPBMC_BAD_WILDCARDS    = 5,  /* Unsupported combination of fields masked
                                     or omitted in the match. */
    OFPBMC_BAD_FIELD        = 6,  /* Unsupported field type in the match. */
    OFPBMC_BAD_VALUE        = 7,  /* Unsupported value in a match field. */
    OFPBMC_BAD_MASK         = 8,  /* Unsupported mask specified in the match,
                                     field is not dl-address or nw-address. */
    OFPBMC_BAD_PREREQ       = 9,  /* A prerequisite was not met. */
    OFPBMC_DUP_FIELD        = 10, /* A field type was duplicated. */
    OFPBMC_EPERM            = 11, /* Permissions error. */
};
```

For the OFPET_FLOW_MOD_FAILED error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_FLOW_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_flow_mod_failed_code {
    OFPFMFC_UNKNOWN      = 0,   /* Unspecified error. */
    OFPFMFC_TABLE_FULL   = 1,   /* Flow not added because table was full. */
    OFPFMFC_BAD_TABLE_ID = 2,   /* Table does not exist */
    OFPFMFC_OVERLAP      = 3,   /* Attempted to add overlapping flow with
                                   CHECK_OVERLAP flag set. */
    OFPFMFC_EPERM        = 4,   /* Permissions error. */
    OFPFMFC_BAD_TIMEOUT  = 5,   /* Flow not added because of unsupported
                                   idle/hard timeout. */
    OFPFMFC_BAD_COMMAND  = 6,   /* Unsupported or unknown command. */
    OFPFMFC_BAD_FLAGS    = 7,   /* Unsupported or unknown flags. */
};
```

For the OFPET_GROUP_MOD_FAILED error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_GROUP_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_group_mod_failed_code {
    OFPGMFC_GROUP_EXISTS         = 0,  /* Group not added because a group ADD
                                          attempted to replace an
                                          already-present group. */
    OFPGMFC_INVALID_GROUP        = 1,  /* Group not added because Group
                                          specified is invalid. */
    OFPGMFC_WEIGHT_UNSUPPORTED   = 2,  /* Switch does not support unequal load
                                          sharing with select groups. */
    OFPGMFC_OUT_OF_GROUPS        = 3,  /* The group table is full. */
    OFPGMFC_OUT_OF_BUCKETS       = 4,  /* The maximum number of action buckets
                                          for a group has been exceeded. */
    OFPGMFC_CHAINING_UNSUPPORTED = 5,  /* Switch does not support groups that
                                          forward to groups. */
```



```
    OFPGMFC_WATCH_UNSUPPORTED    = 6,  /* This group cannot watch the watch_port
                                          or watch_group specified. */
    OFPGMFC_LOOP                 = 7,  /* Group entry would cause a loop. */
    OFPGMFC_UNKNOWN_GROUP        = 8,  /* Group not modified because a group
                                          MODIFY attempted to modify a
                                          non-existent group. */
    OFPGMFC_CHAINED_GROUP        = 9,  /* Group not deleted because another
                                          group is forwarding to it. */
    OFPGMFC_BAD_TYPE             = 10, /* Unsupported or unknown group type. */
    OFPGMFC_BAD_COMMAND          = 11, /* Unsupported or unknown command. */
    OFPGMFC_BAD_BUCKET           = 12, /* Error in bucket. */
    OFPGMFC_BAD_WATCH            = 13, /* Error in watch port/group. */
    OFPGMFC_EPERM                = 14, /* Permissions error. */
};
```

For the OFPET_PORT_MOD_FAILED error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_PORT_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_port_mod_failed_code {
    OFPPMFC_BAD_PORT      = 0,   /* Specified port number does not exist. */
    OFPPMFC_BAD_HW_ADDR   = 1,   /* Specified hardware address does not
                                  * match the port number. */
    OFPPMFC_BAD_CONFIG    = 2,   /* Specified config is invalid. */
    OFPPMFC_BAD_ADVERTISE = 3,   /* Specified advertise is invalid. */
    OFPPMFC_EPERM         = 4,   /* Permissions error. */
};
```

For the OFPET_TABLE_MOD_FAILED error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_TABLE_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_table_mod_failed_code {
    OFPTMFC_BAD_TABLE  = 0,      /* Specified table does not exist. */
    OFPTMFC_BAD_CONFIG = 1,      /* Specified config is invalid. */
    OFPTMFC_EPERM      = 2,      /* Permissions error. */
};
```

For the OFPET_QUEUE_OP_FAILED error type, the following codes are currently defined:

```
/* ofp_error msg 'code' values for OFPET_QUEUE_OP_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request */
enum ofp_queue_op_failed_code {
    OFPQOFC_BAD_PORT   = 0,     /* Invalid port (or port does not exist). */
    OFPQOFC_BAD_QUEUE  = 1,     /* Queue does not exist. */
    OFPQOFC_EPERM      = 2,     /* Permissions error. */
};
```

For the OFPET_SWITCH_CONFIG_FAILED error type, the following codes are currently defined:


```
/* ofp_error_msg 'code' values for OFPET_SWITCH_CONFIG_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_switch_config_failed_code {
    OFPSCFC_BAD_FLAGS  = 0,      /* Specified flags is invalid. */
    OFPSCFC_BAD_LEN    = 1,      /* Specified len is invalid. */
    OFPSCFC_EPERM      = 2,      /* Permissions error. */
};
```

For the OFPET_ROLE_REQUEST_FAILED error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_ROLE_REQUEST_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_role_request_failed_code {
    OFPRRFC_STALE      = 0,      /* Stale Message: old generation_id. */
    OFPRRFC_UNSUP      = 1,      /* Controller role change unsupported. */
    OFPRRFC_BAD_ROLE   = 2,      /* Invalid role. */
};
```

For the OFPET_METER_MOD_FAILED error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_METER_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_meter_mod_failed_code {
    OFPMMFC_UNKNOWN       = 0,  /* Unspecified error. */
    OFPMMFC_METER_EXISTS  = 1,  /* Meter not added because a Meter ADD
                                 * attempted to replace an existing Meter. */
    OFPMMFC_INVALID_METER = 2,      /* Meter not added because Meter specified
                                 * is invalid. */
    OFPMMFC_UNKNOWN_METER = 3,  /* Meter not modified because a Meter
                                   MODIFY attempted to modify a non-existent
                                   Meter. */
    OFPMMFC_BAD_COMMAND   = 4,  /* Unsupported or unknown command. */
    OFPMMFC_BAD_FLAGS     = 5,  /* Flag configuration unsupported. */
    OFPMMFC_BAD_RATE      = 6,  /* Rate unsupported. */
    OFPMMFC_BAD_BURST     = 7,  /* Burst size unsupported. */
    OFPMMFC_BAD_BAND      = 8,  /* Band unsupported. */
    OFPMMFC_BAD_BAND_VALUE = 9, /* Band value unsupported. */
    OFPMMFC_OUT_OF_METERS = 10, /* No more meters available. */
    OFPMMFC_OUT_OF_BANDS  = 11, /* The maximum number of properties
                                 * for a meter has been exceeded. */
};
```

For the OFPET_TABLE_FEATURES_FAILED error type, the following codes are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_TABLE_FEATURES_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_table_features_failed_code {
    OFPTFFC_BAD_TABLE    = 0,      /* Specified table does not exist. */
    OFPTFFC_BAD_METADATA = 1,      /* Invalid metadata mask. */
    OFPTFFC_BAD_TYPE     = 2,      /* Unknown property type. */
    OFPTFFC_BAD_LEN      = 3,      /* Length problem in properties. */
    OFPTFFC_BAD_ARGUMENT = 4,      /* Unsupported property value. */
    OFPTFFC_EPERM        = 5,      /* Permissions error. */
};
```


For the OFPET_EXPERIMENTER error type, the error message is defined by the following structure and fields, followed by experimenter defined data:

```
/* OFPET_EXPERIMENTER: Error message (datapath -> controller). */
struct ofp_error_experimenter_msg {
    struct ofp_header header;

    uint16_t type;            /* OFPET_EXPERIMENTER. */
    uint16_t exp_type;        /* Experimenter defined. */
    uint32_t experimenter;    /* Experimenter ID which takes the same form
                                 as in struct ofp_experimenter_header. */
    uint8_t data[0];          /* Variable-length data.  Interpreted based
                                 on the type and code.  No padding. */
};
OFP_ASSERT(sizeof(struct ofp_error_experimenter_msg) == 16);
```

The experimenter field is the Experimenter ID, which takes the same form as in struct ofp_experimenter (see 7.5.4).

7.5 Symmetric Messages

7.5.1 Hello

The OFPT_HELLO message consists of an OpenFlow header plus a set of variable size hello elements.

```
/* OFPT_HELLO.  This message includes zero or more hello elements having
 * variable size. Unknown elements types must be ignored/skipped, to allow
 * for future extensions. */
struct ofp_hello {
    struct ofp_header header;

    /* Hello element list */
    struct ofp_hello_elem_header elements[0]; /* List of elements - 0 or more */
};
OFP_ASSERT(sizeof(struct ofp_hello) == 8);
```

The version field part of the header field (see 7.1) must be set to the highest OpenFlow protocol version supported by the sender (see 6.3.1).

The elements field is a set of hello elements, containing optional data to inform the initial handshake of the connection. Implementations must ignore (skip) all elements of a Hello message that they do not support. The list of hello elements types that are currently defined are:

```
/* Hello elements types.
 */
enum ofp_hello_elem_type {
    OFPHET_VERSIONBITMAP          = 1,  /* Bitmap of version supported. */
};
```

An element definition contains the element type, length, and any associated data:


```
/* Common header for all Hello Elements */
struct ofp_hello_elem_header {
    uint16_t         type;    /* One of OFPHET_*. */
    uint16_t         length;  /* Length in bytes of this element. */
};
OFP_ASSERT(sizeof(struct ofp_hello_elem_header) == 4);
```

The OFPHET_VERSIONBITMAP element use the following structure and fields:

```
/* Version bitmap Hello Element */
struct ofp_hello_elem_versionbitmap {
    uint16_t         type;    /* OFPHET_VERSIONBITMAP. */
    uint16_t         length;  /* Length in bytes of this element. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the bitmaps, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         bitmaps[0];   /* List of bitmaps - supported versions */
};
OFP_ASSERT(sizeof(struct ofp_hello_elem_versionbitmap) == 4);
```

The bitmaps field indicates the set of versions of the OpenFlow switch protocol a device supports, and may be used during version negotiation (see 6.3.1). The bits of the set of bitmaps are indexed by the *ofp_version* number of the protocol ; if the bit identified by the number of left bitshift equal to a *ofp_version* number is set, this OpenFlow version is supported. The number of bitmaps included in the field depend on the highest version number supported : *ofp_versions* 0 to 31 are encoded in the first bitmap, *ofp_versions* 32 to 63 are encoded in the second bitmap and so on. For example, if a switch supports only version 1.0 (*ofp_version*=0x01) and version 1.3 (*ofp_version*=0x04), the first bitmap would be set to 0x00000012.

7.5.2 Echo Request

An Echo Request message consists of an OpenFlow header plus an arbitrary-length data field. The data field might be a message timestamp to check latency, various lengths to measure bandwidth, or zero-size to verify liveness between the switch and controller.

7.5.3 Echo Reply

An Echo Reply message consists of an OpenFlow header plus the unmodified data field of an echo request message.

In an OpenFlow protocol implementation divided into multiple layers, the echo request/reply logic should be implemented in the "deepest" practical layer. For example, in the OpenFlow reference implementation that includes a userspace process that relays to a kernel module, echo request/reply is implemented in the kernel module. Receiving a correctly formatted echo reply then shows a greater likelihood of correct end-to-end functionality than if the echo request/reply were implemented in the userspace process, as well as providing more accurate end-to-end latency timing.

7.5.4 Experimenter

The Experimenter message is defined as follows:

```
/* Experimenter extension. */
struct ofp_experimenter_header {
    struct ofp_header header;   /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter;      /* Experimenter ID:
                                 * - MSB 0: low-order bytes are IEEE OUI.
                                 * - MSB != 0: defined by ONF. */
    uint32_t exp_type;          /* Experimenter defined. */
    /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_header) == 16);
```

The `experimenter` field is a 32-bit value that uniquely identifies the experimenter. If the most significant byte is zero, the next three bytes are the experimenter's IEEE OUI. If the most significant byte is not zero, it is a value allocated by the Open Networking Foundation. If experimenter does not have (or wish to use) their OUI, they should contact the Open Networking Foundation to obtain a unique experimenter ID.

The rest of the body is uninterpreted by standard OpenFlow processing and is arbitrarily defined by the corresponding experimenter.

If a switch does not understand a experimenter extension, it must send an `OFPT_ERROR` message with a `OFPBRC_BAD_EXPERIMENTER` error code and `OFPET_BAD_REQUEST` error type.

Appendix A Release Notes

This section contains release notes highlighting the main changes between the main versions of the OpenFlow protocol.

The text of the release notes is informative and historical, and should not be considred normative. Many items of the release notes refer to features and text that has been removed, replaced or updated in subsequent versions of this specification, and therefore does not necessarily match the actual specification.

A.1 OpenFlow version 0.2.0

Release date : March 28,2008
Wire Protocol : 1

A.2 OpenFlow version 0.2.1

Release date : March 28,2008
Wire Protocol : 1
No protocol change.

A.3 OpenFlow version 0.8.0

Release date : May 5, 2008
Wire Protocol : 0x83

- Reorganise OpenFlow message types
- Add `OFPP_TABLE` virtual port to send packet-out packet to the tables
- Add global flag `OFPC_SEND_FLOW_EXP` to configure flow expired messages generation
- Add flow priority
- Remove flow Group-ID (experimental QoS support)
- Add Error messages
- Make stat request and stat reply more generic, with a generic header and stat specific body
- Change fragmentation strategy for stats reply, use explicit flag `OFPSF_REPLY_MORE` instead of empty packet
- Add table stats and port stats mesages

A.4 OpenFlow version 0.8.1

Release date : May 20, 2008
Wire Protocol : 0x83
No protocol change.

A.5 OpenFlow version 0.8.2

Release date : October 17, 2008
Wire Protocol : 0x85

- Add Echo Request and Echo Reply messages
- Make all message 64 bits aligned

A.6 OpenFlow version 0.8.9

Release date : December 2, 2008
Wire Protocol : 0x97

A.6.1 IP Netmasks

It is now possible for flow entries to contain IP subnet masks. This is done by changes to the wildcards field, which has been expanded to 32-bits:

```
/* Flow wildcards. */
enum ofp_flow_wildcards {
OFPFW_IN_PORT  = 1 << 0,  /* Switch input port. */
OFPFW_DL_VLAN  = 1 << 1,  /* VLAN. */
OFPFW_DL_SRC   = 1 << 2,  /* Ethernet source address. */
OFPFW_DL_DST   = 1 << 3,  /* Ethernet destination address. */
OFPFW_DL_TYPE  = 1 << 4,  /* Ethernet frame type. */
OFPFW_NW_PROTO = 1 << 5,  /* IP protocol. */
OFPFW_TP_SRC   = 1 << 6,  /* TCP/UDP source port. */
OFPFW_TP_DST   = 1 << 7,  /* TCP/UDP destination port. */

/* IP source address wildcard bit count.  0 is exact match, 1 ignores the
 * LSB, 2 ignores the 2 least-significant bits, ..., 32 and higher wildcard
 * the entire field.  This is the *opposite* of the usual convention where
 * e.g. /24 indicates that 8 bits (not 24 bits) are wildcarded. */
OFPFW_NW_SRC_SHIFT = 8,
OFPFW_NW_SRC_BITS = 6,
OFPFW_NW_SRC_MASK = ((1 << OFPFW_NW_SRC_BITS) - 1) << OFPFW_NW_SRC_SHIFT,
OFPFW_NW_SRC_ALL = 32 << OFPFW_NW_SRC_SHIFT,

/* IP destination address wildcard bit count.  Same format as source. */
OFPFW_NW_DST_SHIFT = 14,
OFPFW_NW_DST_BITS = 6,
OFPFW_NW_DST_MASK = ((1 << OFPFW_NW_DST_BITS) - 1) << OFPFW_NW_DST_SHIFT,
OFPFW_NW_DST_ALL = 32 << OFPFW_NW_DST_SHIFT,

/* Wildcard all fields. */
OFPFW_ALL = ((1 << 20) - 1)
};
```

The source and destination netmasks are each specified with a 6-bit number in the wildcard description. It is interpreted similar to the CIDR suffix, but with the opposite meaning, since this is being used to indicate which bits in the IP address should be treated as "wild". For example, a CIDR suffix of "24" means to use a netmask of "255.255.255.0". However, a wildcard mask value of "24" means that the least-significant 24-bits are wild, so it forms a netmask of "255.0.0.0".

A.6.2 New Physical Port Stats

The `ofp_port_stats` message has been expanded to return more information. If a switch does not support a particular field, it should set the value to have all bits enabled (i.e., a "-1" if the value were treated as signed). This is the new format:

```
/* Body of reply to OFPST_PORT request. If a counter is unsupported, set
 * the field to all ones. */
struct ofp_port_stats {
    uint16_t port_no;
```



```
    uint8_t pad[6];          /* Align to 64-bits. */
    uint64_t rx_packets;     /* Number of received packets. */
    uint64_t tx_packets;     /* Number of transmitted packets. */
    uint64_t rx_bytes;       /* Number of received bytes. */
    uint64_t tx_bytes;       /* Number of transmitted bytes. */
    uint64_t rx_dropped;     /* Number of packets dropped by RX. */
    uint64_t tx_dropped;     /* Number of packets dropped by TX. */
    uint64_t rx_errors;      /* Number of receive errors.  This is a super-set
                                of receive errors and should be great than or
                                equal to the sum of al rx_*_err values. */
    uint64_t tx_errors;      /* Number of transmit errors.  This is a super-set
                                of transmit errors. */
    uint64_t rx_frame_err;   /* Number of frame alignment errors. */
    uint64_t rx_over_err;    /* Number of packets with RX overrun. */
    uint64_t rx_crc_err;     /* Number of CRC errors. */
    uint64_t collisions;     /* Number of collisions. */
};
```

A.6.3 IN_PORT Virtual Port

The behavior of sending out the incoming port was not clearly defined in earlier versions of the specification. It is now forbidden unless the output port is explicitly set to `OFPP_IN_PORT` virtual port (0xfff8) is set. The primary place where this is used is for wireless links, where a packet is received over the wireless interface and needs to be sent to another host through the same interface. For example, if a packet needed to be sent to **all** interfaces on the switch, two actions would need to be specified: "actions=output:ALL,output:IN_PORT".

A.6.4 Port and Link Status and Configuration

The switch should inform the controller of changes to port and link status. This is done with a new flag in `ofp_port_config`:

- `OFPPC_PORT_DOWN` - The port has been configured "down".

... and a new flag in `ofp_port_state`:

- `OFPPS_LINK_DOWN` - There is no physical link present.

The switch should support enabling and disabling a physical port by modifying the `OFPPFL_PORT_DOWN` flag (and mask bit) in the `ofp_port_mod` message. Note that this is **not** the same as adding or removing the interface from the list of OpenFlow monitored ports; it is equivalent to `"ifconfig eth0 down"` on Unix systems.

A.6.5 Echo Request/Reply Messages

The switch and controller can verify proper connectivity through the OpenFlow protocol with the new echo request (`OFPT_ECHO_REQUEST`) and reply (`OFPT_ECHO_REPLY`) messages. The body of the message is undefined and simply contains uninterpreted data that is to be echoed back to the requester. The requester matches the reply with the transaction id from the OpenFlow header.

A.6.6 Vendor Extensions

Vendors are now able to add their own extensions, while still being OpenFlow compliant. The primary way to do this is with the new `OFPT_VENDOR` message type. The message body is of the form:

```
/* Vendor extension. */
struct ofp_vendor {
    struct ofp_header header;   /* Type OFPT_VENDOR. */
    uint32_t vendor;            /* Vendor ID:
                                 * - MSB 0: low-order bytes are IEEE OUI.
                                 * - MSB != 0: defined by OpenFlow
                                 *   consortium. */
    /* Vendor-defined arbitrary additional data. */
};
```

The *vendor* field is a 32-bit value that uniquely identifies the vendor. If the most significant byte is zero, the next three bytes are the vendor's IEEE OUI. If vendor does not have (or wish to use) their OUI, they should contact the OpenFlow consortium to obtain one. The rest of the body is uninterpreted.

It is also possible to add vendor extensions for stats messages with the `OFPST_VENDOR` stats type. The first four bytes of the message are the vendor identifier as described earlier. The rest of the body is vendor-defined.

To indicate that a switch does not understand a vendor extension, a `OFPBRC_BAD_VENDOR` error code has been defined under the `OFPET_BAD_REQUEST` error type.

Vendor-defined actions are described below in the "Variable Length and Vendor Actions" section.

A.6.7 Explicit Handling of IP Fragments

In previous versions of the specification, handling of IP fragments was not clearly defined. The switch is now able to tell the controller whether it is able to reassemble fragments. This is done with the following `capabilities` flag passed in the `ofp_switch` features message:

```
OFPC_IP_REASM     = 1 << 5  /* Can reassemble IP fragments. */
```

The controller can configure fragment handling in the switch through the setting the following new `ofp_config_flags` in the `ofp_switch_config` message:

```
/* Handling of IP fragments. */
OFPC_FRAG_NORMAL   = 0 << 1,  /* No special handling for fragments. */
OFPC_FRAG_DROP     = 1 << 1,  /* Drop fragments. */
OFPC_FRAG_REASM    = 2 << 1,  /* Reassemble (only if OFPC_IP_REASM set). */
OFPC_FRAG_MASK     = 3 << 1
```

"Normal" handling of fragments means that an attempt should be made to pass the fragments through the OpenFlow tables. If any field is not present (e.g., the TCP/UDP ports didn't fit), then the packet should not match any entry that has that field set.

A.6.8 802.1D Spanning Tree

OpenFlow now has a way to configure and view results of on-switch implementations of 802.1D Spanning Tree Protocol.

A switch that implements STP must set the new OFPC_STP bit in the 'capabilities' field of its OFPT_FEATURES_REPLY message. A switch that implements STP at all must make it available on all of its physical ports, but it need not implement it on virtual ports (e.g. OFPP_LOCAL).

Several port configuration flags are associated with STP. The complete set of port configuration flags are:

```
enum ofp_port_config {
   OFPPC_PORT_DOWN    = 1 << 0,  /* Port is administratively down. */
   OFPPC_NO_STP       = 1 << 1,  /* Disable 802.1D spanning tree on port. */
   OFPPC_NO_RECV      = 1 << 2,  /* Drop most packets received on port. */
   OFPPC_NO_RECV_STP  = 1 << 3,  /* Drop received 802.1D STP packets. */
   OFPPC_NO_FLOOD     = 1 << 4,  /* Do not include this port when flooding. */
   OFPPC_NO_FWD       = 1 << 5,  /* Drop packets forwarded to port. */
   OFPPC_NO_PACKET_IN = 1 << 6   /* Do not send packet-in msgs for port. */
};
```

The controller may set OFPPFL_NO_STP to 0 to enable STP on a port or to 1 to disable STP on a port. (The latter corresponds to the Disabled STP port state.) The default is switch implementation-defined; the OpenFlow reference implementation by default sets this bit to 0 (enabling STP).

When OFPPFL_NO_STP is 0, STP controls the OFPPFL_NO_FLOOD and OFPPFL_STP_* bits directly. OFPPFL_NO_FLOOD is set to 0 when the STP port state is Forwarding, otherwise to 1. The bits in OFPPFL_STP_MASK are set to one of the other OFPPFL_STP_* values according to the current STP port state.

When the port flags are changed by STP, the switch sends an OFPT_PORT_STATUS message to notify the controller of the change. The OFPPFL_NO_RECV, OFPPFL_NO_RECV_STP, OFPPFL_NO_FWD, and OFPPFL_NO_PACKET_IN bits in the OpenFlow port flags may be useful for the controller to implement STP, although they interact poorly with in-band control.

A.6.9 Modify Actions in Existing Flow Entries

New ofp_flow_mod commands have been added to support modifying the actions of existing entries: OFPFC_MODIFY and OFPFC_MODIFY_STRICT. They use the match field to describe the entries that should be modified with the supplied actions. OFPFC_MODIFY is similar to OFPFC_DELETE, in that wildcards are "active". OFPFC_MODIFY_STRICT is similar to OFPFC_DELETE_STRICT, in that wildcards are not "active", so both the wildcards and priority must match an entry. When a matching flow is found, only its actions are modified–information such as counters and timers are not reset.

If the controller uses the OFPFC_ADD command to add an entry that already exists, then the new entry replaces the old and all counters and timers are reset.

A.6.10 More Flexible Description of Tables

Previous versions of OpenFlow had very limited abilities to describe the tables supported by the switch. The `n_exact`, `n_compression`, and `n_general` fields in `ofp_switch_features` have been replaced with `n_tables`, which lists the number of tables in the switch.

The behavior of the `OFPST_TABLE` stat reply has been modified slightly. The `ofp_table_stats` body now contains a wildcards field, which indicates the fields for which that particular table supports wildcarding. For example, a direct look-up hash table would have that field set to zero, while a sequentially searched table would have it set to `OFPFW_ALL`. The `ofp_table_stats` entries are returned in the order that packets traverse the tables.

When the controller and switch first communicate, the controller will find out how many tables the switch supports from the Features Reply. If it wishes to understand the size, types, and order in which tables are consulted, the controller sends a `OFPST_TABLE` stats request.

A.6.11 Lookup Count in Tables

Table stats returned `ofp_table_stats` structures now return the number of packets that have been looked up in the table–whether they hit or not. This is stored in the `lookup_count` field.

A.6.12 Modifying Flags in Port-Mod More Explicit

The `ofp_port_mod` is used to modify characteristics of a switch's ports. A supplied `ofp_phy_port` structure describes the behavior of the switch through its `flags` field. However, it's possible that the controller wishes to change a particular flag and may not know the current status of all flags. A `mask` field has been added which has a bit set for each flag that should be changed on the switch.

The new `ofp_port_mod` message looks like the following:

```
/* Modify behavior of the physical port */
struct ofp_port_mod {
    struct ofp_header header;
    uint32_t mask;         /* Bitmap of "ofp_port_flags" that should be
                              changed. */
    struct ofp_phy_port desc;
};
```

A.6.13 New Packet-Out Message Format

The previous version's `packet-out` message treated the variable-length array differently depending on whether the `buffer_id` was set or not. If set, the array consisted of actions to be executed and the `out_port` was ignored. If not, the array consisted of the actual packet that should be placed on the wire through the `out_port` interface. This was a bit ugly, and it meant that in order for a non-buffered packet to have multiple actions executed on it, that a new flow entry be created just to match that entry.

A new format is now used, which cleans the message up a bit. The packet always contains a list of actions. An additional variable-length array follows the list of actions with the contents of the packet if `buffer_id` is not set. This is the new format:

```
struct ofp_packet_out {
    struct ofp_header header;
    uint32_t buffer_id;           /* ID assigned by datapath (-1 if none). */
    uint16_t in_port;             /* Packet's input port (OFPP_NONE if none). */
    uint16_t n_actions;           /* Number of actions. */
    struct ofp_action actions[0]; /* Actions. */
    /* uint8_t data[0]; */        /* Packet data.  The length is inferred
                                     from the length field in the header.
                                     (Only meaningful if buffer_id == -1.) */
};
```

A.6.14 Hard Timeout for Flow Entries

A hard timeout value has been added to flow entries. If set, then the entry must be expired in the specified number of seconds regardless of whether or not packets are hitting the entry. A `hard_timeout` field has been added to the `flow_mod` message to support this. The `max_idle` field has been renamed `idle_timeout`. A value of zero means that a timeout has not been set. If both `idle_timeout` and `hard_timeout` are zero, then the flow is permanent and should not be deleted without an explicit deletion.

The new `ofp_flow_mod` format looks like this:

```
struct ofp_flow_mod {
    struct ofp_header header;
    struct ofp_match match;      /* Fields to match */

    /* Flow actions. */
    uint16_t command;             /* One of OFPFC_*. */
    uint16_t idle_timeout;        /* Idle time before discarding (seconds). */
    uint16_t hard_timeout;        /* Max time before discarding (seconds). */
    uint16_t priority;            /* Priority level of flow entry. */
    uint32_t buffer_id;           /* Buffered packet to apply to (or -1).
                                     Not meaningful for OFPFC_DELETE*. */
    uint32_t reserved;            /* Reserved for future use. */
    struct ofp_action actions[0]; /* The number of actions is inferred from
                                     the length field in the header. */
};
```

Since flow entries can now be expired due to idle or hard timeouts, a `reason` field has been added to the `ofp_flow_expired` message. A value of 0 indicates an idle timeout and 1 indicates a hard timeout:

```
enum ofp_flow_expired_reason {
    OFPER_IDLE_TIMEOUT,         /* Flow idle time exceeded idle_timeout. */
    OFPER_HARD_TIMEOUT          /* Time exceeded hard_timeout. */
};
```

The new `ofp_flow_expired` message looks like the following:


```
struct ofp_flow_expired {
    struct ofp_header header;
    struct ofp_match match;   /* Description of fields */

    uint16_t priority;        /* Priority level of flow entry. */
    uint8_t reason;           /* One of OFPER_*. */
    uint8_t pad[1];           /* Align to 32-bits. */

    uint32_t duration;        /* Time flow was alive in seconds. */
    uint8_t pad2[4];          /* Align to 64-bits. */
    uint64_t packet_count;
    uint64_t byte_count;
};
```

A.6.15 Reworked initial handshake to support backwards compatibility

OpenFlow now includes a basic "version negotiation" capability. When an OpenFlow connection is established, each side of the connection should immediately send an `OFPT_HELLO` message as its first OpenFlow message. The 'version' field in the hello message should be the highest OpenFlow protocol version supported by the sender. Upon receipt of this message, the recipient may calculate the OpenFlow protocol version to be used as the smaller of the version number that it sent and the one that it received.

If the negotiated version is supported by the recipient, then the connection proceeds. Otherwise, the recipient must reply with a message of `OFPT_ERROR` with a 'type' value of `OFPET_HELLO_FAILED`, a 'code' of `OFPHFC_COMPATIBLE`, and optionally an ASCII string explaining the situation in 'data', and then terminate the connection.

The `OFPT_HELLO` message has no body; that is, it consists only of an OpenFlow header. Implementations must be prepared to receive a hello message that includes a body, ignoring its contents, to allow for later extensions.

A.6.16 Description of Switch Stat

The `OFPST_DESC` stat has been added to describe the hardware and software running on the switch:

```
#define DESC_STR_LEN   256
#define SERIAL_NUM_LEN 32
/* Body of reply to OFPST_DESC request.  Each entry is a NULL-terminated
 * ASCII string. */
struct ofp_desc_stats {
    char mfr_desc[DESC_STR_LEN];       /* Manufacturer description. */
    char hw_desc[DESC_STR_LEN];        /* Hardware description. */
    char sw_desc[DESC_STR_LEN];        /* Software description. */
    char serial_num[SERIAL_NUM_LEN];   /* Serial number. */
};
```

It contains a 256 character ASCII description of the manufacturer, hardware type, and software version. It also contains a 32 character ASCII serial number. Each entry is padded on the right with 0 bytes.

A.6.17 Variable Length and Vendor Actions

Vendor-defined actions have been added to OpenFlow. To enable more versatility, actions have switched from fixed-length to variable. All actions have the following header:

```
struct ofp_action_header {
    uint16_t type;                  /* One of OFPAT_*. */
    uint16_t len;                   /* Length of action, including this
                                       header.  This is the length of action,
                                       including any padding to make it
                                       64-bit aligned. */
    uint8_t pad[4];
};
```

The length for actions must always be a multiple of eight to aid in 64-bit alignment. The action types are as follows:

```
enum ofp_action_type {
    OFPAT_OUTPUT,           /* Output to switch port. */
    OFPAT_SET_VLAN_VID,     /* Set the 802.1q VLAN id. */
    OFPAT_SET_VLAN_PCP,     /* Set the 802.1q priority. */
    OFPAT_STRIP_VLAN,       /* Strip the 802.1q header. */
    OFPAT_SET_DL_SRC,       /* Ethernet source address. */
    OFPAT_SET_DL_DST,       /* Ethernet destination address. */
    OFPAT_SET_NW_SRC,       /* IP source address. */
    OFPAT_SET_NW_DST,       /* IP destination address. */
    OFPAT_SET_TP_SRC,       /* TCP/UDP source port. */
    OFPAT_SET_TP_DST,       /* TCP/UDP destination port. */
    OFPAT_VENDOR = 0xffff
};
```

The vendor-defined action header looks like the following:

```
struct ofp_action_vendor_header {
    uint16_t type;                  /* OFPAT_VENDOR. */
    uint16_t len;                   /* Length is 8. */
    uint32_t vendor;                /* Vendor ID, which takes the same form
                                       as in "struct ofp_vendor". */
};
```

The `vendor` field uses the same vendor identifier described earlier in the "Vendor Extensions" section. Beyond using the `ofp_action_vendor` header and the 64-bit alignment requirement, vendors are free to use whatever body for the message they like.

A.6.18 VLAN Action Changes

It is now possible to set the priority field in VLAN tags and stripping VLAN tags is now a separate action. The OFPAT_SET_VLAN_VID action behaves like the former OFPAT_SET_DL_VLAN action, but no longer accepts a special value that causes it to strip the VLAN tag. The OFPAT_SET_VLAN_PCP action modifies the 3-bit priority field in the VLAN tag. For existing tags, both actions only modify the bits associated with the field being updated. If a new VLAN tag needs to be added, the value of all other fields is zero.

The OFPAT_SET_VLAN_VID action looks like the following:

```
struct ofp_action_vlan_vid {
    uint16_t type;                  /* OFPAT_SET_VLAN_VID. */
    uint16_t len;                   /* Length is 8. */
    uint16_t vlan_vid;              /* VLAN id. */
    uint8_t pad[2];
};
```

The OFPAT_SET_VLAN_PCP action looks like the following:

```
struct ofp_action_vlan_pcp {
    uint16_t type;                  /* OFPAT_SET_VLAN_PCP. */
    uint16_t len;                   /* Length is 8. */
    uint8_t vlan_pcp;               /* VLAN priority. */
    uint8_t pad[3];
};
```

The OFPAT_STRIP_VLAN action takes no argument and strips the VLAN tag if one is present.

A.6.19 Max Supported Ports Set to 65280

What: Increase maximum number of ports to support large vendor switches; was previously 256, chosen arbitrarily.

Why: The HP 5412 chassis supports 288 ports of Ethernet, and some Cisco switches go much higher. The current limit (OFPP_MAX) is 255, set to equal the maximum number of ports in a bridge segment in the 1998 STP spec. The RSTP spec from 2004 supports up to 4096 (12 bits) of ports.

How: Change OFPP_MAX to 65280. (However, out of the box, the reference switch implementation supports at most 256 ports.)

A.6.20 Send Error Message When Flow Not Added Due To Full Tables

The switch now sends an error message when a flow is added, but cannot because all the tables are full. The message has an error type of OFPET_FLOW_MOD_FAILED and code of OFPFMFC_ALL_TABLES_FULL. If the Flow-Mod command references a buffered packet, then actions are not performed on the packet. If the controller wishes the packet to be sent regardless of whether or not a flow entry is added, then it should use a Packet-Out directly.

A.6.21 Behavior Defined When Controller Connection Lost

What: Ensure that all switches have at least one common behavior when the controller connection is lost.

Why: When the connection to the controller is lost, the switch should behave in a well-defined way. Reasonable behaviors include 'do nothing - let flows naturally timeout', 'freeze timeouts', 'become learning switch', and 'attempt connection to other controller'. Switches may implement one or more of these, and network admins may want to ensure that if the controller goes out, they know what the network can do.

The first is the simplest: ensure that every switch implements a default of 'do nothing - let flows timeout naturally'. Changes must be done via vendor-specific command line interface or vendor extension OpenFlow messages.

The second may help ensure that a single controller can work with switches from multiple vendors. The different failure behaviors, plus 'other', could be feature bits set for the switch. A switch would still only have to support the default.

The worry here is that we may not be able to enumerate in advance the full range of failure behaviors, which argues for the first approach.

How: Added text to spec: "In the case that the switch loses contact with the controller, the default behavior must be to do nothing - to let flows timeout naturally. Other behaviors can be implemented via vendor-specific command line interface or vendor extension OpenFlow messages."

A.6.22 ICMP Type and Code Fields Now Matchable

What: Allow matching ICMP traffic based on type or code.

Why: We can't distinguish between different types of ICMP traffic (e.g., echo replies vs echo requests vs redirects).

How: Changed spec to allow matching on these fields.

As for implementation: The type and code are each a single byte, so they easily fit in our existing flow structure. Overload the `tp_src` field to ICMP type and `tp_dst` to ICMP code. Since they are only a single byte, they will reside in the low-byte of these two byte fields (stored in network-byte order). This will allow a controller to use the existing wildcard bits to wildcard these ICMP fields.

A.6.23 Output Port Filtering for Delete*, Flow Stats and Aggregate Stats

Add support for listing and deleting entries based on an output port.

To support this, an `out_port` field has been added to the `ofp_flow_mod`, `ofp_flow_stats_request`, and `ofp_aggregate_stats_request` messages. If an `out_port` contains a value other than `OFPP_NONE`, it introduces a constraint when matching. This constraint is that the rule must contain an output action directed at that port. Other constraints such as `ofp_match` structs and priorities are still used; this is purely an *additional* constraint. Note that to get previous behavior, though, `out_port` must be set

to `OFPP_NONE`, since "0" is a valid port id. This only applies to the `delete` and `delete_strict` flow mod commands; the field is ignored by `add`, `modify`, and `modify_strict`.

A.7 OpenFlow version 0.9

Release date : July 20, 2009
Wire Protocol : 0x98

A.7.1 Failover

The reference implementation now includes a simple failover mechanism. A switch can be configured with a list of controllers. If the first controller fails, it will automatically switch over to the second controller on the list.

A.7.2 Emergency Flow Cache

The protocol and reference implementation have been extended to allow insertion and management of emergency flow entries.

Emergency-specific flow entries are inactive until a switch loses connectivity from the controller. If this happens, the switch invalidates all normal flow table entries and copies all emergency flows into the normal flow table.

Upon connecting to a controller again, all entries in the flow cache stay active. The controller then has the option of resetting the flow cache if needed.

A.7.3 Barrier Command

The Barrier Command is a mechanism to get notified when an OpenFlow message has finished executing on the switch. When a switch receives a Barrier message it must first complete all commands sent before the Barrier message before executing any commands after it. When all commands before the Barrier message have completed, it must send a Barrier Reply message back to the controller.

A.7.4 Match on VLAN Priority Bits

There is an optional new feature that allows matching on priority VLAN fields. Pre 0.9, the VLAN id is a field used in identifying a flow, but the priority bits in the VLAN tag are not. In this release we include the priority bits as a separate field to identify flows. Matching is possible as either an exact match on the 3 priority bits, or as a wildcard for the entire 3 bits.

A.7.5 Selective Flow Expirations

Flow expiration messages can now be requested on a per-flow, rather than per-switch granularity.

A.7.6 Flow Mod Behavior

There now is a `CHECK_OVERLAP` flag to flow mods which requires the switch to do the (potentially more costly) check that there doesn't already exist a conflicting flow with the same priority. If there is one, the mod fails and an error code is returned. Support for this flag is required in an OpenFlow switch.

A.7.7 Flow Expiration Duration

The meaning of the "duration" field in the Flow Expiration message has been changed slightly. Previously there were conflicting definitions of this in the spec. In 0.9 the value returned will be the time that the flow was active and not include the timeout period.

A.7.8 Notification for Flow Deletes

If a controller deletes a flow it now receives a notification if the notification bit is set. In previous releases only flow expirations but not delete actions would trigger notifications.

A.7.9 Rewrite DSCP in IP ToS header

There is now an added Flow action to rewrite the DiffServ CodePoint bits part of the IP ToS field in the IP header. This enables basic support for basic QoS with OpenFlow in some switches. A more complete QoS framework is planned for a future OpenFlow release.

A.7.10 Port Enumeration now starts at 1

Previous releases of OpenFlow had port numbers start at 0, release 0.9 changes them to start at 1.

A.7.11 Other changes to the Specification

- 6633/TCP is now the recommended default OpenFlow Port. Long term the goal is to get a IANA approved port for OpenFlow.
- The use of "Type 1" and "Type 0" has been depreciated and references to it have been removed.
- Clarified Matching Behavior for Flow Modification and Stats
- Made explicit that packets received on ports that are disabled by spanning tree must follow the normal flow table processing path.
- Clarified that transaction ID in header should match offending message for `OFPET_BAD_REQUEST`, `OFPET_BAD_ACTION`, `OFPET_FLOW_MOD_FAILED`.
- Clarified the format for the Strip VLAN Action
- Clarify behavior for packets that are buffered on the switch while switch is waiting for a reply from controller
- Added the new `EPERM` Error Type
- Fixed Flow Table Matching Diagram
- Clarified datapath ID 64 bits, up from 48 bits
- Clarified `miss-send-len` and `max-len` of output action

A.8 OpenFlow version 1.0

Release date : December 31, 2009
Wire Protocol : 0x01

A.8.1 Slicing

OpenFlow now supports multiple queues per output port. Queues support the ability to provide minimum bandwidth guarantees; the bandwidth allocated to each queue is configurable. The name slicing is derived from the ability to provide a slice of the available network bandwidth to each queue.

A.8.2 Flow cookies

Flows have been extended to include an opaque identifier, referred to as a cookie. The cookie is specified by the controller when the flow is installed; the cookie will be returned as part of each flow stats and flow expired message.

A.8.3 User-specifiable datapath description

The OFPST_DESC (switch description) reply now includes a datapath description field. This is a user-specifiable field that allows a switch to return a string specified by the switch owner to describe the switch.

A.8.4 Match on IP fields in ARP packets

The reference implementation can now match on IP fields inside ARP packets. The source and destination protocol address are mapped to the nw_src and nw_dst fields respecitively, and the opcode is mapped to the nw_proto field.

A.8.5 Match on IP ToS/DSCP bits

OpenFlow now supports matching on the IP ToS/DSCP bits.

A.8.6 Querying port stats for individual ports

Port stat request messages include a `port_no` field to allow stats for individual ports to be queried. Port stats for all ports can still be requested by specifying `OFPP_NONE` as the port number.

A.8.7 Improved flow duration resolution in stats/expiry messages

Flow durations in stats and expiry messages are now expressed with nanosecond resolution. Note that the accuracy of flow durations in the reference implementation is on the order of milliseconds. (The actual accuracy is in part dependent upon kernel parameters.)

A.8.8 Other changes to the Specification

- remove `multi_phy_tx` spec text and capability bit
- clarify execution order of actions
- replace SSL refs with TLS
- resolve overlap ambiguity
- clarify flow mod to non-existing port
- clarify port definition
- update packet flow diagram
- update header parsing diagram for ICMP
- fix English ambiguity for flow-removed messages
- fix async message English ambiguity
- note that multiple controller support is undefined
- clarify that byte equals octet
- note counter wrap-around
- removed warning not to build a switch from this specification

A.9 OpenFlow version 1.1

Release date : February 28, 2011
Wire Protocol : 0x02

A.9.1 Multiple Tables

Prior versions of the OpenFlow specification did expose to the controller the abstraction of a single table. The OpenFlow pipeline could internally be mapped to multiple tables, such as having a separate wildcard and exact match table, but those tables would always act logically as a single table.

OpenFlow 1.1 introduces a more flexible pipeline with multiple tables. Exposing multiple tables has many advantages. The first advantage is that many hardware have multiple tables internally (for example L2 table, L3 table, multiple TCAM lookups), and the multiple table support of OpenFlow may enable to expose this hardware with greater efficiency and flexibility. The second advantage is that many network deployments combine orthogonal processing of packets (for example ACL, QoS and routing), forcing all those processing in a single table creates huge ruleset due to the cross product of individual rules, multiple tables may decouple properly those processing.

The new OpenFlow pipeline with multiple table is quite different from the simple pipeline of prior OpenFlow versions. The new OpenFlow pipeline expose a set of completely generic tables, supporting the full match and full set of actions. It's difficult to build a pipeline abstraction that represent accurately all possible hardware, therefore OpenFlow 1.1 is based on a generic and flexible pipeline that may be

mapped to the hardware. Some limited table capabilities are available to denote what each table is capable of supporting.

Packets are processed through the pipeline, they are matched and processed in the first table, and may be matched and processed in other tables. As it goes through the pipeline, a packet is associated with an action set, accumulating action, and a generic metadata register. The action set is resolved at the end of the pipeline and applied to the packet. The metadata can be matched and written at each table and enables to carry state between tables.

OpenFlow introduces a new protocol object called instruction to control pipeline processing. Actions which were directly attached to flows in previous versions are now encapsulated in instructions, instructions may apply those actions between tables or accumulate them in the packet action set. Instructions can also change the metadata, or direct packet to another table.

- The switch now expose a pipeline with multiple tables
- Flow entry have instructions to control pipeline processing
- Controller can choose packet traversal of tables via goto instruction
- Metadata field (64 bits) can be set and match in tables
- Packet actions can be merged in packet action set
- Packet action set is executed at the end of pipeline
- Packet actions can be applied between table stages
- Table miss can send to controller, continue to next table or drop
- Rudimentary table capability and configuration

A.9.2 Groups

The new group abstraction enables OpenFlow to represent a set of ports as a single entity for forwarding packets. Different types of groups are provided, to represent different abstractions such as multicasting or multipathing. Each group is composed of a set group buckets, each group bucket contains the set of actions to be applied before forwarding to the port. Groups buckets can also forward to other groups, enabling to chain groups together.

- Group indirection to represent a set of ports
- Group table with 4 types of groups :
  - All - used for multicast and flooding
  - Select - used for multipath
  - Indirect - simple indirection
  - Fast Failover - use first live port
- Group action to direct a flow to a group
- Group buckets contains actions related to the individual port

A.9.3 Tags : MPLS & VLAN

Prior versions of the OpenFlow specification had limited VLAN support, it only supported a single level of VLAN tagging with ambiguous semantic. The new tagging support has explicit actions to add, modify and remove VLAN tags, and can support multiple level of VLAN tagging. It also adds similar support the MPLS shim headers.

- Support for VLAN and QinQ, adding, modifying and removing VLAN headers
- Support for MPLS, adding, modifying and removing MPLS shim headers

A.9.4 Virtual ports

Prior versions of the OpenFlow specification assumed that all the ports of the OpenFlow switch were physical ports. This version of the specification add support for virtual ports, which can represent complex forwarding abstractions such as LAGs or tunnels.

- Make port number 32 bits, enable larger number of ports
- Enable switch to provide virtual ports as OpenFlow ports
- Augment packet-in to report both virtual and physical ports

A.9.5 Controller connection failure

Prior versions of the OpenFlow specification introduced the emergency flow cache as a way to deal with the loss of connectivity with the controller. The emergency flow cache feature was removed in this version of the specification, due to the lack of adoption, the complexity to implement it and other issues with the feature semantic.

This version of the specification adds two simpler modes to deal with the loss of connectivity with the controller. In fail secure mode, the switch continues operating in OpenFlow mode, until it reconnects to a controller. In fail standalone mode, the switch revert to using normal processing (Ethernet switching).

- Remove Emergency Flow Cache from spec
- Connection interruption triggers fail secure or fail standalone mode

A.9.6 Other changes

- Remove 802.1d-specific text from the specification
- Cookie Enhancements Proposal - cookie mask for filtering
- Set_queue action (unbundled from output port action)
- Maskable DL and NW address match fields
- Add TTL decrement, set and copy actions for IPv4 and MPLS
- SCTP header matching and rewriting support
- Set ECN action
- Define message handling : no loss, may reorder if no barrier
- Rename `VENDOR` APIs to `EXPERIMENTER` APIs
- Many other bug fixes, rewording and clarifications

A.10 OpenFlow version 1.2

Release date : December 5, 2011
Wire Protocol : 0x03

Please refers to the bug tracking ID for more details on each change

A.10.1 Extensible match support

Prior versions of the OpenFlow specification used a static fixed length structure to specify `ofp_match`, which prevents flexible expression of matches and prevents inclusion of new match fields. The `ofp_match` has been changed to a TLV structure, called OpenFlow Extensible Match (OXM), which dramatically increases flexibility.

The match fields themselves have been reorganised. In the previous static structure, many fields were overloaded ; for example `tcp.src_port`, `udp.src_port`, and `icmp.code` were using the same field entry. Now, every logical field has its own unique type.

List of features for OpenFlow Extensible Match :

- Flexible and compact TLV structure called OXM (EXT-1)
- Enable flexible expression of match, and flexible bitmasking (EXT-1)
- Pre-requisite system to insure consistency of match (EXT-1)
- Give every match field a unique type, remove overloading (EXT-1)
- Modify VLAN matching to be more flexible (EXT-26)
- Add vendor classes and experimenter matches (EXT-42)
- Allow switches to override match requirements (EXT-56, EXT-33)

A.10.2 Extensible 'set_field' packet rewriting support

Prior versions of the OpenFlow specification were using hand-crafted actions to rewrite header fields. The Extensible `set_field` action reuses the OXM encoding defined for matches, and enables to rewrite any header field in a single action (EXT-13). This allows any new match field, including experimenter fields, to be available for rewrite. This makes the specification cleaner and eases cost of introducing new fields.

- Deprecate most header rewrite actions
- Introduce generic `set-field` action (EXT-13)
- Reuse match TLV structure (OXM) in `set-field` action

A.10.3 Extensible context expression in 'packet-in'

The `packet-in` message did include some of the packet context (ingress port), but not all (metadata), preventing the controller from figuring how match did happen in the table and which flow entries would match or not match. Rather than introduce a hard coded field in the `packet-in` message, the flexible OXM encoding is used to carry packet context.

- Reuse match TLV structure (OXM) to describe metadata in packet-in (EXT-6)
- Include the 'metadata' field in packet-in
- Move ingress port and physical port from static field to OXM encoding
- Allow to optionally include packet header fields in TLV structure

A.10.4 Extensible Error messages via experimenter error type

An experimenter error code has been added, enabling experimenter functionality to generate custom error messages (EXT-2). The format is identical to other experimenter APIs.

A.10.5 IPv6 support added

Basic support for IPv6 match and header rewrite has been added, via the Flexible match support.

- Added support for matching on IPv6 source address, destination address, protocol number, traffic class, ICMPv6 type, ICMPv6 code and IPv6 neighbor discovery header fields (EXT-1)
- Added support for matching on IPv6 flow label (EXT-36)

A.10.6 Simplified behaviour of flow-mod request

The behaviour of flow-mod request has been simplified (EXT-30).

- MODIFY and MODIFY_STRICT commands never insert new flows in the table
- New flag OFPFF_RESET_COUNTS to control counter reset
- Remove quirky behaviour for cookie field.

A.10.7 Removed packet parsing specification

The OpenFlow specification no longer attempts to define how to parse packets (EXT-3). The match fields are only defined logically.

- OpenFlow does not mandate how to parse packets
- Parsing consistency acheived via OXM pre-requisite

A.10.8 Controller role change mechanism

The controller role change mechanism is a simple mechanism to support multiple controllers for failover (EXT-39). This scheme is entirely driven by the controllers ; the switch only need to remember the role of each controller to help the controller election mechanism.

- Simple mechanism to support multiple controllers for failover
- Switches may now connect to multiple controllers in parallel
- Enable each controller to change its roles to equal, master or slave

A.10.9 Other changes

- Per-table metadata bitmask capabilities (EXT-34)
- Rudimentary group capabilities (EXT-61)
- Add hard timeout info in flow-removed messages (OFP-283)
- Add ability for controller to detect STP support(OFP-285)
- Turn off packet buffering with OFPCML_NO_BUFFER (EXT-45)
- Added ability to query all queues (EXT-15)
- Added experimenter queue property (EXT-16)
- Added max-rate queue property (EXT-21)
- Enable deleting flow in all tables (EXT-10)
- Enable switch to check chaining when deleting groups (EXT-12)
- Enable controller to disable buffering (EXT-45)
- Virtual ports renamed logical ports (EXT-78)
- New error messages (EXT-1, EXT-2, EXT-12, EXT-13, EXT-39, EXT-74 and EXT-82)
- Include release notes into the specification document
- Many other bug fixes, rewording and clarifications

A.11 OpenFlow version 1.3

Release date : April 13, 2012
Wire Protocol : 0x04

Please refers to the bug tracking ID for more details on each change

A.11.1 Refactor capabilities negotiation

Prior versions of the OpenFlow specification included limited expression of the capabilities of an OpenFlow switch. OpenFlow 1.3 include a more flexible framework to express capabilities (EXT-123).

The main change is the improved description of table capabilities. Those capabilities have been moved out of the table statistics structure in its own request/reply message, and encoded using a flexible TLV format. This enables the additions of next-table capabilities, table-miss flow entry capabilities and experimenter capabilities.

Other changes include renaming the 'stats' framework into the 'multipart' framework to reflect the fact that it is now used for both statistics and capabilities, and the move of port descriptions into its own multipart message to enable support of a greater number of ports.

List of features for Refactor capabilities negotiation :

- Rename 'stats' framework into the 'multipart' framework.
- Enable 'multipart' requests (requests spanning multiple messages).
- Move port list description to its own multipart request/reply.
- Move table capabilities to its own multipart request/reply.
- Create flexible property structure to express table capabilities.
- Enable to express experimenter capabilities.
- Add capabilities for table-miss flow entries.

- Add next-table (i.e. goto) capabilities

A.11.2 More flexible table miss support

Prior versions of the OpenFlow specification included table configuration flags to select one of three 3 behaviour for handling table-misses (packet not matching any flows in the table). OpenFlow 1.3 replace those limited flags with the table-miss flow entry, a special flow entry describing the behaviour on table miss (EXT-108).

The table-miss flow entry uses standard OpenFlow instructions and actions to process table-miss packets, this enables to use the full flexibility of OpenFlow in processing those packets. All previous behaviour expressed by the table-miss config flags can be expressed using the table-miss flow entry. Many new way of handling table-miss, such as processing table-miss with normal, can now trivially be described by the OpenFlow protocol.

- Remove table-miss config flags (EXT-108).
- Define table-miss flow entry as the all wildcard, lowest priority flow entry (EXT-108).
- Mandate support of the table-miss flow entry in every table to process table-miss packets (EXT-108).
- Add capabilities to describe the table-miss flow entry (EXT-123).
- Change table-miss default to drop packets (EXT-119).

A.11.3 IPv6 Extension Header handling support

Add the ability of match the presence of common IPv6 extension headers, and some anomalous conditions in IPv6 extension headers (EXT-38). A new OXM pseudo header field `OXM_OF_IPV6_EXTHDR` enables to match the following conditions :

- Hop-by-hop IPv6 extension header is present.
- Router IPv6 extension header is present.
- Fragmentation IPv6 extension header is present.
- Destination options IPv6 extension headers is present.
- Authentication IPv6 extension header is present.
- Encrypted Security Payload IPv6 extension header is present.
- No Next Header IPv6 extension header is present.
- IPv6 extension headers out of preferred order.
- Unexpected IPv6 extension header encountered.

A.11.4 Per flow meters

Add support for per-flow meters (EXT-14). Per-flow meters can be attached to flow entries and can measure and control the rate of packets. One of the main applications of per-flow meters is to rate limit packets sent to the controller.

The per-flow meter feature is based on a new flexible meter framework, which includes the ability to describe complex meters through the use of multiple metering bands, metering statistics and capabilities. Currently, only simple rate-limiter meters are defined over this framework. Support for color-aware

meters, which support Diff-Serv style operation and are tightly integrated in the pipeline, was postponed to a later release.

- Flexible meter framework based on per-flow meters and meter bands.
- Meter statistics, including per band statistics.
- Enable to attach meters flexibly to flow entries.
- Simple rate-limiter support (drop packets).

A.11.5 Per connection event filtering

Previous version of the specification introduced the ability for a switch to connect to multiple controllers for fault tolerance and load balancing. Per connection event filtering improves the multi-controller support by enabling each controller to filter events from the switch it does not want (EXT-120).

A new set of OpenFlow messages enables a controller to configure an event filter on its own connection to the switch. Asynchronous messages can be filtered by type and reason. This event filter comes in addition to other existing mechanisms that enable or disable asynchronous messages, for example the generation of flow-removed events can be configured per flow. Each controller can have a separate filter for the slave role and the master/equal role.

- Add asynchronous message filter for each controller connection.
- Controller message to set/get the asynchronous message filter.
- Set default filter value to match OpenFlow 1.2 behaviour.
- Remove `OFPC_INVALID_TTL_TO_CONTROLLER` config flag.

A.11.6 Auxiliary connections

In previous version of the specification, the channel between the switch and the controller is exclusively made of a single TCP connection, which does not allow to exploit the parallelism available in most switch implementations. OpenFlow 1.3 enables a switch to create auxiliary connections to supplement the main connection between the switch and the controller (EXT-114). Auxiliary connections are mostly useful to carry packet-in and packet-out messages.

- Enable switch to create auxiliary connections to the controller.
- Mandate that auxiliary connection can not exist when main connection is not alive.
- Add auxiliary-id to the protocol to disambiguate the type of connection.
- Enable auxiliary connection over UDP and DTLS.

A.11.7 MPLS BoS matching

A new OXM field `OXM_OF_MPLS_BOS` has been added to match the Bottom of Stack bit (BoS) from the MPLS header (EXT-85). The BoS bit indicates if other MPLS shim header are in the payload of the present MPLS packet, and matching this bit can help to disambiguate case where the MPLS label is reused across levels of MPLS encapsulation.

A.11.8 Provider Backbone Bridging tagging

Add support for tagging packet using Provider Backbone Bridging (PBB) encapsulation (EXT-105). This support enables OpenFlow to support various network deployment based on PBB, such as regular PBB and PBB-TE.

- Push and Pop operation to add PBB header as a tag.
- New OXM field to match I-SID for the PBB header.

A.11.9 Rework tag order

In previous version of the specification, the final order of tags in a packet was statically specified. For example, a MPLS shim header was always inserted after all VLAN tags in the packet. OpenFlow 1.3 removes this restriction, the final order of tags in a packet is dictated by the order of the tagging operations, each tagging operation adds its tag in the outermost position (EXT-121).

- Remove defined order of tags in packet from the specification.
- Tags are now always added in the outermost possible position.
- Action-list can add tags in arbitrary order.
- Tag order is predefined for tagging in the action-set.

A.11.10 Tunnel-ID metadata

The logical port abstraction enables OpenFlow to support a wide variety of encapsulations. The tunnel-id metadata `OXM_OF_TUNNEL_ID` is a new OXM field that expose to the OpenFlow pipeline metadata associated with the logical port, most commonly the demultiplexing field from the encapsulation header (EXT-107).

For example, if the logical port perform GRE encapsulation, the tunnel-id field would map to the GRE key field from the GRE header. After decapsulation, OpenFlow would be able to match the GRE key in the tunnel-id match field. Similarly, by setting the tunnel-id, OpenFlow would be able to set the GRE key in an encapsulated packet.

A.11.11 Cookies in packet-in

A cookie field was added to the packet-in message (EXT-7). This field takes its value from the flow the sends the packet to the controller. If the packet was not sent by a flow, this field is set to 0xffffffffffffffff.

Having the cookie in the packet-in enables the controller to more efficiently classify packet-in, rather than having to match the packet against the full flow table.

A.11.12 Duration for stats

A duration field was added to most statistics, including port statistics, group statistics, queue statistics and meter statistics (EXT-102). The duration field enables to more accurately calculate packet and byte rate from the counters included in those statistics.

A.11.13 On demand flow counters

New flow-mod flags have been added to disable packet and byte counters on a per-flow basis. Disabling such counters may improve flow handling performance in the switch.

A.11.14 Other changes

- Fix a bug describing VLAN matching (EXT-145).
- Flow entry description now mention priority (EXT-115).
- Flow entry description now mention timeout and cookies (EXT-147).
- Unavailable counters must now be set to all 1 (EXT-130).
- Correctly refer to flow entry instead of rule (EXT-132).
- Many other bug fixes, rewording and clarifications.

A.12 OpenFlow version 1.3.1

Release date : September 06, 2012
Wire Protocol : 0x04

Please refers to the bug tracking ID for more details on each change

A.12.1 Improved version negotiation

Prior versions of the OpenFlow specification included a simple scheme for version negotiation, picking the lowest of the highest version supported by each side. Unfortunately this scheme does not work properly in all cases, if both implementations don't implement all versions up to their highest version, the scheme can fail to negotiate a version they have in common (EXT-157).

The main change is adding a bitmap of version numbers in the Hello messages using during negotiation. By having the full list of version numbers, negotiation can always negotiate the appropriate version if one is available. This version bitmap is encoded in a flexible TLV format to retain future extensibility of the Hello message.

List of features for Improved version negotiation :

- Hello Elements, new flexible TLV format for Hello message
- Optional version bitmap in Hello messages.
- Improve version negotiation using optional version bitmaps.

A.12.2 Other changes

- Mandate that table-miss flow entry support drop and controller (EXT-158).
- Clarify the mapping of encapsulation data in `OXM_OF_TUNNEL_ID` (EXT-161).
- Rules and restrictions for UDP connections (EXT-162).
- Clarify virtual meters (EXT-165).
- Remove reference to switch fragmentation - confusing (EXT-172).
- Fix meter constant names to always be multipart (`OFPST_ => OFPMT_`) (EXT-184).

- Add `OFPG_*` definitions to spec (EXT-198).
- Add `ofp_instruction` and `ofp_table_feature_prop_header` in spec text (EXT-200).
- Bad error code in connection setup, must be `OFPHFC_INCOMPATIBLE` (EXT-201).
- Instructions must be a multiple of 8 bytes in length (EXT-203).
- Port status includes a reason, not a status (EXT-204).
- Clarify usage of table config field (EXT-205).
- Clarify that required match fields don't need to be supported in every flow table (EXT-206).
- Clarify that prerequisite does not require full match field support (EXT-206).
- Include in the spec missing definitions from openflow.h (EXT-207).
- Fix invalid error code `OFPQCFC_EPERM -> OFPSCFC_EPERM` (EXT-208).
- Clarify PBB language about B-VLAN (EXT-215)
- Fix inversion between source and destination ethernet addresses (EXT-215)
- Clarify how to reorder group buckets, and associated group bucket clarifications (EXT-217).
- Add disclaimer that release notes may not match specification (EXT-218)
- Figure 1 still says "Secure Channel" (EXT-222).
- OpenFlow version must be calculated (EXT-223).
- Meter band drop precedence should be increased, not reduced (EXT-225)
- Fix ambiguous uses of may/can/should/must (EXT-227)
- Fix typos (EXT-228)
- Many typos (EXT-231)

A.13 OpenFlow version 1.3.2

Release date : April 25, 2013
Wire Protocol : 0x04

Please refers to the bug tracking ID for more details on each change

A.13.1 Changes

- Mandate in OXM that 0-bits in mask must be 0-bits in value (EXT-238).
- Allow connection initiated from one of the controllers (EXT-252).
- Add clause on frame misordering to spec (EXT-259).
- Set table features doesn't generate flow removed messages (EXT-266).
- Fix description of set table features error response (EXT-267).
- Define use of `generation_id` in role reply messages (EXT-272).
- Switch with only one flow table are not mandated to implement goto (EXT-280).

A.13.2 Clarifications

- Clarify that MPLS Pop action uses Ethertype regardless of BOS bit (EXT-194).
- Controller message priorities using auxiliary connections (EXT-240).
- Clarify padding rules and variable size arrays (EXT-251).
- Better description buffer-id in flow mod (EXT-257).
- Semantic of `OFPPS_LIVE` (EXT-258).
- Improve multipart introduction (EXT-263).

- Clarify set table features description (EXT-266).
- Clarify meter flags and burst fields (EXT-270).
- Clarify slave access rights (EXT-271).
- Clarify that a switch can't change a controller role (EXT-276).
- Clarify roles of coexisting master and equal controllers (EXT-277).
- Various typos and rewording (EXT-282, EXT-288, EXT-290)

Appendix B Credits

Spec contributions, in alphabetical order:

Anders Nygren, Ben Pfaff, Bob Lantz, Brandon Heller, Casey Barker, Curt Beckmann, Dan Cohn, Dan Talayco, David Erickson, David McDysan, David Ward, Edward Crabbe, Fabian Schneider, Glen Gibb, Guido Appenzeller, Jean Tourrilhes, Johann Tonsing, Justin Pettit, KK Yap, Leon Poutievski, Lorenzo Vicisano, Martin Casado, Masahiko Takahashi, Masayoshi Kobayashi, Navindra Yadav, Nick McKeown, Nico dHeureuse, Peter Balland, Rajiv Ramanathan, Reid Price, Rob Sherwood, Saurav Das, Shashidhar Gandham, Tatsuya Yabe, Yiannis Yiakoumis, Zoltán Lajos Kis.