OpenFlow Switch Specification

Version 1.1.0 Implemented (Wire Protocol 0x02)

February 28, 2011

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 OpenFlow Consortium website (`http://openflow.org`). 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.

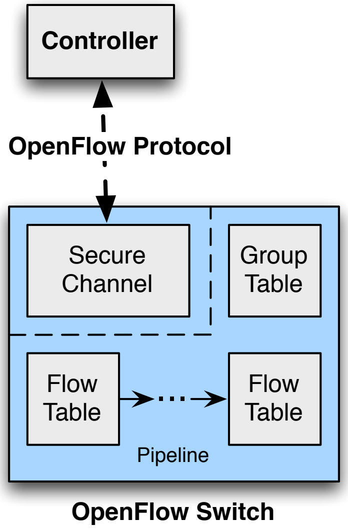


Figure 1: An OpenFlow switch communicates with a controller over a secure connection using the OpenFlow protocol.

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 controller manages the switch via the OpenFlow protocol. Using this protocol, the controller can add, update, and delete *flow entries*, 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 4.1).

Matching starts at the first flow table and may continue to additional flow tables (see 4.1.1). Flow entries match packets in priority order, with the first matching entry in each table being used (see 4.4). 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 switch configuration: the packet may be forwarded to the controller over the OpenFlow channel, dropped, or may continue to the next flow table (see 4.1.1).

Instructions associated with each flow entry describe packet forwarding, packet modification, group table processing, and pipeline processing (see 4.6). 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 4.7).

Flow entries may forward to a *port*. This is usually a physical port, but it may also be a virtual port defined by the switch or a reserved virtual port defined by this specification. Reserved virtual 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 4.9), while switch-defined virtual ports may specify link aggregation groups, tunnels or loopback interfaces (see 4.9).

Flow entries may also point to a group, which specifies additional processing (see 4.2). 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 flows to forward to a single identifier (e.g. IP forwarding to a common next hop). This abstraction allows common output actions across flows 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 4.2.1). 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 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.
- **Pipeline**: the set of linked tables that provide matching, forwarding, and packet modifications in an OpenFlow switch.
- **Port**: where packets enter and exit the OpenFlow pipeline. May be a physical port, a virtual port defined by the switch, or a virtual port defined by the OpenFlow protocol. Reserved virtual ports are ports reserved by this specification (see 4.9). Switch-defined virtual ports are higher level abstractions that may be defined in the switch using non-OpenFlow methods (e.g. link aggregation groups, tunnels, loopback interfaces).
- **Match Field**: a field against which a packet is matched, including packet headers, the ingress port, and the metadata value.
- **Metadata**: a maskable register value that is used to carry information from one table to the next.
- **Instruction**: an operation that either contains a *set* of actions to add to the action set, contains a *list* of actions to apply immediately to the packet, or modifies pipeline processing.
- **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.

- **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.

4 OpenFlow Tables

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

4.1 Flow Table

A flow table consists of flow entries.

Match Fields	Counters	Instructions

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

Each flow table entry (see Table 1) contains:

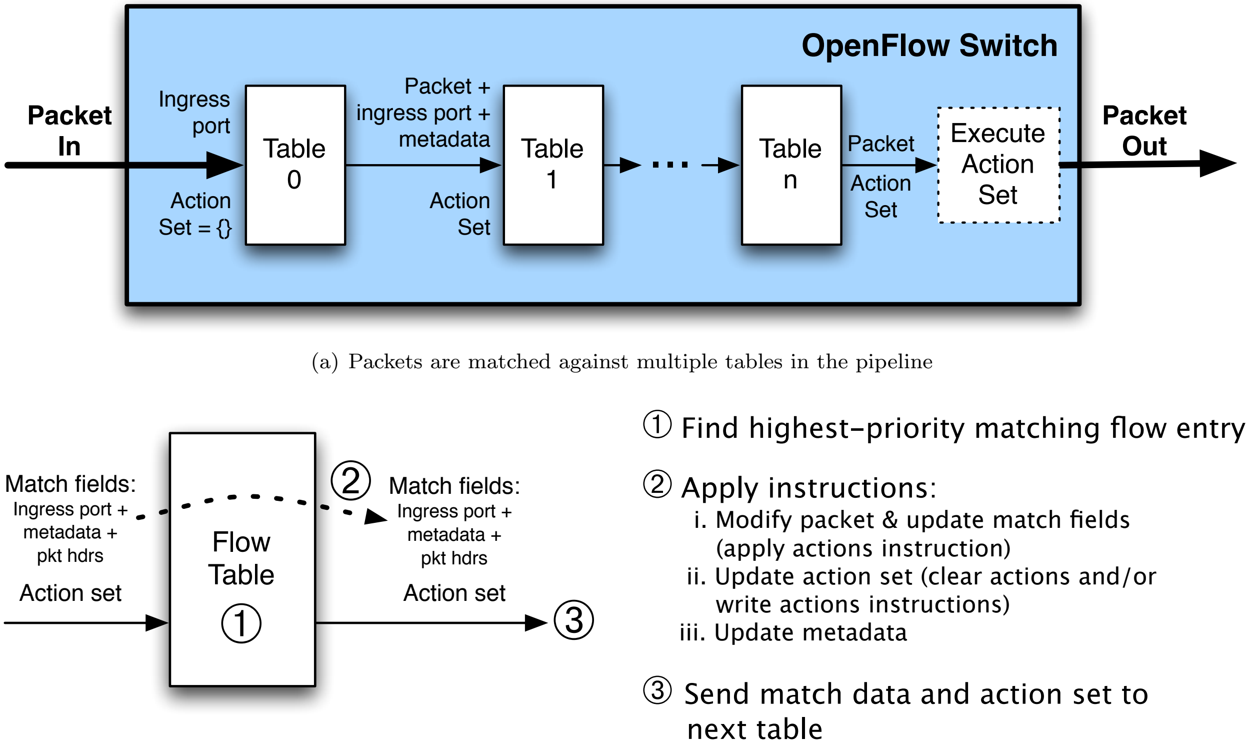
- **match fields**: to match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.
- **counters**: to update for matching packets
- **instructions** to modify the action set or pipeline processing

4.1.1 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, ACL and QoS processing. Those switches should 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 switches may also allow a packet to go from the OpenFlow pipeline to the normal pipeline through the *NORMAL* and *FLOOD* virtual ports (see 4.9).

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 with only a single flow table is valid, in this case pipeline processing is greatly simplified.


OpenFlow Switch
Packet In
Ingress port
Action Set = {}
Table 0
Packet + ingress port + metadata
Action Set
Table 1
Table n
Packet
Action Set
Execute Action Set
Packet Out
Match fields:
Ingress port + metadata + pkt hdrs
Action set
Flow Table
①
②
③
① Find highest-priority matching flow entry
② Apply instructions:
i. Modify packet & update match fields (apply actions instruction)
ii. Update action set (clear actions and/or write actions instructions)
iii. Update metadata
③ Send match data and action set to next table

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

(b) Per-table packet processing

Figure 2: Packet flow through the processing pipeline

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 entries of flow table 0. Other flow tables may be used depending on the outcome of the match in the first table.

If the packet matches a flow entry in a flow table, the corresponding instruction set is executed (see 4.4). The instructions in the flow entry may explicitly direct the packet to another flow table (using the Goto Instruction, see 4.6), 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 4.7).

If the packet does not match a flow entry in a flow table, this is a table miss. The behavior on table miss depends on the table configuration; the default is to send packets to the controller over the control channel via a packet-in message (see 5.1.2), another options is to drop the packet. A table can also specify that on a table miss the packet processing should continue; in this case the packet is processed by the next sequentially numbered table.

4.2 Group Table

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

Each group entry (see Table 2) contains:

Group Identifier	Group Type	Counters	Action Buckets

Table 2: A group entry consists of a group identifier, a group type, counters, and a list of action buckets.

- **group identifier**: a 32 bit unsigned integer uniquely identifying the group
- **group type**: to determine group semantics (see Section 4.2.1)
- **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

4.2.1 Group Types

The following group types are defined:

- **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 should include an extra bucket which includes an output action to the `OFPP_IN_PORT` virtual port.
- **select**: Execute one bucket in the group. Packets are sent to 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. 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.
- **indirect**: Execute the one defined bucket in this group. Allows multiple flows 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.
- **fast failover**: Execute the first live bucket. Each action bucket is associated with a specific port and/or group that controls its liveness. 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 5.8).

4.3 Match Fields

Table 3 shows the match fields an incoming packet is compared against. Each entry contains a specific value, or ANY, which matches any value. If the switch supports arbitrary bitmasks on the Ethernet source and/or destinations fields, or on the IP source and/or destination fields, these masks can more precisely specify matches. The fields in the OpenFlow tuple are listed in Table 3 and details on the properties of each field are described in Table 4. 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.

Ingress Port	Metadata	Ether src	Ether dst	Ether type	VLAN id	VLAN priority	MPLS label	MPLS traffic class	IPv4 src	IPv4 dst	IPv4 proto / ARP opcode	IPv4 ToS bits	TCP/ UDP / SCTP src port ICMP Type	TCP/ UDP / SCTP dst port ICMP Code

Table 3: Fields from packets used to match against flow entries.

4.4 Matching

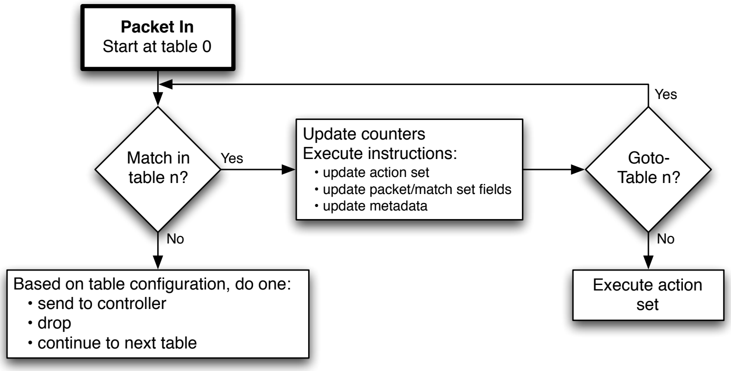


Figure 3: Flowchart detailing packet flow through an OpenFlow switch.

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 lookup in other flow tables (see 4.1.1). Match fields used for table lookups depend on the packet type as in Figure 4.

A packet matches a flow table entry if the values in the match fields used for the lookup (as defined in Figure 4) match those defined in the flow table. If a flow table field has a value of ANY, it matches all possible values in the header.

To handle the various Ethernet framing types, matching the Ethernet type is handled based on the packet frame content. In general, the Ethernet type matched by OpenFlow is the one describing what is considered by OpenFlow as the payload of the packet. If the packet has VLAN tags, the Ethernet type matched is the one found after all the VLAN tags. An exception to that rule is packets with MPLS tags where OpenFlow can not determine the Ethernet type of the MPLS payload of the packet.

If the packet is an Ethernet II frame, the Ethernet type of the Ethernet header (after all VLAN tags) is matched against the flow's Ethernet type. If the packet is an 802.3 frame with a 802.2 LLC header, a SNAP header and Organizationally Unique Identifier (OUI) of 0x000000, the SNAP protocol id is matched against the flow's Ethernet type. A flow entry that specifies an Ethernet type of 0x05FF, matches all 802.3 frames without a SNAP header and those with SNAP headers that do not have an OUI of 0x000000.

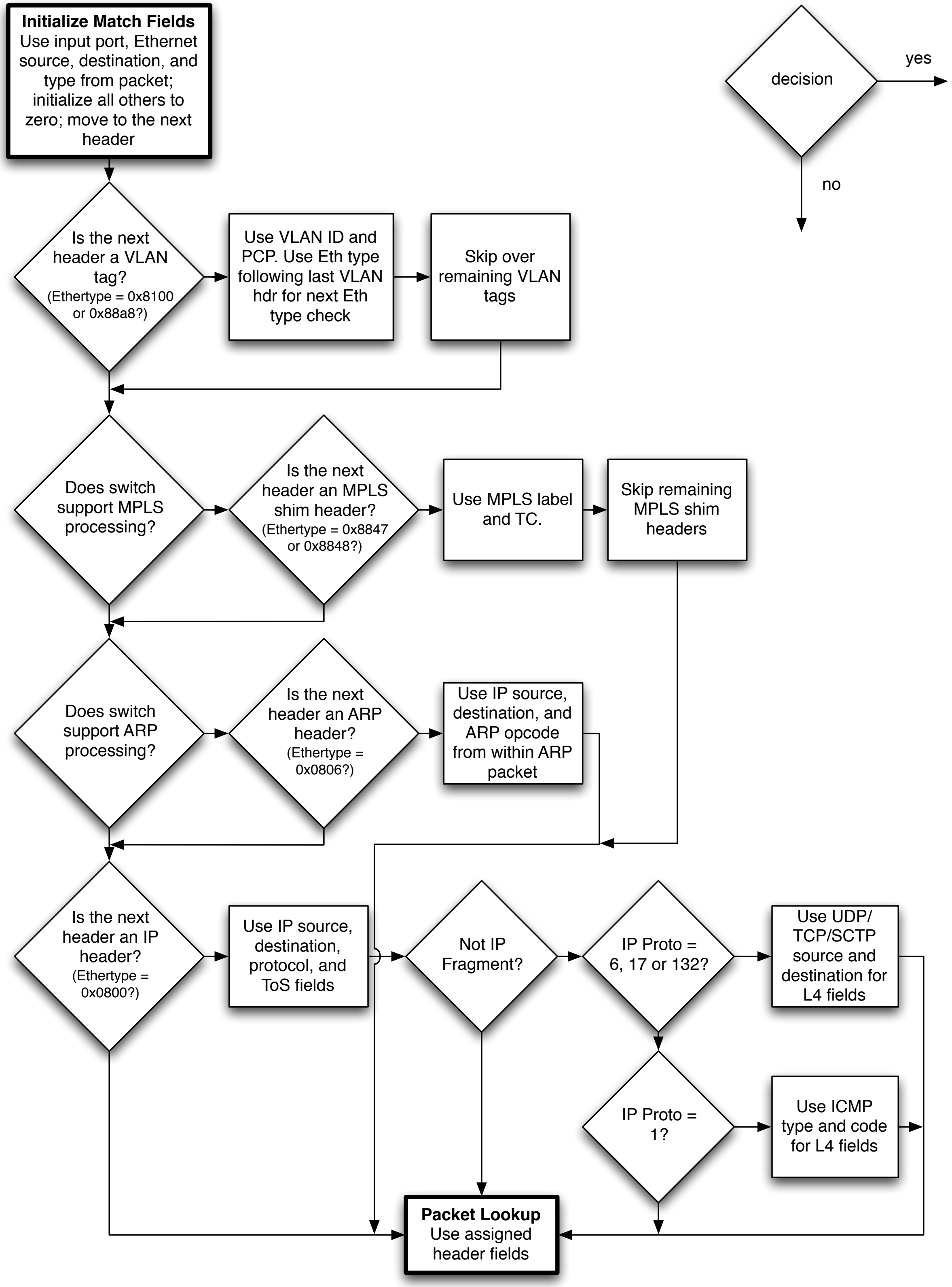

Initialize Match Fields
Use input port, Ethernet source, destination, and type from packet; initialize all others to zero; move to the next header
decision
yes
no
Is the next header a VLAN tag? (Ethertype = 0x8100 or 0x88a8?)
Use VLAN ID and PCP. Use Eth type following last VLAN hdr for next Eth type check
Skip over remaining VLAN tags
Does switch support MPLS processing?
Is the next header an MPLS shim header? (Ethertype = 0x8847 or 0x8848?)
Use MPLS label and TC.
Skip remaining MPLS shim headers
Does switch support ARP processing?
Is the next header an ARP header? (Ethertype = 0x0806?)
Use IP source, destination, and ARP opcode from within ARP packet
Is the next header an IP header? (Ethertype = 0x0800?)
Use IP source, destination, protocol, and ToS fields
Not IP Fragment?
IP Proto = 6, 17 or 132?
Use UDP/ TCP/SCTP source and destination for L4 fields
IP Proto = 1?
Use ICMP type and code for L4 fields
Packet Lookup
Use assigned header fields

Figure 4: Flowchart showing how match fields are parsed for matching.

Field	Bits	When applicable	Notes
Ingress Port	32	All packets	Numerical representation of incoming port, starting at 1. This may be a physical or switch-defined virtual port.
Metadata	64	Table 1 and above	
Ethernet source address	48	All packets on enabled ports	Can use arbitrary bitmask
Ethernet destination address	48	All packets on enabled ports	Can use arbitrary bitmask
Ethernet type	16	All packets on enabled ports	Ethernet type of the OpenFlow packet payload, after VLAN tags. 802.3 frames have special handling.
VLAN id	12	All packets with VLAN tags	VLAN identifier of *outermost* VLAN tag.
VLAN priority	3	All packets with VLAN tags	VLAN PCP field of *outermost* VLAN tag.
MPLS label	20	All packets with MPLS tags	Match on *outermost* MPLS tag.
MPLS traffic class	3	All packets with MPLS tags	Match on *outermost* MPLS tag.
IPv4 source address	32	All IPv4 and ARP packets	Can use subnet mask or arbitrary bitmask
IPv4 destination address	32	All IPv4 and ARP packets	Can use subnet mask or arbitrary bitmask
IPv4 protocol / ARP opcode	8	All IPv4 and IPv4 over Ethernet, ARP packets	Only the lower 8 bits of the ARP opcode are used
IPv4 ToS bits	6	All IPv4 packets	Specify as 8-bit value and place ToS in upper 6 bits.
Transport source port / ICMP Type	16	All TCP, UDP, SCTP, and ICMP packets	Only lower 8 bits used for ICMP Type
Transport destination port / ICMP Code	16	All TCP, UDP, SCTP, and ICMP packets	Only lower 8 bits used for ICMP Code

Table 4: Field lengths and the way they must be applied to flow entries.

The switch should apply the instruction set and update the associated counters of *only* the highest-priority flow entry matching the packet. If there are multiple matching flow entries with the same highest priority, the matching flow entry is explicitly undefined. This case can only arise when a controller writer never sets the `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 A.3.2).

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

4.5 Counters

Counters may be maintained for each table, flow, port, queue, group, and bucket. 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.

Duration refers to the amount of time the flow has been installed in the switch. 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 wrap around with no overflow indicator. If a specific numeric counter is not available in the switch, its value should be set to -1.

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

Table 5: List of counters

4.6 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. Supported instructions include:

- **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 4.8).
- **Clear-Actions**: Clears all the actions in the action set immediately.
- **Write-Actions *action(s)***: Merges the specified action(s) into the current action set (see 4.7). If an action of the given type exists in the current set, overwrite it, otherwise add it.
- **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).
- **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 flows of last table of the pipeline can not include this instruction (see 4.1.1).

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 *Clear-Actions* instruction is executed before the *Write-Actions* instruction, and that *Goto-Table* is executed last.

A switch may 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. Flow tables may not support every match and every instruction.

4.7 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 an instruction set does not contain a *Goto-Table* instruction, pipeline processing stops and the actions in the action set are executed.

An action set contains a maximum of one action of each type. When multiple actions of the same type are required, e.g. pushing multiple MPLS labels or popping multiple MPLS labels, the *Apply-Actions* instruction may be used (see 4.8).

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**: apply all tag push actions to the packet
4. **copy TTL outwards**: apply copy TTL outwards action to the packet
5. **decrement TTL**: apply decrement TTL action to the packet
6. **set**: apply all set-field actions to the packet
7. **qos**: apply all QoS actions, such as set_queue to the packet
8. **group**: if a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list
9. **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; a group bucket may specify another group, in which case the execution of actions traverses all the groups specified by the group configuration.

4.8 Action List

The *Apply-Actions* instruction and the *Packet-out* message include an action list. The semantic 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 action start 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 list contains an output action, a copy of the packet is forwarded in its current state to the desired port. If the list contains a 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 4.1.1). The action set of the packet is unchanged by the execution of the action list.

4.9 Actions

A switch is not required to support all action types — just those marked "Required Actions" below. When connecting to the controller, a switch indicates which of the "Optional Actions" it supports.

Required Action: *Output.* The Output action forwards a packet to a specified port. OpenFlow switches must support forwarding to physical ports and switch-defined virtual ports. Standard ports are defined as physical ports, switch-defined virtual ports, and the `LOCAL` port if supported (excluding other reserved virtual ports). OpenFlow switches must also support forwarding to the following reserved virtual ports:

- **ALL:** Send the packet out all standard ports, but not to the ingress port or ports that are configured `OFPPC_NO_FWD`.
- **CONTROLLER:** Encapsulate and send the packet to the controller.
- **TABLE:** Submit the packet to the first flow table so that the packet can be processed through the regular OpenFlow pipeline. Only valid in the action set of a packet-out message.
- **IN_PORT:** Send the packet out the ingress port.

Optional Action: *Output.* The switch may optionally support forwarding to the following reserved virtual ports:

- **LOCAL:** Send the packet to the switch's local networking stack. The local port enables remote entities to interact with the switch via the OpenFlow network, rather than via a separate control network. With a suitable set of default rules it can be used to implement an in-band controller connection.
- **NORMAL:** Process the packet using the traditional non-OpenFlow pipeline of the switch (see 4.1.1). If the switch cannot forward packets from the OpenFlow pipeline to the normal pipeline, it must indicate that it does not support this action.
- **FLOOD:** Flood the packet using the normal pipeline of the switch (see 4.1.1). In general, send the packet out all standard ports, but not to the ingress port, or ports that are in `OFPPS_BLOCKED` state. The switch may also use the packet VLAN ID to select which ports to flood to.

OpenFlow-only switches do not support output actions to the **NORMAL** port and **FLOOD** port, while *OpenFlow-hybrid* switches may support them. 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 4.2.1).

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 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 A.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.

The ordering of header fields/tags is:

Ethernet	VLAN	MPLS	ARP/IP	TCP/UDP/SCTP (IP-only)

Newly pushed tags should *always* be inserted as the outermost tag in this ordering. When a new VLAN tag is pushed, it should be the outermost VLAN tag inserted immediately after the Ethernet header. Likewise, when a new MPLS tag is pushed, it should be the outermost MPLS tag, inserted as a shim header after any VLAN tags.

Note: Refer to section 4.9.1 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.
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).

Table 6: Push/pop tag actions.

Optional Action: *Set-Field.* The various Set-Field actions modify the values of the respective header field in the packet. While not strictly required, the actions shown in Table 7 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).

Action	**Associated Data**	**Description**
Set Ethernet source MAC address	48 bits: New source MAC address	Replace the existing Ethernet source MAC address.
Set Ethernet destination MAC address	48 bits: New destination MAC address	Replace the existing Ethernet destination MAC address.
Set VLAN ID	12 bits: New VLAN ID	Replace the existing VLAN ID. Only applies to packets with an existing VLAN tag.

Continued on next page

Table 7 – continued from previous page

Action	Associated Data	Description
Set VLAN priority	3 bits: New VLAN priority	Replace the existing VLAN priority. Only applies to packets with an existing VLAN tag.
Set MPLS label	20 bits: New MPLS label	Replace the existing MPLS label. Only applies to packets with an existing MPLS shim header.
Set MPLS traffic class	3 bits: New MPLS traffic class	Replace the existing MPLS traffic class. Only applies to packets with an existing MPLS shim header.
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 IPv4 source address	32 bits: New IPv4 source address	Replace the existing IP source address with new value and update the IP checksum (and TCP/UDP/SCTP checksum if applicable). This action is only applicable to IPv4 packets.
Set IPv4 destination address	32 bits: New IPv4 destination address	Replace the existing IP destination address with and update the IP checksum (and TCP/UDP/SCTP checksum if applicable). This action is only applied to IPv4 packets.
Set IPv4 ToS bits	6 bits: New IPv4 ToS	Replace the existing IP ToS and update the IP checksum. Only applies to IPv4 packets.
Set IPv4 ECN bits	2 bits: New IPv4 ECN	Replace the existing IP ECN value and update the IP checksum. Only applies to IPv4 packets.
Set IPv4 TTL	8 bits: New IPv4 TTL	Replace the existing IP TTL and update the IP checksum. Only applies to IPv4 packets.
Decrement IPv4 TTL	-	Decrement the IP TTL field and update the IP checksum. Only applies to IPv4 packets.
Set transport source port	16 bits: New TCP, UDP or SCTP source port	Replace the existing TCP/UDP/SCTP source port with new value and update the TCP/UDP/SCTP checksum. This action is only applicable to TCP, UDP and SCTP packets.
Set transport destination port	16 bits: New TCP, UDP or SCTP destination port	Replace the existing TCP/UDP/SCTP destination port with new value and update the TCP/UDP/SCTP checksum Only applies to TCP, UDP and SCTP 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.

Continued on next page

Table 7 – concluded from previous page

Action	Associated Data	Description
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: Set-Field actions.

4.9.1 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

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.

5 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.

Support for multiple simultaneous controllers is currently undefined.

5.1 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.

5.1.1 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 capabilities of a switch by sending a features request; the switch must respond with a features reply that specifies the 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 flows/groups in the OpenFlow tables and to set switch port properties.

Read-State: Read-State messages are used by the controller to collect statistics from the switch.

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.

5.1.2 Asynchronous

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

Packet-in: For all packets that do not have a matching flow entry, a packet-in event may be sent to the controller (depending on the table configuration). For all packets forwarded to the **CONTROLLER** virtual port, a packet-in event is always sent to the controller. If the switch has sufficient memory to buffer packets that are sent to the controller, the packet-in events contain some fraction of the packet header (by default 128 bytes) and a buffer ID to be used by the controller when it is ready for the switch to forward the packet. Switches that do not support internal buffering (or have run out of internal buffering) must send the full packet to the controller as part of the event. Buffered packets will usually be processed via a **Packet-out** message from the controller, or automatically expired after some time.

Flow-Removed: When a flow entry is added to the switch by a flow modify message, an idle timeout value indicates when the entry should be removed due to a lack of activity, as well as a hard timeout value that indicates when the entry should be removed, regardless of activity. The flow modify message also specifies whether the switch should send a flow removed message to the controller when the flow expires. Flow delete requests should generate flow removed messages for any flows with the `OFPFF_SEND_FLOW_REM` flag set.

Port-status: The switch is expected to send port-status messages to the controller as port configuration state changes. These events include change in port status events (for example, if it was brought down directly by a user).

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

5.1.3 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 can be used to measure the latency or bandwidth of a controller-switch connection, as well as verify its liveness.

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.

5.2 Connection Setup

The switch must be able to establish communication with a controller at a user-configurable (but otherwise fixed) IP address, using a user-specified port. If the switch knows the IP address of the controller, 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. Future versions of the protocol specification will describe a dynamic controller discovery protocol in which the IP address and port for communicating with the controller is determined at runtime.

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. 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 an `OFPT_ERROR` message with a `type` field of `OFPET_HELLO_FAILED`, a `code` field of `OFPHFC_COMPATIBLE`, and optionally an ASCII string explaining the situation in `data`, and then terminate the connection.

5.3 Connection Interruption

In the case that a switch loses contact with the current controller, as a result of an echo request timeout, TLS session timeout, or other disconnection, it should attempt to contact one or more backup controllers. The ordering by which a switch contacts backup controllers is not specified by the protocol.

The switch should *immediately* enter either “fail secure mode” or “fail standalone mode” if it loses connection to the controller, 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 current controller are dropped. Flows 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` port; in other words, the switch acts as a legacy Ethernet switch or router.

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. Configuration of the default set of flow entries to be used at startup is outside the scope of the OpenFlow protocol.

5.4 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).

5.5 Message Handling

The OpenFlow protocol provides reliable message delivery and processing, but does *not* automatically provide acknowledgements or ensure ordered message processing.

Message Delivery: Messages are guaranteed delivery, unless the connection 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 internal state changes, such as flow-removed or packet-in messages. However, 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.

Controllers are free to drop messages, but should respond to hello and echo messages to prevent the switch from dropping 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, flows 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 (e.g. a flow add with a following packet-out to `OFPP_TABLE`), they should be separated by a barrier message.

5.6 Flow Table Modification Messages

Flow table modification messages can have the following types:

```
enum ofp_flow_mod_command {
    OFPFC_ADD,             /* New flow. */
    OFPFC_MODIFY,          /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT,   /* Modify entry strictly matching wildcards and
                              priority. */
    OFPFC_DELETE,          /* Delete all matching flows. */
    OFPFC_DELETE_STRICT    /* 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 valid (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 counters and duration, must be cleared from the table, and the new flow entry added. 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_STRICT for the old flow 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. For **modify** requests, if no flow currently residing in the requested table matches the request, and if the `cookie_mask` field contains 0, the **modify** acts like an **add**, and the new flow entry must be inserted with zeroed counters.

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 matches, 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 *non-strict* versions, the wildcards are active and all flows that match the description are modified or removed. In the *strict* versions, all fields, including the wildcards and priority, are strictly matched against the entry, and only an identical flow is modified or removed. For example, if a message to remove entries is sent that has all the wildcard flags set, the `OFPFC_DELETE` command would delete all flows from all tables, while the `OFPFC_DELETE_STRICT` command would only delete a rule that applies to all packets at the specified priority.

For *non-strict* **modify** and **delete** commands that contain wildcards, a match will occur when a flow entry exactly matches or is more specific than the description in the flow_mod command. For example, if a `OFPFC_DELETE` command says to delete all flows with a destination port of 80, then a flow entry that is all wildcards will not be deleted. However, a `OFPFC_DELETE` command that is all wildcards 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.

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 rule must contain an *output* action directed at the specified port in the actions associated with that rule. This constraint is limited to only the actions directly associated with the rule. 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's `cookie` value must be equal. In other words, (flow.cookie & flow_mod.cookie_mask) == (flow_mod.cookie & flow_mod.cookie_mask).

If the flow modification message specifies an invalid table or `0xFF`, the switch should send an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_BAD_TABLE_ID` code.

If a switch cannot find any space in the requested table in which to add the incoming flow entry, the switch should 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 instructions requested contain an Experimenter instruction and the particular experimenter instruction is unsupported the switch must return an `ofp_error_msg` with `OFPET_BAD_INSTRUCTION` type and `OFPBIC_UNSUP_EXP_INST`.

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 `wildcards` field that is unsupported in the table, the switch must return an `ofp_error_msg` with `OFPET_BAD_MATCH` type and `OFPBMC_BAD_WILDCARDS` 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 values that cannot be matched, for example, a VLAN ID greater than 4095 and not one of the reserved values, or a ToS value with one of the two lower 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 may 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 should 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 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 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.

5.7 Flow Removal

Each flow entry has an idle_timeout and a hard_timeout associated with it. If either value is non-zero, the switch must note the flow'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. 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. In addition, the controller may actively remove flow entries by sending **delete** flow table modification messages (OFPFC_DELETE or OFPFC_DELETE_STRICT).

When a flow entry is removed, 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 time of removal.

5.8 Group Table Modification Messages

Group table modification messages can have the following types:

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

The action set for each bucket must be validated using the same rules as those for flow mods (Section 5.6), 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 5.6).

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.

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. If a group mod is sent such that a forwarding loop would be created, the switch should 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.

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 flows that forward to the group 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.

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 spec (such as Spanning Tree or a KeepAlive mechanism). At a minimum, the port should not be considered live 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.

Appendix A The OpenFlow Protocol

The heart of the OpenFlow spec 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.

A.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 current draft phase of the OpenFlow Protocol, the most significant bit will be set to indicate an experimental version and the lower bits will indicate a revision number. The current version is 0x02 . 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,               /* Symmetric message */
    OFPT_ERROR,               /* Symmetric message */
    OFPT_ECHO_REQUEST,        /* Symmetric message */
    OFPT_ECHO_REPLY,          /* Symmetric message */
    OFPT_EXPERIMENTER,        /* Symmetric message */

    /* Switch configuration messages. */
    OFPT_FEATURES_REQUEST,    /* Controller/switch message */
    OFPT_FEATURES_REPLY,      /* Controller/switch message */
    OFPT_GET_CONFIG_REQUEST,  /* Controller/switch message */
    OFPT_GET_CONFIG_REPLY,    /* Controller/switch message */
    OFPT_SET_CONFIG,          /* Controller/switch message */
    /* Asynchronous messages. */
    OFPT_PACKET_IN,           /* Async message */
    OFPT_FLOW_REMOVED,        /* Async message */
    OFPT_PORT_STATUS,         /* Async message */

    /* Controller command messages. */
    OFPT_PACKET_OUT,          /* Controller/switch message */
    OFPT_FLOW_MOD,            /* Controller/switch message */
    OFPT_GROUP_MOD,           /* Controller/switch message */
    OFPT_PORT_MOD,            /* Controller/switch message */
    OFPT_TABLE_MOD,           /* Controller/switch message */

    /* Statistics messages. */
    OFPT_STATS_REQUEST,       /* Controller/switch message */
    OFPT_STATS_REPLY,         /* Controller/switch message */

    /* Barrier messages. */
    OFPT_BARRIER_REQUEST,     /* Controller/switch message */
    OFPT_BARRIER_REPLY,       /* Controller/switch message */

    /* Queue Configuration messages. */
    OFPT_QUEUE_GET_CONFIG_REQUEST,  /* Controller/switch message */
    OFPT_QUEUE_GET_CONFIG_REPLY,     /* Controller/switch message */
};
```

A.2 Common Structures

This section describes structures used by multiple messages.

A.2.1 Port Structures

The OpenFlow pipeline receives and sends packets on ports. The switch may define physical and virtual ports, and the OpenFlow specification defines some reserved virtual ports.

The physical ports, switch-defined virtual ports, and the OFPP_LOCAL 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_MAX_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 switch ports. */
    OFPP_MAX        = 0xffffff00,

    /* Fake output "ports". */
    OFPP_IN_PORT    = 0xfffffff8,  /* Send the packet out the input port.  This
                                      virtual 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.

A.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 flows on it. Flows 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. */
    uint16_t len;          /* Length in bytes of this queue desc. */
    uint8_t pad[2];        /* 64-bit alignment. */
    struct ofp_queue_prop_header properties[0]; /* List of properties. */
};
OFP_ASSERT(sizeof(struct ofp_packet_queue) == 8);
```

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

```
enum ofp_queue_properties {
    OFPQT_NONE = 0,       /* No property defined for queue (default). */
    OFPQT_MIN_RATE,       /* Minimum datarate guaranteed. */
                          /* Other types should be added here
                           * (i.e. max rate, precedence, etc). */
};
```

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);
```

Currently, there is only a minimum-rate type queue, described by the `ofp_queue_prop_min_rate` structure:

```
/* 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);
```

A.2.3 Flow Match Structures

When describing a flow entry, the following structures are used:

```
/* 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 "standard" type corresponds to ofp_match and
 * must be supported by all OpenFlow switches. Extensions that define other
 * match types may be published on the OpenFlow wiki. Support for extensions is
 * optional.
 */
enum ofp_match_type {
    OFPMT_STANDARD,           /* The match fields defined in the ofp_match
                                 structure apply */
};

/* Fields to match against flows */
struct ofp_match {
    uint16_t type;             /* One of OFPMT_* */
    uint16_t length;           /* Length of ofp_match */
    uint32_t in_port;          /* Input switch port. */
    uint32_t wildcards;        /* Wildcard fields. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_src_mask[OFP_ETH_ALEN]; /* Ethernet source address mask. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint8_t dl_dst_mask[OFP_ETH_ALEN]; /* Ethernet destination address mask. */
    uint16_t dl_vlan;          /* Input VLAN id. */
    uint8_t dl_vlan_pcp;       /* Input VLAN priority. */
    uint8_t pad1[1];           /* Align to 32-bits */
    uint16_t dl_type;          /* Ethernet frame type. */
    uint8_t nw_tos;            /* IP ToS (actually DSCP field, 6 bits). */
    uint8_t nw_proto;          /* IP protocol or lower 8 bits of
                                * ARP opcode. */
    uint32_t nw_src;           /* IP source address. */
    uint32_t nw_src_mask;      /* IP source address mask. */
    uint32_t nw_dst;           /* IP destination address. */
    uint32_t nw_dst_mask;      /* IP destination address mask. */
    uint16_t tp_src;           /* TCP/UDP/SCTP source port. */
    uint16_t tp_dst;           /* TCP/UDP/SCTP destination port. */
    uint32_t mpls_label;       /* MPLS label. */
    uint8_t mpls_tc;           /* MPLS TC. */
    uint8_t pad2[3];           /* Align to 64-bits */
    uint64_t metadata;         /* Metadata passed between tables. */
    uint64_t metadata_mask;    /* Mask for metadata. */
};
OFP_ASSERT(sizeof(struct ofp_match) == OFPMT_STANDARD_LENGTH);
```

The `type` field is set to `OFPMT_STANDARD` and `length` field is set to `OFPMT_STANDARD_LENGTH` by default. If the match needs to be extended, these fields may be modified.

The constant `OFPMT_STANDARD_LENGTH` is defined to be 88 .

Protocol-specific fields within `ofp_match` will be ignored within a single table when the corresponding protocol is not specified in the match. The MPLS match fields will be ignored unless the Ethertype is specified as MPLS. Likewise, the IP header and transport header fields will be ignored unless the Ethertype is specified as either IPv4 or ARP. The `tp_src` and `tp_dst` fields will be ignored unless the network protocol specified is as TCP, UDP or SCTP. Fields that are ignored don't need to be wildcarded and should be set to 0.

The `wildcards` field has a number of flags that may be set:

```
/* Flow wildcards. */
enum ofp_flow_wildcards {
    OFPFW_IN_PORT     = 1 << 0,  /* Switch input port. */
    OFPFW_DL_VLAN     = 1 << 1,  /* VLAN id. */
    OFPFW_DL_VLAN_PCP = 1 << 2,  /* VLAN priority. */
    OFPFW_DL_TYPE     = 1 << 3,  /* Ethernet frame type. */
    OFPFW_NW_TOS      = 1 << 4,  /* IP ToS (DSCP field, 6 bits). */
    OFPFW_NW_PROTO    = 1 << 5,  /* IP protocol. */
    OFPFW_TP_SRC      = 1 << 6,  /* TCP/UDP/SCTP source port. */
    OFPFW_TP_DST      = 1 << 7,  /* TCP/UDP/SCTP destination port. */
    OFPFW_MPLS_LABEL  = 1 << 8,  /* MPLS label. */
    OFPFW_MPLS_TC     = 1 << 9,  /* MPLS TC. */

    /* Wildcard all fields. */
    OFPFW_ALL           = ((1 << 10) - 1)
};
```

The metadata field is used to pass information between lookups across multiple tables. This value can be masked using the metadata mask (which is `0xFFFFFFFFFFFFFFFF` by default).

There are also four match field masks: `dl_src_mask`, `dl_dst_mask`, `nw_src_mask`, and `nw_dst_mask`. These mask fields allow the corresponding fields (`dl_src`, `dl_dst`, `nw_src`, and `nw_dst`) to be masked arbitrarily. The masks are defined such that a 1 in a given bit position indicates a "don't care" match for the same bit in the corresponding field, whereas a 0 means match the bit exactly.

If no wildcards are set and the `_mask` fields are all zero, then the `ofp_match` exactly describes a flow, over the entire OpenFlow n-tuple. On the other extreme, if all the wildcard flags and `_mask` fields are set, then every flow will match.

Setting the `OFPFW_DL_VLAN` bit in the `wildcards` field specifies that a flow 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 dl_vlan are:

```
/* The VLAN id is 12-bits, so we can use the entire 16 bits to indicate
 * special conditions.
 */
enum ofp_vlan_id {
    OFPVID_ANY  = 0xfffe, /* Indicate that a VLAN id is set but don't care
                             about it's value. Note: only valid when specifying
                             the VLAN id in a match */
    OFPVID_NONE = 0xffff, /* No VLAN id was set. */
};
```

The dl_vlan_pcp field must be ignored when the OFPFW_DL_VLAN wildcard bit is set or when the dl_vlan value is set to OFPVID_NONE.

Table 9 summarizes the combinations of wildcard bits and field values for particular matches.

Wildcard bit	dl_vlan value	Matching packets
OFPFW_DL_VLAN	*	Packets with and without a VLAN tag
-	OFPVID_NONE	Only packets *without* a VLAN tag
-	OFPVID_ANY	Only packets *with* a VLAN tag regardless of it's value

Table 9: Match combinations for VLAN tags.

A.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_EXPERIMENTER = 0xFFFF  /* Experimenter instruction */
};
```

The instruction set is described in section 4.6. Flow tables may support a subset of instruction types.

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];  /* 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.

A.2.5 Action Structures

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

```
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_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_NW_TOS,       /* IP ToS (DSCP field, 6 bits). */
    OFPAT_SET_NW_ECN,       /* IP ECN (2 bits). */
    OFPAT_SET_TP_SRC,       /* TCP/UDP/SCTP source port. */
    OFPAT_SET_TP_DST,       /* TCP/UDP/SCTP destination port. */
    OFPAT_COPY_TTL_OUT,     /* Copy TTL "outwards" -- from next-to-outermost to
                               outermost */
    OFPAT_COPY_TTL_IN,      /* Copy TTL "inwards" -- from outermost to
                               next-to-outermost */
    OFPAT_SET_MPLS_LABEL,   /* MPLS label */
    OFPAT_SET_MPLS_TC,      /* MPLS TC */
    OFPAT_SET_MPLS_TTL,     /* MPLS TTL */
    OFPAT_DEC_MPLS_TTL,     /* Decrement MPLS TTL */

    OFPAT_PUSH_VLAN,        /* Push a new VLAN tag */
    OFPAT_POP_VLAN,         /* Pop the outer VLAN tag */
    OFPAT_PUSH_MPLS,        /* Push a new MPLS tag */
    OFPAT_POP_MPLS,         /* Pop the outer MPLS tag */
    OFPAT_SET_QUEUE,        /* Set queue id when outputting to a port */
    OFPAT_GROUP,            /* Apply group. */
    OFPAT_SET_NW_TTL,       /* IP TTL. */
    OFPAT_DEC_NW_TTL,       /* Decrement IP TTL. */
    OFPAT_EXPERIMENTER = 0xffff
};
```

Output, group, and set-queue actions are described in Section 4.9, tag push/pop actions are described in Table 6, and Set-Field actions are described in Table 7. 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.*/
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 `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 a zero-size `packet_in` message. The `port` specifies the port through which the packet should be sent.

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 to an already-configured queue, 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 to a specific queue, therefore the Set-Queue action is not supported. The user can still use these queues and map flows 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 VLAN ID* action uses the following structure and fields:

```
/* Action structure for OFPAT_SET_VLAN_VID. */
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];
};
OFP_ASSERT(sizeof(struct ofp_action_vlan_vid) == 8);
```

The `vlan_vid` field is 16 bits long, when an actual VLAN id is only 12 bits.

A *Set VLAN priority* action uses the following structure and fields:

```
/* Action structure for OFPAT_SET_VLAN_PCP. */
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];
};
OFP_ASSERT(sizeof(struct ofp_action_vlan_pcp) == 8);
```

The `vlan_pcp` field is 8 bits long, but only the lower 3 bits have meaning.

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

```
/* Action structure for OFPAT_SET_MPLS_LABEL. */
struct ofp_action_mpls_label {
    uint16_t type;                  /* OFPAT_SET_MPLS_LABEL. */
    uint16_t len;                   /* Length is 8. */
    uint32_t mpls_label;            /* MPLS label */
};
OFP_ASSERT(sizeof(struct ofp_action_mpls_label) == 8);
```

The `mpls_label` field is 32 bits long, but only the lower 20 bits have meaning.

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

```
/* Action structure for OFPAT_SET_MPLS_TC. */
struct ofp_action_mpls_tc {
    uint16_t type;                  /* OFPAT_SET_MPLS_TC. */
    uint16_t len;                   /* Length is 8. */
    uint8_t mpls_tc;                /* MPLS TC */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_mpls_tc) == 8);
```

The `mpls_tc` field is 8 bits long, but only the lower 3 bits have meaning.

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 Ethernet source address* action and *Set Ethernet destination address* action use the following structure and fields:

```
/* Action structure for OFPAT_SET_DL_SRC/DST. */
struct ofp_action_dl_addr {
    uint16_t type;                  /* OFPAT_SET_DL_SRC/DST. */
    uint16_t len;                   /* Length is 16. */
    uint8_t dl_addr[OFP_ETH_ALEN];  /* Ethernet address. */
    uint8_t pad[6];
};
OFP_ASSERT(sizeof(struct ofp_action_dl_addr) == 16);
```

The `dl_addr` field is the MAC address to set.

A *Set IPv4 source address* action and *Set IPv4 destination address* action use the following structure and fields:

```
/* Action structure for OFPAT_SET_NW_SRC/DST. */
struct ofp_action_nw_addr {
    uint16_t type;                  /* OFPAT_SET_TW_SRC/DST. */
    uint16_t len;                   /* Length is 8. */
    uint32_t nw_addr;               /* IP address. */
};
OFP_ASSERT(sizeof(struct ofp_action_nw_addr) == 8);
```

The `nw_addr` field is the IP address to set.

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

```
/* Action structure for OFPAT_SET_NW_TOS. */
struct ofp_action_nw_tos {
    uint16_t type;                  /* OFPAT_SET_TW_SRC/DST. */
    uint16_t len;                   /* Length is 8. */
    uint8_t nw_tos;                 /* IP ToS (DSCP field, 6 bits). */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_nw_tos) == 8);
```

The `nw_tos` field is the 6 upper bits of the ToS field to set, in the original bit positions (shifted to the left by 2).

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

```
/* Action structure for OFPAT_SET_NW_ECN. */
struct ofp_action_nw_ecn {
    uint16_t type;                  /* OFPAT_SET_TW_SRC/DST. */
    uint16_t len;                   /* Length is 8. */
    uint8_t nw_ecn;                 /* IP ECN (2 bits). */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_nw_ecn) == 8);
```

The nw_ecn field is the 2 lower bits of the ECN field to set, in the original bit positions.

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 *Set transport source port* action and *Set transport destination port* action use the following structure and fields:

```
/* Action structure for OFPAT_SET_TP_SRC/DST. */
struct ofp_action_tp_port {
    uint16_t type;                  /* OFPAT_SET_TP_SRC/DST. */
    uint16_t len;                   /* Length is 8. */
    uint16_t tp_port;               /* TCP/UDP/SCTP port. */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_tp_port) == 8);
```

The tp_port field is the TCP/UDP/SCTP/other port to set.

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 and *Push MPLS header* action use the following structure and fields:

```
/* Action structure for OFPAT_PUSH_VLAN/MPLS. */
struct ofp_action_push {
    uint16_t type;                  /* OFPAT_PUSH_VLAN/MPLS. */
    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 or new MPLS header.

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 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 payload.

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.

A.3 Controller-to-Switch Messages

A.3.1 Handshake

Upon TLS session establishment, the controller sends an OFPT_FEATURES_REQUEST message. This message does not contain a body beyond the OpenFlow header. The switch responds 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 pad[3];         /* Align to 64-bits. */

    /* Features. */
    uint32_t capabilities;  /* Bitmap of support "ofp_capabilities". */
    uint32_t reserved;

    /* Port info.*/
    struct ofp_port ports[0];  /* Port definitions.  The number of ports
                                  is inferred from the length field in
```

```
                                    the header. */
};
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. Switches that support buffering can send only the number of bytes specified in the switch configuration or send to controller action (see A.3.2 and **??**).

The n_tables field describes the number of tables supported by the switch, each of which can have a different set of supported wildcard bits 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 OFPST_TABLE stats request. A switch must return these tables in the order the packets traverse the tables.

The capabilities field uses 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_ARP_MATCH_IP   = 1 << 7   /* Match IP addresses in ARP pkts. */
};
```

The actions field is a bitmap of actions supported by the switch. The list of actions is found in Section 4.9; all actions marked Required must be supported. Experimenter actions should *not* be reported via this bitmask. The bitmask uses the values from ofp_action_type as the number of bits to shift left for an associated action. For example, OFPAT_SET_DL_VLAN would use the flag 0x00000002.

The ports field is an array of ofp_port structures that describe all the ports in the system that support OpenFlow. The number of port elements is inferred from the length field in the OpenFlow header.

A.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;             /* OFPC_* flags. */
    uint16_t miss_send_len;     /* Max bytes of new flow that datapath should
                                   send to the controller. */
};
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,

    /* TTL processing - applicable for IP and MPLS packets */
    OFPC_INVALID_TTL_TO_CONTROLLER = 1 << 2, /* Send packets with invalid TTL
                                                ie. 0 or 1 to controller */
};
```

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 OFPC_INVALID_TTL_TO_CONTROLLER flag indicates whether packets with invalid IP TTL or MPLS TTL should be dropped or sent to the controller. The flag is cleared by default, causing invalid packets to get dropped.

The miss_send_len field defines the number of bytes of each packet sent to the controller as a result of both flow table misses and flow table hits with the controller as the destination. If this field equals 0, the switch must send a zero-size packet_in message.

A.3.3 Flow Table Configuration

The controller can configure and query table state in the switch with the OFP_TABLE_MOD and OFPST_TABLE_STATS requests, respectively. The switch responds to a table stats request with a OFPT_STATS_REPLY message.

```
/* Configure/Modify behavior of a flow table */
struct ofp_table_mod {
    struct ofp_header header;
    uint8_t table_id;       /* ID of the table, 0xFF 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 0xFF, the configuration is applied to all tables in the switch.

The config field is a bitmap that is used to configure the default behavior of unmatched packets. By default, any packet that does not match a table is sent to the controller for processing using a OFPT_PACKET_IN message. This behavior can be modified by using the following flags:

```
/* Flags to indicate behavior of the flow table for unmatched packets.
   These flags are used in ofp_table_stats messages to describe the current
   configuration and in ofp_table_mod messages to configure table behavior. */
enum ofp_table_config {
    OFPTC_TABLE_MISS_CONTROLLER = 0,      /* Send to controller. */
    OFPTC_TABLE_MISS_CONTINUE   = 1 << 0, /* Continue to the next table in the
                                             pipeline (OpenFlow 1.0
                                             behavior). */
    OFPTC_TABLE_MISS_DROP       = 1 << 1, /* Drop the packet. */
    OFPTC_TABLE_MISS_MASK       = 3
};
```

The OFPTC_TABLE_MISS_CONTINUE flag directs unmatched packets to the next table in the pipeline, except for the last table of the pipeline where unmatched packets are sent to the controller. This behavior is similar to the multiple table match process in the OpenFlow 1.0 specification. The OFPTC_TABLE_MISS_DROP flag drops unmatched packets.

A.3.4 Modify State Messages

Modify Flow Entry Message Modifications to the 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 */
    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 -1).
                                     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;               /* One of OFPFF_*. */
    uint8_t pad[2];
    struct ofp_match match;       /* Fields to match */
    struct ofp_instruction instructions[0]; /* Instruction set */
};
OFP_ASSERT(sizeof(struct ofp_flow_mod) == 136);
```

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 5.6). 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 is inserted in a table through an OFPC_ADD message, its cookie field is set to the provided value. When a flow is modified (OFPC_MODIFY or OFPC_MODIFY_STRICT messages), the cookie field is ignored.

If the cookie_mask field is non-zero, it is used with the cookie field to restrict flow matching while modifying or deleting flows. When used in a modify, a non-zero cookie_mask also prevents a new flow from getting inserted. This field is ignored by OFPC_ADD messages. The cookie_mask field's behavior is explained in Section 5.6.

The table_id field specifies the table into which the flow should be inserted. Table 0 signifies the first table in the pipeline. The behavior of table_id value 0xFF is undefined.

The command field must be one of the following:

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

The differences between OFPFC_MODIFY and OFPFC_MODIFY_STRICT are explained in Section 5.6 and differences between OFPFC_DELETE and OFPFC_DELETE_STRICT are explained in Section 5.6.

The idle_timeout and hard_timeout fields control how quickly flows expire. When a flow is inserted in a table, its idle_timeout and hard_timeout fields are set with the values from the message. When a flow is modified (OFPC_MODIFY or OFPC_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 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 OFPC_ADD, OFPC_MODIFY or OFPC_MODIFY_STRICT messages.

The buffer_id refers to a packet buffered at the switch and sent to the controller by a packet-in message. A flow mod that includes a valid buffer_id is effectively equivalent to sending a two-message sequence of a flow mod and a packet-out to OFPP_TABLE, 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 OFPC_DELETE and OFPC_DELETE_STRICT flow mod messages.

The out_port and out_group fields optionally filter the scope of OFPC_DELETE and OFPC_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 rule 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. This field is ignored by OFPC_ADD, OFPC_MODIFY or OFPC_MODIFY_STRICT messages.

The flags field may include the follow 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. */
};
```

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

When the OFPFF_CHECK_OVERLAP flag is set, the switch must check that there are no conflicting entries with the same priority. If there is one, the flow mod fails and an error code is returned.

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

The instructions field contains the instruction set for the flow entry when adding or modifying entries. The switch should return an error with OFPET_BAD_ACTION and OFPBAC_UNSUPPORTED_ORDER type and code if the instruction set contains an Apply-Actions instruction and the action list contains an action order that the switch cannot support.

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 bucket length 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 5.8.

The command field must be one of the following:

```
/* Group commands */
enum ofp_group_mod_command {
    OFPGC_ADD,              /* New group. */
    OFPGC_MODIFY,           /* Modify all matching groups. */
    OFPGC_DELETE,           /* 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,      /* All (multicast/broadcast) group.  */
    OFPGT_SELECT,   /* Select group. */
    OFPGT_INDIRECT, /* Indirect group. */
    OFPGT_FF        /* Fast failover group. */
};
```

Buckets 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]; /* The action length is inferred
                                           from the length field in the
                                           header. */
};
OFP_ASSERT(sizeof(struct ofp_bucket) == 16);
```

The weight field is only defined for select groups. 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.

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.

A.3.5 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) */
    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);
```

A.3.6 Read State Messages

While the system is running, the datapath may be queried about its current state using the `OFPT_STATS_REQUEST` message:

```
struct ofp_stats_request {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPST_* constants. */
    uint16_t flags;             /* OFPSF_REQ_* flags (none yet defined). */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the request. */
};
OFP_ASSERT(sizeof(struct ofp_stats_request) == 16);
```

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

```
struct ofp_stats_reply {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPST_* constants. */
    uint16_t flags;             /* OFPSF_REPLY_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the reply. */
};
OFP_ASSERT(sizeof(struct ofp_stats_reply) == 16);
```

The only value defined for `flags` in a reply is whether more replies will follow this one - this has the value `0x0001`. To ease implementation, the switch is allowed to send replies with no additional entries. However, it must always send another reply following a message with the *more* flag set. The transaction ids (xid) 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_stats_types {
    /* Description of this OpenFlow switch.
     * The request body is empty.
     * The reply body is struct ofp_desc_stats. */
    OFPST_DESC,

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

    /* Aggregate flow statistics.
     * The request body is struct ofp_aggregate_stats_request.
     * The reply body is struct ofp_aggregate_stats_reply. */
    OFPST_AGGREGATE,

    /* Flow table statistics.
```

```
     * The request body is empty.
     * The reply body is an array of struct ofp_table_stats. */
    OFPST_TABLE,

    /* Port statistics.
     * The request body is struct ofp_port_stats_request.
     * The reply body is an array of struct ofp_port_stats. */
    OFPST_PORT,

    /* Queue statistics for a port
     * The request body defines the port
     * The reply body is an array of struct ofp_queue_stats */
    OFPST_QUEUE,

    /* Group counter statistics.
     * The request body is empty.
     * The reply is struct ofp_group_stats. */
    OFPST_GROUP,

    /* Group description statistics.
     * The request body is empty.
     * The reply body is struct ofp_group_desc_stats. */
    OFPST_GROUP_DESC,

    /* Experimenter extension.
     * The request and reply bodies begin with a 32-bit experimenter ID,
     * which takes the same form as in "struct ofp_experimenter_header".
     * The request and reply bodies are otherwise experimenter-defined. */
    OFPST_EXPERIMENTER = 0xffff
};
```

In all types of statistics reply, if a specific numeric counter is not available in the switch, its value should be set to -1. Counters wrap around with no overflow indicator.

Description Statistics Information about the switch manufacturer, hardware revision, software revision, serial number, and a description field is available from the `OFPST_DESC` stats request type:

```
/* 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. */
    char dp_desc[DESC_STR_LEN];        /* Human readable description of datapath. */
};
OFP_ASSERT(sizeof(struct ofp_desc_stats) == 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 . Note: [1] 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 (§ A.3.1).

Individual Flow Statistics Information about individual flows is requested with the `OFPST_FLOW` stats request type:

[1] Added to address concerns raised in https://mailman.stanford.edu/pipermail/openflow-spec/2009-September/000504.html

```
/* Body for ofp_stats_request of type OFPST_FLOW. */
struct ofp_flow_stats_request {
    uint8_t table_id;        /* ID of table to read (from ofp_table_stats),
                                0xff for all tables. */
    uint8_t pad[3];          /* Align to 64 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. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 120);
```

The `match` field contains a description of the flows that should be matched and may contain wildcards. This field's matching behavior is described in Section 5.6.

The `table_id` field indicates the index of a single table to read, or `0xff` 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 rule 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 5.6.
The `body` of the reply consists of an array of the following:

```
/* Body of reply to OFPST_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. Only meaningful
                                when this is not an exact-match entry. */
    uint16_t idle_timeout;   /* Number of seconds idle before expiration. */
    uint16_t hard_timeout;   /* Number of seconds before expiration. */
    uint8_t pad2[6];         /* 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. */
    struct ofp_instruction instructions[0]; /* Instruction set. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 136);
```

The fields consist of those provided in the `flow_mod` that created these, plus the table into which the entry was inserted, the packet count, and the byte count.

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

Aggregate Flow Statistics Aggregate information about multiple flows is requested with the `OFPST_AGGREGATE` stats request type:

```
/* Body for ofp_stats_request of type OFPST_AGGREGATE. */
struct ofp_aggregate_stats_request {
    uint8_t table_id;         /* ID of table to read (from ofp_table_stats)
                                 0xff for all tables. */
    uint8_t pad[3];           /* Align to 64 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. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_request) == 120);
```

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

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

```
/* Body of reply to OFPST_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);
```

Table Statistics Information about tables is requested with the `OFPST_TABLE` stats 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 OFPST_TABLE request. */
struct ofp_table_stats {
    uint8_t table_id;        /* Identifier of table.  Lower numbered tables
                                are consulted first. */
    uint8_t pad[7];          /* Align to 64-bits. */
    char name[OFP_MAX_TABLE_NAME_LEN];
    uint32_t wildcards;      /* Bitmap of OFPFMF_* wildcards that are
                                supported by the table. */
    uint32_t match;          /* Bitmap of OFPFMF_* that indicate the fields
                                the table can match on. */
    uint32_t instructions;   /* Bitmap of OFPIT_* values supported. */
    uint32_t write_actions;  /* Bitmap of OFPAT_* that are supported
                                by the table with OFPIT_WRITE_ACTIONS. */
    uint32_t apply_actions;  /* Bitmap of OFPAT_* that are supported
```

```
                                by the table with OFPIT_APPLY_ACTIONS. */
    uint32_t config;         /* Bitmap of OFPTC_* values */
    uint32_t max_entries;    /* Max number of entries supported. */
    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) == 88);
```

The body 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 entries are returned in the order that packets traverse the tables.

The write_actions field is a bitmap of actions supported by the table using the OFPIT_WRITE_ACTIONS instruction, whereas the apply_actions field refers to the OFPIT_APPLY_ACTIONS instruction. The list of actions is found in Section 4.9. Experimenter actions should *not* be reported via this bitmask. The bitmask uses the values from ofp_action_type as the number of bits to shift left for an associated action. For example, OFPAT_SET_DL_VLAN would use the flag 0x00000002.

OFP_MAX_TABLE_NAME_LEN is 32 .

Port Statistics Information about ports is requested with the OFPST_PORT stats request type:

```
/* Body for ofp_stats_request of type OFPST_PORT. */
struct ofp_port_stats_request {
    uint32_t port_no;        /* OFPST_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 OFPST_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. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats) == 104);
```

Queue Statistics The OFPST_QUEUE stats 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.

```
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. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats) == 32);
```

Group Statistics The OFPST_GROUP stats 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 OFPST_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 OFPST_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. */
    struct ofp_bucket_counter bucket_stats[0];
};
OFP_ASSERT(sizeof(struct ofp_group_stats) == 32);
```

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);
```

Group Description Statistics The OFPST_GROUP_DESC stats 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 OFPST_GROUP_DESC request. */
struct ofp_group_desc_stats {
    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];
};
OFP_ASSERT(sizeof(struct ofp_group_desc_stats) == 8);
```

Fields for group description stats are the same as those used with the ofp_group_mod struct.

Experimenter Statistics Experimenter-specific stats messages are requested with the OFPST_EXPERIMENTER stats type. The first four bytes of the message are the experimenter identifier. The rest of the body is experimenter-defined.

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 experimenter does not have (or wish to use) their OUI, they should contact the OpenFlow consortium to obtain one.

A.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 (-1 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. */
    /* uint8_t data[0]; */        /* Packet data.  The length is inferred
                                     from the length field in the header.
                                     (Only meaningful if buffer_id == -1.) */
};
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 -1, then the packet data is included in the data array.

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 virtual port as an output action to process the packet through the existing flow entries, starting at the first flow table. If OFPP_TABLE is specified, the in_port field is used as the ingress port in the flow table lookup. The in_port field must be set to either a valid switch port or OFPP_CONTROLLER.

Packets sent to OFPP_TABLE may be forwarded back to the controller as the result of a flow 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 was committed to the flow table prior to execution of OFPP_TABLE.

A.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.

A.4 Asynchronous Messages

A.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. */
    uint32_t in_port;       /* Port on which frame was received. */
    uint32_t in_phy_port;   /* Physical Port on which frame was received. */
    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 */
    uint8_t data[0];        /* Ethernet frame, halfway through 32-bit word,
                               so the IP header is 32-bit aligned.  The
                               amount of data is inferred from the length
                               field in the header.  Because of padding,
                               offsetof(struct ofp_packet_in, data) ==
                               sizeof(struct ofp_packet_in) - 2. */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 24);
```

The in_phy_port is the physical port on which the packet was received. The in_port is the virtual port through which a packet was received, or physical port if the packet was not received on a virtual port. The port referenced by the in_port field must be the port used for matching flows (see 4.4) and must be available to OpenFlow processing (i.e. OpenFlow can forward packet to this port, depending on port flags).

For example, consider a packet received on a tunnel interface. This tunnel interface is defined over a link aggregation group (LAG) with two physical port members and the tunnel interface is the virtual port bound to OpenFlow. In this case, the in_port is the tunnel port_no and the in_phy_port is the physical port_no member of the LAG on which the tunnel is configured. If a packet is received directly on a physical port and not processed by a virtual port, in_port should have the same value as in_phy_port.

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 because of a flow table miss, 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, the entire packet is included in the data portion, and the buffer_id is -1.

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 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,          /* No matching flow. */
    OFPR_ACTION             /* Action explicitly output to controller. */
};
```

A.4.2 Flow Removed Message

If the controller has requested to be notified when flows time out, 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. */
    uint8_t pad2[2];          /* Align to 64-bits. */
    uint64_t packet_count;
    uint64_t byte_count;
    struct ofp_match match;   /* Description of fields. */
};
OFP_ASSERT(sizeof(struct ofp_flow_removed) == 136);
```

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

The reason field is one of the following:

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

The duration_sec and duration_nsec fields are described in Section A.3.6.

The idle_timeout field is 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 was active, as well as the amount of time the flow received traffic.

The packet_count and byte_count indicate the number of packets and bytes that were associated with this flow, respectively. The switch should return a value of -1 for unavailable counters.

A.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 status can be one of the following values:

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

A.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.

Error codes ending in _EPERM correspond to a permissions error generated by an entity between a controller and switch, such as an OpenFlow hypervisor.

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,         /* Hello protocol failed. */
    OFPET_BAD_REQUEST,          /* Request was not understood. */
    OFPET_BAD_ACTION,           /* Error in action description. */
    OFPET_BAD_INSTRUCTION,      /* Error in instruction list. */
    OFPET_BAD_MATCH,            /* Error in match. */
    OFPET_FLOW_MOD_FAILED,      /* Problem modifying flow entry. */
    OFPET_GROUP_MOD_FAILED,     /* Problem modifying group entry. */
    OFPET_PORT_MOD_FAILED,      /* Port mod request failed. */
    OFPET_TABLE_MOD_FAILED,     /* Table mod request failed. */
    OFPET_QUEUE_OP_FAILED,      /* Queue operation failed. */
    OFPET_SWITCH_CONFIG_FAILED, /* Switch config request failed. */
};
```

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,        /* No compatible version. */
    OFPHFC_EPERM                /* 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,         /* ofp_header.version not supported. */
    OFPBRC_BAD_TYPE,            /* ofp_header.type not supported. */
    OFPBRC_BAD_STAT,            /* ofp_stats_request.type not supported. */
    OFPBRC_BAD_EXPERIMENTER,    /* Experimenter id not supported
                                 * (in ofp_experimenter_header
                                 * or ofp_stats_request or ofp_stats_reply). */
    OFPBRC_BAD_SUBTYPE,         /* Experimenter subtype not supported. */
    OFPBRC_EPERM,               /* Permissions error. */
    OFPBRC_BAD_LEN,             /* Wrong request length for type. */
    OFPBRC_BUFFER_EMPTY,        /* Specified buffer has already been used. */
    OFPBRC_BUFFER_UNKNOWN,      /* Specified buffer does not exist. */
    OFPBRC_BAD_TABLE_ID         /* Specified table-id invalid or does not
                                 * exist. */
};
```

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,           /* Unknown action type. */
    OFPBAC_BAD_LEN,            /* Length problem in actions. */
    OFPBAC_BAD_EXPERIMENTER,   /* Unknown experimenter id specified. */
    OFPBAC_BAD_EXPERIMENTER_TYPE, /* Unknown action type for experimenter id. */
    OFPBAC_BAD_OUT_PORT,       /* Problem validating output port. */
    OFPBAC_BAD_ARGUMENT,       /* Bad action argument. */
    OFPBAC_EPERM,              /* Permissions error. */
    OFPBAC_TOO_MANY,           /* Can't handle this many actions. */
    OFPBAC_BAD_QUEUE,          /* Problem validating output queue. */
    OFPBAC_BAD_OUT_GROUP,      /* Invalid group id in forward action. */
    OFPBAC_MATCH_INCONSISTENT, /* Action can't apply for this match. */
    OFPBAC_UNSUPPORTED_ORDER,  /* Action order is unsupported for the action
  list in an Apply-Actions instruction */
    OFPBAC_BAD_TAG,            /* Actions uses an unsupported
                                  tag/encap. */
};
```

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,       /* Unknown instruction. */
    OFPBIC_UNSUP_INST,         /* Switch or table does not support the
                                  instruction. */
    OFPBIC_BAD_TABLE_ID,       /* Invalid Table-ID specified. */
    OFPBIC_UNSUP_METADATA,     /* Metadata value unsupported by datapath. */
    OFPBIC_UNSUP_METADATA_MASK,/* Metadata mask value unsupported by
                                  datapath. */
    OFPBIC_UNSUP_EXP_INST,     /* Specific experimenter instruction
                                  unsupported. */
};
```

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,            /* Unsupported match type specified by the
                                   match */
    OFPBMC_BAD_LEN,             /* Length problem in match. */
    OFPBMC_BAD_TAG,             /* Match uses an unsupported tag/encap. */
    OFPBMC_BAD_DL_ADDR_MASK,    /* Unsupported datalink addr mask - switch does
                                   not support arbitrary datalink address
                                   mask. */
    OFPBMC_BAD_NW_ADDR_MASK,    /* Unsupported network addr mask - switch does
                                   not support arbitrary network address
                                   mask. */
    OFPBMC_BAD_WILDCARDS,       /* Unsupported wildcard specified in the
                                   match. */
    OFPBMC_BAD_FIELD,/* Unsupported field in the match. */
    OFPBMC_BAD_VALUE,/* Unsupported value in a match field. */
};
```

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,            /* Unspecified error. */
    OFPFMFC_TABLE_FULL,         /* Flow not added because table was full. */
    OFPFMFC_BAD_TABLE_ID,       /* Table does not exist */
    OFPFMFC_OVERLAP,            /* Attempted to add overlapping flow with
                                   CHECK_OVERLAP flag set. */
    OFPFMFC_EPERM,              /* Permissions error. */
    OFPFMFC_BAD_TIMEOUT,        /* Flow not added because of unsupported
                                   idle/hard timeout. */
    OFPFMFC_BAD_COMMAND,        /* Unsupported or unknown command. */
};
```

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,             /* Group not added because a group ADD
                                       * attempted to replace an
                                       * already-present group. */
    OFPGMFC_INVALID_GROUP,            /* Group not added because Group specified
                                       * is invalid. */
    OFPGMFC_WEIGHT_UNSUPPORTED,       /* Switch does not support unequal load
                                       * sharing with select groups. */
    OFPGMFC_OUT_OF_GROUPS,            /* The group table is full. */
    OFPGMFC_OUT_OF_BUCKETS,           /* The maximum number of action buckets
                                       * for a group has been exceeded. */
    OFPGMFC_CHAINING_UNSUPPORTED,     /* Switch does not support groups that
                                       * forward to groups. */
    OFPGMFC_WATCH_UNSUPPORTED,        /* This group cannot watch the
                                         watch_port or watch_group specified. */
    OFPGMFC_LOOP,                     /* Group entry would cause a loop. */
    OFPGMFC_UNKNOWN_GROUP,            /* Group not modified because a group
                                         MODIFY attempted to modify a
                                         non-existent group. */
};
```

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,           /* Specified port number does not exist. */
    OFPPMFC_BAD_HW_ADDR,        /* Specified hardware address does not
                                 * match the port number. */
    OFPPMFC_BAD_CONFIG,         /* Specified config is invalid. */
    OFPPMFC_BAD_ADVERTISE       /* Specified advertise is invalid. */
};
```

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,          /* Specified table does not exist. */
    OFPTMFC_BAD_CONFIG          /* Specified config is invalid. */
};
```

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,          /* Invalid port (or port does not exist). */
    OFPQOFC_BAD_QUEUE,         /* Queue does not exist. */
    OFPQOFC_EPERM              /* 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,          /* Specified flags is invalid. */
    OFPSCFC_BAD_LEN             /* Specified len is invalid. */
};
```

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 should match that of the offending message.

A.5 Symmetric Messages

A.5.1 Hello

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.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.

A.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.

A.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 OpenFlow
                                 *   consortium. */
    uint8_t pad[4];
    /* 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 experimenter 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.

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 B Credits

Spec contributions, in alphabetical order:

Ben Pfaff, Bob Lantz, Brandon Heller, Casey Barker, Dan Cohn, Dan Talayco, David Erickson, Edward Crabbe, Glen Gibb, Guido Appenzeller, Jean Tourrilhes, Justin Pettit, KK Yap, Leon Poutievski, Martin Casado, Masahiko Takahashi, Masayoshi Kobayashi, Nick McKeown, Peter Balland, Rajiv Ramanathan, Reid Price, Rob Sherwood, Saurav Das, Tatsuya Yabe, Yiannis Yiakoumis, Zoltán Lajos Kis.