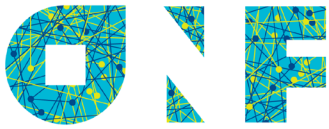


OPEN NETWORKING
FOUNDATION

OpenFlow Switch Specification

Version 1.2 (Wire Protocol 0x03)
December 5, 2011

ONF Document Type: OpenFlow Spec
ONF Document Name: openflow-spec-v1.2

Disclaimer

OpenFlow 1.2

December 2011

The Open Networking Foundation hereby issues OpenFlow 1.2, following this page. OpenFlow 1.2 encompasses the switch specification, an evolution from OpenFlow 1.1, 1.0, and previous versions. The family of OpenFlow standards will include, at the appropriate time, other components, including a Configuration and Management Protocol (OF-Config), Testing and Interoperability specifications for conformance, performance, and interoperability (OF-Test), and others as the need arises.

OpenFlow 1.2, the Switch Specification, describes the formats and protocols by which an OpenFlow Switch receives, reacts to, and responds to messages from an OpenFlow Controller. The objective of this dialogue is to instruct the forwarding plane of the OpenFlow Switch to treat incoming packets in a particular way.

The OpenFlow 1.2 Switch Specification builds significantly upon previous releases in many ways, including these significant improvements:

- It adds support for IPv6. In addition to the previous support for IPv4, MPLS, and L2 headers, OpenFlow 1.2 now supports matching on IPv6 source address, destination address, protocol number, traffic class, ICMPv6 type, ICMPv6 code, IPv6 neighbor discovery header fields, and IPv6 flow labels.
- It adds support for extensible matches. By employing a TLV structure, the protocol allows far greater flexibility for the treatment of current and future protocols.
- It allows experimenter extensions through dedicated fields and code points assigned by ONF.

Appendix B of the OpenFlow Switch Specification contains release notes for all versions of OpenFlow starting with OpenFlow 0.2.0. The release notes for OpenFlow 1.2 may be found in section B.10.

OpenFlow Switch Specification

Version 1.2 (Wire Protocol 0x03)

December 5, 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 Open Networking Foundation website (`https://www.opennetworking.org/standards/open-flow`). This specification covers the components and the basic functions of the switch, and the OpenFlow protocol to manage an OpenFlow switch from a remote controller.

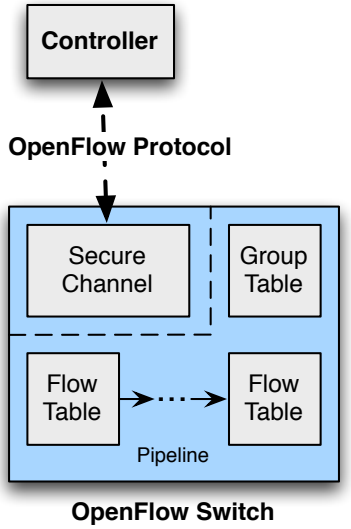


Figure 1: Main components of an OpenFlow switch.

2 Switch Components

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

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

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

Instructions associated with each flow entry either contain actions or modify pipeline processing (see 5.6). Actions included in instructions describe packet forwarding, packet modification and group table processing. Pipeline processing instructions allow packets to be sent to subsequent tables for further

processing and allow information, in the form of metadata, to be communicated between tables. Table pipeline processing stops when the instruction set associated with a matching flow entry does not specify a next table; at this point the packet is usually modified and forwarded (see 5.7).

Flow entries may forward to a *port*. This is usually a physical port, but it may also be a logical port defined by the switch or a reserved port defined by this specification (see 4.1). Reserved ports may specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as "normal" switch processing (see 5.9), while switch-defined logical ports may specify link aggregation groups, tunnels or loopback interfaces (see 5.9).

Actions associated with flow entries may also direct packets to a group, which specifies additional processing (see 5.4). 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 5.4.1). 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.
- **Port**: where packets enter and exit the OpenFlow pipeline (see 4.1). May be a physical port, a logical port defined by the switch, or a reserved port defined by the OpenFlow protocol.
- **Pipeline**: the set of linked flow tables that provide matching, forwarding, and packet modifications in an OpenFlow switch.
- **Flow Table**: A stage of the pipeline, contains flow entries.
- **Flow Entry**: an element in a Flow Table use to match and process packets. It contains a set of match fields for matching packets, a set of counters to track packets, and a set of instructions to apply.
- **Match Field**: a field against which a packet is matched, including packet headers, the ingress port, and the metadata value. A match field may be wildcarded (match any value) and in some case bitmasked.
- **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.
- **Controller**: An entity interacting with the OpenFlow switch using the OpenFlow protocol.

4 OpenFlow Ports

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

4.1 OpenFlow ports

The OpenFlow port are the network interface for passing packet between OpenFlow processing and the rest of the network. OpenFlow switches connect logically to each other via their OpenFlow ports.

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

OpenFlow packets are received on an **ingress port**, processed by the OpenFlow pipeline (see 5.1) which may forward them to an **output port**. The packet ingress port is a property of the packet throughout the OpenFlow pipeline and represent the OpenFlow port on which the packet was received into the OpenFlow switch, it can be used when matching packet (see 5.3). The OpenFlow pipeline can decide to send the packet on an output port using the output action (see 5.9), which defines how the packets goes back to the network.

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

4.2 Standard ports

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

Standard ports can be used as ingress and output ports, can also be used in groups (see 5.4), and have port counters (see 5.5).

4.3 Physical ports

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

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

4.4 Logical ports

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

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

The only difference between *physical ports* and *logical ports* is that when a packet received on a logical port is sent to the controller, both its logical port and its underlying physical port are reported to the controller (see A.4.1).

4.5 Reserved ports

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

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

- *Required:* **ALL:** Represent all ports the switch can use for forwarding a specific packet. Can be used only as an output port, send a copy of the packet to all standard ports, excluding the packet ingress port and ports that are configured `OFPPC_NO_FWD`.
- *Required:* **CONTROLLER:** Represent the control channel with the OpenFlow controller. Can be used as an ingress port or as an output port. When used as an output port, encapsulate and send the packet to the controller. When used as an ingress port, identify a packet originating from the controller.
- *Required:* **TABLE:** Represent the start of the OpenFlow pipeline. This port is only valid in an output action in the action list of a *packet-out* message, submit the packet to the first flow table so that the packet can be processed through the regular OpenFlow pipeline.
- *Required:* **IN_PORT:** Represent the packet ingress port. Can be used only as an output port, send the packet out its ingress port.
- *Required:* **ANY:** Special value used in some OpenFlow commands when no port is specified (port wildcarded). Can not be used as an ingress port nor as an output port.
- *Optional:* **LOCAL:** Represent the switch's local networking stack. Can be used as an ingress port or as an output port. 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.

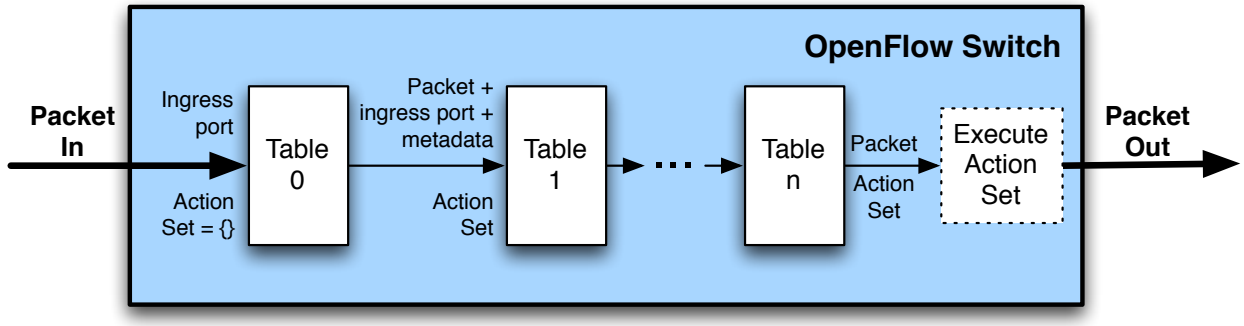
- *Optional:* **NORMAL:** Represent the traditional non-OpenFlow pipeline of the switch (see 5.1). Can be used only as an output port, process the packet using the normal pipeline. If the switch cannot forward packets from the OpenFlow pipeline to the normal pipeline, it must indicate that it does not support this action.
- *Optional:* **FLOOD:** Represent flooding using the normal pipeline of the switch (see 5.1). Can be used only as an output port, 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 the **NORMAL** port and **FLOOD** port, while *OpenFlow-hybrid* switches may support them (see 5.1). Forwarding packets to the **FLOOD** port depends on the switch implementation and configuration, while forwarding using a **group** of type *all* enables the controller to more flexibly implement flooding (see 5.4.1).

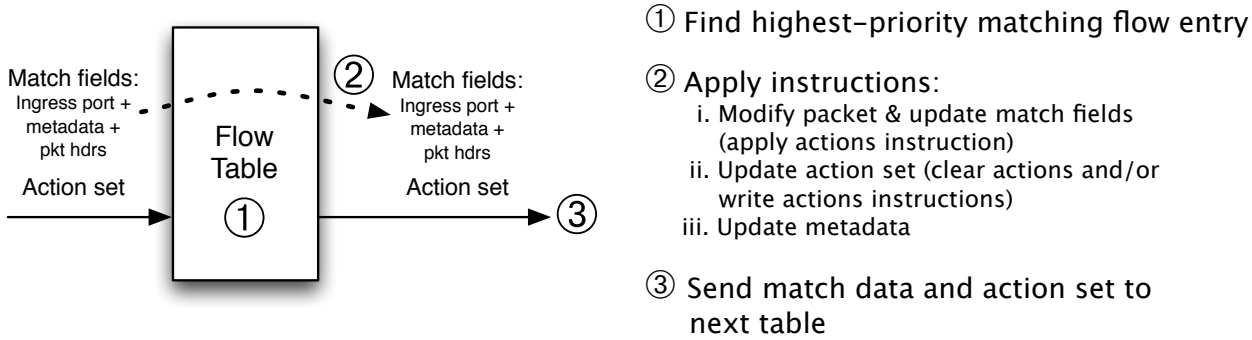
5 OpenFlow Tables

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

5.1 Pipeline Processing



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



(b) Per-table packet processing

Figure 2: Packet flow through the processing pipeline

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

OpenFlow-hybrid switches support both OpenFlow operation and *normal* Ethernet switching operation, i.e. traditional L2 Ethernet switching, VLAN isolation, L3 routing (IPv4 routing, IPv6 routing...), ACL and QoS processing. Those switches 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* reserved ports (see 4.5).

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

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

When processed by a flow table, the packet is matched against the flow entries of the flow table to select a flow entry (see 5.3). If a flow entry is found, the instruction set included in that flow entry is executed, those instructions may explicitly direct the packet to another flow table (using the Goto Instruction, see 5.6), 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 5.7).

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 (see A.3.3). The default is to send packets to the controller over the control channel via a packet-in message (see 6.1.2), 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.

5.2 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) is identified by its match fields and 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

5.3 Matching

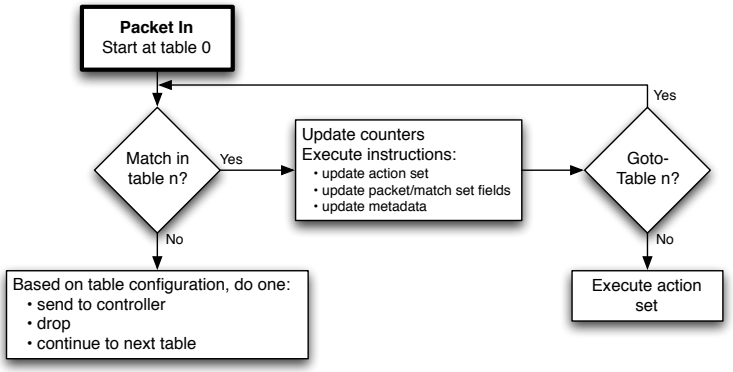


Figure 3: Flowchart detailing packet flow through an OpenFlow switch.

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

Packet match fields are extracted from the packet. Packet match fields used for table lookups depend on the packet type, and typically include various packet header fields, such as Ethernet source address or IPv4 destination address (see A.2.3). In addition to packet headers, matches can also be performed against the ingress port and metadata fields. Metadata may be used to pass information between tables in a switch. The packet match fields represent the packet in its current state, if actions applied in a previous table using the *Apply-Actions* changed the packet headers, those changes are reflected in the packet match fields.

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

The packet is matched against the table and *only* the highest priority flow entry that matches the packet must be selected. The counters associated with the selected flow entry must be updated and the instruction set included in the selected flow entry must be applied. If there are multiple matching flow entries with the same highest priority, the selected flow entry is explicitly undefined. This case can only arise when a controller writer never sets the `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.

5.4 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).

Group Identifier	Group Type	Counters	Action Buckets

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

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

- **group identifier**: a 32 bit unsigned integer uniquely identifying the group
- **group type**: to determine group semantics (see Section 5.4.1)
- **counters**: updated when packets are processed by a group
- **action buckets**: an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters

5.4.1 Group Types

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

- *Required:* **all**: Execute all buckets in the group. This group is used for multicast or broadcast forwarding. The packet is effectively cloned for each bucket; one packet is processed for each bucket of the group. If a bucket directs a packet explicitly out the ingress port, this packet clone is dropped. If the controller writer wants to forward out the ingress port, the group should include an extra bucket which includes an output action to the `OFPP_IN_PORT` reserved port.
- *Optional:* **select**: Execute one bucket in the group. Packets are processed by a single bucket in the group, based on a switch-computed selection algorithm (e.g. hash on some user-configured tuple or simple round robin). All configuration and state for the selection algorithm is external to OpenFlow. The selection algorithm should implement equal load sharing and can optionally be based on bucket weights. When a port specified in a bucket in a select group goes down, the switch may restrict bucket selection to the remaining set (those with forwarding actions to live ports) instead of dropping packets destined to that port. This behavior may reduce the disruption of a downed link or switch.
- *Required:* **indirect**: Execute the one defined bucket in this group. This group supports only a single bucket. Allows multiple 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.
- *Optional:* **fast failover**: Execute the first live bucket. Each action bucket is associated with a specific port and/or group that controls its liveness. The buckets are evaluated in the order defined by the group, and the first bucket which is associated with a live port/group is selected. This group type enables the switch to change forwarding without requiring a round trip to the controller. If no buckets are live, packets are dropped. This group type must implement a *liveness mechanism*(see 6.9).

5.5 Counters

Counters are 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 3 contains the set of counters defined by the OpenFlow specification. A switch is not required to

support all counters, just those marked *"Required"* in Table 3.

Duration refers to the amount of time the flow has been installed in the switch, and must be tracked with second precision. The Receive Errors field is the total of all receive and collision errors defined in Table 3, as well as any others not called out in the table.

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

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

Table 3: List of counters

5.6 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.

A switch is not required to support all action types, just those marked *"Required Instruction"* below. The

controller can also query the switch about which of the *"Optional Instruction"* it supports.

- *Optional Instruction:* **Apply-Actions *action(s)***: Applies the specific action(s) immediately, without any change to the Action Set. This instruction may be used to modify the packet between two tables or to execute multiple actions of the same type. The actions are specified as an action list (see 5.8).
- *Optional Instruction:* **Clear-Actions**: Clears all the actions in the action set immediately.
- *Required Instruction:* **Write-Actions *action(s)***: Merges the specified action(s) into the current action set (see 5.7). If an action of the given type exists in the current set, overwrite it, otherwise add it.
- *Optional Instruction:* **Write-Metadata *metadata / mask***: Writes the masked metadata value into the metadata field. The mask specifies which bits of the metadata register should be modified (i.e. new_metadata = old_metadata & ˜mask | value & mask).
- *Required Instruction:* **Goto-Table *next-table-id***: Indicates the next table in the processing pipeline. The table-id must be greater than the current table-id. The flows of last table of the pipeline can not include this instruction (see 5.1).

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 (see 6.7). Flow tables may not support every match, every instruction and every actions.

5.7 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 the instruction set of a flow entry does not contain a *Goto-Table* instruction, pipeline processing stops and the actions in the action set of the packet are executed.

An action set contains a maximum of one action of each type. The *set-field* actions are identified by their field types, therefore the action set contains a maximum of one *set-field* action for each field type (i.e. multiple fields can be set). When multiple actions of the same type are required, e.g. pushing multiple MPLS labels or popping multiple MPLS labels, the *Apply-Actions* instruction may be used (see 5.8).

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 if the switch supports it; a group bucket may specify another group, in which case the execution of actions traverses all the groups specified by the group configuration.

5.8 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 an action list 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 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 5.1). The action set of the packet is unchanged by the execution of the action list.

5.9 Actions

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

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

Optional Action: **Set-Queue**. The set-queue action sets the queue id for a packet. When the packet is forwarded to a port using the output action, the queue id determines which queue attached to this port is used for 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 4. 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 5.9.1 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 4: Push/pop tag actions.

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

Optional Action: **Change-TTL**. The various Change-TTL actions modify the values of the IPv4 TTL, IPv6 Hop Limit or MPLS TTL in the packet. While not strictly required, the actions shown in Table 5 greatly increase the usefulness of an OpenFlow implementation for implementing routing functions. Change-TTL actions should *always* be applied to the outermost-possible header.

Action	**Associated Data**	**Description**
Set MPLS TTL	8 bits: New MPLS TTL	Replace the existing MPLS TTL. Only applies to packets with an existing MPLS shim header.
Decrement MPLS TTL	-	Decrement the MPLS TTL. Only applies to packets with an existing MPLS shim header.
Set IP TTL	8 bits: New IP TTL	Replace the existing IPv4 TTL or IPv6 Hop Limit and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Table 5 – Continued on next page		

Table 5 – concluded from previous page

Action	Associated Data	Description
Decrement IP TTL	-	Decrement the IPv4 TTL or IPv6 Hop Limit field and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Copy TTL outwards	-	Copy the TTL from next-to-outermost to outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or IP-to-MPLS.
Copy TTL inwards	-	Copy the TTL from outermost to next-to-outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or MPLS-to-IP.

Table 5: Change-TTL actions.

5.9.1 Default values for fields on push

Field values for all fields specified in Table 6 should be copied from existing outer headers to new outer headers when executing a push action. New fields listed in Table 6 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 6: Existing fields that may be copied into new fields on a push action.

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

6 OpenFlow Channel

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

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

6.1 OpenFlow Protocol Overview

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

6.1.1 Controller-to-Switch

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

Features: The controller may request the 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 flow/group entries in the OpenFlow tables and to set switch port properties.

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

6.1.2 Asynchronous

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

Packet-in: Transfer the control of a packet to the controller. For all packets that do not have a matching flow entry, a packet-in event may be sent to the controller, depending on the table configuration (see 5.1). For all packets forwarded to the **CONTROLLER** reserved port, a packet-in event is always sent to controllers (see 5.9).

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

If the packet is buffered, the number of bytes of the original packet to include in the packet-in can be configured. By default, it is 128 bytes, for table miss it can be configured in the switch configuration (see A.3.2), for packet forwarded to the controller it can be configured in the output action (see A.2.5).

Flow-Removed: Inform the controller about the removal of a flow entry from a flow table. Flow-Removed messages are only sent for flow with the `OFPFF_SEND_FLOW_REM` flag set. They are generated as the result of a controller flow delete requests or the switch flow expiry process when one of the flow timeout is exceeded (see 6.8).

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

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

6.1.3 Symmetric

Symmetric messages are sent without solicitation, in either direction.

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

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

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

6.2 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.

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.

6.3 Multiple Controllers

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

When OpenFlow operation is initiated, the switch must connect to all controllers it is configured with, and try to maintain connection with all of them concurrently. Many controllers may send controller-to-switch commands to the switch, the reply or error messages related to those command must only be sent

on the controller connection associated with that command. Asynchronous messages may need to be send to multiple controllers, the message is duplicated for each eligible controller connection and each message sent when the respective controller connection allows it.

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

A controller can request its role to be changed to OFPCR_ROLE_SLAVE. In this role, the controller has read-only access to the switch. The controller does not receive switch asynchronous messages, apart from Port-status messages. The controller is denied ability to send controller-to-switch commands that modify the state of the switch, OFPT_PACKET_OUT, OFPT_FLOW_MOD, OFPT_GROUP_MOD, OFPT_PORT_MOD and OFPT_TABLE_MOD. If the controller sends one of those commands, the switch must reply with an OFPT_ERROR message with a type field of OFPET_BAD_REQUEST, a code field of OFPBRC_IS_SLAVE. Other controller-to-switch messages, such as OFPT_STATS_REQUEST and OFPT_ROLE_REQUEST, should be processed normally.

A controller can request its role to be changed to OFPCR_ROLE_MASTER. This role is similar to OFPCR_ROLE_EQUAL and has full access to the switch, the only difference is that the switch make sure there is only a single controller in this role. When a controller change its role to OFPCR_ROLE_MASTER, the switch change all other controllers which role is OFPCR_ROLE_MASTER to have the role OFPCR_ROLE_SLAVE. When the switch perform such role change, no message is generated to the controller which role is changed (in most case that controller is no longer reachable).

A switch may be simultaneously connected to multiple controllers in Equal state, multiple controllers in Slave state, and at most one controller in Master state. Each controller may communicate its role to the switch via a OFPT_ROLE_REQUEST message, and the switch must remember the role of each controller connection. A controller may change role at any time.

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

On receiving a OFPT_ROLE_REQUEST with role equal to OFPCR_ROLE_MASTER or OFPCR_ROLE_SLAVE the switch must compare the generation_id in the message against the largest generation id seen so far. A message with a generation_id smaller than a previously seen generation id must be considered stale and discarded. The switch must respond to stale messages with an error message with type OFPET_ROLE_REQUEST_FAILED and code OFPRRFC_STALE.

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

On switch startup:
```
generation_is_defined = false;
```

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

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

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

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

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

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

6.4 Connection Interruption

In the case that a switch loses contact with all controllers, as a result of echo request timeouts, TLS session timeouts, or other disconnections, the switch should *immediately* enter either "fail secure mode" or "fail standalone mode", depending upon the switch implementation and configuration. In "fail secure mode", the only change to switch behavior is that packets and messages destined to the controllers are dropped. 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, until is successfully connects to a controller. Configuration of the default set of flow entries to be used at startup is outside the scope of the OpenFlow protocol.

6.5 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).

6.6 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. The policing of packet destined to the controller using QoS actions or rate limiting is advised, to prevent denial of service of the controller connection, and is outside the scope of the present specification.

Controllers are free to 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.

6.7 Flow Table Modification Messages

Flow table modification messages can have the following types:

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

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

For non-overlapping **add** requests, or those with no overlap checking, the switch must insert the flow entry in the requested table. If a flow entry with identical match fields and priority already resides in the requested table, then that entry, including its duration, must be cleared from the table, and the new flow entry added. If the `OFPFF_RESET_COUNTS` flag is set, the flow entry counters must be cleared, otherwise they should be copied from the replaced flow. 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. If the `OFPFF_RESET_COUNTS` flag is set, the flow entry counters must be cleared. For **modify** requests, if no flow currently residing in the requested table matches the request, no error is recorded, and no flow table modification occurs.

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

Modify and **delete** flow_mod commands have *non-strict* versions (`OFPFC_MODIFY` and `OFPFC_DELETE`) and *strict* versions (`OFPFC_MODIFY_STRICT` or `OFPFC_DELETE_STRICT`). In the *strict* versions, the set of match fields, all match fields, including their masks, and the priority, are strictly matched against the entry, and only an identical flow is modified or removed. For example, if a message to remove entries is sent that has no match fields included, the `OFPFC_DELETE` command would delete all flows from the 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, all flows that match the flow_mod description are modified or removed. In the *non-strict* versions, a match will occur when a flow entry exactly matches or is more specific than the description in the flow_mod command; in the flow_mod the missing match fields are wildcarded, field masks are active, and other flow_mod fields such as priority are ignored. For example, if a `OFPFC_DELETE` command says to delete all flows with a destination port of 80, then a flow entry that wildcards all match fields will not be deleted. However, a `OFPFC_DELETE` command that wildcards all match fields will delete an entry that matches all port 80 traffic. This same interpretation of mixed wildcard and exact match fields also applies to individual and aggregate flows stats requests.

Delete commands can be optionally filtered by destination group or output port. If the `out_port` field contains a value other than `OFPP_ANY`, it introduces a constraint when matching. This constraint is that each matching 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).

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

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

If a switch cannot find any space in the requested table in which to add the incoming flow entry, the switch 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 match in a flow mod message specifies a field that is unsupported in the table, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_FIELD code. If the match in a flow mod message specifies a field more than once, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_DUP_FIELD code. If the match in a flow mod message specifies a field but fail to specify its associated prerequisites, for example specifies an IPv4 address without matching the EtherType to 0x800, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_PREREQ code.

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

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

If any action references a port that will never be valid on a switch, the switch must return an ofp_error_msg with OFPET_BAD_ACTION type and OFPBAC_BAD_OUT_PORT code. If the referenced port may be valid in the future, e.g. when a linecard is added to a chassis switch, or a port is dynamically added to a software switch, the switch 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 an action list contain a sequence of actions that the switch can not support in the specified order, the switch should return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_UNSUPPORTED_ORDER` code.

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

6.8 Flow Removal

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

The switch **flow expiry** mechanism that is run by the switch independantly of the controller and is based on the state and configuration of flow entries. Each flow entry has an `idle_timeout` and a `hard_timeout` associated with it. If 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. The switch must implement flow expiry and remove flow entries from the flow table when one of their timeout is exceeded.

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

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

6.9 Group Table Modification Messages

Group table modification messages can have the following types:

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

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

the switch supports it.

The action set for each bucket must be validated using the same rules as those for flow mods (Section 6.7), with additional group-specific checks. If an action in one of the buckets is invalid or unsupported, the switch should return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and code corresponding to the error (see 6.7).

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 containing this group in a Group action are also removed. The group type need not be specified for the **delete** request. **Delete** also differs from an **add** or **modify** with no buckets specified in that future attempts to **add** the group identifier will not result in a group exists error. If one wishes to effectively delete a group yet leave in flow entries using it, that group can be cleared by sending a **modify** with no buckets specified.

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

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

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

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

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

- A port is considered live if it has the `OFPPS_LIVE` flag set in its port state. Port liveness may be managed by code outside of the OpenFlow portion of a switch, defined outside of the OpenFlow 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 0x03 . The `length` field indicates the total length of the message, so no additional framing is used to distinguish one frame from the next. The `type` can have the following values:

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

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

    /* Asynchronous messages. */
    OFPT_PACKET_IN          = 10, /* Async message */
    OFPT_FLOW_REMOVED       = 11, /* Async message */
    OFPT_PORT_STATUS        = 12, /* Async message */

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

    /* Statistics messages. */
    OFPT_STATS_REQUEST      = 18, /* Controller/switch message */
    OFPT_STATS_REPLY        = 19, /* Controller/switch message */

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

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

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

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 logical ports, and the OpenFlow specification defines some reserved ports (see 4.1).

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

```
/* Description of a port */
struct ofp_port {
```

```
    uint32_t port_no;
    uint8_t pad[4];
    uint8_t hw_addr[OFP_ETH_ALEN];
    uint8_t pad2[2];                  /* Align to 64 bits. */
    char name[OFP_MAX_PORT_NAME_LEN]; /* Null-terminated */

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

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

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

The `port_no` field uniquely identifies a port within a switch. The `hw_addr` field typically is the MAC address for the port; `OFP_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 and logical switch ports. */
    OFPP_MAX        = 0xffffff00,

    /* Reserved OpenFlow Port (fake output "ports"). */
    OFPP_IN_PORT    = 0xfffffff8,  /* Send the packet out the input port.  This
                                      reserved port must be explicitly used
                                      in order to send back out of the input
                                      port. */
    OFPP_TABLE      = 0xfffffff9,  /* Submit the packet to the first flow table
                                      NB: This destination port can only be
                                      used in packet-out messages. */
    OFPP_NORMAL     = 0xfffffffa,  /* Process with normal L2/L3 switching. */
    OFPP_FLOOD      = 0xfffffffb,  /* All physical ports in VLAN, except input
                                      port and those blocked or link down. */
    OFPP_ALL        = 0xfffffffc,  /* All physical ports except input port. */
    OFPP_CONTROLLER = 0xfffffffd,  /* Send to controller. */
    OFPP_LOCAL      = 0xfffffffe,  /* Local openflow "port". */
    OFPP_ANY        = 0xffffffff   /* Wildcard port used only for flow mod
                                      (delete) and flow stats requests. Selects
                                      all flows regardless of output port
                                      (including flows with no output port). */
};
```

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

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

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

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

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

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

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

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

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

Each queue property description starts with a common header:

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

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

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

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

```
/* Max-Rate queue property description. */
struct ofp_queue_prop_max_rate {
    struct ofp_queue_prop_header prop_header; /* prop: OFPQT_MAX, len: 16. */
    uint16_t rate;        /* In 1/10 of a percent; >1000 -> disabled. */
```

```
    uint8_t pad[6];       /* 64-bit alignment */
};
OFP_ASSERT(sizeof(struct ofp_queue_prop_max_rate) == 16);
```

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

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

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

A.2.3 Flow Match Structures

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

A.2.3.1 Flow Match Header

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

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

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

```
/* The match type indicates the match structure (set of fields that compose the
 * match) in use. The match type is placed in the type field at the beginning
 * of all match structures. The "OpenFlow Extensible Match" type corresponds
 * to OXM TLV format described below and must be supported by all OpenFlow
 * switches. Extensions that define other match types may be published on the
 * ONF wiki. Support for extensions is optional.
 */
enum ofp_match_type {
    OFPMT_STANDARD = 0,       /* Deprecated. */
    OFPMT_OXM      = 1,       /* OpenFlow Extensible Match */
};
```

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

A.2.3.2 Flow Match Field Structures

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

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

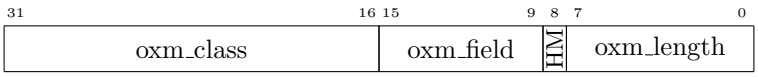


Figure 4: OXM TLV header layout

The OXM TLV's header fields are defined in Table 7

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

Table 7: OXM TLV header fields

The `oxm_class` is a OXM match class that contains related match types, and is described in section A.2.3.3. `oxm_field` is an class-specific value, identifying one of the match types within the match class. The combination of `oxm_class` and `oxm_field` (the most-significant 23 bits of the header) are collectively `oxm_type`. The `oxm_type` normally designates a protocol header field, such as the Ethernet type, but it can also refer to packet metadata, such as the switch port on which a packet arrived.

`oxm_hasmask` defines if the OXM TLV contains a bitmask, more details is explained in section A.2.3.5.

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

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

A.2.3.3 OXM classes

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

The following OXM classes are defined:

```
/* OXM Class IDs.
 * The high order bit differentiate reserved classes from member classes.
 * Classes 0x0000 to 0x7FFF are member classes, allocated by ONF.
 * Classes 0x8000 to 0xFFFE are reserved classes, reserved for standardisation.
```

```
 */
enum ofp_oxm_class {
    OFPXMC_NXM_0          = 0x0000,    /* Backward compatibility with NXM */
    OFPXMC_NXM_1          = 0x0001,    /* Backward compatibility with NXM */
    OFPXMC_OPENFLOW_BASIC = 0x8000,    /* Basic class for OpenFlow */
    OFPXMC_EXPERIMENTER   = 0xFFFF,    /* Experimenter class */
};
```

The class `OFPXMC_OPENFLOW_BASIC` contains the basic set of OpenFlow match fields (see A.2.3.7). The optional class `OFPXMC_EXPERIMENTER` is used for experimenter matches (see A.2.3.8). Other ONF reserved classes are reserved for future uses such as modularisation of the specification. The first two ONF member classes `OFPXMC_NXM_0` and `OFPXMC_NXM_1` are reserved for backward compatibility with the Nicira Extensible Match (NXM) specification.

A.2.3.4 Flow Matching

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

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

- If `oxm_hasmask` is 0, the OXM TLV's body contains a value for the field, called `oxm_value`. The OXM TLV match matches only packets in which the corresponding field equals `oxm_value`.
- If `oxm_hasmask` is 1, then the `oxm_entry`'s body contains a value for the field (`oxm_value`), followed by a bitmask of the same length as the value, called `oxm_mask`. The OXM TLV match matches only packets in which the corresponding field equals `oxm_value` for the bits defined by `oxm_mask`.

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

A.2.3.5 Flow Match Field Masking

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

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

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

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

These restrictions are detailed in specifications for individual fields. A switch may accept an `oxm_hasmask` or `oxm_mask` value that the specification disallows, but only if the switch correctly implements support for that `oxm_hasmask` or `oxm_mask` value. A switch must reject an attempt to set up a flow that contains a `oxm_hasmask` or `oxm_mask` value that it does not support (see 6.7).

A.2.3.6 Flow Match Field Prerequisite

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

For example:

- An OXM TLV for `oxm_type`=OXM_OF_IPV4_SRC is allowed only if it is preceded by another entry with `oxm_type=OXM_OF_ETH_TYPE`, `oxm_hasmask`=0, and `oxm_value`=0x0800. That is, matching on the IPv4 source address is allowed only if the Ethernet type is explicitly set to IPv4.
- An OXM TLV for `oxm_type`=OXM_OF_TCP_SRC is allowed only if it is preceded by an entry with `oxm_type`=OXM_OF_ETH_TYPE, `oxm_hasmask`=0, `oxm_value`=0x0800 or 0x86dd, and another with `oxm_type`=OXM_OF_IP_PROTO, `oxm_hasmask`=0, `oxm_value`=6, in that order. That is, matching on the TCP source port is allowed only if the Ethernet type is IP and the IP protocol is TCP.
- An OXM TLV for `oxm_type`=OXM_OF_MPLS_LABEL is allowed only if it is preceded by an entry with `oxm_type`=OXM_OF_ETH_TYPE, `oxm_hasmask`=0, `oxm_value`=0x8847 or 0x8848.
- An OXM TLV for `oxm_type`=OXM_OF_VLAN_PCP is allowed only if it is preceded by an entry with `oxm_type`=OXM_OF_VLAN_VID, `oxm_value`!=OFPVID_NONE.

These restrictions are be noted in specifications for individual fields. A switch may implement relaxed versions of these restrictions. For example, a switch may accept no prerequisite at all. A switch must reject an attempt to set up a flow that violates its restrictions (see 6.7), and must deal with inconsistent matches created by the lack of prerequisite (for example matching both a TCP source port and a UDP destination port).

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

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

Any given `oxm_type` may appear in an OpenFlow match at most once, otherwise the switch must generate an error (see 6.7). A switch may implement a relaxed version of this rule and may allow in some cases a `oxm_type` to appear multiple time in an OpenFlow match, however the behaviour of matching is then implementation-defined.

A.2.3.7 Flow Match Fields

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

```
/* OXM Flow match field types for OpenFlow basic class. */
enum oxm_ofb_match_fields {
    OFPXMT_OFB_IN_PORT        = 0,  /* Switch input port. */
    OFPXMT_OFB_IN_PHY_PORT    = 1,  /* Switch physical input port. */
    OFPXMT_OFB_METADATA       = 2,  /* Metadata passed between tables. */
    OFPXMT_OFB_ETH_DST        = 3,  /* Ethernet destination address. */
    OFPXMT_OFB_ETH_SRC        = 4,  /* Ethernet source address. */
    OFPXMT_OFB_ETH_TYPE       = 5,  /* Ethernet frame type. */
    OFPXMT_OFB_VLAN_VID       = 6,  /* VLAN id. */
    OFPXMT_OFB_VLAN_PCP       = 7,  /* VLAN priority. */
    OFPXMT_OFB_IP_DSCP        = 8,  /* IP DSCP (6 bits in ToS field). */
```

```
    OFPXMT_OFB_IP_ECN         = 9,  /* IP ECN (2 bits in ToS field). */
    OFPXMT_OFB_IP_PROTO       = 10, /* IP protocol. */
    OFPXMT_OFB_IPV4_SRC       = 11, /* IPv4 source address. */
    OFPXMT_OFB_IPV4_DST       = 12, /* IPv4 destination address. */
    OFPXMT_OFB_TCP_SRC        = 13, /* TCP source port. */
    OFPXMT_OFB_TCP_DST        = 14, /* TCP destination port. */
    OFPXMT_OFB_UDP_SRC        = 15, /* UDP source port. */
    OFPXMT_OFB_UDP_DST        = 16, /* UDP destination port. */
    OFPXMT_OFB_SCTP_SRC       = 17, /* SCTP source port. */
    OFPXMT_OFB_SCTP_DST       = 18, /* SCTP destination port. */
    OFPXMT_OFB_ICMPV4_TYPE    = 19, /* ICMP type. */
    OFPXMT_OFB_ICMPV4_CODE    = 20, /* ICMP code. */
    OFPXMT_OFB_ARP_OP         = 21, /* ARP opcode. */
    OFPXMT_OFB_ARP_SPA        = 22, /* ARP source IPv4 address. */
    OFPXMT_OFB_ARP_TPA        = 23, /* ARP target IPv4 address. */
    OFPXMT_OFB_ARP_SHA        = 24, /* ARP source hardware address. */
    OFPXMT_OFB_ARP_THA        = 25, /* ARP target hardware address. */
    OFPXMT_OFB_IPV6_SRC       = 26, /* IPv6 source address. */
    OFPXMT_OFB_IPV6_DST       = 27, /* IPv6 destination address. */
    OFPXMT_OFB_IPV6_FLABEL    = 28, /* IPv6 Flow Label */
    OFPXMT_OFB_ICMPV6_TYPE    = 29, /* ICMPv6 type. */
    OFPXMT_OFB_ICMPV6_CODE    = 30, /* ICMPv6 code. */
    OFPXMT_OFB_IPV6_ND_TARGET = 31, /* Target address for ND. */
    OFPXMT_OFB_IPV6_ND_SLL    = 32, /* Source link-layer for ND. */
    OFPXMT_OFB_IPV6_ND_TLL    = 33, /* Target link-layer for ND. */
    OFPXMT_OFB_MPLS_LABEL     = 34, /* MPLS label. */
    OFPXMT_OFB_MPLS_TC        = 35, /* MPLS TC. */
};
```

A switch is not required to support all match field types, just those listed in the Table 8. Those required match fields don't need to be implemented in the same table lookup. The controller can query the switch about which other fields it supports.

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

Table 8: Required match fields.

All match fields have different size, prerequisites and masking capability, as specified in Table 9. If not explictely specified in the field description, each field type refer the the outermost occurrence of the field in the packet headers.

Field	Bits	Mask	Pre-requisite	Description
OXM_OF_IN_PORT	32	No	None	Ingress port. Numerical representation of incoming port, starting at 1. This may be a physical or switch-defined logical port.
OXM_OF_IN_PHY_PORT	32	No	IN_PORT present	Physical port. In ofp_packet_in messages, underlying physical port when packet received on a logical port.
OXM_OF_METADATA	64	Yes	None	Table metadata. Used to pass information between tables.
OXM_OF_ETH_DST	48	Yes	None	Ethernet destination MAC address.
OXM_OF_ETH_SRC	48	Yes	None	Ethernet source MAC address.
OXM_OF_ETH_TYPE	16	No	None	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_VLAN_VID	12+1	Yes	None	VLAN-ID from 802.1Q header. The CFI bit indicate the presence of a valid VLAN-ID, see below.
OXM_OF_VLAN_PCP	3	No	VLAN_VID!=NONE	VLAN-PCP from 802.1Q header.
OXM_OF_IP_DSCP	6	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	Diff Serv Code Point (DSCP). Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_ECN	2	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	ECN bits of the IP header. Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_PROTO	8	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd	IPv4 or IPv6 protocol number.
OXM_OF_IPV4_SRC	32	Yes	ETH_TYPE=0x0800	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	32	Yes	ETH_TYPE=0x0800	IPv4 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_TCP_SRC	16	No	IP_PROTO=6	TCP source port
OXM_OF_TCP_DST	16	No	IP_PROTO=6	TCP destination port
OXM_OF_UDP_SRC	16	No	IP_PROTO=17	UDP source port
OXM_OF_UDP_DST	16	No	IP_PROTO=17	UDP destination port
OXM_OF_SCTP_SRC	16	No	IP_PROTO=132	SCTP source port
OXM_OF_SCTP_DST	16	No	IP_PROTO=132	SCTP destination port
OXM_OF_ICMPV4_TYPE	8	No	IP_PROTO=1	ICMP type
OXM_OF_ICMPV4_CODE	8	No	IP_PROTO=1	ICMP code
OXM_OF_ARP_OP	16	No	ETH_TYPE=0x0806	ARP opcode
OXM_OF_ARP_SPA	32	Yes	ETH_TYPE=0x0806	Source IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_TPA	32	Yes	ETH_TYPE=0x0806	Target IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_SHA	48	Yes	ETH_TYPE=0x0806	Source Ethernet address in the ARP payload.
OXM_OF_ARP_THA	48	Yes	ETH_TYPE=0x0806	Target Ethernet address in the ARP payload.
OXM_OF_IPV6_SRC	128	Yes	ETH_TYPE=0x86dd	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	128	Yes	ETH_TYPE=0x86dd	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_FLABEL	20	Yes	ETH_TYPE=0x86dd	IPv6 flow label.
OXM_OF_ICMPV6_TYPE	8	No	IP_PROTO=58	ICMPv6 type
OXM_OF_ICMPV6_CODE	8	No	IP_PROTO=58	ICMPv6 code
OXM_OF_IPV6_ND_TARGET	128	No	ICMPV6_TYPE=135 or ICMPV6_TYPE=136	The target address in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_SLL	48	No	ICMPV6_TYPE=135	The source link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_TLL	48	No	ICMPV6_TYPE=136	The target link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_MPLS_LABEL	20	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The LABEL in the first MPLS shim header.

Table 9 – Continued on next page

Table 9 – concluded from previous page

Field	Bits	Mask	Pre-requisite	Description
OXM_OF_MPLS_TC	3	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The TC in the first MPLS shim header.

Table 9: Match fields details.

The ingress port is a valid standard OpenFlow port, either a physical, a logical port, the OFPP_LOCAL reserved port or the OFPP_CONTROLLER reserved port.

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

Omitting the OFPXMT_OFB_VLAN_VID 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 OFPXMT_OFB_VLAN_VID are:

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

The OFPXMT_OFB_VLAN_PCP field must be rejected when the OFPXMT_OFB_VLAN_VID field is wildcarded (not present) or when the value of OFPXMT_OFB_VLAN_VID is set to OFPVID_NONE.

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

Table 10: Match combinations for VLAN tags.

Table 10 summarizes the combinations of wildcard bits and field values for particular matches.

A.2.3.8 Experimenter Flow Match Fields

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

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

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 5.6. 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       = 0,  /* Output to switch port. */
    OFPAT_COPY_TTL_OUT = 11, /* Copy TTL "outwards" -- from next-to-outermost
                                to outermost */
    OFPAT_COPY_TTL_IN  = 12, /* Copy TTL "inwards" -- from outermost to
                                next-to-outermost */
    OFPAT_SET_MPLS_TTL = 15, /* MPLS TTL */
    OFPAT_DEC_MPLS_TTL = 16, /* Decrement MPLS TTL */

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

Output, group, and set-queue actions are described in Section 5.9, tag push/pop actions are described in Table 4, and Set-Field actions are described from their OXM types in Table 9. An action definition contains the action type, length, and any associated data:

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

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

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

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

```
enum ofp_controller_max_len {
OFPCML_MAX       = 0xffe5, /* maximum max_len value which can be used
                              to request a specific byte length. */
```

```
OFPCML_NO_BUFFER = 0xffff  /* indicates that no buffering should be
                              applied and the whole packet is to be
                              sent to the controller. */
};
```

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

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

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

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

A switch may support only queues that are tied to specific PCP/ToS bits. In that case, we cannot map an arbitrary flow 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 MPLS TTL* action uses the following structure and fields:

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

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

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

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

```
/* Action structure for OFPAT_SET_NW_TTL. */
struct ofp_action_nw_ttl {
    uint16_t type;                  /* OFPAT_SET_NW_TTL. */
    uint16_t len;                   /* Length is 8. */
    uint8_t nw_ttl;                 /* IP TTL */
```

```
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_nw_ttl) == 8);
```

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

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

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

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

A *Push VLAN header* action 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.

Set Field actions uses the following structure and fields:

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

The `field` contains a header field described using a single OXM TLV structure (see A.2.3). Set-Field actions are defined by `oxm_type`, the type of the OXM TLV, and modify the corresponding header field in the packet with the value of `oxm_value`, the payload of the OXM TLV. The value of `oxm_hasmask` must be zero and no `oxm_mask` is included. The match of the flow entry must contain the OXM prerequisite curresponding to the field to be set (see A.2.3.6), otherwise an error must be generated (see 6.7).

The type of a set-field action can be any valid OXM header type, the list of possible OXM types are described in Section A.2.3.7 and Table 9. Both Set-Field actions for OXM types `OFPXMT_OFB_IN_PORT` and `OFPXMT_OFB_METADATA` are not supported, because those are not header fields. The Set-Field action overwrite the header field specified by the OXM type, and perform the necessary CRC recalculation based on the header field. The OXM fields refers to the outermost-possible occurence in the header, unless the field type explictely specifies otherwise, and therefore in general the set-field actions applies to the outermost-possible header (e.g. a "Set VLAN ID" set-field action always sets the ID of the outermost VLAN tag).

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

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

The `experimenter` field is the Experimenter ID, which takes the same form as in struct `ofp_experimenter` (see A.5.4).

A.3 Controller-to-Switch Messages

A.3.1 Handshake

Upon 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 using *packet-in* messages (see 6.1.2).

The `n_tables` field describes the number of tables supported by the switch, each of which can have a different set of supported match fields, actions and number of entries. When the controller and switch first communicate, the controller will find out how many tables the switch supports from the Features Reply. If it wishes to understand the size, types, and order in which tables are consulted, the controller sends a `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_PORT_BLOCKED   = 1 << 8   /* Switch will block looping ports. */
};
```

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

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. See
                                   ofp_controller_max_len for valid values.
                                   */
};
OFP_ASSERT(sizeof(struct ofp_switch_config) == 12);
```

The configuration flags include the following:

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

    /* TTL processing - applicable for IP and MPLS packets */
    OFPC_INVALID_TTL_TO_CONTROLLER = 1 << 2, /* Send packets with invalid TTL
                                                to the 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 for processing using a OFPT_PACKET_IN message with reason OFPR_INVALID_TTL. The flag is cleared by default, causing packets with invalid TTL to get dropped. Checking for invalid TTL does not need to be done for every packets, however it must be done at a minimum every time a *decrement TTL* action is applied to a packet.

The miss_send_len field defines the number of bytes of each packet sent to the controller as a result of flow table miss when configured to generate packet-in messages. If this field equals 0, the switch must send zero bytes of the packet in the ofp_packet_in message. If the value is set to OFPCML_NO_BUFFER the complete packet must be included in the message, and should not be buffered.

A.3.3 Flow Table Configuration

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

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

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

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

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

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

The config field is a bitmap that is 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 with reason OFPR_NO_MATCH. 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

A.3.4.1 Modify Flow Entry Message

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

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

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

The cookie field is an opaque data value chosen by the controller. This value appears in flow removed messages and flow statistics, and can also be used to filter flow statistics, flow modification and flow deletion (see 6.7). 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), its cookie field is unchanged.

If the cookie_mask field is non-zero, it is used with the cookie field to restrict flow matching while modifying or deleting flows. This field is ignored by OFPC_ADD messages. The cookie_mask field's behavior is explained in Section 6.7.

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

The command field must be one of the following:

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

The differences between OFPFC_MODIFY and OFPFC_MODIFY_STRICT are explained in Section 6.7 and differences between OFPFC_DELETE and OFPFC_DELETE_STRICT are explained in Section 6.7.

The idle_timeout and hard_timeout fields control how quickly flows expire (see 6.8). 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 messages when matching and adding flows, and for OFPC_MODIFY_STRICT or OFPC_DELETE_STRICT messages when matching flows.

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. */
    OFPFF_RESET_COUNTS  = 1 << 2   /* Reset flow packet and byte counts. */
};
```

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

When the `OFPFF_CHECK_OVERLAP` flag is set, the switch must check that there are no conflicting entries with the same priority prior to inserting it in the flow table. If there is one, the flow mod fails and an error message is returned (see 6.7).

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. If the instruction set is not valid or supported, the switch must generate an error (see 6.7).

A.3.4.2 Modify Group Entry Message

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

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

The semantics of the type and group fields are explained in Section 6.9.

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

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

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

```
/* Group types.  Values in the range [128, 255] are reserved for experimental
 * use. */
enum ofp_group_type {
```

```
    OFPGT_ALL      = 0, /* All (multicast/broadcast) group.  */
    OFPGT_SELECT   = 1, /* Select group. */
    OFPGT_INDIRECT = 2, /* Indirect group. */
    OFPGT_FF       = 3, /* Fast failover group. */
};
```

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

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

A.3.4.3 Port Modification Message

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

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

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

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

A.3.5 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 = 0,

    /* 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 = 1,

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

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

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

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

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

    /* Group description statistics.
     * The request body is empty.
     * The reply body is an array of struct ofp_group_desc_stats. */
    OFPST_GROUP_DESC = 7,

    /* Group features.
     * The request body is empty.
     * The reply body is struct ofp_group_features_stats. */
    OFPST_GROUP_FEATURES = 8,

    /* Experimenter extension.
     * The request and reply bodies begin with
     * struct ofp_experimenter_stats_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 the maximum field value (the unsigned equivalent of -1). Counters are unsigned and wrap around with no overflow indicator.

A.3.5.1 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 . The `dp_desc` field is a free-form string to describe the datapath for debugging purposes, e.g., "switch3 in room 3120". As such, it is not guaranteed to be unique and should not be used as the primary identifier for the datapath—use the `datapath_id` field from the switch features instead (see A.3.1).

A.3.5.2 Individual Flow Statistics

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

```
/* 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),
                                 OFPTT_ALL for all tables. */
```

```
    uint8_t pad[3];           /* Align to 32 bits. */
    uint32_t out_port;        /* Require matching entries to include this
                                 as an output port.  A value of OFPP_ANY
                                 indicates no restriction. */
    uint32_t out_group;       /* Require matching entries to include this
                                 as an output group.  A value of OFPG_ANY
                                 indicates no restriction. */
    uint8_t pad2[4];          /* Align to 64 bits. */
    uint64_t cookie;          /* Require matching entries to contain this
                                 cookie value */
    uint64_t cookie_mask;     /* Mask used to restrict the cookie bits that
                                 must match. A value of 0 indicates
                                 no restriction. */
    struct ofp_match match;   /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 40);
```

The `match` field contains a description of the flows that should be matched and may contain wildcarded and masked fields. This field's matching behavior is described in Section 6.7.

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

The `out_port` and `out_group` fields optionally filter by output port and group. If either `out_port` or `out_group` contain a value other than `OFPP_ANY` and `OFPG_ANY` respectively, it introduces a constraint when matching. This constraint is that the 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 6.7.

The `body` of the reply to a `OFPST_FLOW` stats request 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. */
    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. Variable size. */
    //struct ofp_instruction instructions[0]; /* Instruction set. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 56);
```

The fields consist of those provided in the `flow_mod` that created the flow, plus the `table_id` 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 second precision; higher precision is encouraged where available.

A.3.5.3 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)
                                 OFPTT_ALL for all tables. */
    uint8_t pad[3];           /* Align to 32 bits. */
    uint32_t out_port;        /* Require matching entries to include this
                                 as an output port.  A value of OFPP_ANY
                                 indicates no restriction. */
    uint32_t out_group;       /* Require matching entries to include this
                                 as an output group.  A value of OFPG_ANY
                                 indicates no restriction. */
    uint8_t pad2[4];          /* Align to 64 bits. */
    uint64_t cookie;          /* Require matching entries to contain this
                                 cookie value */
    uint64_t cookie_mask;     /* Mask used to restrict the cookie bits that
                                 must match. A value of 0 indicates
                                 no restriction. */
    struct ofp_match match;   /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_request) == 40);
```

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

A.3.5.4 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];
    uint64_t match;          /* Bitmap of (1 << OFPXMT_*) that indicate the
                                fields the table can match on. */
    uint64_t wildcards;      /* Bitmap of (1 << OFPXMT_*) wildcards that are
                                supported by the table. */
    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. */
    uint64_t write_setfields;/* Bitmap of (1 << OFPXMT_*) header fields that
                                can be set with OFPIT_WRITE_ACTIONS. */
    uint64_t apply_setfields;/* Bitmap of (1 << OFPXMT_*) header fields that
```

```
                                can be set with OFPIT_APPLY_ACTIONS. */
    uint64_t metadata_match; /* Bits of metadata table can match. */
    uint64_t metadata_write; /* Bits of metadata table can write. */
    uint32_t instructions;   /* Bitmap of OFPIT_* values supported. */
    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) == 128);
```

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

The `match` field indicates the fields for which that particular table supports matching on (see A.2.3.7). For example, if the table can match the ingress port, the lowest order bit would be set, and a table that support matching on all fields defined by the specification would have it set to `OFPXMT_OFB_ALL`.

The `wildcards` field 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 TCAM or sequentially searched table would have it set to `OFPXMT_OFB_ALL`.

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

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

The `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 5.9. 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_OUTPUT` would use the flag `0x00000001`.

The `write_setfields` field is a bitmap of Set-Field actions type supported by the table using the `OFPIT_WRITE_ACTIONS` instruction, whereas the `apply_setfields` field refers to the `OFPIT_APPLY_ACTIONS` instruction.

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

A.3.5.5 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);
```

A.3.5.6 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);
```

A.3.5.7 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);
```

A.3.5.8 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.

A.3.5.9 Group Features Statistics

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

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

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

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

A.3.5.10 Experimenter Statistics

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

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

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

The `experimenter` field is the Experimenter ID, which takes the same form as in struct `ofp_experimenter` (see A.5.4).

A.3.6 Queue Configuration Messages

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

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

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

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

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

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 (OFP_NO_BUFFER
                                     if none). */
    uint32_t in_port;             /* Packet's input port or OFPP_CONTROLLER. */
    uint16_t actions_len;         /* Size of action array in bytes. */
    uint8_t pad[6];
    struct ofp_action_header actions[0]; /* Action list. */
    /* 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 OFP_NO_BUFFER, then the packet data is included in the data array.

The in_port field is the ingress port that must be associated with the packet for OpenFlow processing. It must be set to either a valid standard switch port or OFPP_CONTROLLER.

The action field is an action list defining how the packet should be processed by the switch. It may include packet modification, group processing and an output port. The action list of an OFPT_PACKET_OUT message can also specify the OFPP_TABLE reserved port as an output action to process the packet through the 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.

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.3.9 Role Request Message

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

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

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

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

If the role value is OFPCR_ROLE_MASTER, all other controllers which role was OFPCR_ROLE_MASTER are changed to OFPCR_ROLE_SLAVE. If the role value is OFPCR_ROLE_NOCHANGE, the current role of the controller is not changed ; this enable a controller to query its current role without changing it.

Upon receipt of a OFPT_ROLE_REQUEST message, the switch must return a OFPT_ROLE_REPLY message. The structure of this message is exactly the same as the OFPT_ROLE_REQUEST message, and the field `role` is the current role of the controller.

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

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. */
    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 */
    struct ofp_match match; /* Packet metadata. Variable size. */
    /* Followed by:
     *   - Exactly 2 all-zero padding bytes, then
     *   - An Ethernet frame whose length is inferred from header.length.
     * The padding bytes preceding the Ethernet frame ensure that the IP
     * header (if any) following the Ethernet header is 32-bit aligned.
     */
    //uint8_t pad[2];       /* Align to 64 bit + 16 bit */
    //uint8_t data[0];      /* Ethernet frame */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 24);
```

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 - either because of no available buffers, or because of explicitly requested via OFPCML_NO_BUFFER - the entire packet is included in the data portion, and the buffer_id is OFP_NO_BUFFER.

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

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

The reason field can be any of these values:

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

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

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

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

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

A.4.2 Flow Removed Message

If the controller has requested to be notified when flows time out or are deleted from tables, the datapath does this with the `OFPT_FLOW_REMOVED` message:

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

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

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

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

The `reason` field is one of the following:

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

The `duration_sec` and `duration_nsec` fields are described in Section A.3.5.2.

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

With the above three fields, one can find both the amount of time the flow 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. Those counters should behave like other statistics counters (see A.3.5) ; they are unsigned and should be set to the maximum field value if not available.

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     = 0,          /* The port was added. */
    OFPPR_DELETE  = 1,          /* The port was removed. */
    OFPPR_MODIFY  = 2,          /* 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.

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

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

Currently defined error types are:

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

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

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

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

For the `OFPET_BAD_REQUEST` error `type`, the following `code`s are currently defined:

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

For the `OFPET_BAD_ACTION` error `type`, the following `code`s are currently defined:

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

For the `OFPET_BAD_INSTRUCTION` error `type`, the following `code`s are currently defined:

```
/* ofp_error_msg 'code' values for OFPET_BAD_INSTRUCTION.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_instruction_code {
    OFPBIC_UNKNOWN_INST     = 0, /* Unknown instruction. */
```

```
    OFPBIC_UNSUP_INST       = 1, /* Switch or table does not support the
                                    instruction. */
    OFPBIC_BAD_TABLE_ID     = 2, /* Invalid Table-ID specified. */
    OFPBIC_UNSUP_METADATA   = 3, /* Metadata value unsupported by datapath. */
    OFPBIC_UNSUP_METADATA_MASK = 4, /* Metadata mask value unsupported by
                                       datapath. */
    OFPBIC_BAD_EXPERIMENTER = 5, /* Unknown experimenter id specified. */
    OFPBIC_BAD_EXP_TYPE     = 6, /* Unknown instruction for experimenter id. */
    OFPBIC_BAD_LEN          = 7, /* Length problem in instructions. */
    OFPBIC_EPERM            = 8, /* Permissions error. */
};
```

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

```
/* ofp_error_msg 'code' values for OFPET_BAD_MATCH.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_match_code {
    OFPBMC_BAD_TYPE         = 0,  /* Unsupported match type specified by the
                                     match */
    OFPBMC_BAD_LEN          = 1,  /* Length problem in match. */
    OFPBMC_BAD_TAG          = 2,  /* Match uses an unsupported tag/encap. */
    OFPBMC_BAD_DL_ADDR_MASK = 3,  /* Unsupported datalink addr mask - switch
                                     does not support arbitrary datalink
                                     address mask. */
    OFPBMC_BAD_NW_ADDR_MASK = 4,  /* Unsupported network addr mask - switch
                                     does not support arbitrary network
                                     address mask. */
    OFPBMC_BAD_WILDCARDS    = 5,  /* Unsupported combination of fields masked
                                     or omitted in the match. */
    OFPBMC_BAD_FIELD        = 6,  /* Unsupported field type in the match. */
    OFPBMC_BAD_VALUE        = 7,  /* Unsupported value in a match field. */
    OFPBMC_BAD_MASK         = 8,  /* Unsupported mask specified in the match,
                                     field is not dl-address or nw-address. */
    OFPBMC_BAD_PREREQ       = 9,  /* A prerequisite was not met. */
    OFPBMC_DUP_FIELD        = 10, /* A field type was duplicated. */
    OFPBMC_EPERM            = 11, /* Permissions error. */
};
```

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

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

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

```
/* ofp_error_msg 'code' values for OFPET_GROUP_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_group_mod_failed_code {
    OFPGMFC_GROUP_EXISTS         = 0,  /* Group not added because a group ADD
                                          attempted to replace an
                                          already-present group. */
    OFPGMFC_INVALID_GROUP        = 1,  /* Group not added because Group
```

```
                                        specified is invalid. */
    OFPGMFC_WEIGHT_UNSUPPORTED   = 2,  /* Switch does not support unequal load
                                        sharing with select groups. */
    OFPGMFC_OUT_OF_GROUPS        = 3,  /* The group table is full. */
    OFPGMFC_OUT_OF_BUCKETS       = 4,  /* The maximum number of action buckets
                                        for a group has been exceeded. */
    OFPGMFC_CHAINING_UNSUPPORTED = 5,  /* Switch does not support groups that
                                        forward to groups. */
    OFPGMFC_WATCH_UNSUPPORTED    = 6,  /* This group cannot watch the watch_port
                                        or watch_group specified. */
    OFPGMFC_LOOP                 = 7,  /* Group entry would cause a loop. */
    OFPGMFC_UNKNOWN_GROUP        = 8,  /* Group not modified because a group
                                        MODIFY attempted to modify a
                                        non-existent group. */
    OFPGMFC_CHAINED_GROUP        = 9,  /* Group not deleted because another
                                        group is forwarding to it. */
    OFPGMFC_BAD_TYPE             = 10, /* Unsupported or unknown group type. */
    OFPGMFC_BAD_COMMAND          = 11, /* Unsupported or unknown command. */
    OFPGMFC_BAD_BUCKET           = 12, /* Error in bucket. */
    OFPGMFC_BAD_WATCH            = 13, /* Error in watch port/group. */
    OFPGMFC_EPERM                = 14, /* Permissions error. */
};
```

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

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

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

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

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

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

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

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

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

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

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

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

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

The experimenter field is the Experimenter ID, which takes the same form as in struct ofp_experimenter (see A.5.4).

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 ONF. */
    uint32_t exp_type;          /* Experimenter defined. */
    /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_header) == 16);
```

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

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

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

Appendix B Release Notes

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

B.1 OpenFlow version 0.2.0

Release date : March 28,2008
Wire Protocol : 1

B.2 OpenFlow version 0.2.1

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

B.3 OpenFlow version 0.8.0

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

- Reorganise OpenFlow message types
- Add `OFPP_TABLE` virtual port to send packet-out packet to the tables
- Add global flag `OFPC_SEND_FLOW_EXP` to configure flow expired messages generation
- Add flow priority
- Remove flow Group-ID (experimental QoS support)

- Add Error messages
- Make stat request and stat reply more generic, with a generic header and stat specific body
- Change fragmentation strategy for stats reply, use explicit flag `OFPSF_REPLY_MORE` instead of empty packet
- Add table stats and port stats mesages

B.4 OpenFlow version 0.8.1

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

B.5 OpenFlow version 0.8.2

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

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

B.6 OpenFlow version 0.8.9

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

B.6.1 IP Netmasks

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

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

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

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

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

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

B.6.2 New Physical Port Stats

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

```
/* Body of reply to OFPST_PORT request. If a counter is unsupported, set
 * the field to all ones. */
struct ofp_port_stats {
    uint16_t port_no;
    uint8_t pad[6];          /* Align to 64-bits. */
    uint64_t rx_packets;     /* Number of received packets. */
    uint64_t tx_packets;     /* Number of transmitted packets. */
    uint64_t rx_bytes;       /* Number of received bytes. */
    uint64_t tx_bytes;       /* Number of transmitted bytes. */
    uint64_t rx_dropped;     /* Number of packets dropped by RX. */
    uint64_t tx_dropped;     /* Number of packets dropped by TX. */
    uint64_t rx_errors;      /* Number of receive errors.  This is a super-set
                                of receive errors and should be great than or
                                equal to the sum of al rx_*_err values. */
    uint64_t tx_errors;      /* Number of transmit errors.  This is a super-set
                                of transmit errors. */
    uint64_t rx_frame_err;   /* Number of frame alignment errors. */
    uint64_t rx_over_err;    /* Number of packets with RX overrun. */
    uint64_t rx_crc_err;     /* Number of CRC errors. */
    uint64_t collisions;     /* Number of collisions. */
};
```

B.6.3 IN_PORT Virtual Port

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

B.6.4 Port and Link Status and Configuration

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

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

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

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

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

B.6.5 Echo Request/Reply Messages

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

B.6.6 Vendor Extensions

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

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

The *vendor* field is a 32-bit value that uniquely identifies the vendor. If the most significant byte is zero, the next three bytes are the vendor's IEEE OUI. If vendor does not have (or wish to use) their OUI, they should contact the OpenFlow consortium to obtain one. The rest of the body is uninterpreted.

It is also possible to add vendor extensions for stats messages with the `OFPST_VENDOR` stats type. The first four bytes of the message are the vendor identifier as described earlier. The rest of the body is vendor-defined.

To indicate that a switch does not understand a vendor extension, a `OFPBRC_BAD_VENDOR` error code has been defined under the `OFPET_BAD_REQUEST` error type.

Vendor-defined actions are described below in the "Variable Length and Vendor Actions" section.

B.6.7 Explicit Handling of IP Fragments

In previous versions of the specification, handling of IP fragments was not clearly defined. The switch is now able to tell the controller whether it is able to reassemble fragments. This is done with the following `capabilities` flag passed in the `ofp_switch` features message:

```
OFPC_IP_REASM     = 1 << 5  /* Can reassemble IP fragments. */
```

The controller can configure fragment handling in the switch through the setting the following new `ofp_config_flags` in the `ofp_switch_config` message:

```
/* Handling of IP fragments. */
OFPC_FRAG_NORMAL   = 0 << 1,  /* No special handling for fragments. */
OFPC_FRAG_DROP     = 1 << 1,  /* Drop fragments. */
OFPC_FRAG_REASM    = 2 << 1,  /* Reassemble (only if OFPC_IP_REASM set). */
OFPC_FRAG_MASK     = 3 << 1
```

"Normal" handling of fragments means that an attempt should be made to pass the fragments through the OpenFlow tables. If any field is not present (e.g., the TCP/UDP ports didn't fit), then the packet should not match any entry that has that field set.

B.6.8 802.1D Spanning Tree

OpenFlow now has a way to configure and view results of on-switch implementations of 802.1D Spanning Tree Protocol.

A switch that implements STP must set the new `OFPC_STP` bit in the 'capabilities' field of its `OFPT_FEATURES_REPLY` message. A switch that implements STP at all must make it available on all of its physical ports, but it need not implement it on virtual ports (e.g. `OFPP_LOCAL`).

Several port configuration flags are associated with STP. The complete set of port configuration flags are:

```
enum ofp_port_config {
   OFPPC_PORT_DOWN    = 1 << 0,  /* Port is administratively down. */
   OFPPC_NO_STP       = 1 << 1,  /* Disable 802.1D spanning tree on port. */
   OFPPC_NO_RECV      = 1 << 2,  /* Drop most packets received on port. */
   OFPPC_NO_RECV_STP  = 1 << 3,  /* Drop received 802.1D STP packets. */
   OFPPC_NO_FLOOD     = 1 << 4,  /* Do not include this port when flooding. */
   OFPPC_NO_FWD       = 1 << 5,  /* Drop packets forwarded to port. */
   OFPPC_NO_PACKET_IN = 1 << 6   /* Do not send packet-in msgs for port. */
};
```

The controller may set `OFPPFL_NO_STP` to 0 to enable STP on a port or to 1 to disable STP on a port. (The latter corresponds to the Disabled STP port state.) The default is switch implementation-defined; the OpenFlow reference implementation by default sets this bit to 0 (enabling STP).

When `OFPPFL_NO_STP` is 0, STP controls the `OFPPFL_NO_FLOOD` and `OFPPFL_STP_*` bits directly. `OFPPFL_NO_FLOOD` is set to 0 when the STP port state is Forwarding, otherwise to 1. The bits in `OFPPFL_STP_MASK` are set to one of the other `OFPPFL_STP_*` values according to the current STP port state.

When the port flags are changed by STP, the switch sends an `OFPT_PORT_STATUS` message to notify the controller of the change. The `OFPPFL_NO_RECV`, `OFPPFL_NO_RECV_STP`, `OFPPFL_NO_FWD`, and `OFPPFL_NO_PACKET_IN` bits in the OpenFlow port flags may be useful for the controller to implement STP, although they interact poorly with in-band control.

B.6.9 Modify Actions in Existing Flow Entries

New `ofp_flow_mod` commands have been added to support modifying the actions of existing entries: `OFPFC_MODIFY` and `OFPFC_MODIFY_STRICT`. They use the match field to describe the entries that should be modified with the supplied actions. `OFPFC_MODIFY` is similar to `OFPFC_DELETE`, in that wildcards are "active". `OFPFC_MODIFY_STRICT` is similar to `OFPFC_DELETE_STRICT`, in that wildcards are not "active", so both the wildcards and priority must match an entry. When a matching flow is found, only its actions are modified–information such as counters and timers are not reset.

If the controller uses the `OFPFC_ADD` command to add an entry that already exists, then the new entry replaces the old and all counters and timers are reset.

B.6.10 More Flexible Description of Tables

Previous versions of OpenFlow had very limited abilities to describe the tables supported by the switch. The `n_exact`, `n_compression`, and `n_general` fields in `ofp_switch_features` have been replaced with `n_tables`, which lists the number of tables in the switch.

The behavior of the `OFPST_TABLE` stat reply has been modified slightly. The `ofp_table_stats` body

now contains a wildcards field, which indicates the fields for which that particular table supports wildcarding. For example, a direct look-up hash table would have that field set to zero, while a sequentially searched table would have it set to `OFPFW_ALL`. The `ofp_table_stats` entries are returned in the order that packets traverse the tables.

When the controller and switch first communicate, the controller will find out how many tables the switch supports from the Features Reply. If it wishes to understand the size, types, and order in which tables are consulted, the controller sends a `OFPST_TABLE` stats request.

B.6.11 Lookup Count in Tables

Table stats returned `ofp_table_stats` structures now return the number of packets that have been looked up in the table–whether they hit or not. This is stored in the `lookup_count` field.

B.6.12 Modifying Flags in Port-Mod More Explicit

The `ofp_port_mod` is used to modify characteristics of a switch's ports. A supplied `ofp_phy_port` structure describes the behavior of the switch through its `flags` field. However, it's possible that the controller wishes to change a particular flag and may not know the current status of all flags. A `mask` field has been added which has a bit set for each flag that should be changed on the switch.

The new `ofp_port_mod` message looks like the following:

```
/* Modify behavior of the physical port */
struct ofp_port_mod {
    struct ofp_header header;
    uint32_t mask;         /* Bitmap of "ofp_port_flags" that should be
                              changed. */
    struct ofp_phy_port desc;
};
```

B.6.13 New Packet-Out Message Format

The previous version's `packet-out` message treated the variable-length array differently depending on whether the `buffer_id` was set or not. If set, the array consisted of actions to be executed and the `out_port` was ignored. If not, the array consisted of the actual packet that should be placed on the wire through the `out_port` interface. This was a bit ugly, and it meant that in order for a non-buffered packet to have multiple actions executed on it, that a new flow entry be created just to match that entry.

A new format is now used, which cleans the message up a bit. The packet always contains a list of actions. An additional variable-length array follows the list of actions with the contents of the packet if `buffer_id` is not set. This is the new format:

```
struct ofp_packet_out {
    struct ofp_header header;
    uint32_t buffer_id;           /* ID assigned by datapath (-1 if none). */
    uint16_t in_port;             /* Packet's input port (OFPP_NONE if none). */
    uint16_t n_actions;           /* Number of actions. */
    struct ofp_action actions[0]; /* Actions. */
    /* uint8_t data[0]; */        /* Packet data.  The length is inferred
                                     from the length field in the header.
                                     (Only meaningful if buffer_id == -1.) */
};
```

B.6.14 Hard Timeout for Flow Entries

A hard timeout value has been added to flow entries. If set, then the entry must be expired in the specified number of seconds regardless of whether or not packets are hitting the entry. A `hard_timeout` field has

been added to the `flow_mod` message to support this. The `max_idle` field has been renamed `idle_timeout`. A value of zero means that a timeout has not been set. If both `idle_timeout` and `hard_timeout` are zero, then the flow is permanent and should not be deleted without an explicit deletion.

The new `ofp_flow_mod` format looks like this:

```
struct ofp_flow_mod {
    struct ofp_header header;
    struct ofp_match match;      /* Fields to match */

    /* Flow actions. */
    uint16_t command;             /* One of OFPFC_*. */
    uint16_t idle_timeout;        /* Idle time before discarding (seconds). */
    uint16_t hard_timeout;        /* Max time before discarding (seconds). */
    uint16_t priority;            /* Priority level of flow entry. */
    uint32_t buffer_id;           /* Buffered packet to apply to (or -1).
                                     Not meaningful for OFPFC_DELETE*. */
    uint32_t reserved;            /* Reserved for future use. */
    struct ofp_action actions[0]; /* The number of actions is inferred from
                                     the length field in the header. */
};
```

Since flow entries can now be expired due to idle or hard timeouts, a `reason` field has been added to the `ofp_flow_expired` message. A value of 0 indicates an idle timeout and 1 indicates a hard timeout:

```
enum ofp_flow_expired_reason {
    OFPER_IDLE_TIMEOUT,         /* Flow idle time exceeded idle_timeout. */
    OFPER_HARD_TIMEOUT          /* Time exceeded hard_timeout. */
};
```

The new `ofp_flow_expired` message looks like the following:

```
struct ofp_flow_expired {
    struct ofp_header header;
    struct ofp_match match;   /* Description of fields */

    uint16_t priority;        /* Priority level of flow entry. */
    uint8_t reason;           /* One of OFPER_*. */
    uint8_t pad[1];           /* Align to 32-bits. */

    uint32_t duration;        /* Time flow was alive in seconds. */
    uint8_t pad2[4];          /* Align to 64-bits. */
    uint64_t packet_count;
    uint64_t byte_count;
};
```

B.6.15 Reworked initial handshake to support backwards compatibility

OpenFlow now includes a basic "version negotiation" capability. When an OpenFlow connection is established, each side of the connection should immediately send an `OFPT_HELLO` message as its first OpenFlow message. The 'version' field in the hello message should be the highest OpenFlow protocol version supported by the sender. Upon receipt of this message, the recipient may calculate the OpenFlow protocol version to be used as the smaller of the version number that it sent and the one that it received.

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

The OFPT_HELLO message has no body; that is, it consists only of an OpenFlow header. Implementations must be prepared to receive a hello message that includes a body, ignoring its contents, to allow for later extensions.

B.6.16 Description of Switch Stat

The OFPST_DESC stat has been added to describe the hardware and software running on the switch:

```
#define DESC_STR_LEN   256
#define SERIAL_NUM_LEN 32
/* Body of reply to OFPST_DESC request.  Each entry is a NULL-terminated
 * ASCII string. */
struct ofp_desc_stats {
    char mfr_desc[DESC_STR_LEN];       /* Manufacturer description. */
    char hw_desc[DESC_STR_LEN];        /* Hardware description. */
    char sw_desc[DESC_STR_LEN];        /* Software description. */
    char serial_num[SERIAL_NUM_LEN];   /* Serial number. */
};
```

It contains a 256 character ASCII description of the manufacturer, hardware type, and software version. It also contains a 32 character ASCII serial number. Each entry is padded on the right with 0 bytes.

B.6.17 Variable Length and Vendor Actions

Vendor-defined actions have been added to OpenFlow. To enable more versatility, actions have switched from fixed-length to variable. All actions have the following header:

```
struct ofp_action_header {
    uint16_t type;                  /* One of OFPAT_*. */
    uint16_t len;                   /* Length of action, including this
                                       header.  This is the length of action,
                                       including any padding to make it
                                       64-bit aligned. */
    uint8_t pad[4];
};
```

The length for actions must always be a multiple of eight to aid in 64-bit alignment. The action types are as follows:

```
enum ofp_action_type {
    OFPAT_OUTPUT,           /* Output to switch port. */
    OFPAT_SET_VLAN_VID,     /* Set the 802.1q VLAN id. */
    OFPAT_SET_VLAN_PCP,     /* Set the 802.1q priority. */
    OFPAT_STRIP_VLAN,       /* Strip the 802.1q header. */
    OFPAT_SET_DL_SRC,       /* Ethernet source address. */
    OFPAT_SET_DL_DST,       /* Ethernet destination address. */
    OFPAT_SET_NW_SRC,       /* IP source address. */
    OFPAT_SET_NW_DST,       /* IP destination address. */
    OFPAT_SET_TP_SRC,       /* TCP/UDP source port. */
    OFPAT_SET_TP_DST,       /* TCP/UDP destination port. */
    OFPAT_VENDOR = 0xffff
};
```

The vendor-defined action header looks like the following:

```
struct ofp_action_vendor_header {
    uint16_t type;                  /* OFPAT_VENDOR. */
    uint16_t len;                   /* Length is 8. */
    uint32_t vendor;                /* Vendor ID, which takes the same form
                                       as in "struct ofp_vendor". */
};
```

The vendor field uses the same vendor identifier described earlier in the "Vendor Extensions" section. Beyond using the ofp_action_vendor header and the 64-bit alignment requirement, vendors are free to use whatever body for the message they like.

B.6.18 VLAN Action Changes

It is now possible to set the priority field in VLAN tags and stripping VLAN tags is now a separate action. The OFPAT_SET_VLAN_VID action behaves like the former OFPAT_SET_DL_VLAN action, but no longer accepts a special value that causes it to strip the VLAN tag. The OFPAT_SET_VLAN_PCP action modifies the 3-bit priority field in the VLAN tag. For existing tags, both actions only modify the bits associated with the field being updated. If a new VLAN tag needs to be added, the value of all other fields is zero.

The OFPAT_SET_VLAN_VID action looks like the following:

```
struct ofp_action_vlan_vid {
    uint16_t type;                  /* OFPAT_SET_VLAN_VID. */
    uint16_t len;                   /* Length is 8. */
    uint16_t vlan_vid;              /* VLAN id. */
    uint8_t pad[2];
};
```

The OFPAT_SET_VLAN_PCP action looks like the following::

```
struct ofp_action_vlan_pcp {
    uint16_t type;                  /* OFPAT_SET_VLAN_PCP. */
    uint16_t len;                   /* Length is 8. */
    uint8_t vlan_pcp;               /* VLAN priority. */
    uint8_t pad[3];
};
```

The OFPAT_STRIP_VLAN action takes no argument and strips the VLAN tag if one is present.

B.6.19 Max Supported Ports Set to 65280

What: Increase maximum number of ports to support large vendor switches; was previously 256, chosen arbitrarily.

Why: The HP 5412 chassis supports 288 ports of Ethernet, and some Cisco switches go much higher. The current limit (OFPP_MAX) is 255, set to equal the maximum number of ports in a bridge segment in the 1998 STP spec. The RSTP spec from 2004 supports up to 4096 (12 bits) of ports.

How: Change OFPP_MAX to 65280. (However, out of the box, the reference switch implementation supports at most 256 ports.)

B.6.20 Send Error Message When Flow Not Added Due To Full Tables

The switch now sends an error message when a flow is added, but cannot because all the tables are full. The message has an error type of OFPET_FLOW_MOD_FAILED and code of OFPFMFC_ALL_TABLES_FULL. If the Flow-Mod command references a buffered packet, then actions are not performed on the packet. If the controller wishes the packet to be sent regardless of whether or not a flow entry is added, then it should use a Packet-Out directly.

B.6.21 Behavior Defined When Controller Connection Lost

What: Ensure that all switches have at least one common behavior when the controller connection is lost.

Why: When the connection to the controller is lost, the switch should behave in a well-defined way. Reasonable behaviors include 'do nothing - let flows naturally timeout', 'freeze timeouts', 'become learning switch', and 'attempt connection to other controller'. Switches may implement one or more of these, and network admins may want to ensure that if the controller goes out, they know what the network can do.

The first is the simplest: ensure that every switch implements a default of 'do nothing - let flows timeout naturally'. Changes must be done via vendor-specific command line interface or vendor extension OpenFlow messages.

The second may help ensure that a single controller can work with switches from multiple vendors. The different failure behaviors, plus 'other', could be feature bits set for the switch. A switch would still only have to support the default.

The worry here is that we may not be able to enumerate in advance the full range of failure behaviors, which argues for the first approach.

How: Added text to spec: "In the case that the switch loses contact with the controller, the default behavior must be to do nothing - to let flows timeout naturally. Other behaviors can be implemented via vendor-specific command line interface or vendor extension OpenFlow messages."

B.6.22 ICMP Type and Code Fields Now Matchable

What: Allow matching ICMP traffic based on type or code.

Why: We can't distinguish between different types of ICMP traffic (e.g., echo replies vs echo requests vs redirects).

How: Changed spec to allow matching on these fields.

As for implementation: The type and code are each a single byte, so they easily fit in our existing flow structure. Overload the `tp_src` field to ICMP type and `tp_dst` to ICMP code. Since they are only a single byte, they will reside in the low-byte of these two byte fields (stored in network-byte order). This will allow a controller to use the existing wildcard bits to wildcard these ICMP fields.

B.6.23 Output Port Filtering for Delete*, Flow Stats and Aggregate Stats

Add support for listing and deleting entries based on an output port.

To support this, an `out_port` field has been added to the `ofp_flow_mod`, `ofp_flow_stats_request`, and `ofp_aggregate_stats_request` messages. If an `out_port` contains a value other than `OFPP_NONE`, it introduces a constraint when matching. This constraint is that the rule must contain an output action directed at that port. Other constraints such as `ofp_match` structs and priorities are still used; this is purely an *additional* constraint. Note that to get previous behavior, though, `out_port` must be set to `OFPP_NONE`, since "0" is a valid port id. This only applies to the `delete` and `delete_strict` flow mod commands; the field is ignored by `add`, `modify`, and `modify_strict`.

B.7 OpenFlow version 0.9

Release date : July 20, 2009
Wire Protocol : 0x98

B.7.1 Failover

The reference implementation now includes a simple failover mechanism. A switch can be configured with a list of controllers. If the first controller fails, it will automatically switch over to the second controller on the list.

B.7.2 Emergency Flow Cache

The protocol and reference implementation have been extended to allow insertion and management of emergency flow entries.
Emergency-specific flow entries are inactive until a switch loses connectivity from the controller. If this happens, the switch invalidates all normal flow table entries and copies all emergency flows into the normal flow table.
Upon connecting to a controller again, all entries in the flow cache stay active. The controller then has the option of resetting the flow cache if needed.

B.7.3 Barrier Command

The Barrier Command is a mechanism to get notified when an OpenFlow message has finished executing on the switch. When a switch receives a Barrier message it must first complete all commands sent before the Barrier message before executing any commands after it. When all commands before the Barrier message have completed, it must send a Barrier Reply message back to the controller.

B.7.4 Match on VLAN Priority Bits

There is an optional new feature that allows matching on priority VLAN fields. Pre 0.9, the VLAN id is a field used in identifying a flow, but the priority bits in the VLAN tag are not. In this release we include the priority bits as a separate field to identify flows. Matching is possible as either an exact match on the 3 priority bits, or as a wildcard for the entire 3 bits.

B.7.5 Selective Flow Expirations

Flow expiration messages can now be requested on a per-flow, rather than per-switch granularity.

B.7.6 Flow Mod Behavior

There now is a `CHECK_OVERLAP` flag to flow mods which requires the switch to do the (potentially more costly) check that there doesn't already exist a conflicting flow with the same priority. If there is one, the mod fails and an error code is returned. Support for this flag is required in an OpenFlow switch.

B.7.7 Flow Expiration Duration

The meaning of the "duration" field in the Flow Expiration message has been changed slightly. Previously there were conflicting definitions of this in the spec. In 0.9 the value returned will be the time that the flow was active and not include the timeout period.

B.7.8 Notification for Flow Deletes

If a controller deletes a flow it now receives a notification if the notification bit is set. In previous releases only flow expirations but not delete actions would trigger notifications.

B.7.9 Rewrite DSCP in IP ToS header

There is now an added Flow action to rewrite the DiffServ CodePoint bits part of the IP ToS field in the IP header. This enables basic support for basic QoS with OpenFlow in in some switches. A more complete QoS framework is planned for a future OpenFlow release.

B.7.10 Port Enumeration now starts at 1

Previous releases of OpenFlow had port numbers start at 0, release 0.9 changes them to start at 1.

B.7.11 Other changes to the Specification

- 6633/TCP is now the recommended default OpenFlow Port. Long term the goal is to get a IANA approved port for OpenFlow.
- The use of "Type 1" and "Type 0" has been depreciated and references to it have been removed.
- Clarified Matching Behavior for Flow Modification and Stats
- Made explicit that packets received on ports that are disabled by spanning tree must follow the normal flow table processing path.
- Clarified that transaction ID in header should match offending message for `OFPET_BAD_REQUEST`, `OFPET_BAD_ACTION`, `OFPET_FLOW_MOD_FAILED`.
- Clarified the format for the Strip VLAN Action
- Clarify behavior for packets that are buffered on the switch while switch is waiting for a reply from controller
- Added the new `EPERM` Error Type
- Fixed Flow Table Matching Diagram
- Clarified datapath ID 64 bits, up from 48 bits
- Clarified `miss-send-len` and `max-len` of output action

B.8 OpenFlow version 1.0

Release date : December 31, 2009
Wire Protocol : 0x01

B.8.1 Slicing

OpenFlow now supports multiple queues per output port. Queues support the ability to provide minimum bandwidth guarantees; the bandwidth allocated to each queue is configurable. The name slicing is derived from the ability to provide a slice of the available network bandwidth to each queue.

B.8.2 Flow cookies

Flows have been extended to include an opaque identifier, referred to as a cookie. The cookie is specified by the controller when the flow is installed; the cookie will be returned as part of each flow stats and flow expired message.

B.8.3 User-specifiable datapath description

The OFPST_DESC (switch description) reply now includes a datapath description field. This is a user-specifiable field that allows a switch to return a string specified by the switch owner to describe the switch.

B.8.4 Match on IP fields in ARP packets

The reference implementation can now match on IP fields inside ARP packets. The source and destination protocol address are mapped to the nw_src and nw_dst fields respecitively, and the opcode is mapped to the nw_proto field.

B.8.5 Match on IP ToS/DSCP bits

OpenFlow now supports matching on the IP ToS/DSCP bits.

B.8.6 Querying port stats for individual ports

Port stat request messages include a `port_no` field to allow stats for individual ports to be queried. Port stats for all ports can still be requested by specifying `OFPP_NONE` as the port number.

B.8.7 Improved flow duration resolution in stats/expiry messages

Flow durations in stats and expiry messages are now expressed with nanosecond resolution. Note that the accuracy of flow durations in the reference implementation is on the order of milliseconds. (The actual accuracy is in part dependent upon kernel parameters.)

B.8.8 Other changes to the Specification

- remove `multi_phy_tx` spec text and capability bit
- clarify execution order of actions
- replace SSL refs with TLS
- resolve overlap ambiguity
- clarify flow mod to non-existing port
- clarify port definition
- update packet flow diagram
- update header parsing diagram for ICMP
- fix English ambiguity for flow-removed messages
- fix async message English ambiguity
- note that multiple controller support is undefined
- clarify that byte equals octet
- note counter wrap-around
- removed warning not to build a switch from this specification

B.9 OpenFlow version 1.1

Release date : February 28, 2011
Wire Protocol : 0x02

B.9.1 Multiple Tables

- The switch now expose a pipeline with multiple tables.
- Flow entry have instruction to control pipeline processing
- Controller can choose packet traversal of tables via goto instruction
- Metadata field (64 bits) can be set and match in tables
- Packet actions can be merged in packet action set
- Packet action set is executed at the end of pipeline
- Packet actions can be applied between table stages
- Table miss can send to controller, continue to next table or drop
- Rudimentary table capability and configuration

B.9.2 Groups

- Group indirection to represent a set of ports
- Group table with 4 types of groups :
  - All - used for multicast and flooding
  - Select - used for multipath
  - Indirect - simple indirection
  - Fast Failover - use first live port
- Group action to direct a flow to a group

B.9.3 Tags : MPLS & VLAN

- Support for VLAN and QinQ, adding, modifying and removing VLAN headers
- Support for MPLS, adding, modifying and removing MPLS shim headers

B.9.4 Virtual ports

- Make port number 32 bits, enable larger number of ports
- Enable switch to provide virtual port as OpenFlow ports
- Augment packet-in to report both virtual and physical ports

B.9.5 Other changes

- Remove 802.1d-specific text from spec
- Remove Emergency Flow Cache from spec
- Cookie Enhancements Proposal
- Set_queue action (unbundled from output port)
- Maskable DL and NW address match fields
- Add TTL decrement, set and copy actions for IPv4 and MPLS
- SCTP header matching and rewriting support
- Set ECN action
- Connection interruption trigger fail secure or fail standalone mode
- Define message handling : no loss, may reorder if no barrier
- Rename `VENDOR` APIs to `EXPERIMENTER` APIs
- Many other bug fixes, rewording and clarifications

B.10 OpenFlow version 1.2

Release date : (Target) December, 2011
Wire Protocol : 0x03

Please refers to the bug tracking ID for more details on each change

B.10.1 Extensible match support

Prior versions of the OpenFlow specification used a static fixed length structure to specify `ofp_match`, which prevents flexible expression of matches and prevents inclusion of new match fields. The `ofp_match` has been changed to a TLV structure, called OpenFlow Extensible Match (OXM), which dramatically increases flexibility.

The match fields themselves have been reorganised. In the previous static structure, many fields were overloaded ; for example `tcp.src_port`, `udp.src_port`, and `icmp.code` were using the same field entry. Now, every logical field has its own unique type.

List of features for OpenFlow Extensible Match :

- Flexible and compact TLV structure called OXM (EXT-1)
- Enable flexible expression of match, and flexible bitmasking (EXT-1)
- Pre-requisite system to insure consistency of match (EXT-1)
- Give every match field a unique type, remove overloading (EXT-1)
- Modify VLAN matching to be more flexible (EXT-26)
- Add vendor classes and experimenter matches (EXT-42)
- Allow switches to override match requirements (EXT-56, EXT-33)

B.10.2 Extensible 'set_field' packet rewriting support

Prior versions of the OpenFlow specification were using hand-crafted actions to rewrite header fields. The Extensible `set_field` action reuses the OXM encoding defined for matches, and enables to rewrite any header field in a single action (EXT-13). This allows any new match field, including experimenter fields, to be available for rewrite. This makes the specification cleaner and eases cost of introducing new fields.

- Deprecate most header rewrite actions
- Introduce generic `set-field` action (EXT-13)
- Reuse match TLV structure (OXM) in `set-field` action

B.10.3 Extensible context expression in 'packet-in'

The `packet-in` message did include some of the packet context (ingress port), but not all (metadata), preventing the controller from figuring how match did happen in the table and which flow entries would match or not match. Rather than introduce a hard coded field in the `packet-in` message, the flexible OXM encoding is used to carry packet context.

- Reuse match TLV structure (OXM) to describe metadata in packet-in (EXT-6)
- Include the 'metadata' field in packet-in
- Move ingress port and physical port from static field to OXM encoding
- Allow to optionally include packet header fields in TLV structure

B.10.4 Extensible Error messages via experimenter error type

An experimenter error code has been added, enabling experimenter functionality to generate custom error messages (EXT-2). The format is identical to other experimenter APIs.

B.10.5 IPv6 support added

Basic support for IPv6 match and header rewrite has been added, via the Flexible match support.

- Added support for matching on IPv6 source address, destination address, protocol number, traffic class, ICMPv6 type, ICMPv6 code and IPv6 neighboor discovery header fields (EXT-1)
- Added support for matching on IPv6 flow label (EXT-36)

B.10.6 Simplified behaviour of flow-mod request

The behaviour of flow-mod request has been simplified (EXT-30).

- MODIFY and MODIFY_STRICT commands never insert new flows in the table
- New flag OFPFF_RESET_COUNTS to control counter reset

- Remove quirky behaviour for cookie field.

B.10.7 Removed packet parsing specification

The OpenFlow specification no longer attempts to define how to parse packets (EXT-3). The match fields are only defined logically.

- OpenFlow does not mandate how to parse packets
- Parsing consistency acheived via OXM pre-requisite

B.10.8 Controller role change mechanism

The controller role change mechanism is a simple mechanism to support multiple controllers for failover (EXT-39). This scheme is entirely driven by the controllers ; the switch only need to remember the role of each controller to help the controller election mechanism.

- Simple mechanism to support multiple controllers for failover
- Switches may now connect to multiple controllers in parallel
- Enable each controller to change its roles to equal, master or slave

B.10.9 Other changes

- Per-table metadata bitmask capabilities (EXT-34)
- Rudimentary group capabilities (EXT-61)
- Add hard timeout info in flow-removed messages (OFP-283)
- Add ability for controller to detect STP support(OFP-285)
- Turn off packet buffering with OFPCML_NO_BUFFER (EXT-45)
- Added ability to query all queues (EXT-15)
- Added experimenter queue property (EXT-16)
- Added max-rate queue property (EXT-21)
- Enable deleting flow in all tables (EXT-10)
- Enable switch to check chaining when deleting groups (EXT-12)
- Enable controller to disable buffering (EXT-45)
- Virtual ports renamed logical ports (EXT-78)
- New error messages (EXT-1, EXT-2, EXT-12, EXT-13, EXT-39, EXT-74 and EXT-82)
- Include release notes into the specification document
- Many other bug fixes, rewording and clarifications

Appendix C Credits

Spec contributions, in alphabetical order:

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