OpenFlow Switch Specification

Version 0.9.0 ( Wire Protocol 0x98 )

July 20, 2009

Current Maintainer: Brandon Heller (brandonh@stanford.edu)

1 Introduction

This document describes the requirements of an OpenFlow Switch. We recommend that you read the latest version of the OpenFlow whitepaper before reading this specification. The whitepaper is available on the OpenFlow Consortium website (http://OpenFlowSwitch.org). This specification covers the components and the basic functions of the switch, and the OpenFlow protocol to manage an OpenFlow switch from a remote controller.

Version 1.0 of this document will be the first for which official vendor support is expected. Versions before 1.0 will be marked “Draft”, and will include the header: “Do not build a switch from this specification!” We hope to generate feedback prior to Version 1.0 from switch designers and network researchers, so that the set of features defined in Version 1.0 enables production deployments on a variety of vendor hardware.

2 Switch Components

An OpenFlow Switch consists of a flow table, which performs packet lookup and forwarding, and a secure channel to an external controller (Figure[1]). The controller manages the switch over the secure channel using the OpenFlow protocol.

The flow table contains a set of flow entries (header values to match packets against), activity counters, and a set of zero or more actions to apply to matching packets. All packets processed by the switch are compared against the flow table. If a matching entry is found, any actions for that entry are performed on the packet (e.g., the action might be to forward a packet out a specified port). If no match is found, the packet is forwarded to the controller over the secure channel. The controller is responsible for determining how to handle packets without valid flow entries, and it manages the switch flow table by adding and removing flow entries.
3 Flow Table

This section describes the components of flow table entries and the process by which incoming packets are matched against flow table entries.

Table 1: A flow entry consists of header fields, counters, and actions.

Each flow table entry (see Table 1) contains:

- header fields to match against packets
- counters to update for matching packet
- actions to apply to matching packets

3.1 Header Fields

Table 2: Fields from packets used to match against flow entries.

Table 2 shows the header fields an incoming packet is compared against. Each entry contains a specific value, or ANY, which matches any value. If the
switch supports subnet masks on the IP source and/or destination fields, these can more precisely specify matches. The fields in the OpenFlow 11-tuple are listed in Table 2 and details on the properties of each field are described in Table 3.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>When applicable</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingress Port</td>
<td>(Implementation dependent)</td>
<td>All packets</td>
<td>Numerical representation of incoming port, starting at 1.</td>
</tr>
<tr>
<td>Ethernet source address</td>
<td>48</td>
<td>All packets on enabled ports</td>
<td></td>
</tr>
<tr>
<td>Ethernet destination address</td>
<td>48</td>
<td>All packets on enabled ports</td>
<td></td>
</tr>
<tr>
<td>Ethernet type</td>
<td>16</td>
<td>All packets on enabled ports</td>
<td>An OpenFlow switch is required to match the type in both standard Ethernet and 802.2 with a SNAP header and OUI of 0x0000000. The special value of 0x05FF is used to match all 802.3 packets without SNAP headers.</td>
</tr>
<tr>
<td>VLAN id</td>
<td>12</td>
<td>All packets of Ethernet type 0x8100</td>
<td></td>
</tr>
<tr>
<td>VLAN priority</td>
<td>3</td>
<td>All packets of Ethernet type 0x8100</td>
<td>VLAN PCP field</td>
</tr>
<tr>
<td>IP source address</td>
<td>32</td>
<td>All IP packets</td>
<td>Can be subnet masked</td>
</tr>
<tr>
<td>IP destination address</td>
<td>32</td>
<td>All IP packets</td>
<td>Can be subnet masked</td>
</tr>
<tr>
<td>IP protocol</td>
<td>8</td>
<td>All IP packets</td>
<td></td>
</tr>
<tr>
<td>Transport source port / ICMP Type</td>
<td>16</td>
<td>All TCP, UDP, and ICMP packets</td>
<td>Only lower 8 bits used for ICMP Type</td>
</tr>
<tr>
<td>Transport destination port / ICMP Code</td>
<td>16</td>
<td>All TCP, UDP, and ICMP packets</td>
<td>Only lower 8 bits used for ICMP Code</td>
</tr>
</tbody>
</table>

Table 3: Field lengths and the way they must be applied to flow entries.

Switch designers are free to implement the internals in any way convenient provided that correct functionality is preserved. For example, while a flow may have multiple forward actions, each specifying a different port, a switch designer may choose to implement this as a single bitmask within the hardware forwarding table.
3.2 Counters

Counters are maintained per-table, per-flow, and per-port. OpenFlow-compliant counters may be implemented in software and maintained by polling hardware counters with more limited ranges.

Table 4 contains the required set of counters. Duration refers to the number of seconds a flow has been active. The Receive Errors field includes all explicitly specified errors, including frame, overrun, and CRC errors, plus any others.

<table>
<thead>
<tr>
<th>Counter</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per Table</td>
<td></td>
</tr>
<tr>
<td>Active Entries</td>
<td>32</td>
</tr>
<tr>
<td>Packet Lookups</td>
<td>64</td>
</tr>
<tr>
<td>Packet Matches</td>
<td>64</td>
</tr>
<tr>
<td>Per Flow</td>
<td></td>
</tr>
<tr>
<td>Received Packets</td>
<td>64</td>
</tr>
<tr>
<td>Received Bytes</td>
<td>64</td>
</tr>
<tr>
<td>Duration</td>
<td>32</td>
</tr>
<tr>
<td>Per Port</td>
<td></td>
</tr>
<tr>
<td>Received Packets</td>
<td>64</td>
</tr>
<tr>
<td>Transmitted Packets</td>
<td>64</td>
</tr>
<tr>
<td>Received Bytes</td>
<td>64</td>
</tr>
<tr>
<td>Transmitted Bytes</td>
<td>64</td>
</tr>
<tr>
<td>Receive Drops</td>
<td>64</td>
</tr>
<tr>
<td>Transmit Drops</td>
<td>64</td>
</tr>
<tr>
<td>Receive Errors</td>
<td>64</td>
</tr>
<tr>
<td>Transmit Errors</td>
<td>64</td>
</tr>
<tr>
<td>Receive Frame Alignment Errors</td>
<td>64</td>
</tr>
<tr>
<td>Receive Overrun Errors</td>
<td>64</td>
</tr>
<tr>
<td>Receive CRC Errors</td>
<td>64</td>
</tr>
<tr>
<td>Collisions</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 4: Required list of counters for use in statistics messages.

3.3 Actions

Each flow entry is associated with zero or more actions that dictate how the switch handles matching packets. Actions need not be executed in the order in which they are specified in the flow entry. If no actions are present, the packet is dropped.

A switch is not required to support all action types — just those marked “Required Actions” below. When connecting to the controller, a switch indicates which of the “Optional Actions” it supports. OpenFlow-compliant switches come in two types: OpenFlow-only, and OpenFlow-enabled.

OpenFlow-only switches support only the required actions below, while OpenFlow-enabled switches, routers, and access points may also support the NORMAL action. Either type of switch can also support the FLOOD action.
**Required Action:** *Forward*. OpenFlow switches must support forwarding the packet to physical ports and the following virtual ones:

- **ALL**: Send the packet out all interfaces, not including the incoming interface.
- **CONTROLLER**: Encapsulate and send the packet to the controller.
- **LOCAL**: Send the packet to the switch's local networking stack.
- **TABLE**: Perform actions in flow table. Only for packet-out messages.
- **IN_PORT**: Send the packet out the input port.

**Optional Action:** *Forward*. The switch may optionally support the following virtual ports:

- **NORMAL**: Process the packet using the traditional forwarding path supported by the switch (i.e., traditional L2, VLAN, and L3 processing.) The switch may check the VLAN field to determine whether or not to forward the packet along the normal processing route. If the switch cannot forward entries for the OpenFlow-specific VLAN back to the normal processing route, it must indicate that it does not support this action.
- **FLOOD**: Flood the packet along the minimum spanning tree, not including the incoming interface.

Ideally, a switch will support flow-entries that can forward packets to any combination of the physical and virtual ports. For example, this could be expressed internally in the switch with a bitmap that includes all the physical and virtual ports.

However, some switches will not be able to support any combination. Therefore, the requirement is that the switch support sending to any combination of physical ports and the Controller virtual port simultaneously. All other combinations are desired, but optional.

The controller will only ask the switch to send to multiple physical ports simultaneously if the switch indicates it supports this behavior in the initial handshake (see section 5.3.1).

**Required Action:** *Drop*. A flow-entry with no specified action indicates that all matching packets should be dropped.

**Optional Action:** *Modify-Field*. While not strictly required, the actions shown in Table 5 greatly increase the usefulness of an OpenFlow implementation. To aid integration with existing networks, we suggest that VLAN modification actions be supported.
### Table 5: Field-modify actions.

<table>
<thead>
<tr>
<th>Action</th>
<th>Associated Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set VLAN ID</td>
<td>12 bits</td>
<td>If no VLAN is present, a new header is added with the specified VLAN ID and priority of zero. If a VLAN header already exists, the VLAN ID is replaced with the specified value.</td>
</tr>
<tr>
<td>Set VLAN priority</td>
<td>3 bits</td>
<td>If no VLAN is present, a new header is added with the specified priority and a VLAN ID of zero. If a VLAN header already exists, the priority field is replaced with the specified value.</td>
</tr>
<tr>
<td>Strip VLAN header</td>
<td>-</td>
<td>Strip VLAN header if present.</td>
</tr>
<tr>
<td>Modify Ethernet source MAC address</td>
<td>48 bits: Value with which to replace existing source MAC address</td>
<td>Replace the existing Ethernet source MAC address with the new value.</td>
</tr>
<tr>
<td>Modify Ethernet destination MAC address</td>
<td>48 bits: Value with which to replace existing destination MAC address</td>
<td>Replace the existing Ethernet destination MAC address with the new value.</td>
</tr>
<tr>
<td>Modify IPv4 source address</td>
<td>32 bits: Value with which to replace existing IPv4 source address</td>
<td>Replace the existing IP source address with new value and update the IP checksum (and TCP/UDP checksum if applicable). This action is only applicable to IPv4 packets.</td>
</tr>
<tr>
<td>Modify IPv4 destination address</td>
<td>32 bits: Value with which to replace existing IPv4 destination address</td>
<td>Replace the existing IP destination address with new value and update the IP checksum (and TCP/UDP checksum if applicable). This action is only applied to IPv4 packets.</td>
</tr>
<tr>
<td>Modify IPv4 ToS bits</td>
<td>6 bits: Value with which to replace existing IPv4 ToS field</td>
<td>Replace the existing IP ToS field. This action is only applied to IPv4 packets.</td>
</tr>
<tr>
<td>Modify transport source port</td>
<td>16 bits: Value with which to replace existing TCP or UDP source port</td>
<td>Replace the existing TCP/UDP source port with new value and update the TCP/UDP checksum. This action is only applicable to TCP and UDP packets.</td>
</tr>
<tr>
<td>Modify transport destination port</td>
<td>16 bits: Value with which to replace existing TCP or UDP destination port</td>
<td>Replace the existing TCP/UDP destination port with new value and update the TCP/UDP checksum. This action is only applied to TCP and UDP packets.</td>
</tr>
</tbody>
</table>
3.4 Matching

Figure 2: The functions performed on a packet as it moves through an OpenFlow switch. As discussed in Section 4.5, support for 802.1D is optional.

Figure 3: A flow table showing how a packet is matched against a flow entry.

On receipt of a packet, an OpenFlow Switch performs the functions shown in Figure 2.

The flow table is checked for a matching flow entry. The header fields used for the table lookup depend on the packet type as described below (and shown in Figure 3).

- Rules specifying an ingress port are matched against the physical port that received the packet.
- The Ethernet headers as specified in Table 2 are used for all packets.
• If the packet is a VLAN (Ethernet type 0x8100), the VLAN ID and PCP fields are used in the lookup.

• For IP packets (Ethernet type equal to 0x0800), the lookup fields also include those in the IP header.

• For IP packets that are TCP or UDP (IP protocol is equal to 6 or 17), the lookup includes the transport ports.

• For IP packets that are ICMP (IP protocol is equal to 1), the lookup includes the Type and Code fields.

• For IP packets with nonzero fragment offset or More Fragments bit set, the transport ports are set to zero for the lookup.

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

To handle the various Ethernet framing types, matching the Ethernet type is handled in a slightly different manner. If the packet is an Ethernet II frame, the Ethernet type is handled in the expected way. If the packet is an 802.3 frame with a SNAP header and Organizationally Unique Identifier (OUI) of 0x000000, the SNAP protocol id is matched against the flows Ethernet type. A flow entry that specifies an Ethernet type of 0x05FF, matches all Ethernet 802.2 frames without a SNAP header and those with SNAP headers that do not have an OUI of 0x000000.

Packets are matched against flow entries based on prioritization. An entry that specifies an exact match (i.e., it has no wildcards) is always the highest priority. All wildcard entries have a priority associated with them. Higher priority entries must match before lower priority ones. If multiple entries have the same priority, the switch is free to choose any ordering. Higher numbers have higher priorities.

For each packet that matches a flow entry, the associated counters for that entry are updated. If no matching entry can be found for a packet, the packet is sent to the controller over the secure channel.

4 Secure Channel

The secure 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 send packets out the switch.

Between the datapath and the secure channel, the interface is implementation-specific, however all secure channel messages must be formatted according to the OpenFlow protocol.
4.1 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.

4.1.1 Controller-to-Switch

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

**Features:** Upon SSL session establishment, the controller sends a features request message to the switch. The switch must reply with a features reply that specifies the capabilities supported by the switch.

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

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

**Read-State:** Read-State messages are used by the controller to collect statistics from the switches flow-tables, ports and the individual flow entries.

**Send-Packet:** These are used by the controller to send packets out of a specified port on the switch.

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

4.1.2 Asynchronous

Asynchronous messages are sent without solicitation from the switch to the controller and denote a change in switch or network state. The four main asynchronous message types are described below.

**Packet-in:** For all packets that do not have a matching flow entry, a packet-in event is sent to the controller (or if a packet matches an entry with a “send to controller” action). If the switch has sufficient memory to buffer packets that are sent to the controller, the packet-in events contain some fraction of the packet header (by default 128 bytes) and a buffer ID to be used by the
controller when it is ready for the switch to forward the packet. Switches that do not support internal buffering (or have run out of internal buffering) must send the full packet to the controller as part of the event.

**Flow-Removed:** When a flow entry is added to the switch by a flow modify message, an idle timeout value indicates when the entry should be removed due to a lack of activity, as well as a hard timeout value that indicates when the entry should be removed, regardless of activity. The flow modify message also specifies whether the switch should send a flow removed message to the controller when the flow expires. Delete messages may also cause flow removed messages.

**Port-status:** The switch is expected to send port-status messages to the controller as port configuration state changes. These events include change in port status (for example, if it was brought down directly by a user) or a change in port status as specified by 802.1D (see Section 4.5 for a description of 802.1D support requirements).

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

### 4.1.3 Symmetric

Symmetric messages are sent without solicitation, in either direction.

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

**Echo:** Echo request/reply messages can be sent from either the switch or the controller, and must return an echo reply. They can be used to indicate the latency, bandwidth, and/or liveness of a controller-switch connection.

**Vendor:** Vendor 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.

### 4.2 Connection Setup

The switch must be able to establish the communication at a user-configurable (but otherwise fixed) IP address, using a user-specified port. Traffic to and from the secure channel is not checked against the flow table. Therefore, the switch must identify incoming traffic as local before checking it against the flow table. Future versions of the protocol specification will describe a dynamic controller discovery protocol in which the IP address and port for communicating with the controller is determined at runtime.
When an OpenFlow connection is first established, each side of the connection must immediately send an `OFPT_HELLO` message with the `version` field set to the highest OpenFlow protocol version supported by the sender. Upon receipt of this message, the recipient may calculate the OpenFlow protocol version to be used as the smaller of the version number that it sent and the one that it received.

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

### 4.3 Connection Interruption

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

If some number of attempts to contact a controller (zero or more) fail, the switch must enter “emergency mode” and immediately reset the current TCP connection. In emergency mode, the matching process is dictated by the emergency flow table entries (those marked with the emergency bit when added to the switch). All normal entries are deleted when entering emergency mode.

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

The first time a switch starts up, it is considered to be in emergency mode. Configuration of the default set of flow entries is outside the scope of the OpenFlow protocol.

### 4.4 Encryption

The switch and controller communicate through an SSL connection. The SSL connection is initiated by the switch on startup to the controllers server, 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).
4.5 Spanning Tree

OpenFlow switches may optionally support 802.1D Spanning Tree Protocol. Those switches that do support it are expected to process all 802.1D packets locally before performing flow lookup. A switch that implements STP must set the `OFPC_STP` bit in the ‘capabilities’ field of its `OFPT_FEATURES_REPLY` message. A switch that implements STP must make it available on all of its physical ports, but it need not implement it on virtual ports (e.g. `OFPP_LOCAL`).

Port status, as specified by the spanning tree protocol, is then used to limit packets forwarded to the `OFP_FLOOD` port to only those ports along the spanning tree. Port changes as a result of the spanning tree are sent to the controller via port-update messages. Note that forward actions that specify the outgoing port or `OFP_ALL` ignore the port status set by the spanning tree protocol. Packets received on ports that are disabled by spanning tree must follow the normal flow table processing path.

Switches that do not support 802.1D spanning tree must allow the controller to specify the port status for packet flooding through the port-mod messages.

4.6 Flow Table Modification Messages

Flow table modification messages can have the following types:

```c
enum ofp_flow_mod_command {
    OFPPC_ADD,         /* New flow. */
    OFPPC_MODIFY,      /* Modify all matching flows. */
    OFPPC_MODIFY_STRICT, /* Modify entry strictly matching wildcards */
    OFPPC_DELETE,      /* Delete all matching flows. */
    OFPPC_DELETE_STRICT /* Strictly match wildcards and priority. */
};
```

For ADD requests with the `OFPFF_CHECK_OVERLAP` flag set, the switch must first check for any overlapping flow entries. An overlapping entry is one with same priority as the request and at least one wildcard field, such that some flow entry exists that could match both. If an overlap conflict is found, the switch must refuse the addition and respond with an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_OVERLAP` code.

For valid (non-overlapping) ADD requests, or those with no overlap checking, the switch must insert the flow entry at the lowest numbered table for which the switch supports all wildcards set in the `flow_match` struct, and for which the priority would be observed during the matching process. If a flow entry with identical header fields and priority already resides in any table, then that entry, including its counters, must be removed, and the new flow entry added.

If a switch cannot find any 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_ALL_TABLES_FULL` code.
For **MODIFY** requests, if a flow entry with identical header fields does not current reside in any table, the **MODIFY** acts like an **ADD**, and the new flow entry must be inserted with zeroed counters. Otherwise, the actions field is changed on the existing entry and its counters and idle time fields are left unchanged.

For **DELETE** requests, if no flow entry matches, no error is recorded, and no flow table modification occurs. If flow entries match, and must be deleted, then each normal entry with the `OFPFF_SEND_FLOW_REM` flag set should generate a flow removed message. Deleted emergency flow entries generate no flow removed messages.

**MODIFY** and **DELETE** flow mod commands have corresponding _STRICT versions. Without _STRICT appended, the wildcards are active and all flows that match the description are modified or removed. If _STRICT is appended, all fields, including the wildcards and priority, are strictly matched against the entry, and only an identical flow is modified or removed. For example, if a message to remove entries is sent that has all the wildcard flags set, the **DELETE** command would delete all flows from all tables, while the **DELETE_STRICT** command would only delete a rule that applies to all packets at the specified priority.

For non-strict **MODIFY** and **DELETE** commands that contain wildcards, a match will occur when a flow entry exactly matches or is more specific than the description in the flow mod command. For example, if a **DELETE** command says to delete all flows with a destination port of 80, then a flow entry that is all wildcards will not be deleted. However, a **DELETE** command that is all wildcards will delete an entry that matches all port 80 traffic. This same interpretation of mixed wildcard and exact header fields also applies to individual and aggregate flows stats.

**DELETE** and **DELETE_STRICT** commands can be optionally filtered by output port. If the `out_port` field 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. This field is ignored by **ADD**, **MODIFY**, and **MODIFY_STRICT** messages.

Emergency flow mod messages must have timeout values set to zero. Otherwise, the switch must refuse the addition and respond with an `ofp.error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_BAD_EMERG_TIMEOUT` code.

### 4.7 Flow Removal

Each flow entry has an **idle_timeout** and a **hard_timeout** associated with it. If no packet has matched the rule in the last **idle_timeout** seconds, or it has been
hard_timeout seconds since the flow was inserted, the switch removes the entry and sends a flow removed message. In addition, the controller is able to actively remove entries by sending a flow message with the DELETE or DELETE_STRICT command. Like the message used to add the entry, a removal message contains a description, which may include wild cards.

5 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 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.

5.1 OpenFlow Header

Each OpenFlow message begins with the OpenFlow header:

```c
/* 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 0x98. The final version for a Type 0 switch will be 0x00. 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:

```c
enum ofp_type {
    /* Immutable messages. */
    OFPT_HELLO, /* Symmetric message */
    OFPT_ERROR, /* Symmetric message */
    OFPT_ECHO_REQUEST, /* Symmetric message */
    OFPT_ECHO_REPLY, /* Symmetric message */
    OFPT_VENDOR, /* Symmetric message */

    /* Switch configuration messages. */
    OFPT_FEATURES_REQUEST, /* Controller/switch message */

    OFPT_PACKET_IN, /* Switch with packet */
    OFPT_FLOW_REMOVED, /* Switch removed entry */
    OFPT_PORT_STATUS, /* Switch port status */
    OFPT_PACKET_OUT, /* Switch send packet */
    OFPT_FLOW_MOD, /* Switch modify entry */
    OFPT_PACKET_OUT_REPLY, /* Switch reply to PACKET_OUT */
    OFPT_FLOW_MOD_REPLY, /* Switch reply to PACKET_IN */
    OFPT_SET_CONFIG, /* Switch set configuration */
    OFPT_GET_CONFIG_REQUEST, /* Switch get configuration */
    OFPT_GET_CONFIG_REPLY, /* Switch get configuration */
    OFPT_barrier,

    /* Extensions. */
    OFPT_VENDOR_1, /* Vendor 1 */
    OFPT_VENDOR_2, /* Vendor 2 */
    OFPT_VENDOR_3, /* Vendor 3 */
    OFPT_VENDOR_4, /* Vendor 4 */
    OFPT_VENDOR_5, /* Vendor 5 */
    OFPT_VENDOR_6, /* Vendor 6 */
    OFPT_VENDOR_7, /* Vendor 7 */
    OFPT_VENDOR_8, /* Vendor 8 */

    OFPT_EXPERIMENTAL, /* Experimental messages */

    MAX_OFPT
};
OFP_ASSERT(sizeof(struct ofp_type) == 2);
```

```c
/* Header on all OpenFlow packets. */
```
5.2 Common Structures

This section describes structures used by multiple messages.

5.2.1 Port Structures

Physical ports are described with the following structure:

```c
/* Description of a physical port */
struct ofp_phy_port {
    uint16_t port_no;
    uint8_t hw_addr[OFP_ETH_ALEN];
    uint8_t 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. */
};
OFP_ASSERT(sizeof(struct ofp_phy_port) == 48);
```

The `port_no` field is a value the datapath associates with a physical port. 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 spanning tree and administrative settings with the following structure:

```c
/* Flags to indicate behavior of the physical port. These flags are
 * used in ofp_phy_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_STP = 1 << 1, /* Disable 802.1D spanning tree on port. */
    OFPPC_NO_RECV = 1 << 2, /* Drop all packets except 802.1D spanning
               tree packets. */
    OFPPC_NO_RECV_STP = 1 << 3, /* Drop received 802.1D STP packets. */
    OFPPC_NO_FWD = 1 << 4, /* Do not include this port when flooding. */
    OFPPC_NO_PACKET_IN = 1 << 6 /* Do not send packet-in msgs for port. */
};
```

The port config bits indicate whether a port has been administratively brought down, options for handling 802.1D spanning tree packets, and how to handle incoming and outgoing packets. These bits, configured over multiple switches, enable an OpenFlow network to safely flood packets along either a custom or 802.1D spanning tree.

The controller may set `OFPPF_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 `OFPPF_NO_STP` is 0, STP controls the `OFPPF_NO_FWD` and `OFPPF_STP_*` bits directly. `OFPPF_NO_FWD` is set to 0 when the STP port state is Forward- ing, otherwise to 1. The bits in `OFPPF_STP_MASK` are set to one of the other `OFPPF_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 `OFPPF_NO_RECV`, `OFPPF_NO_RECV_STP`, `OFPPF_NO_FWD`, and `OFPPF_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.

The `state` field describes the spanning tree state and whether a physical link is present, with the following structure:

```c
/* 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. */

    /* The OFPPS_STP_* bits have no effect on switch operation. */
}```
* controller must adjust OFPPC_NO_RECV, OFPPC_NO_FWD, and
  * OFPPC_NO_PACKET_IN appropriately to fully implement an 802.1D spanning
  * tree. */

OFPPS_STP_LISTEN = 0 << 8, /* Not learning or relaying frames. */
OFPPS_STP猢EARN = 1 << 8, /* Learning but not relaying frames. */
OFPPS_STP_FORWARD = 2 << 8, /* Learning and relaying frames. */
OFPPS_STP_BLOCK = 3 << 8, /* Not part of spanning tree. */
OFPPS_STP_MASK = 3 << 8 /* Bit mask for OFPPS_STP_* values. */

};

All port state bits are read-only, representing spanning tree and physical link
state.

The port numbers use the following conventions:

/* Port numbering. Physical ports are numbered starting from 1. */
enum ofp_port {
  /* Maximum number of physical switch ports. */
  OFPP_MAX = 0xff00,

  /* Fake output "ports". */
  OFPP_IN_PORT = 0xfff8, /* Send the packet out the input port. This
  virtual port must be explicitly used
  in order to send back out of the input
  port. */

  OFPP_TABLE = 0xffff, /* Perform actions in flow table.
  NB: This can only be the destination
  port for packet-out messages. */

  OFPP_NORMAL = 0xfffe, /* Process with normal L2/L3 switching. */

  OFPP_FLOOD = 0xfffb, /* All physical ports except input port and
  those disabled by STP. */

  OFPP_ALL = 0xfffc, /* All physical ports except input port. */

  OFPP_CONTROLLER = 0xfffd, /* Send to controller. */

  OFPP_LOCAL = 0xfffe, /* Local openflow "port". */

  OFPP_NONE = 0xffff /* Not associated with a physical port. */

};

The curr, advertised, supported, and peer fields indicate link modes (10M
to 10G full and half-duplex), link type (copper/fiber) and link features (autone-
gotiation and pause). Port features are represent by the following structure:

/* Features of physical 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_COPPER = 1 << 7, /* Copper medium. */
  OFPPF_FIBER = 1 << 8, /* Fiber medium. */
  OFPPF_AUTONEG = 1 << 9, /* Auto-negotiation. */
  OFPPF_PAUSE = 1 << 10, /* Pause. */
  OFPPF_PAUSEASYM = 1 << 11 /* Asymmetric pause. */

};

Multiple of these flags may be set simultaneously.
5.2.2  Flow Match Structures

When describing a flow entry, the following structure is used:

```c
/* Fields to match against flows */
struct ofp_match {
    /* Wildcard fields. */
    uint32_t wildcards;
    /* Input switch port. */
    uint16_t in_port;
    /* Ethernet source address. */
    uint8_t dl_src[OFP_ETH_ALEN];
    /* Ethernet destination address. */
    uint8_t dl_dst[OFP_ETH_ALEN];
    /* Input VLAN id. */
    uint16_t dl_vlan;
    /* Input VLAN priority. */
    uint8_t dl_vlan_pcp;
    /* Align to 64-bits */
    uint8_t pad1[1];
    /* Ethernet frame type. */
    uint16_t dl_type;
    /* IP protocol. */
    uint8_t nw_proto;
    /* Align to 64-bits */
    uint8_t pad2[1];
    /* IP source address. */
    uint32_t nw_src;
    /* IP destination address. */
    uint32_t nw_dst;
    /* TCP/UDP source port. */
    uint16_t tp_src;
    /* TCP/UDP destination port. */
    uint16_t tp_dst;
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

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

```c
/* Flow wildcards. */
enum ofp_flow_wildcards {
    OFPFW_IN_PORT = 1 << 0, /* Switch input port. */
    OFPFW_DL_VLAN = 1 << 1, /* VLAN id. */
    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. */
    OFPFW_IN_PORT = OFPFW_DL_VLAN = OFPFW_DL_SRC = OFPFW_DL_DST = OFPFW_DL_TYPE = OFPFW_NW_PROTO = OFPFW_TP_SRC = OFPFW_TP_DST = (1 << 21) - 1 /* VLAN priority. */
};
```

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

```c
/* Flow wildcards. */
enum ofp_flow_wildcards {
    OFPFW_IN_PORT = 1 << 0, /* Switch input port. */
    OFPFW_DL_VLAN = 1 << 1, /* VLAN id. */
    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. */
    OFPFW_IN_PORT = OFPFW_DL_VLAN = OFPFW_DL_SRC = OFPFW_DL_DST = OFPFW_DL_TYPE = OFPFW_NW_PROTO = OFPFW_TP_SRC = OFPFW_TP_DST = (1 << 21) - 1 /* VLAN priority. */
};
```
If no wildcards are set, then the `ofp_match` exactly describes a flow, over the entire OpenFlow 11-tuple. On the other extreme, if all the wildcard flags are set, then every flow will match.

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”.

### 5.2.3 Flow Action Structures

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

```c
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_NW_TOS,    /* IP ToS/DSCP field (6 bits). */
    OFPAT_SET_TP_SRC,    /* TCP/UDP source port. */
    OFPAT_SET_TP_DST,    /* TCP/UDP destination port. */
    OFPAT_VENDOR = 0xffff
};
```

Output actions are described in Section 3.3, while Field-Modify actions are described in Table 5. An action definition contains the action type, length, and any associated data:

```c
/* 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];
};
OFPP_ASSERT(sizeof(struct ofp_action_header) == 8);
```

An `action_output` has the following fields:

```c
/* 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. */
struct ofp_action_output {
    uint16_t type;    /* OFPAT_OUTPUT. */
    uint16_t len;     /* Length is 8. */
    uint16_t port;    /* Output port. */
    uint16_t max_len; /* Max length to send to controller. */
};
OFP_ASSERT(sizeof(struct ofp_action_output) == 8);

The `max_len` indicates the maximum amount of data from a packet that should be sent when the port is OFPP_CONTROLLER. If `max_len` is zero, the switch must send a zero-size `packet_in` message. The `port` specifies the physical port from which packets should be sent.

An `action_vlan_vid` has the following fields:

/* Action structure for OFPAT_SET_VLAN_VID. */
struct ofp_action_vlan_vid {
    uint16_t type;    /* OFPAT_SET_VLAN_VID. */
    uint16_t len;     /* Length is 8. */
    uint16_t vlan_vid; /* VLAN id. */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_vlan_vid) == 8);

The `vlan_vid` field is 16 bits long, when an actual VLAN id is only 12 bits. The value `0xffff` is used to indicate that no VLAN id was set.

An `action_vlan_pcp` has the following fields:

/* Action structure for OFPAT_SET_VLAN_PCP. */
struct ofp_action_vlan_pcp {
    uint16_t type;    /* OFPAT_SET_VLAN_PCP. */
    uint16_t len;     /* Length is 8. */
    uint8_t vlan_pcp; /* VLAN priority. */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_vlan_pcp) == 8);

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

An `action_strip_vlan` takes no arguments and consists only of a generic `ofp_action_header`. This action strips the VLAN tag if one is present.

An `action_dl_addr` has the following fields:

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

An `action_nw_addr` has the following fields:

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

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

An `action_nw_tos` has the following fields:

```c
/* Action structure for OFPAT_SET_NW_TOS. */
struct ofp_action_nw_tos {
    uint16_t type;    /* OFPAT_SET_TW_SRC/DST. */
    uint16_t len;     /* Length is 8. */
    uint8_t nw_tos;   /* IP ToS/DSCP (6 bits). */
    uint8_t pad[3];  /* 6 upper bits of the ToS field to set, in the original bit positions (shifted to the left by 2). */
};
OFP_ASSERT(sizeof(struct ofp_action_nw_tos) == 8);
```

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

An `action_tp_port` has the following fields:

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

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

An `action_vendor` has the following fields:

```c
/* Action header for OFPAT_VENDOR. The rest of the body is vendor-defined. */
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_header". */
};
OFP_ASSERT(sizeof(struct ofp_action_vendor_header) == 8);
```

The `vendor` field is the Vendor ID, which takes the same form as in `ofp_vendor`. 
5.3 Controller-to-Switch Messages

5.3.1 Handshake

Upon SSL 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:

```c
/* 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 actions; /* Bitmap of supported "ofp_action_type"s. */
    /* Port info. */
    struct ofp_phy_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_tables` field describes the number of tables supported by the switch, each of which can have a different set of supported wildcard bits and number of entries. When the controller and switch first communicate, the controller will find out how many tables the switch supports from the Features Reply. If it wishes to understand the size, types, and order in which tables are consulted, the controller sends a OFPST_TABLE stats request. A switch must return these tables in the order the packets traverse the tables, with all exact-match tables listed before all tables with wildcards.

The `capabilities` field uses the following flags:

```c
/* 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. */
};
```
DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

OFPC_STP = 1 << 3, /* 802.1d spanning tree. */
OFPC_MULTI_PHY_TX = 1 << 4, /* Supports transmitting through multiple
physical interfaces */
OFPC_IP_REASM = 1 << 5 /* Can reassemble IP fragments. */

The actions field is a bitmap of supported actions on the hardware. It uses
the values from ofp_action_type as the number of bits to shift left for an asso-
ciated action. For example, OFPAT_SET_DL_VLAN would use the flag 0x00000002.

The ports field is an array of ofp_phy_port structures that describe all the
physical ports in the system that support OpenFlow. The number of port ele-
ments is inferred from the length field in the OpenFlow header.

5.3.2 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, respec-
tively. 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. */
};
OFF_ASSERT(sizeof(struct ofp_switch_config) == 12);

The configuration flags include the following:

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

The OFPC_FRAG_ flags indicate whether IP fragments should be treated nor-
manly, dropped, or reassembled. “Normal” handling of fragments means that an
attempt should be made to pass the fragments through the OpenFlow tables.
If any field is not present (e.g., the TCP/UDP ports didn’t fit), then the packet
should not match any entry that has that field set.

The miss_send_len field defines the number of bytes of each packet sent to
the controller as a result of both flow table misses and flow table hits with
the controller as the destination. If this field equals 0, the switch must send a
zero-size packet_in message.
DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

5.3.3 Modify State Messages

Modify Flow Entry Message  Modifications to the flow table from the controller are done with the OFPT_FLOW_MOD message:

```c
/* Flow setup and teardown (controller -> datapath). */
struct ofp_flow_mod {
    struct ofp_header header; /* Fields to match */
    struct ofp_match 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). */
    uint16_t out_port; /* For OFPFC_DELETE* commands, require matching entries to include this as an output port. A value of OFPP_NONE indicates no restriction. */
    uint16_t flags; /* One of OFPFF_* */
    uint32_t reserved; /* Reserved for future use. */
    struct ofp_action_header actions[0]; /* The action length is inferred from the length field in the header. */

};
```

OFP_ASSERT(sizeof(struct ofp_flow_mod) == 68);

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

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

The differences between OFPFC_MODIFY and OFPFC_MODIFY_STRICT are explained in Section 4.6. Differences between OFPFC_DELETE and OFPFC_DELETE_STRICT are explained in Section 4.6.

The `idle_timeout` and `hard_timeout` fields control how quickly flows expire.

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 field is only relevant for flow entries with wildcard fields. The priority field indicates table priority, where higher numbers are higher priorities; the switch must keep the highest-priority wildcard entries in the lowest-numbered (fastest) wildcard table, to ensure correctness. It is the responsibility of each switch implementer to ensure that exact entries always match before wildcards entries, regardless of the table configuration.

The buffer_id refers to a buffered packet sent by the OFPT_PACKET_IN message.

The out_port field optionally filters the scope of DELETE and DELETE_STRICT messages by output port. If 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 disable output port filtering, out_port must be set to OFPP_NONE, since 0 is a valid port id. This field is ignored by ADD, MODIFY, and MODIFY_STRICT messages.

The flags field may include the follow flags:

```c
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_EMERG = 1 << 2 /* Remark this is for emergency. */
};
```

When the OFPFF_SEND_FLOW_REM flag is set, the switch must send a flow removed message when the flow expires. The default is for the switch to not send flow removed messages for newly added flows.

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

When the OFPFF_EMERG flag is set, the switch must consider this flow entry as an emergency entry, and only use it for forwarding when disconnected from the controller.

**Port Modification Message**  The controller uses the OFPT_PORT_MOD message to modify the behavior of the physical port:

```c
/* Modify behavior of the physical port */
struct ofp_port_mod {
  struct ofp_header header;
```
uint16_t port_no;
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_phy_port struct. */

uint32_t config; /* Bitmap of OFPPC_* flags. */
uint32_t mask; /* Bitmap of OFPPC_* flags to be changed. */
uint32_t advertise; /* Bitmap of "ofp_port_features"s. Zero all bits to prevent any action taking place. */
uint8_t pad[4]; /* Pad to 64-bits. */

OFP_ASSERT(sizeof(struct ofp_port_mod) == 32);

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.

5.3.4 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 body[0]; /* Body of the request. */
};
```

OFP_ASSERT(sizeof(struct ofp_stats_request) == 12);

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 body[0]; /* Body of the reply. */
};
```

OFP_ASSERT(sizeof(struct ofp_stats_reply) == 12);

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.
```
DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

• The reply body is struct ofp_desc_stats. */
OFPST_DESC,

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

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

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

/* Physical port statistics.
• The request body is empty.
• The reply body is an array of struct ofp_port_stats. */
OFPST_PORT,

/* Vendor extension.
• The request and reply bodies begin with a 32-bit vendor ID, which takes
  the same form as in "struct ofp_vendor_header". The request and reply
  bodies are otherwise vendor-defined. */
OFPST_VENDOR = 0xffff

};

Description Statistics Information about the switch manufacturer, hardware revision, software revision, and serial number 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. */
};
OFP_ASSERT(sizeof(struct ofp_desc_stats) == 800);

Each entry is ASCII formatted and padded on the right with 0 bytes. DESC_STR_LEN is 256 and SERIAL_NUM_LEN is 32.

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 {
    struct ofp_match match; /* Fields to match. */
    uint8_t table_id; /* ID of table to read (from ofp_table_stats).
                        0xff for all tables or 0xfe for emergency. */

};
The `match` field contains a description of the flows that should be matched and may contain wildcards. This field’s matching behavior is described in Section 4.6.

The `table_id` field indicates the index of a single table to read, or `0xff` for all tables.

The `out_port` field optionally filters by output port. If `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 are still used; this is purely an additional constraint. Note that to disable output port filtering, `out_port` must be set to `OFPP_NONE`, since `0` is a valid port id.

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

```c
/* 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;
    struct ofp_match match;  /* Description of fields. */
    uint32_t duration;  /* Time flow has been alive in seconds. */
    uint16_t priority;  /* Priority of the entry. Only meaningful when this is not an exact-match entry. */
    uint16_t idle_timeout;  /* Number of seconds idle before expiration. */
    uint16_t hard_timeout;  /* Number of seconds before expiration. */
    uint16_t pad2;  /* Pad to 64 bits. */
    uint64_t packet_count;  /* Number of packets in flow. */
    uint64_t byte_count;  /* Number of bytes in flow. */
    struct ofp_action_header actions[0];  /* Actions. */
};
```

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

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

```c
/* Body for ofp_stats_request of type OFPST_AGGREGATE. */
struct ofp_aggregate_stats_request {
    struct ofp_match match;  /* Fields to match. */
    uint8_t table_id;  /* ID of table to read (from ofp_table_stats) */
    0xff for all tables or 0xfe for emergency. */
```
uint8_t pad; /* Align to 32 bits. */
uint16_t out_port; /* Require matching entries to include this
as an output port. A value of OFPP_NONE
indicates no restriction. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_request) == 44);

The **match** field contains a description of the flows that should be matched and may contain wildcards. This field’s matching behavior is described in Section 4.3.

The **table_id** field indicates the index of a single table to read, or 0xff for all tables.

The **out_port** field optionally filters by output port. If **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 are still used; this is purely an additional constraint. Note that to disable output port filtering, **out_port** must be set to OFPP_NONE, since 0 is a valid port id.

The body of the reply consists of the following:

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

**Table Statistics** Information about tables is requested with the **OFPST_TABLE** stats request type. The request does not contain any data in the body.

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

```c
/* 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[3]; /* Align to 32-bits. */
    char name[OFP_MAX_TABLE_NAME_LEN];
    uint32_t wildcards; /* Bitmap of OFPFW_* wildcards that are
        supported by the table. */
    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) == 64);
```
The **body** contains a **wildcards** field, which indicates the fields for which that particular table supports wildcarding. For example, a direct look-up hash table would have that field set to zero, while a sequentially searched table would have it set to `OFPFW_ALL`. The entries are returned in the order that packets traverse the tables.

`OFP_MAX_TABLE_NAME_LEN` is 32.

**Port Statistics** Information about physical ports is requested with the `OFPST_PORT` 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:

```c
/* 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 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);
```

The switch should return a value of -1 for unavailable counters.

**Vendor Statistics** Vendor-specific stats messages are requested with the `OFPST_VENDOR` stats type. The first four bytes of the message are the vendor identifier. The rest of the body is vendor-defined.

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.
5.3.5 Send Packet Message

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

```c
/* Send packet (controller -> datapath). */
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 actions_len; /* Size of action array in bytes. */
    struct ofp_action_header 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.) */
};
OFP_ASSERT(sizeof(struct ofp_packet_out) == 16);
```

The `buffer_id` is the same given in the `ofp_packet_in` message. If the `buffer_id` is -1, then the packet data is included in the data array. If OFPP_TABLE is specified as the output port of an action, the `in_port` in the `packet_out` message is used in the flow table lookup.

5.3.6 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 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.

5.4 Asynchronous Messages

5.4.1 Packet-In Message

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

```c
/* 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. */
    uint16_t in_port; /* Port on which frame was received. */
    uint8_t reason; /* Reason packet is being sent (one of OFPR_*) */
    uint8_t pad;
    uint8_t data[0]; /* Ethernet frame, halfway through 32-bit word, so the IP header is 32-bit aligned. The amount of data is inferred from the length field in the header. Because of padding, offsetof(struct ofp_packet_in, data) == sizeof(struct ofp_packet_in) - 2. */
};
```
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 action_output of the flow setup request are sent. If the packet is sent because of a flow table miss, then at least miss_send_len bytes from the OFPT_SET_CONFIG message are sent. The default miss_send_len is 128 bytes. If the packet is not buffered, the entire packet is included in the data portion, and the buffer_id is -1.

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

The reason field can be any of these values:

```c
enum ofp_packet_in_reason {
    OFPR_NO_MATCH, /* No matching flow. */
    OFPR_ACTION /* Action explicitly output to controller. */
};
```

### 5.4.2 Flow Removed Message

If the controller has requested to be notified when flows time out, the datapath does this with the OFPT_FLOW_REMOVED message:

```c
/* Flow removed (datapath -> controller). */
struct ofp_flow_removed {
    struct ofp_header header;
    struct ofp_match match; /* Description of fields. */
    uint16_t priority; /* Priority level of flow entry. */
    uint8_t reason; /* One of OFPRR_*. */
    uint8_t pad[1]; /* Align to 32-bits. */
    uint32_t duration; /* Time flow was alive in seconds. */
    uint16_t idle_timeout; /* Idle timeout from original flow mod. */
    uint8_t pad2[6]; /* Align to 64-bits. */
    uint64_t packet_count;
    uint64_t byte_count;
};
OFP_ASSERT(sizeof(struct ofp_flow_removed) == 80);
```

The match and priority fields are the same as those used in the flow setup request.
The `reason` field is one of the following:

```c
enum ofp_flow_removed_reason {
    OFPRR_IDLE_TIMEOUT,  /* Flow idle time exceeded idle_timeout. */
    OFPRR_HARD_TIMEOUT,  /* Time exceeded hard_timeout. */
    OFPRR_DELETE         /* Evicted by a DELETE flow mod. */
};
```

The `duration` field indicates the number of seconds the flow was alive (present in the flow table).

The `idle_timeout` field is directly copied from the flow mod that created this entry.

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

The `packet_count` and `byte_count` indicate the number of packets and bytes that were associated with this flow, respectively.

### 5.4.3 Port Status Message

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

```c
struct ofp_port_status {
    struct ofp_header header;
    uint8_t reason;   /* One of OFPPR_. */
    uint8_t pad[7];   /* Align to 64-bits. */
    struct ofp_phy_port desc;
};
```

The status can be one of the following values:

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

### 5.4.4 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:

```c
struct ofp_error_msg {
    /* OFPT_ERROR: Error message (datapath -> controller). */
};
```
DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

```c
struct ofp_header header;
  uint16_t type;
  uint16_t code;
  uint8_t data[0];    /* Variable-length data. Interpreted based
                     on the type and code. */
};
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`; in most cases this is the message that caused the problem.

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

Currently defined error types are:

```c
/* Values for 'type' in ofp_error_message. These values are immutable: they
 * will not change in future versions of the protocol (although new values may
 * be added). */
enum ofp_error_type {
  OFPET_HELLO_FAILED, /* Hello protocol failed. */
  OFPET_BAD_REQUEST,  /* Request was not understood. */
  OFPET_BAD_ACTION,   /* Error in action description. */
  OFPET_FLOW_MOD_FAILED /* Problem modifying flow entry. */
};
```

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

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

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

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

```c
/* 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 {
  OFPBRPC_BAD_VERSION,  /* ofp_header.version not supported. */
  OFPBRPC_BAD_TYPE,     /* ofp_header.type not supported. */
  OFPBRPC_BAD_STAT,     /* ofp_stats_request.type not supported. */
  OFPBRPC_BAD_VENDOR,   /* Vendor not supported (in ofp_vendor_header
                           * or ofp_stats_request or ofp_stats_reply). */
  OFPBRPC_BAD_SUBTYPE,  /* Vendor subtype not supported. */
  OFPBRPC_EPERM         /* Permissions error. */
};
```
The data field contains at least 64 bytes of the failed request.

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

```c
/* ofp_error_msg 'code' values for OFPET_BAD_ACTION. 'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_action_code {
    OFPBAC_BAD_TYPE, /* Unknown action type. */
    OFPBAC_BAD_LEN, /* Length problem in actions. */
    OFPBAC_BAD_VENDOR, /* Unknown vendor id specified. */
    OFPBAC_BAD_VENDOR_TYPE, /* Unknown action type for vendor id. */
    OFPBAC_BAD_OUT_PORT, /* Problem validating output action. */
    OFPBAC_BAD_ARGUMENT, /* Bad action argument. */
    OFPBAC_EPERM /* Permissions error. */
};
```

The data field contains at least 64 bytes of the failed request.

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

```c
/* 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_ALL_TABLES_FULL, /* Flow not added because of full tables. */
    OFPFMFC_OVERLAP, /* Attempted to add overlapping flow with
          * CHECK_OVERLAP flag set. */
    OFPFMFC_EPERM, /* Permissions error. */
    OFPFMFC_BAD_EMERG_TIMEOUT /* Flow not added because of non-zero idle/hard
          * timeout. */
};
```

The data field contains at least 64 bytes of the failed request.

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

### 5.5 Symmetric Messages

#### 5.5.1 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.

#### 5.5.2 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.
5.5.3 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.

5.5.4 Vendor

The Vendor message is defined as follows:

```c
/* Vendor extension. */
struct ofp_vendor_header {
    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. */
};
OFP_ASSERT(sizeof(struct ofp_vendor_header) == 12);
```

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.

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