OPEN NETWORKING
FOUNDATION

OpenFlow Switch Specification

Version 1.0.0 (Wire Protocol 0x01)
December 31, 2009

ONF TS-001

OpenFlow™

ONF Document Type: OpenFlow Spec
ONF Document Name: openflow-spec-v1.0.0

Disclaimer

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Without limitation, ONF disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and ONF disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

No license, express or implied, by estoppel or otherwise, to any Open Networking Foundation or Open Networking Foundation member intellectual property rights is granted herein.

Except that a license is hereby granted by ONF to copy and reproduce this specification for internal use only.

Contact the Open Networking Foundation at https://www.opennetworking.org for information on specification licensing through membership agreements.

Any marks and brands contained herein are the property of their respective owners.

WITHOUT LIMITING THE DISCLAIMER ABOVE, THIS SPECIFICATION OF THE OPEN NETWORKING FOUNDATION ("ONF") IS SUBJECT TO THE ROYALTY FREE, REASONABLE AND NONDISCRIMINATORY ("RANDZ") LICENSING COMMITMENTS OF THE MEMBERS OF ONF PURSUANT TO THE ONF INTELLECTUAL PROPERTY RIGHTS POLICY. ONF DOES NOT WARRANT THAT ALL NECESSARY CLAIMS OF PATENT WHICH MAY BE IMPLICATED BY THE IMPLEMENTATION OF THIS SPECIFICATION ARE OWNED OR LICENSABLE BY ONF'S MEMBERS AND THEREFORE SUBJECT TO THE RANDZ COMMITMENT OF THE MEMBERS.

OpenFlow Switch Specification

Version 1.0.0 (Wire Protocol 0x01)

December 31, 2009

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.

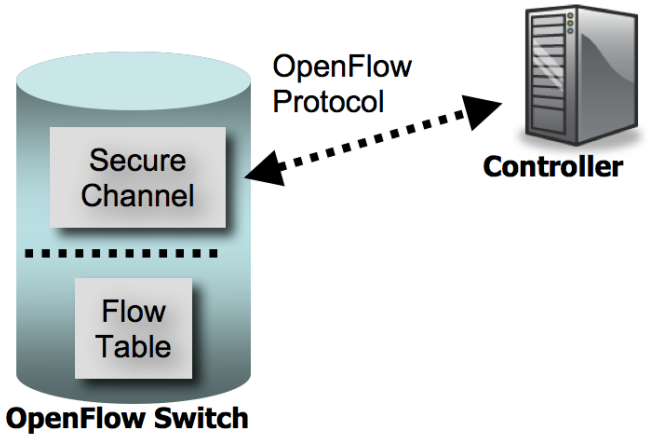

Figure 1: An OpenFlow switch communicates with a controller over a secure connection using the OpenFlow protocol.

DELL - Exhibit 1009 - Page 3

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.

Flow entries may forward packets to one or more OpenFlow ports. In general, these are physical ports, but the protocol does not preclude abstractions like port aggregations or VLAN traffic on a port appearing as an OpenFlow port. OpenFlow ports have limited state such as "up", "down" and whether spanning tree flood packets should be forwarded out the port. Additional configuration of ports may handled by the OpenFlow configuration protocol. There are several OpenFlow virtual ports used to indicate, for example, flooding or the ingress port (see 3.3).

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.

Header Fields	Counters	Actions

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 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

DELL - Exhibit 1009 - Page 4

more precisely specify matches. The fields in the OpenFlow 12-tuple are listed in Table 2 and details on the properties of each field are described in Table 3.

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, per-port and per queue. 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 time the flow has been installed in the switch. The Receive Errors field includes all explicitly specified errors, including frame, overrun, and CRC errors, plus any others. Counters wrap around with no overflow indicator. In this document, the phrase byte refers to 8-bit octets.

3.3 Actions

Each flow entry is associated with zero or more actions that dictate how the switch handles matching packets. If no forward actions are present, the packet is dropped. Action lists for *inserted* flow entries MUST be processed in the order specified. However, there is no packet output ordering guaranteed within a port. For example, an action list may result in two packets sent to two different VLANs on a single port. These two packets may be arbitrarily re-ordered, but the packet bodies must match those generated from a sequential execution of the actions.

A switch may reject a flow entry if it cannot process the action list in the order specified, in which case it should immediately return an unsupported flow error (see 4.6). Ordering within a port may vary between vendor switch implementations.

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

Ingress Port	Ether source	Ether dst	Ether type	VLAN id	VLAN priority	IP src	IP dst	IP proto	IP ToS bits	TCP/UDP src port	TCP/UDP dst port

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

DELL - Exhibit 1009 - Page 5

Field	Bits	When applicable	Notes
Ingress Port	(Implementation dependent)	All packets	Numerical representation of incoming port, starting at 1.
Ethernet source address	48	All packets on enabled ports	
Ethernet destination address	48	All packets on enabled ports	
Ethernet type	16	All packets on enabled ports	An OpenFlow switch is required to match the type in both standard Ethernet and 802.2 with a SNAP header and OUI of 0x000000. The special value of 0x05FF is used to match all 802.3 packets without SNAP headers.
VLAN id	12	All packets of Ethernet type 0x8100	
VLAN priority	3	All packets of Ethernet type 0x8100	VLAN PCP field
IP source address	32	All IP and ARP packets	Can be subnet masked
IP destination address	32	All IP and ARP packets	Can be subnet masked
IP protocol	8	All IP and IP over Ethernet, ARP packets	Only the lower 8 bits of the ARP opcode are used
IP ToS bits	6	All IP packets	Specify as 8-bit value and place ToS in upper 6 bits.
Transport source port / ICMP Type	16	All TCP, UDP, and ICMP packets	Only lower 8 bits used for ICMP Type
Transport destination port / ICMP Code	16	All TCP, UDP, and ICMP packets	Only lower 8 bits used for ICMP Code

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

DELL - Exhibit 1009 - Page 6

Counter	Bits
Per Table	
Active Entries	32
Packet Lookups	64
Packet Matches	64
Per Flow	
Received Packets	64
Received Bytes	64
Duration (seconds)	32
Duration (nanoseconds)	32
Per Port	
Received Packets	64
Transmitted Packets	64
Received Bytes	64
Transmitted Bytes	64
Receive Drops	64
Transmit Drops	64
Receive Errors	64
Transmit Errors	64
Receive Frame Alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64
Per Queue	
Transmit Packets	64
Transmit Bytes	64
Transmit Overrun Errors	64

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

DELL - Exhibit 1009 - Page 7

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 switchs 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.

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).

Optional Action: *Enqueue*. The enqueue action forwards a packet through a queue attached to a port. Forwarding behavior is dictated by the configuration of the queue and is used to provide basic Quality-of-Service (QoS) support (see section 5.2.2).

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.

DELL - Exhibit 1009 - Page 8

Action	Associated Data	Description
Set VLAN ID	12 bits	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.
Set VLAN priority	3 bits	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.
Strip VLAN header	-	Strip VLAN header if present.
Modify Ethernet source MAC address	48 bits: Value with which to replace existing source MAC address	Replace the existing Ethernet source MAC address with the new value
Modify Ethernet destination MAC address	48 bits: Value with which to replace existing destination MAC address	Replace the existing Ethernet destination MAC address with the new value.
Modify IPv4 source address	32 bits: Value with which to replace existing IPv4 source address	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.
Modify IPv4 destination address	32 bits: Value with which to replace existing IPv4 destination address	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.
Modify IPv4 ToS bits	6 bits: Value with which to replace existing IPv4 ToS field	Replace the existing IP ToS field. This action is only applied to IPv4 packets.
Modify transport source port	16 bits: Value with which to replace existing TCP or UDP source port	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.
Modify transport destination port	16 bits: Value with which to replace existing TCP or UDP destination port	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.

Table 5: Field-modify actions.

DELL - Exhibit 1009 - Page 9

3.4 Matching

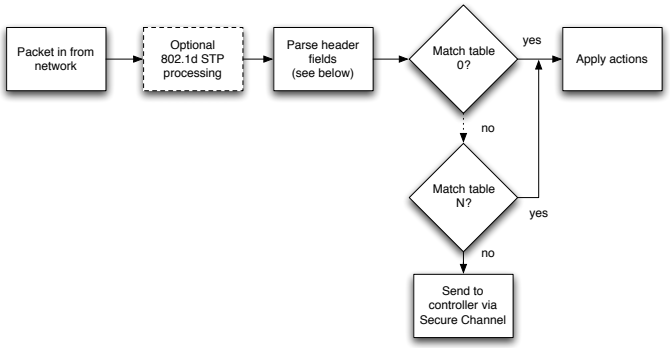

Figure 2: Packet flow in an OpenFlow switch. As discussed in Section 4.5, support for 802.1D is optional.

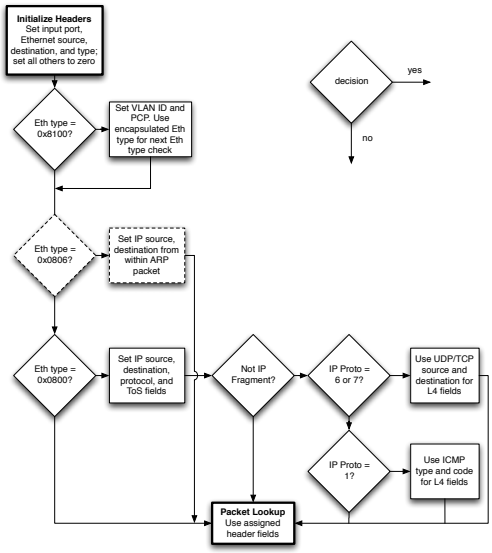

Figure 3: Flowchart showing how header fields are parsed for matching.

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

DELL - Exhibit 1009 - Page 10

- If the packet is a VLAN (Ethernet type 0x8100), the VLAN ID and PCP fields are used in the lookup.
- (Optional) For ARP packets (Ethernet type equal to 0x0806), the lookup fields may also include the contained IP source and destination fields.
- 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 prototcol 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.

DELL - Exhibit 1009 - Page 11

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.

Support for multiple simultaneous controllers is currently undefined.

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 Transport Layer Security (TLS) 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 switchs 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 the controller soliciting them from a switch. Switches send asynchronous messages to the controller to denote a

DELL - Exhibit 1009 - Page 12

packet arrival, switch state change, or error. The four main asynchronous message types are described below.

Packet-in: For all packets that do not have a matching flow entry, a packet-in event 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. Flow modify messages which delete flows 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.

DELL - Exhibit 1009 - Page 13

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, TLS 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.

DELL - Exhibit 1009 - Page 14

4.4 Encryption

The switch and controller communicate through a TLS connection. The TLS connection is initiated by the switch on startup to the controller's 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:

```
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. */
};
```

For ADD requests with the `OFPFF_CHECK_OVERLAP` flag set, the switch must first check for any overlapping flow entries. 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.

DELL - Exhibit 1009 - Page 15

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.

If the action list in a flow mod message 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.

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

DELL - Exhibit 1009 - Page 16

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.

If a switch cannot process the action list for any flow mod message in the order specified, it MUST immediately return an `OFPET_FLOW_MOD_FAILED` : `OFPFMFC_UNSUPPORTED` error and reject the flow.

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:

```
/* Header on all OpenFlow packets. */
struct ofp_header {
    uint8_t version;    /* OFP_VERSION. */
```


DELL - Exhibit 1009 - Page 17

```
    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 0x01 . 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:

```
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_FEATURES_REPLY,      /* Controller/switch message */
    OFPT_GET_CONFIG_REQUEST,  /* Controller/switch message */
    OFPT_GET_CONFIG_REPLY,    /* Controller/switch message */
    OFPT_SET_CONFIG,          /* Controller/switch message */

    /* Asynchronous messages. */
    OFPT_PACKET_IN,           /* Async message */
    OFPT_FLOW_REMOVED,        /* Async message */
    OFPT_PORT_STATUS,         /* Async message */

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

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

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

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

};
```


DELL - Exhibit 1009 - Page 18

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:

```
/* Description of a physical port */
struct ofp_phy_port {
    uint16_t port_no;
    uint8_t hw_addr[OFP_ETH_ALEN];
    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. */
};
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:

```
/* 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_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 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

DELL - Exhibit 1009 - Page 19

802.1D spanning tree.

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.

The state field describes the spanning tree state and whether a physical link is present, with 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. */

    /* The OFPPS_STP_* bits have no effect on switch operation.  The
     * 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_LEARN   = 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
```


DELL - Exhibit 1009 - Page 20

```
                                port. */
    OFPP_TABLE      = 0xfff9,  /* Perform actions in flow table.
                                  NB: This can only be the destination
                                  port for packet-out messages. */
    OFPP_NORMAL     = 0xfffa,  /* 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 (autonegotiation 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_PAUSE_ASYM = 1 << 11  /* Asymmetric pause. */
};
```

Multiple of these flags may be set simultaneously.

5.2.2 Queue Structures

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

A queue is described by the `ofp_packet_queue` structure:

```
/* Full description for a queue. */
struct ofp_packet_queue {
    uint32_t queue_id;     /* id for the specific queue. */
    uint16_t len;          /* Length in bytes of this queue desc. */
    uint8_t pad[2];        /* 64-bit alignment. */
    struct ofp_queue_prop_header properties[0]; /* List of properties. */
};
OFP_ASSERT(sizeof(struct ofp_packet_queue) == 8);
```

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

DELL - Exhibit 1009 - Page 21

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

Each queue property description starts with a common header:

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

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

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

5.2.3 Flow Match Structures

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

```
/* Fields to match against flows */
struct ofp_match {
    uint32_t wildcards;        /* Wildcard fields. */
    uint16_t in_port;          /* Input switch port. */
    uint8_t dl_src[OFP_ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[OFP_ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan;          /* Input VLAN id. */
    uint8_t dl_vlan_pcp;       /* Input VLAN priority. */
    uint8_t pad1[1];           /* Align to 64-bits */
    uint16_t dl_type;          /* Ethernet frame type. */
    uint8_t nw_tos;            /* IP ToS (actually DSCP field, 6 bits). */
    uint8_t nw_proto;          /* IP protocol or lower 8 bits of
                                * ARP opcode. */
    uint8_t pad2[2];           /* Align to 64-bits */
    uint32_t nw_src;           /* IP source address. */
    uint32_t nw_dst;           /* IP destination address. */
    uint16_t tp_src;           /* TCP/UDP source port. */
    uint16_t tp_dst;           /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 40);
```

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

```
/* Flow wildcards. */
enum ofp_flow_wildcards {
```


DELL - Exhibit 1009 - Page 22

```
    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. */

    /* 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,

    OFPFW_DL_VLAN_PCP = 1 << 20,  /* VLAN priority. */
    OFPFW_NW_TOS = 1 << 21,  /* IP ToS (DSCP field, 6 bits). */

    /* Wildcard all fields. */
    OFPFW_ALL = ((1 << 22) - 1)
};
```

If no wildcards are set, then the `ofp_match` exactly describes a flow, over the entire OpenFlow 12-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".

5.2.4 Flow Action Structures

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

```
enum ofp_action_type {
    OFPAT_OUTPUT,           /* Output to switch port. */
    OFPAT_SET_VLAN_VID,     /* Set the 802.1q VLAN id. */
    OFPAT_SET_VLAN_PCP,     /* Set the 802.1q priority. */
    OFPAT_STRIP_VLAN,       /* Strip the 802.1q header. */
    OFPAT_SET_DL_SRC,       /* Ethernet source address. */
```


DELL - Exhibit 1009 - Page 23

```
    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_ENQUEUE,          /* Output to queue.  */
    OFPAT_VENDOR = 0xffff
};
```

Output and enqueue 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:

```
/* 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 `action_output` has the following fields:

```
/* Action structure for OFPAT_OUTPUT, which sends packets out 'port'.
 * When the 'port' is the OFPP_CONTROLLER, 'max_len' indicates the max
 * number of bytes to send.  A 'max_len' of zero means no bytes of the
 * packet should be sent.*/
struct ofp_action_output {
    uint16_t type;                  /* OFPAT_OUTPUT. */
    uint16_t len;                   /* Length is 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.

The enqueue action maps a flow to an already-configured queue, regardless of the TOS and VLAN PCP bits. The packet should not change after an enqueue 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

DELL - Exhibit 1009 - Page 24

action ENQUEUE is not supported. The user can still use these queues and map flows to them by setting the relevant fields (TOS, VLAN PCP).

The enqueue action has the following fields:

```
/* OFPAT_ENQUEUE action struct: send packets to given queue on port. */
struct ofp_action_enqueue {
    uint16_t type;            /* OFPAT_ENQUEUE. */
    uint16_t len;             /* Len is 16. */
    uint16_t port;            /* Port that queue belongs. Should
                                 refer to a valid physical port
                                 (i.e. < OFPP_MAX) or OFPP_IN_PORT. */
    uint8_t pad[6];           /* Pad for 64-bit alignment. */
    uint32_t queue_id;        /* Where to enqueue the packets. */
};
OFP_ASSERT(sizeof(struct ofp_action_enqueue) == 16);
```

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. */
```


DELL - Exhibit 1009 - Page 25

```
    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:

```
/* 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:

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

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

An `action_tp_port` has the following fields:

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

```
/* 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 a multiple of 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 struct `ofp_vendor`.

DELL - Exhibit 1009 - Page 26

5.3 Controller-to-Switch Messages

5.3.1 Handshake

Upon TLS session establishment, the controller sends an OFPT_FEATURES_REQUEST message. This message does not contain a body beyond the OpenFlow header. The switch responds with an OFPT_FEATURES_REPLY message:

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

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

    uint8_t n_tables;       /* Number of tables supported by datapath. */
    uint8_t pad[3];         /* Align to 64-bits. */

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

```
/* 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. */
```


DELL - Exhibit 1009 - Page 27

```
    OFPC_STP            = 1 << 3,  /* 802.1d spanning tree. */
    OFPC_RESERVED       = 1 << 4,  /* Reserved, must be zero. */
    OFPC_IP_REASM       = 1 << 5,  /* Can reassemble IP fragments. */
    OFPC_QUEUE_STATS    = 1 << 6,  /* Queue statistics. */
    OFPC_ARP_MATCH_IP   = 1 << 7   /* Match IP addresses in ARP pkts. */
};
```

The `actions` field is a bitmap of actions supported by the switch. The list of actions is found in Section~3.3; all actions marked Required must be supported. Vendor actions should *not* be reported via this bitmask. The bitmask uses the values from `ofp_action_type` as the number of bits to shift left for an associated action. For example, `OFPAT_SET_DL_VLAN` would use the flag `0x00000002`.

The `ports` field is an array of `ofp_phy_port` structures that describe all the physical ports in the system that support OpenFlow. The number of port elements 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, respectively. The switch responds to a configuration request with an `OFPT_GET_CONFIG_REPLY` message; it does not reply to a request to set the configuration.

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

```
/* Switch configuration. */
struct ofp_switch_config {
    struct ofp_header header;
    uint16_t flags;             /* OFPC_* flags. */
    uint16_t miss_send_len;     /* Max bytes of new flow that datapath should
                                   send to the controller. */
};
OFP_ASSERT(sizeof(struct ofp_switch_config) == 12);
```

The configuration flags include the following:

```
enum ofp_config_flags {
    /* Handling of IP fragments. */
    OFPC_FRAG_NORMAL   = 0,  /* No special handling for fragments. */
    OFPC_FRAG_DROP     = 1,  /* 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 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.

DELL - Exhibit 1009 - Page 28

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.

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:

```
/* Flow setup and teardown (controller -> datapath). */
struct ofp_flow_mod {
    struct ofp_header header;
    struct ofp_match match;      /* Fields to match */
    uint64_t cookie;             /* Opaque controller-issued identifier. */

    /* 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*. */
    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_*. */
    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) == 72);
```

The `cookie` field is an opaque data value that is set by the controller. It is not used in any matching functions, and thus does not need to reside in hardware. The value -1 (0xffffffffffffffff) is reserved and must not be used. It is required that when `command` is `OFPC_MODIFY` or `OFPC_MODIFY_STRICT` that matched flows have their `cookie` field updated appropriately.

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

```
enum ofp_flow_mod_command {
    OFPFC_ADD,              /* New flow. */
    OFPFC_MODIFY,           /* Modify all matching flows. */
    OFPFC_MODIFY_STRICT,    /* Modify entry strictly matching wildcards */
    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 and differences between `OFPFC_DELETE` and `OFPFC_DELETE_STRICT`

DELL - Exhibit 1009 - Page 29

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:

```
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

DELL - Exhibit 1009 - Page 30

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:

```
/* 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 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;
    uint16_t port;         /* Port to be queried. Should refer
                              to a valid physical port (i.e. < OFPP_MAX) */
    uint8_t pad[2];        /* 32-bit alignment. */
};
OFP_ASSERT(sizeof(struct ofp_queue_get_config_request) == 12);
```

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;
```


DELL - Exhibit 1009 - Page 31

```
    uint16_t port;
    uint8_t pad[6];
    struct ofp_packet_queue queues[0]; /* List of configured queues. */
};
OFP_ASSERT(sizeof(struct ofp_queue_get_config_reply) == 16);
```

5.3.5 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 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.
     * 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,
```


DELL - Exhibit 1009 - Page 32

```
    /* 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 struct ofp_port_stats_request.
     * The reply body is an array of struct ofp_port_stats. */
    OFPST_PORT,

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

    /* 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, serial number, and a description field is available from the `OFPST_DESC` stats request type:

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

Each entry is ASCII formatted and padded on the right with null bytes (\0). `DESC_STR_LEN` is 256 and `SERIAL_NUM_LEN` is 32 . Note: [1] the `dp_desc` field is a free-form string to describe the datapath for debugging purposes, e.g., "switch3 in room 3120". As such, it is not guaranteed to be unique and should not be used as the primary identifier for the datapath—use the `datapath_id` field from the switch features instead (§˜5.3.1).

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),
```

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

DELL - Exhibit 1009 - Page 33

```
                                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_flow_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 4.6.

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:

```
/* 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_sec;    /* Time flow has been alive in seconds. */
    uint32_t duration_nsec;   /* Time flow has been alive in nanoseconds beyond
                                 duration_sec. */
    uint16_t priority;        /* Priority of the entry. Only meaningful
                                 when this is not an exact-match entry. */
    uint16_t idle_timeout;    /* Number of seconds idle before expiration. */
    uint16_t hard_timeout;    /* Number of seconds before expiration. */
    uint8_t pad2[6];          /* Align to 64-bits. */
    uint64_t cookie;          /* Opaque controller-issued identifier. */
    uint64_t packet_count;    /* Number of packets in flow. */
    uint64_t byte_count;      /* Number of bytes in flow. */
    struct ofp_action_header actions[0]; /* Actions. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 88);
```

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

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

DELL - Exhibit 1009 - Page 34

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 {
    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 4.6.

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 the following:

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

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

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

```
/* Body of reply to OFPST_TABLE request. */
struct ofp_table_stats {
    uint8_t table_id;        /* Identifier of table.  Lower numbered tables
                                are consulted first. */
    uint8_t pad[3];          /* Align to 32-bits. */
    char name[OFP_MAX_TABLE_NAME_LEN];
    uint32_t wildcards;      /* Bitmap of OFPFW_* wildcards that are
```


DELL - Exhibit 1009 - Page 35

```
                                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:

```
/* Body for ofp_stats_request of type OFPST_PORT. */
struct ofp_port_stats_request {
    uint16_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_NONE). */
    uint8_t pad[6];
};
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_NONE`.

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 {
    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. */
```


DELL - Exhibit 1009 - Page 36

```
    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.

Queue Statistics The `OFPST_QUEUE` stats request message provides queue statistics for one or more ports. The request body consists of a `port_no` field identifying the port and a `queue_id`. `OFPP_ALL` refers to all ports, while `OFPQ_ALL` refers to all queues configured at a port.

```
struct ofp_queue_stats_request {
    uint16_t port_no;        /* All ports if OFPT_ALL. */
    uint8_t pad[2];          /* Align to 32-bits. */
    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 {
    uint16_t port_no;
    uint8_t pad[2];          /* Align to 32-bits. */
    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);
```

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.

5.3.6 Send Packet Message

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

```
/* Send packet (controller -> datapath). */
struct ofp_packet_out {
    struct ofp_header header;
    uint32_t buffer_id;           /* ID assigned by datapath (-1 if none). */
    uint16_t in_port;             /* Packet's input port (OFPP_NONE if none). */
    uint16_t actions_len;         /* Size of action array in bytes. */
```


DELL - Exhibit 1009 - Page 37

```
    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.

5.3.7 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:

```
/* 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. */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 20);
```

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,

DELL - Exhibit 1009 - Page 38

the entire packet is included in the data portion, and the `buffer_id` is -1.

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

The reason field can be any of these values:

```
/* Why is this packet being sent to the controller? */
enum ofp_packet_in_reason {
    OFPR_NO_MATCH,          /* No matching flow. */
    OFPR_ACTION             /* Action explicitly output to controller. */
};
```

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:

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

    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_sec;    /* Time flow was alive in seconds. */
    uint32_t duration_nsec;   /* Time flow was alive in nanoseconds beyond
                                 duration_sec. */
    uint16_t idle_timeout;    /* Idle timeout from original flow mod. */
    uint8_t pad2[2];          /* Align to 64-bits. */
    uint64_t packet_count;
    uint64_t byte_count;
};
OFP_ASSERT(sizeof(struct ofp_flow_removed) == 88);
```

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

The `reason` field is one of the following:

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


DELL - Exhibit 1009 - Page 39

The `duration_sec` and `duration_nsec` fields are described in Section 5.3.5.

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:

```
/* 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_phy_port desc;
};
OFP_ASSERT(sizeof(struct ofp_port_status) == 64);
```

The `status` can be one of the following values:

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

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:

```
/* 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. */
};
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

DELL - Exhibit 1009 - Page 40

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:

```
/* 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. */
    OFPET_PORT_MOD_FAILED,      /* Port mod request failed. */
    OFPET_QUEUE_OP_FAILED       /* Queue operation failed. */
};
```

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

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

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

For the `OFPET_BAD_REQUEST` error `type`, the following `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,         /* ofp_header.version not supported. */
    OFPBRC_BAD_TYPE,            /* ofp_header.type not supported. */
    OFPBRC_BAD_STAT,            /* ofp_stats_request.type not supported. */
    OFPBRC_BAD_VENDOR,          /* Vendor not supported (in ofp_vendor_header
                                 * or ofp_stats_request or ofp_stats_reply). */
    OFPBRC_BAD_SUBTYPE,         /* Vendor subtype not supported. */
    OFPBRC_EPERM,               /* Permissions error. */
    OFPBRC_BAD_LEN,             /* Wrong request length for type. */
    OFPBRC_BUFFER_EMPTY,        /* Specified buffer has already been used. */
    OFPBRC_BUFFER_UNKNOWN       /* Specified buffer does not exist. */
};
```

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

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

DELL - Exhibit 1009 - Page 41

```
/* 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. */
    OFPBAC_TOO_MANY,           /* Can't handle this many actions. */
    OFPBAC_BAD_QUEUE           /* Problem validating output queue. */
};
```

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

For the `OFPET_FLOW_MOD_FAILED` error `type`, the following `code`s 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_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. */
    OFPFMFC_BAD_COMMAND,        /* Unknown command. */
    OFPFMFC_UNSUPPORTED         /* Unsupported action list - cannot process in
                                 * the order specified. */
};
```

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

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

```
/* ofp_error_msg 'code' values for OFPET_PORT_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_port_mod_failed_code {
    OFPPMFC_BAD_PORT,            /* Specified port does not exist. */
    OFPPMFC_BAD_HW_ADDR,         /* Specified hardware address is wrong. */
};
```

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

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

```
/* ofp_error msg 'code' values for OFPET_QUEUE_OP_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request */
enum ofp_queue_op_failed_code {
    OFPQOFC_BAD_PORT,           /* Invalid port (or port does not exist). */
```


DELL - Exhibit 1009 - Page 42

```
    OFPQOFC_BAD_QUEUE,          /* Queue does not exist. */
    OFPQOFC_EPERM               /* Permissions error. */
};
```

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:

```
/* 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
```


DELL - Exhibit 1009 - Page 43

```
                                *   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.

6 Appendix B: Credits

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

Spec contributions, in alphabetical order:

Ben Pfaff, Brandon Heller, Dan Talayco, David Erickson, Glen Gibb, Guido Appenzeller, Jean Tourrilhes, Justin Pettit, KK Yap, Martin Casado, Masayoshi Kobayashi, Nick McKeown, Peter Balland, Reid Price, Rob Sherwood, Yiannis Yiakoumis.

DELL - Exhibit 1009 - Page 44