

OpenFlow Switch Specification

Version 0.8.2 Draft (Wire Protocol 0x85)

October 9, 2008

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.

OpenFlow Switches will be of “Type 0” or “Type 1”, depending on their capabilities. Type 0 represents the minimum requirements for any conforming OpenFlow Switch. Type 1 requirements will be a superset of Type 0, and remain to be defined. It is expected that commercial OpenFlow Switches will initially be of Type 0, evolving to Type 1; and that vendors will support additional features over time. However, all switches are expected to use the same OpenFlow Protocol for communication between switch and controller. For the remainder of this version of the document, unless otherwise specified, all references to an OpenFlow Switch refer to Type 0.

Version 1.0 of this document will be the first to specify a Type 0 switch suitable for implementation. Versions before Version 1.0 will be marked “Draft”, and will include the header: “Do not build a switch from this specification!” We hope to generate feedback from versions prior to Version 1.0 from switch designers and network researchers.

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.

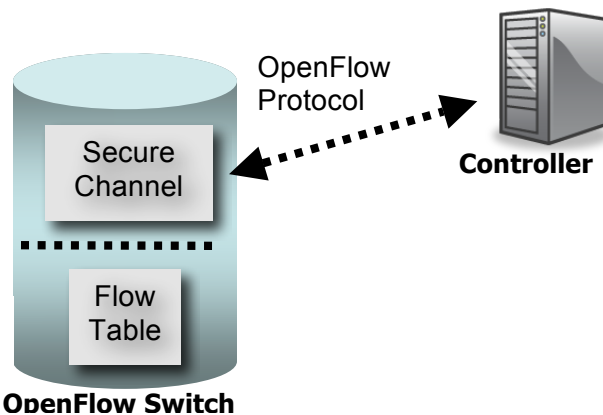


Figure 1: An OpenFlow switch communicates with a controller over a secure connection using the OpenFlow protocol.

3 Flow Table

The OpenFlow flow table is a set of flow entries, each entry containing: (1) The header values to match packets against, (2) Counters which are updated on each matching packet, and (3) A set of actions to apply to matching packets (Table 1).

3.1 Flow Entry

Header fields	Counters	Actions
---------------	----------	---------

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

3.1.1 Header Fields

Figure 2 shows the header fields an incoming packet is compared against. Each entry contains a specific value, or an ANY, which matches any value. The field properties are described in Table 2.

Ingress port	Ethernet source	Ethernet dst	Ethernet Type	VLAN id	IP src	IP dst	IP proto	TCP/UDP src port	TCP/UDP dst port
--------------	-----------------	--------------	---------------	---------	--------	--------	----------	------------------	------------------

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

Field	Bits	When applicable	Notes
Ingress Port	(Implementation dependent)	All packets	Numerical representation of incoming port.
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	A Type 0 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	
IP source address	32	All IP packets	
IP destination address	32	All IP packets	
IP protocol	8	All IP packets	
Transport source port	16	All TCP and UDP packets	
Transport destination port	16	All TCP and UDP packets	

Table 2: Field lengths and the way they should be applied to flow entries.

The field types contained in this document are derived from the OpenFlow protocol and used to describe the valid ranges. 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.

3.1.2 Counters

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

Table 3 contains the required set of counters for Type 0 switches.

Counter	Bits	Description
Per Table		
Packet Match	64	Number of matching packets
Per Flow		
Packet	64	Number of packets received
Byte	64	Number of bytes received
Duration	32	Number of seconds flow has been active
Per Port		
Received (rx)	64	Number of packets received
Transmitted (tx)	64	Number of packets sent
Dropped	64	Number of packets dropped

Table 3: Required list of counters for use in statistic messages of tables, flows and ports.

3.1.3 Actions

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

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

The OpenFlow whitepaper lists four actions that all Type 0 switches must implement: (1) Forward to external port(s), (2) Drop, (3) Encapsulate and forward to controller over Secure Channel, and (4) Forward to the normal forwarding path of this switch (e.g. for normal Layer 2 and Layer 3 processing). This specification defines these actions in a uniform way, by combining them into two required actions: *Forward* and *Drop*.

Required Action: *Forward*. A Type 0 switch is required to support forwarding the packet to physical ports and the following virtual ones:

- **NORMAL:** Process the packet using the traditional forwarding path supported by the switch (i.e., traditional L2, VLAN, and L3 processing.) At a minimum, the normal processing path must support L2 learning and forwarding. A type 0 switch may be implemented to check on the VLAN field to determine whether or not to forward the packet along the normal processing route.
- **FLOOD:** Flood the packet along the minimum spanning tree, not including the incoming interface.
- **ALL:** Send the packet out all interfaces, not including the incoming interface.
- **CONTROLLER:** Encapsulate and send the packet to the controller.
- **LOCAL:** Send the packet to the switch’s local networking stack.

Ideally, a switch will support flow-entries that can forward packets to any combination of the physical and virtual ports. For example, this could be expressed internally in the switch with a bitmap that includes all the physical and virtual ports. However, some switches will not be able to support any combination.

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

Therefore, the requirement is that the switch support sending to any combination of physical ports and the “Controller” virtual port simultaneously. All other combinations are desired, but optional.

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

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

Optional Actions: While not strictly required, the following actions greatly increase the usefulness of an OpenFlow implementation. To aid integration with existing networks, we suggest that the VLAN modification action be supported.

Action	Associated Data	Description
Add/Modify/Remove VLAN tag	16 bits	<p>If no VLAN is present on the packet, a new header is added with the specified VLAN ID (lower 12 bits).</p> <p>If a VLAN header already exists, the VLAN ID is replaced with the specified value.</p> <p>If all 16 bits of the action value are set (0xffff) then remove VLAN tag.</p>
Modify IPv4 source address	32 bits: Value with which to replace existing IPv4 source address	<p>Replace the existing IP source address with new value and update the IP checksum (and TCP/UDP checksum if applicable).</p> <p>This action is only applicable to IPv4 packets.</p>
Modify IPv4 destination address	32 bits: Value with which to replace existing IPv4 destination address	<p>Replace the existing IP destination address with new value and update the IP checksum (and TCP/UDP checksum if applicable).</p> <p>This action is only applied to IPv4 packets.</p>
Modify transport source port	16 bits: Value with which to replace existing TCP or UDP source port	<p>Replace the existing TCP/UDP source port with new value and update the TCP/UDP checksum.</p> <p>This action is only applicable to TCP and UDP packets.</p>
Modify transport destination port	16 bits: Value with which to replace existing TCP or UDP destination port	<p>Replace the existing TCP/UDP destination port with new value and update the TCP/UDP checksum</p> <p>This action is only applied to TCP and UDP packets.</p>

Table 4: Optional actions in a Type 1 switch.

3.2 Dataflow

On receipt of a packet, an OpenFlow Switch performs the functions shown in Figure 3.

The flow table is checked for a matching flow entry. The header fields used for the table lookup depend on the packet type as described below (and shown in Figure 4).

- Rules specifying an ingress port are matched against the physical port that received the packet.
- The Ethernet headers as specified in Table 2 are used for all packets.
- If the packet is a VLAN (Ethernet type 0x8100), the VLAN ID is used in the lookup.
- For IP packets (Ethernet type equal to 0x0800), the lookup fields also include those in the IP header.
- For IP packets that are TCP or UDP (IP protocol is equal to 6 or 17), the lookup includes the transport ports.

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's protocol id is matched against the flow's 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 other 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 that eases implementation.

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.

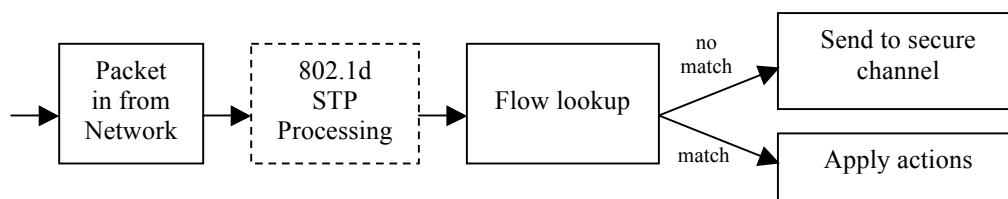


Figure 3: The functions performed on a packet as it moves through an OpenFlow switch. As discussed in Section 0, support for 802.1d is optional in Type 0 switches.

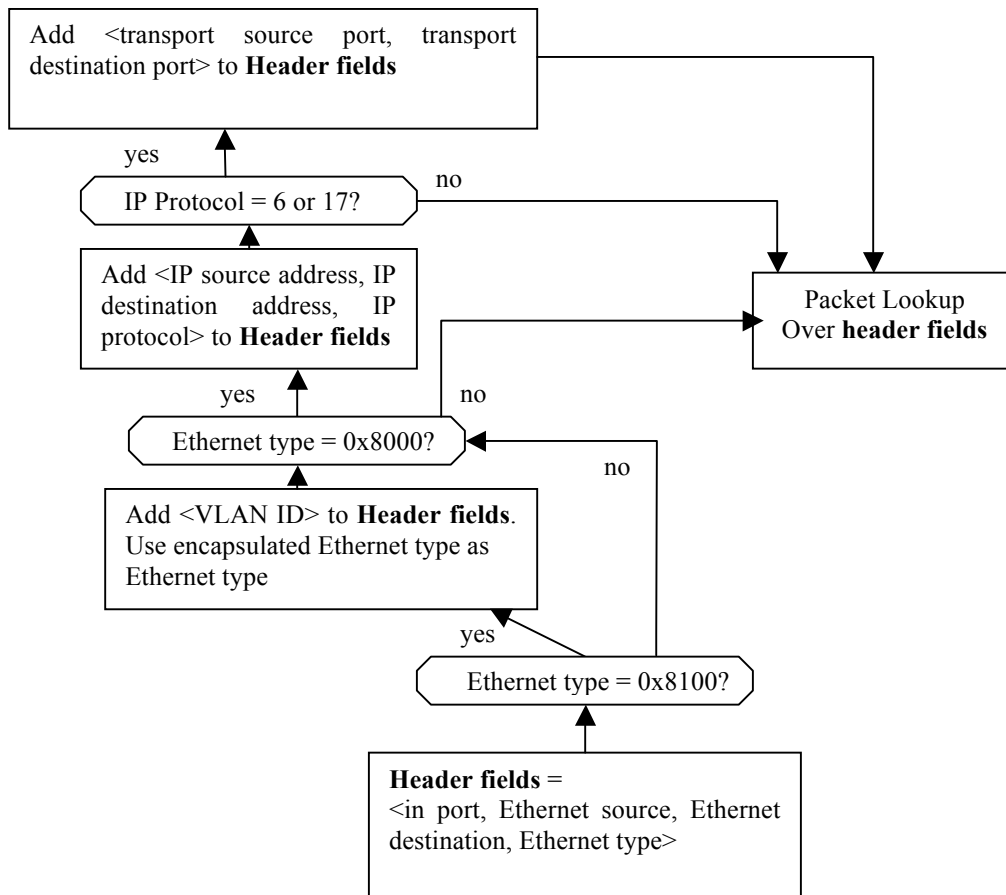


Figure 4: A flow table showing how a packet is matched against a flow entry.

3.3 STP Support

Type 0 switches may optionally support the 802.1d spanning tree protocols. Those switches that do support it are expected to process all 802.1d packets locally before performing flow lookup. 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.

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.

3.4 Interface with Secure Channel

The interface between the datapath and the secure channel are implementation-specific. However, the controller must be able to have the flow table perform the following functions based on the OpenFlow messages:

- Add flow table entry
- Delete flow table entry
- Forward packet out of port
- Send packet to the controller

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

- Update and report counter values for flow entries, flow tables and switch ports

3.4.1 Flow Removal

Each flow entry has a maximum idle time associated with it. If no packet has matched the rule in that time, the switch removes the entry and sends a flow expiration 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.

If the DELETE command is used, the wildcards are “active” and all flows that match the description are removed. If the DELETE_STRICT command is used, all fields, including the wildcards and priority, are strictly matched against the entry, and only an identical flow is removed from the table. 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.

4 Secure Channel

4.1 Switch/Controller Connection

The switch and controller communicate through an SSL connection. 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.

The SSL connection is initiated by the switch on startup to the controller’s server, which is located by default on TCP port 976. 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).

4.2 OpenFlow Protocol Overview

The controller configures and manages the switch, and receives events from the switch, via the OpenFlow protocol transmitted using the SSL connection. This description of the OpenFlow protocol is not intended to be complete but rather provide a general idea of the communication flow and capabilities. A comprehensive protocol specification will be released in the future as a separate document.

The OpenFlow protocol supports two message types, *controller/switch* and *asynchronous*, each with multiple sub-types. Controller/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. The main message types used by OpenFlow are described below.

4.2.1 Controller/Switch

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

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

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

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

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 switch's 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.

4.2.2 Asynchronous

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

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

Flow Expiration: When a flow entry is added to the switch, an idle timeout value is included that indicates when the entry should be removed due to a lack of activity. In the configuration message, the controller can indicate that it wishes to be informed when a flow expires. If this flag is set, the switch sends a flow expiration event that includes the duration of the flow and the number of packets and bytes sent.

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 ports status as specified by 802.1d (see Section 3.3 for a description of 802.1d support requirements).

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

4.2.3 Symmetric

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.

5 Appendix A: The OpenFlow Protocol

5.1 OpenFlow Header

All OpenFlow messages are sent in big-endian format. Each OpenFlow message begins with the OpenFlow header:

```

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

```

The version specifies the OpenFlow protocol version being used. During this evolution 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 0x85. 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 {
    OFPT_FEATURES_REQUEST, /* 0 Controller/switch message */
    OFPT_FEATURES_REPLY, /* 1 Controller/switch message */
    OFPT_GET_CONFIG_REQUEST, /* 2 Controller/switch message */
    OFPT_GET_CONFIG_REPLY, /* 3 Controller/switch message */
    OFPT_SET_CONFIG, /* 4 Controller/switch message */
    OFPT_PACKET_IN, /* 5 Async message */
    OFPT_PACKET_OUT, /* 6 Controller/switch message */
    OFPT_FLOW_MOD, /* 7 Controller/switch message */
    OFPT_FLOW_EXPIRED, /* 8 Async message */
    OFPT_TABLE, /* 9 Controller/switch message */
    OFPT_PORT_MOD, /* 10 Controller/switch message */
    OFPT_PORT_STATUS, /* 11 Async message */
    OFPT_ERROR_MSG, /* 12 Async message */
    OFPT_STATS_REQUEST, /* 13 Controller/switch message */
    OFPT_STATS_REPLY, /* 14 Controller/switch message */
    OFPT_ECHO_REQUEST, /* 15 Symmetric message */
    OFPT_ECHO_REPLY /* 16 Symmetric message */
};

```

5.2 Common Structures

A number of structures are used by many of the OpenFlow messages. 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[ETH_ALEN];
    uint8_t name[OFPT_MAX_PORT_NAME_LEN]; /* Null-terminated */
    uint32_t flags; /* Bitmap of "ofp_port_flags". */
    uint32_t speed; /* Current speed in Mbps */
    uint32_t features; /* Bitmap of supported "ofp_port_features"s. */
};

```

The name field is a null-terminated string containing a human-readable name for the interface. The value of OFPT_MAX_PORT_NAME_LEN is 16. The port_no field is a value the datapath associates with a physical port. The only flag currently defined is:

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

```
/* Flags to indicate behavior of the physical port */
enum ofp_port_flags {
    OFPPFL_NO_FLOOD = 1 << 0, /* Do not include this port when flooding */
};
```

The port numbers use the following conventions:

```
/* Port numbering. Physical ports are numbered starting from 0. */
enum ofp_port {
    /* Maximum number of physical switch ports. */
    OFPP_MAX = 0x100,

    /* Fake output "ports". */
    OFPP_TABLE = 0xffff9, /* Perform actions in flow table.
        * NB: This can only be the destination
        * port for packet-out messages.
        */
    OFPP_NORMAL = 0xffffa, /* Process with normal L2/L3 switching */
    OFPP_FLOOD = 0xffffb, /* All physical ports except input port and
        those disabled by STP. */
    OFPP_ALL = 0xffffc, /* All physical ports except input port. */
    OFPP_CONTROLLER = 0xffffd, /* Send to controller. */
    OFPP_LOCAL = 0xffffe, /* Local OpenFlow "port". */
    OFPP_NONE = 0xfffff /* Not associated with a physical port. */
};
```

The `features` field describes the characteristics of this port. Multiple of these flags may be set simultaneously:

```
/* 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. */
};
```

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

```
struct ofp_match {
    uint16_t wildcard; /* Wildcard fields. */
    uint16_t in_port; /* Input switch port. */
    uint8_t dl_src[ETH_ALEN]; /* Ethernet source address. */
    uint8_t dl_dst[ETH_ALEN]; /* Ethernet destination address. */
    uint16_t dl_vlan; /* Input VLAN. */
    uint16_t dl_type; /* Ethernet frame type. */
    uint32_t nw_src; /* IP source address. */
    uint32_t nw_dst; /* IP destination address. */
    uint8_t nw_proto; /* IP protocol. */
    uint8_t pad[3]; /* Align to 32-bits */
    uint16_t tp_src; /* TCP/UDP source port. */
    uint16_t tp_dst; /* TCP/UDP destination port. */
};
OFP_ASSERT(sizeof(struct ofp_match) == 36);
```

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

```
/* Flow wildcards. */
enum ofp_flow_wildcards {
    OFFFW_IN_PORT = 1 << 0, /* Switch input port. */
    OFFFW_DL_VLAN = 1 << 1, /* VLAN. */
    OFFFW_DL_SRC = 1 << 2, /* Ethernet source address. */
    OFFFW_DL_DST = 1 << 3, /* Ethernet destination address. */
};
```

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

```
OFFFW_DL_TYPE = 1 << 4, /* Ethernet frame type. */
OFFFW_NW_SRC = 1 << 5, /* IP source address. */
OFFFW_NW_DST = 1 << 6, /* IP destination address. */
OFFFW_NW_PROTO = 1 << 7, /* IP protocol. */
OFFFW_TP_SRC = 1 << 8, /* TCP/UDP source port. */
OFFFW_TP_DST = 1 << 9, /* TCP/UDP destination port. */
OFFFW_ALL = (1 << 10) - 1
};
```

If no wildcards are set, then the `ofp_match` exactly describes a flow. On the other extreme, if all the wildcard flags are set, then every flow will match.

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

```
enum ofp_action_type {
    OFFPAT_OUTPUT, /* Output to switch port. */
    OFFPAT_SET_DL_VLAN, /* VLAN. */
    OFFPAT_SET_DL_SRC, /* Ethernet source address. */
    OFFPAT_SET_DL_DST, /* Ethernet destination address. */
    OFFPAT_SET_NW_SRC, /* IP source address. */
    OFFPAT_SET_NW_DST, /* IP destination address. */
    OFFPAT_SET_TP_SRC, /* TCP/UDP source port. */
    OFFPAT_SET_TP_DST /* TCP/UDP destination port. */
};
```

An action definition contains the action type and any associated data:

```
struct ofp_action {
    uint16_t type; /* One of OFFPAT_* */
    union {
        struct ofp_action_output output; /* OFFPAT_OUTPUT: output struct. */
        uint16_t vlan_id; /* OFFPAT_SET_DL_VLAN: VLAN id. */
        uint8_t dl_addr[ETH_ALEN]; /* OFFPAT_SET_DL_SRC/DST */
        uint32_t nw_addr OFFPACKED; /* OFFPAT_SET_NW_SRC/DST */
        uint16_t tp; /* OFFPAT_SET_TP_SRC/DST */
    } arg;
};
OFFP_ASSERT(sizeof(struct ofp_action) == 8);
```

The `vlan_id` 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, or if used as an action, that the VLAN header should be stripped.

An `action_output` has the following fields:

```
/* An output action sends packets out 'port'. When the 'port' is the
 * OFFP_CONTROLLER, 'max_len' indicates the max number of bytes to
 * send. A 'max_len' of zero means the entire packet should be sent. */
struct ofp_action_output {
    uint16_t max_len;
    uint16_t port;
};
```

The `port` specifies the physical port from which packet packets should be sent. The `max_len` indicates the maximum amount of data from a packet that should be sent when the port is `OFFP_CONTROLLER`. If `max_len` is zero, then the entire packet should be sent.

5.3 Controller/Switch Messages

5.3.1 Handshake

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

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

```
/* Switch features. */
struct ofp_switch_features {
    struct ofp_header header;
    uint64_t datapath_id; /* Datapath unique ID. Only the lower 48-bits
                           are meaningful. */

    /* Table info. */
    uint32_t n_exact; /* Max exact-match table entries. */
    uint32_t n_compression; /* Max entries compressed on service port. */
    uint32_t n_general; /* Max entries of arbitrary form. */

    /* Buffer limits. A datapath that cannot buffer reports 0.*/
    uint32_t buffer_mb; /* Space for buffering packets, in MB. */
    uint32_t n_buffers; /* Max packets buffered at once. */

    /* Features. */
    uint32_t capabilities; /* Bitmap of support "ofp_capabilities". */
    uint32_t actions; /* Bitmap of supported "ofp_action_type"s. */
    uint8_t pad[4]; /* Align to 64-bits. */

    /* 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) == 48);
```

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

The `capabilities` field uses the following flags to indicate supported capabilities of the datapath:

```
/* Capabilities supported by the datapath. */
enum ofp_capabilities {
    OFPC_FLOW_STATS = 1 << 0, /* Flow statistics. */
    OFPC_TABLE_STATS = 1 << 1, /* Table statistics. */
    OFPC_PORT_STATS = 1 << 2, /* Port statistics. */
    OFPC_STP = 1 << 3, /* 802.11d spanning tree. */
    OFPC_MULTI_PHY_TX = 1 << 4 /* Supports transmitting through multiple
                                physical interfaces */
};
```

The `ports` field is an array of `ofp_phy_port` structures that describe all the physical ports in the system. 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);
```

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

The only configuration flag currently defined is `OFPC_SEND_FLOW_EXP`, which has the value `0x0001`. When this flag is set, the switch sends flow expiration messages.

5.3.3 Modify State Messages

5.3.3.1 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 */

    /* Flow actions. */
    uint16_t command; /* One of OFFFC_*. */
    uint16_t max_idle; /* Idle time before discarding (seconds). */
    uint32_t buffer_id; /* Buffered packet to apply to (or -1). */
    uint16_t priority; /* Priority level of flow entry. */
    uint8_t pad[2]; /* Align to 32-bits. */
    uint32_t reserved; /* Reserved for future use. */
    struct ofp_action actions[0]; /* The number of actions is inferred from
    the length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_flow_mod) == 60);
```

The `buffer_id` refers to a buffered packet sent by the `OFPT_PACKET_IN` message. If a flow is added with a `max_idle` value of `0x0000`, then the entry will never time out. The values of `command` can be:

```
enum ofp_flow_mod_command {
    OFFFC_ADD, /* New flow. */
    OFFFC_DELETE /* Delete all matching flows. */
    OFFFC_DELETE_STRICT /* Strictly match wildcards */
};
```

The differences between `OFFFC_DELETE` and `OFFFC_DELETE_STRICT` are explained more fully in Section 3.4.1.

5.3.3.2 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;
    struct ofp_phy_port desc;
};
OFP_ASSERT(sizeof(struct ofp_port_mod) == 44);
```

5.3.4 Read State Messages

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

```
struct ofp_stats_request {
    struct ofp_header header;
    uint16_t type; /* One of the OFPST_* constants. */
    uint16_t flags; /* OFPSF_REQ_* flags (none yet defined). */
    uint8_t body[0]; /* Body of the request. */
};
OFP_ASSERT(sizeof(struct ofp_stats_request) == 12);
```

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

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

```
struct ofp_stats_reply {
    struct ofp_header header;
    uint16_t type;           /* One of the OFPST_* constants. */
    uint16_t flags;         /* OFPST_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 {
    /* Individual flow statistics.
     * The request body is struct ofp_flow_stats_request.
     * The reply body is an array of struct ofp_flow_stats. */
    OFPST_FLOW,

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

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

    /* Physical port statistics.
     * The request body is empty.
     * The reply body is an array of struct ofp_port_stats. */
    OFPST_PORT
};
```

5.3.4.1 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)
                          * or 0xff for all tables. */
    uint8_t pad[3];       /* Align to 32-bits */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 40);
```

The `match` field contains a description of the flows that should be matched and may contain wildcards.

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

```
/* Statistics about flows that match the "match" field */
struct ofp_flow_stats {
    uint16_t length;       /* Length of this entry. */
    uint8_t table_id;     /* ID of table flow came from. */
    uint8_t pad;
    struct ofp_match match; /* Description of fields. */
    uint32_t duration;    /* Time flow has been alive in seconds. */
    uint16_t priority;    /* Priority of the entry. Only meaningful
                          * when this is not an exact-match entry. */
    uint16_t max_idle;    /* Number of seconds idle before expiration. */
    uint64_t packet_count; /* Number of packets in flow. */
};
```

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

```
uint64_t byte_count; /* Number of bytes in flow. */
struct ofp_action actions[0]; /* Actions. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 64);
```

5.3.4.2 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)
                      or 0xff for all tables. */
    uint8_t pad[3]; /* Align to 32 bits. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_request) == 40);
```

The `match` field contains a description of the flows that should be matched and may contain wildcards.

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);
```

5.3.4.3 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;
    uint8_t pad[3]; /* Align to 32-bits */
    char name[OFPMAX_TABLE_NAME_LEN];
    uint32_t max_entries; /* Max number of entries supported */
    uint32_t active_count; /* Number of active entries */
    uint8_t pad2[4]; /* Align to 64 bits. */
    uint64_t matched_count; /* Number of packets that hit table */
};
OFP_ASSERT(sizeof(struct ofp_table_stats) == 56);
```

5.3.4.4 Port Statistics

Information about physical ports is requested with the `OFPST_PORT` stats request type. The request does not contain any data in the `body`.

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

```
/* Statistics about a particular port */
struct ofp_port_stats {
    uint16_t port_no;
    uint8_t pad[6]; /* Align to 64-bits */
    uint64_t rx_count; /* Number of received packets */
    uint64_t tx_count; /* Number of transmitted packets */
    uint64_t drop_count; /* Number of packets dropped by interface */
}
```

DO NOT BUILD A SWITCH FROM THIS SPECIFICATION!

```
OFPP_ASSERT(sizeof(struct ofp_port_stats) == 32);
```

5.3.5 Send Packet Message

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

```
/* 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 out_port; /* Output port. */
    union {
        struct ofp_action actions[0]; /* buffer_id != -1 */
        uint8_t data[0]; /* buffer_id == -1 */
    };
};
OFPP_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 data is included in the `data` array. Otherwise, the `actions` field is used to indicate actions that should be applied to the packet associated with the `buffer_id` when it reaches the datapath.

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[]; /* 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. */
};
OFPP_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 flow setup request are sent. If the packet is sent because of a flow table miss, then at least `miss_send_len` from the `OFPT_SET_CONFIG` message are sent. If the value was never sent, the default is 128 bytes. If packet is not buffered, the entire packet is included in the `data` portion, and the `buffer_id` is `-1`. The `reason` field can be any of these values:

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

5.4.2 Flow Expiration Message

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

```
/* Flow expiration (datapath -> controller). */
struct ofp_flow_expired {
    struct ofp_header header;
    struct ofp_match match; /* Description of fields */

    uint16_t priority; /* Priority level of flow entry. */
    uint8_t pad[2]; /* Align to 32-bits. */

    uint32_t duration; /* Time flow was alive in seconds. */
    uint8_t pad2[4]; /* Align to 64-bits. */
    uint64_t packet_count;
    uint64_t byte_count;
};
OFP_ASSERT(sizeof(struct ofp_flow_expired) == 72);
```

The `match` and `priority` fields is the same as was used in the flow setup request. The `duration` field indicates the number of seconds 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[3]; /* Align to 32-bits */
    struct ofp_phy_port desc;
};
OFP_ASSERT(sizeof(struct ofp_port_status) == 48);
```

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_MOD, /* 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:

```
/* 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 on the `type` and `code`; in most cases this is the message that caused the problem. The exact definition of errors will be specified in a future appendix.

5.5 Symmetric Messages

5.5.1 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.

5.5.2 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.