

OpenFlow Hardware Abstraction API Specification – DRAFT

Version 0.4 based on OpenFlow 0.9.0

September 16, 2009

Contributors: David Erickson, Mikio Hara, Brandon Heller, Bob Lantz, Justin Pettit,
Ben Pfaff, Dan Talayco

1 Introduction

This document describes the OpenFlow Hardware Abstraction API, a programming interface inside the OpenFlow switch software architecture. The interface provides isolation between vendor specific hardware details and OpenFlow switch implementations. It abstracts the physical hardware port to an OpenFlow port type and the hardware packet processing tables to flow manipulation operations.

1.1 Component Overview

The OpenFlow system includes a Controller which generates OpenFlow commands and one or more OpenFlow switches which accept the commands from the Controller and process packets according to the resulting state of the Flow Tables it maintains. This spec focuses on the OpenFlow switch software architecture.

The important abstractions for this description include:

- **Port Interface:** A conduit through which packets may be sent or received. Typically associated to a physical port, but some abstractions may group physical ports (e.g., trunking) or associate multiple interfaces to a physical port (e.g., each with a different VLAN). An interface may have additional state such as “up/down” or spanning tree state. In addition, an interface may maintain statistics on its use.
- **Flow Table and Datapath:** A Flow Table is a prioritized list of flow descriptions, each with associated actions. There may be multiple Flow Tables in a switch (for instance, one in hardware, one in software). Each Flow Table is associated with a Datapath. The Datapath accepts packets to be processed according to the rules in its Flow Table. It updates the packets and forwards them according to the actions of the matching flow entry (if a match is found). Packets may arrive from a port interface on the switch or from the OpenFlow protocol stack. A Flow Table and Datapath may be implemented in hardware or in software.

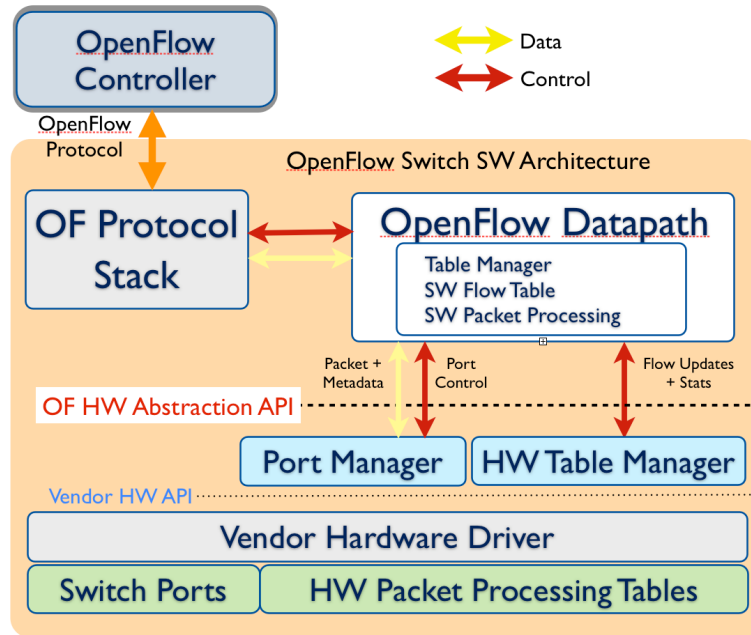


Figure 1: High Level OpenFlow Switch Software Architecture

Figure 1 illustrates how the OpenFlow Hardware Abstraction API fits into the software architecture of an OpenFlow switch. The main components of this architecture include:

- **OpenFlow Protocol Stack:** Makes connection with a Controller and parses OF commands as they arrive. It is called *secchan* or *protocold* in some distributions.
- **OpenFlow Datapath:** The main controlling component for OpenFlow. It usually maintains a software flow table and processes packets from the datapath. (Some applications may disable or limit this function). It coordinates the multiple flow tables (software and hardware) in the system. It handles packets arriving from the hardware to be processed by the software flow table or forwarded to the controller.
- **Port Manager:** Implements the needed functionality to expose a port from the hardware (physical or otherwise) to OpenFlow. May provide some control or expose port state (enabled, link state, spanning tree state, etc.) Allows access to port statistics. Packet data flows from the hardware through the port manager to the OpenFlow datapath.
- **Hardware Table Manager:** Implements the needed functionality to expose hardware packet processing tables to OpenFlow. Converts OpenFlow flow insertion messages into hardware specific table entries. Allows access to flow table and per-flow statistics.

In addition to identifying these high level components, one of the main considerations of the software architecture is to identify the mechanisms by which these modules communicate. In the simplest case,

the two modules are assumed to be in the same address space. In this case, a “driver” structure may be passed between the modules during initialization allowing one module to call functions implemented by the other module.

When two modules are not in the same address space, things become much more complicated because the facilities for communication between them will depend on the operating system and how the modules are deployed. Most frequently, one module is located in the OS kernel and the other in a user space.

We assume that the modules connected by the OpenFlow hardware abstraction APIs (port and flow table) lie in the same address space.

Let’s go through a couple of example operations in the system. We’ll look at a flow insertion operation followed by the packet flow through the Datapath. We suppose an OpenFlow switch with a hardware Datapath with limited resources and a software Datapath.

Note that if a flow description in the hardware table is matched, the software table will not be consulted.

- *Flow Insertion*

The OpenFlow Controller generates a flow insertion (for example, because an initial packet in the flow was forwarded to the Controller and the Controller’s programming accepted the flow for setup). The Controller sends the insertion via the OpenFlow Protocol to the OpenFlow Switch. The protocol stack on the Switch identifies this as a Flow Table manipulation command and forwards the information to the OpenFlow Datapath Manager. Normally, the Datapath Manager will attempt to insert the flow into both the hardware and software datapath flow tables.

If the Table Manager determines that the flow has lower priority than one only installed in the software table, it should not insert the new flow in hardware. This is because the new flow would overshadow the higher priority entry and packets would be incorrectly forwarded.

- *Packet Flow*

Packets arriving at the hardware will go through the hardware datapath. If the packet matches a flow stored in the hardware tables, it will normally be updated and sent out a port based on the action given for that flow. If the packet does not match (or is directed to the CPU subsystem by the flow actions), it is forwarded through the vendor hardware layer, through the OpenFlow hardware API to the software OpenFlow Datapath Manager.

The software Datapath may be able to match the packet with a flow in the software Flow Table and send the packet back to the hardware. If not, it will send the packet to the protocol stack for encapsulation and forwarding to the Controller.

1.2 API Purpose

The purpose of the OpenFlow HW Abstraction API is to allow the software components of an OpenFlow Switch to control a hardware Datapath and hardware port interfaces. It should provide an extensible mechanism to expose vendor specific hardware features to the OpenFlow Controller when such exposure is necessary or useful.

1.3 API Goals

The OpenFlow HW Abstraction API has the following goals:

- **Protect Proprietary Code:** Vendor proprietary code may be written to support this API and released as a binary to protect the intellectual property of the vendor.
- **Insulate OpenFlow Switch Code:** Allow the OpenFlow Switch software components to be written independent of the underlying hardware. This will facilitate porting the OpenFlow Switch code to new hardware.
- **Insulate Vendor Code:** When a new version of OpenFlow is released, this API should permit the reuse of existing HW drivers which support this API. This is particularly important if the HW driver is released as a binary object.
- **Flexible Deployments:** In environments such as Linux with a user/kernel space distinction, the API should function on either side of the division or across it. More generally, the API should not impede the creation of library to support the functionality.

1.4 API Requirements

The API must support the following operations:

- **Flow Table Maintenance** The API must allow the insertion, deletion and modification of flow descriptions used in the hardware forwarding Datapath.
- **Packet Receive and Transmit** The API must support packet reception from and transmission to the hardware.
- **Port Status and Control** The API must allow the control of physical ports and to allow the gathering of their status. Some additional support for hardware port characteristics may need to be exposed (for example, trunk state). Since this is likely to be highly vendor specific, it must be done in a generic and extensible manner.
- **Statistics Gathering** The API must allow the gathering of per-flow and per-port statistics.
- **Expose Vendor Features** The API must provide an extensible mechanism allowing the exposure of vendor specific features to an OpenFlow Controller.

1.5 Assumptions

- **Limited state below the OpenFlow API.** The hardware API implementation should be assumed to maintain no state about flows other than having stored the matches and actions in its tables. For example, it will have no information about what flow descriptions may be stored in the software Flow Table.
- **Limited capacity.** It should be assumed that the hardware has limited capacity and may not be able to install a particular flow.

-
- **Capacity estimates.** Any capacity reported by the hardware (such as number of supported flows) should be considered an estimate and not a firm limit. Note in particular the hardware resources required to install one particular flow may vary according to the details of the flow description.

2 The OpenFlow API Description

2.1 API Overview

Keeping in mind the API requirements above, the API is broken into two parts.

- **of_table:** This API exposes the hardware support for the insertion and maintenance of flow definitions. This is meant to provide an abstraction of how hardware implements flow descriptions.
- **of_port:** This API provides an abstraction of the interface through which packets flow. It provides the means of sending and receiving packets, controlling the port and reporting the status of the port.

C language data structures, enumerated types and function prototypes (function hooks) are used to describe the API.

2.2 The Packet Abstraction

The packet abstraction is very simple. There is a pointer to a monolithic databuffer, an integer length and an opaque pointer to an OS specific representation of the packet.

Currently no allocation or free routines are specified.

2.3 The Port Abstraction

The OpenFlow representation of a port is a small integer. It is currently 1-based in 0.9.

We make no assumption about whether the port refers to a specific physical port or to some virtualization such as a VLAN on a physical port or an aggregation of physical ports.

Currently port objects are “stateless” in the sense that no structure representing the state of the port is maintained. Queries regarding the state of the port are done through function calls.

The port object is simply a driver table providing function calls to create and manipulate port objects. Creating a port binds an OpenFlow number to the port. Once created, this number is used to identify the port for future calls.

2.4 Conventions

Unless otherwise noted, function calls will return an error code with 0 meaning success and a value < 0 in the case of an error. Error codes may be hardware specific, although a set of basic error codes are defined in `types.h`.

The file names for the OpenFlow Hardware API are generic, such as `types.h`. Thus it is important that references to these files in `#include` C directives use the directory name, currently `ofhw-abs`.

2.5 Putting the Pieces Together

Here are the high level order requirements for the initialization of the components.

- **Datapath Setup:** The datapath component is setup and exposes its functions for the asynchronous events: flow-delete, port-change and packet-in. It exposes its hooks for port-attach and table-attach.
- **Port Setup:** The port component is setup. It sets the hooks in the datapath for port-attach.
- **Table Setup:** The table component is setup. It sets the hook in the datapath for table-create.
- **Datapath Initialization:** The datapath initialization function is called. It verifies that the port- and table-create hooks are instantiated.
 - The port-create function is called by datapath. The port component is initialized, instantiating the hardware ports and binding with OS objects if necessary. Datapath receives the port driver structure back from the call.
 - The table-create function is called by datapath. The table component is initialized, setting up the hardware table as necessary. Datapath receives the table driver structure back from the call.
- The datapath initializes any default flows needed for the configuration (such as an entry for force all packets to the CPU).
- The datapath is told by an external mechanism to open port objects; it calls the open function from the port driver.

3 The API Definitions

3.1 Basic Types

The following basic types are defined in `types.h`.

3.2 Datapath Exported Functions

The following functions are exported `datapath.h`.

3.3 The Port Driver

Type/Function	Member/Param	Notes
os_pkt_t		Opaque reference to OS packet object
of_packet_t		OpenFlow packet structure
	data	Pointer to start of packet data
	length	Length of packet data in bytes
of_port_t		OpenFlow port type: uint32_t
of_datapath_t		Opaque reference to OpenFlow datapath
of_table_attach_f		Function to attach a table driver to datapath
	dp	Pointer to controlling datapath
	Returns	Pointer to flow table driver structure (see below)
of_port_attach_f		Function to attach a port driver to datapath
	dp	Pointer to controlling datapath
	Returns	Pointer to port driver structure (see below)
of_flow_id_t		Experimental flow id (uint32_t)

Table 1: Basic Types

Function	Param	Notes
of_table_flow_delete		Notify the datapath that a table has deleted a flow entry
	dp	Reference to controlling datapath
	table	Pointer to table structure from which flow was deleted
	flow	Pointer to flow object that was deleted from the table
	reason	Indicates why flow was removed (details TBD)
	Returns	void
of_port_change		Notify the datapath that a port state change has occurred
	dp	Reference to controlling datapath
	port	Reference to the port with the state change
	state	Indication of new state. details TBD
	Returns	void
of_port_packet_in		Forward a received packet to the datapath
	dp	Reference to controlling datapath
	port	Reference to the RX port
	packet	Pointer to the OpenFlow packet object
	reason	Indicates why the packet was received (details TBD)
	flow_id	Experimental: What flow matched the packet
	Returns	TBD, but probably an indication of whether the packet is handled

Table 2: Datapath Functions

Function	Param	Notes
open		Bind a HW port to an OpenFlow port number and initialize it
	name	String to identify the intended hardware port to open
	of_port	OpenFlow port identifier to bind to port
	enabled	Should the port be enabled after initialization (TBD)
	Returns	0 on success, or a negative error code
close		Remove the port from OpenFlow processing
	of_port	OpenFlow port identifier to close
	Returns	0 on success, or a negative error code
packet_send		Send the packet out the port
	of_port	Forward packet to this OpenFlow port
	pkt	Pointer to OpenFlow packet structure to forward
	Returns	0 on success, or a negative error code
enable_set		Enable/disable a port from receiving and forwarding packets
	of_port	OpenFlow port to control
	enabled	True/false to enable/disable port
	Returns	0 on success, or a negative error code
enable_get		Query port enable state
	of_port	OpenFlow port to query
	Returns	0 if port is disabled, 1 if enabled, or a negative error code
link_get		Query port link state
	of_port	OpenFlow port to query
	Returns	0 if link is down, 1 if up, or a negative error code
stats_get		Query port statistics
	of_port	OpenFlow port to query
	stats	Pointer to stats structure to be filled in
	Returns	0 on success, or a negative error code
stats_clear		OPTIONAL: Clear the statistics state
	of_port	OpenFlow port to control
	Returns	0 on success, or a negative error code
ioctl		OPTIONAL: Port control extensions; TBD Should more IO be supported?
	of_port	OpenFlow port to control
	request	Code indicating operation
	io_param	Pointer to <code>uint32_t</code> parameter
	Returns	0 on success, or a negative error code

Table 3: The `of_port_driver` Structure Functions

3.4 The Flow Data Type

The flow data type derives from the OpenFlow spec. It references `ofp_action_header` and `ofp_match` from `openflow.h`. The structure of `_action_list_t` is a length (in bytes) followed by an array of actions. The key structure is `of_flow_t` given by the following:

```
struct of_flow_s {
    of_flow_match_t key;          /* Match criteria */
    of_flow_id_t flow_id;        /* ID for this flow (experimental) */
    of_action_list_t *actions;   /* Actions for matching packet */

    uint16_t priority;          /* Relative priority of this flow */
    of_duration_t idle_timeout; /* Idle time before discarding */
    of_duration_t hard_timeout; /* Hard expiration time */
    of_time_t created;          /* When the flow was created */

    /* Dynamic information */
    uint32_t flow_state;         /* Bitmap of OFFS_ flags */
    of_time_t last_used;        /* Last used time. */
    uint64_t packet_count;      /* Stats for the flow */
    uint64_t byte_count;

    hw_flow_cookie_t hw_cookie; /* HW layer implementation cookie */
};
```

3.5 The Table Driver

The table driver consists of a set of functions to manipulate and configure the flow table representing the hardware datapath.

One important note that varies from previous implementations: The lower level driver for the table is responsible for timeout processing of expired flow entries. The evicted entries will be communicated to the datapath through the previously mentioned `of_table_flow_delete` function.

The table statistics structure is quite simple.

```
typedef struct of_table_stats_s {
    unsigned int n_flows;        /* Current active flows. */
    unsigned long int n_inserts; /* Flows inserted (total) */
    unsigned long int n_lookup;  /* Packets looked up. */
    unsigned long int n_matched; /* Packets that matched in table. */
} of_table_stats_t;
```

The following flags indicate the table capabilities.

```
OFTC_ITER          /* Supports iteration */
```

```
OFTC_FLOW_PACKET    /* Supports packet counters per flow */
OFTC_FLOW_BYTE      /* Supports byte counters per flow */
```

The capabilities are exchanged with the following structure.

```
/* Static table capabilities/info structure */
typedef struct of_table_caps_s {
    const char *name;          /* Optional: Human-readable name. */
    uint32_t wc_supported;     /* Bitmap of OFPFW_* supported wildcards */
    uint32_t actions_supported; /* Bitmap of OFPAT_* supported actions */
    unsigned int max_flows;    /* Flow capacity (estimate). */
    uint32_t flags;           /* Bitmap of OFTC_* flags */
} of_table_caps_t;
```

In addition to a table capabilities structure, the driver structure returned by the `of_table_attach` has the following function calls.

Function	Params	Notes
insert		Inserts flow into table, replacing any duplicate flow.
	table	Pointer to table structure identifying table
	flow	Flow description of flow to be inserted
	Returns	0 on success, or a negative error code: RESOURCE, MATCH or ACTION
modify		Modifies the actions for flows in table that match key.
	table	Pointer to table structure identifying table
	key	Key of flows to match to be modified
	priority	See strict
	strict	Boolean: If set, wildcards and priority must match
	actions	List of actions to replace for matching flows
	Returns	The number of flows modified, or a negative error code
delete		Delete the flows matching the given key
	table	Pointer to table structure identifying table
	key	Key of flows to match to be deleted
	out_port	If not OFPP_NONE, match only entries with output action to this port
	priority	See strict
	strict	Boolean: If set, wildcards and priority must match
	Returns	The number of flows deleted, or a negative error code
flow_stats_get		Get the statistics for a given flow
	table	Pointer to table structure identifying table
	flow	Flow description to match; statistics members in this struct are updated
	Returns	0 on success, or a negative error code
table_stats_get		Get the statistics for a given table
	table	Pointer to table structure identifying table
	stats	Structure to be filled out with table statistics
	Returns	0 on success, or a negative error code
destroy		Remove all flows from table and deactivate table
	table	Pointer to table structure identifying table
iterate		OPTIONAL: Iterate across flow entries in the table
	table	Pointer to table structure identifying table to search
	key	If non NULL, a pointer to a flow description; only make callback on matching flows
	out_port	If a valid port number, only make callback flows with an output action to this port
	position	Used to track position across multiple calls of iterate. Set to 0 on initial call.
	callback	Pointer to function with flow and cookie parameters that is called by the iteration; it should return 0 to continue the iteration
	private	Cookie passed to callback as second parameter
Returns	0 on success or non-zero if iteration did not complete	
lookup		OPTIONAL: Searches table for a flow matching key
	table	Pointer to table structure identifying table to search
	key	Pointer to match structure to search for
	Returns	Pointer to flow entry matching key if found; NULL otherwise
ioctl		OPTIONAL: Table control extensions; TBD Should more IO be supported?
	table	OpenFlow table to control
	request	Code indicating operation
	io_param	Pointer to uint32_t parameter; see Open Issues
	Returns	0 on success, or a negative error code

Table 4: The of_table Driver Functions

4 Open Questions

- **Asynchronous Flow Update Information Returned to Controller:** When the switch deletes flows, either through a timeout mechanism or when it is given a pattern of flows to delete, it only returns the number of flows deleted. For these APIs, a handle pointer is passed to the routine. If non-NULL, the low level routine is expected to malloc sufficient space to report the flows that were deleted by the operation. The caller will then free the object. (Alternatively, a “free” operation could be provided to insulate from OS specific memory management).
- **Support Polling:** Resolution: Polling will not be supported across the OpenFlow Hardware Abstraction API. The API will require the use of callback functions for asynchronous events. Lower level parts of the implementation may provide polling of the hardware and generate the necessary asynchronous events as needed. This implies that the OS facilities for polling (timers, threads) are available to the lower parts of the implementation.
- **Port Complications:** Hardware trunks, software trunks, port groups, virtual ports, VLANs
- **Network HW Support:** How generic should the target hardware support be? Can we assume Ethernet for now since OF does?
- **Spanning Tree Support:** Should spanning tree get and set (or, optionally, only get) be supported in the of_port module? If so, is it per-VLAN? Or should the notion of spanning-tree-group be introduced?
- **ioctl Parameters:** Assuming the port and table ioctl functions are supported (which is probably also an open question) what should the I/O parameters be? (1) Just an integer? (2) void * input? (3) void ** output handle malloc'd by driver?

5 Attributions

TBD: Fill in details of contributions.