

OPEN NETWORKING FOUNDATION

OpenFlow Switch Specification

Version 1.5.1 (Protocol version 0x06)

March 26, 2015

ONF TS-025



Copyright © 2015; Open Networking Foundation

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 `http://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.

 © 2015; The Open Networking Foundation

Contents

1 Introduction **11**

2 Switch Components **11**

3 Glossary **12**

4 OpenFlow Ports **15**
4.1 OpenFlow Ports . . . 15
4.2 Standard Ports . . . 15
4.3 Physical Ports . . . 16
4.4 Logical Ports . . . 16
4.5 Reserved Ports . . . 16
4.6 Port changes . . . 17
4.7 Port recirculation . . . 18

5 OpenFlow Tables **18**
5.1 Pipeline Processing . . . 19
5.1.1 Pipeline Consistency . . . 21
5.2 Flow Tables and flow entries . . . 22
5.3 Matching . . . 22
5.4 Table-miss . . . 24
5.5 Instructions . . . 25
5.6 Action Set . . . 26
5.7 List of Actions . . . 28
5.8 Actions . . . 28
5.8.1 Default values for fields on push . . . 30
5.9 Counters . . . 31
5.10 Group Table . . . 33
5.10.1 Group Types . . . 33
5.10.2 Group Liveness Monitoring . . . 34
5.11 Meter Table . . . 34
5.11.1 Meter Bands . . . 35
5.12 Ingress and egress processing differences . . . 37

6 OpenFlow Channel and Control Channel **38**
6.1 OpenFlow Switch Protocol Overview . . . 38
6.1.1 Controller-to-Switch . . . 38
6.1.2 Asynchronous . . . 39
6.1.3 Symmetric . . . 40
6.2 Message Handling . . . 40
6.3 OpenFlow Channel Connections . . . 41
6.3.1 Connection URI . . . 42
6.3.2 Alternate connection transports . . . 43
6.3.3 Connection Setup . . . 43
6.3.4 Connection Maintenance . . . 44
6.3.5 Connection Interruption . . . 45

 © 2015; The Open Networking Foundation

6.3.6 Encryption . . . 46
6.3.7 Multiple Controllers . . . 46
6.3.8 Auxiliary Connections . . . 49
6.4 Flow Table Modification Messages . . . 51
6.5 Flow Removal . . . 52
6.6 Flow Table Synchronisation . . . 53
6.7 Group Table Modification Messages . . . 55
6.8 Meter Modification Messages . . . 57
6.9 Bundle Messages . . . 57
6.9.1 Bundle overview . . . 57
6.9.2 Bundle example usage . . . 58
6.9.3 Bundle error processing . . . 58
6.9.4 Bundle atomic modifications . . . 59
6.9.5 Bundle parallelism . . . 59
6.9.6 Scheduled Bundles . . . 60
6.9.6.1 The Scheduled Bundle Procedure . . . 60
6.9.6.2 Discarding Scheduled Bundles . . . 61
6.9.6.3 Timekeeping and Synchronization . . . 61
6.9.6.4 Scheduling Tolerance . . . 62

7 The OpenFlow Switch Protocol 62
7.1 Protocol basic format . . . 63
7.1.1 OpenFlow Header . . . 63
7.1.2 Padding . . . 65
7.1.3 Reserved and unsupported values and bit positions . . . 65
7.2 Common Structures . . . 66
7.2.1 Port Structures . . . 66
7.2.1.1 Port Description Structures . . . 66
7.2.1.2 Port Description Properties . . . 68
7.2.2 Header Types Structure . . . 72
7.2.3 Flow Match Structures . . . 74
7.2.3.1 Flow Match Header . . . 74
7.2.3.2 Flow Match Field Structures . . . 75
7.2.3.3 OXM classes . . . 75
7.2.3.4 Flow Matching . . . 76
7.2.3.5 Flow Match Field Masking . . . 77
7.2.3.6 Flow Match Field Prerequisite . . . 77
7.2.3.7 Flow Match Fields . . . 78
7.2.3.8 Header Match Fields . . . 80
7.2.3.9 Pipeline Match Fields . . . 83
7.2.3.10 Packet registers . . . 85
7.2.3.11 Packet Type Match Field . . . 85
7.2.3.12 Experimenter Flow Match Fields . . . 86
7.2.4 Flow Stats Structures . . . 87
7.2.4.1 Flow Stats Header . . . 87
7.2.4.2 Flow Stat Field Structures . . . 87
7.2.4.3 OXS classes . . . 88

 © 2015; The Open Networking Foundation

7.2.4.4 Flow Stat Fields . . . 89
7.2.4.5 Experimenter Flow Stat Fields . . . 90
7.2.5 Flow Instruction Structures . . . 90
7.2.6 Action Structures . . . 93
7.2.6.1 Output Action Structures . . . 94
7.2.6.2 Group Action Structures . . . 95
7.2.6.3 Set-Queue Action Structure . . . 95
7.2.6.4 Meter Action Structure . . . 95
7.2.6.5 TTL Actions Structures . . . 96
7.2.6.6 Push and Pop Actions Structures . . . 97
7.2.6.7 Set-Field Action Structure . . . 98
7.2.6.8 Copy-Field Action Structure . . . 99
7.2.6.9 Experimenter Action Structure . . . 100
7.2.7 Controller Status Structure . . . 101
7.2.8 Experimenter Structure . . . 103
7.3 Controller-to-Switch Messages . . . 104
7.3.1 Handshake . . . 104
7.3.2 Switch Configuration . . . 105
7.3.3 Flow Table Configuration . . . 106
7.3.4 Modify State Messages . . . 106
7.3.4.1 Modify Flow Table Message . . . 106
7.3.4.2 Modify Flow Entry Message . . . 110
7.3.4.3 Modify Group Entry Message . . . 113
7.3.4.4 Port Modification Message . . . 118
7.3.4.5 Meter Modification Message . . . 120
7.3.5 Multipart Messages . . . 123
7.3.5.1 Description . . . 127
7.3.5.2 Individual Flow Descriptions . . . 127
7.3.5.3 Individual Flow Statistics . . . 129
7.3.5.4 Aggregate Flow Statistics . . . 130
7.3.5.5 Port Statistics . . . 131
7.3.5.6 Port Description . . . 134
7.3.5.7 Queue Statistics . . . 135
7.3.5.8 Queue Descriptions . . . 136
7.3.5.9 Group Statistics . . . 138
7.3.5.10 Group Description . . . 139
7.3.5.11 Group Features . . . 140
7.3.5.12 Meter Statistics . . . 141
7.3.5.13 Meter Descriptions . . . 142
7.3.5.14 Meter Features . . . 143
7.3.5.15 Controller Status Multipart . . . 144
7.3.5.16 Table Statistics . . . 144
7.3.5.17 Table Description . . . 144
7.3.5.18 Table Features . . . 145
7.3.5.19 Flow Monitoring . . . 153
7.3.5.20 Bundle Features Multipart . . . 159
7.3.5.21 Experimenter Multipart . . . 162

 © 2015; The Open Networking Foundation

7.3.6 Packet-Out Message . . . 162
7.3.7 Barrier Message . . . 164
7.3.8 Role Request Message . . . 164
7.3.9 Bundle messages . . . 165
7.3.9.1 Bundle control messages . . . 165
7.3.9.2 Bundle Add message . . . 166
7.3.9.3 Bundle flags . . . 167
7.3.9.4 Bundle properties . . . 167
7.3.9.5 Time Format . . . 168
7.3.9.6 Creating and opening a bundle . . . 169
7.3.9.7 Adding messages to a bundle . . . 169
7.3.9.8 Closing a bundle . . . 170
7.3.9.9 Committing Bundles . . . 171
7.3.9.10 Discarding Bundles . . . 172
7.3.9.11 Other bundle error conditions . . . 172
7.3.10 Set Asynchronous Configuration Message . . . 173
7.4 Asynchronous Messages . . . 175
7.4.1 Packet-In Message . . . 175
7.4.2 Flow Removed Message . . . 177
7.4.3 Port Status Message . . . 178
7.4.4 Controller Role Status Message . . . 179
7.4.5 Table Status Message . . . 181
7.4.6 Request Forward Message . . . 181
7.4.7 Controller Status Message . . . 182
7.5 Symmetric Messages . . . 182
7.5.1 Hello . . . 182
7.5.2 Echo Request . . . 184
7.5.3 Echo Reply . . . 184
7.5.4 Error Message . . . 184
7.5.4.1 Hello Failed error codes . . . 185
7.5.4.2 Bad Request error codes . . . 186
7.5.4.3 Bad Action error codes . . . 187
7.5.4.4 Bad Instruction error codes . . . 189
7.5.4.5 Bad Match error codes . . . 190
7.5.4.6 Flow-mod Failed error codes . . . 191
7.5.4.7 Group-mod Failed error codes . . . 192
7.5.4.8 Port-mod Failed error codes . . . 194
7.5.4.9 Table-mod Failed error codes . . . 194
7.5.4.10 Queue-op Failed error codes . . . 194
7.5.4.11 Switch Config Failed error codes . . . 195
7.5.4.12 Role Request Failed error codes . . . 195
7.5.4.13 Meter-mod Failed error codes . . . 195
7.5.4.14 Table-features Failed error codes . . . 196
7.5.4.15 Bad Property error codes . . . 197
7.5.4.16 Async Config Failed error codes . . . 197
7.5.4.17 Flow-monitor Failed error codes . . . 197
7.5.4.18 Bundle Failed error codes . . . 198

 © 2015; The Open Networking Foundation

7.5.4.19 Experimenter Error Message . . . 199
7.5.5 Experimenter Message . . . 200

A Header file openflow.h 200

B Release Notes 242
B.1 OpenFlow version 0.2.0 . . . 242
B.2 OpenFlow version 0.2.1 . . . 243
B.3 OpenFlow version 0.8.0 . . . 243
B.4 OpenFlow version 0.8.1 . . . 243
B.5 OpenFlow version 0.8.2 . . . 243
B.6 OpenFlow version 0.8.9 . . . 243
B.6.1 IP Netmasks . . . 244
B.6.2 New Physical Port Stats . . . 244
B.6.3 IN_PORT Virtual Port . . . 245
B.6.4 Port and Link Status and Configuration . . . 245
B.6.5 Echo Request/Reply Messages . . . 245
B.6.6 Vendor Extensions . . . 246
B.6.7 Explicit Handling of IP Fragments . . . 246
B.6.8 802.1D Spanning Tree . . . 247
B.6.9 Modify Actions in Existing Flow Entries . . . 247
B.6.10 More Flexible Description of Tables . . . 248
B.6.11 Lookup Count in Tables . . . 248
B.6.12 Modifying Flags in Port-Mod More Explicit . . . 248
B.6.13 New Packet-Out Message Format . . . 248
B.6.14 Hard Timeout for Flow Entries . . . 249
B.6.15 Reworked initial handshake to support backwards compatibility . . . 250
B.6.16 Description of Switch Stat . . . 250
B.6.17 Variable Length and Vendor Actions . . . 251
B.6.18 VLAN Action Changes . . . 252
B.6.19 Max Supported Ports Set to 65280 . . . 252
B.6.20 Send Error Message When Flow Not Added Due To Full Tables . . . 252
B.6.21 Behavior Defined When Controller Connection Lost . . . 253
B.6.22 ICMP Type and Code Fields Now Matchable . . . 253
B.6.23 Output Port Filtering for Delete*, Flow Stats and Aggregate Stats . . . 253
B.7 OpenFlow version 0.9 . . . 254
B.7.1 Failover . . . 254
B.7.2 Emergency Flow Cache . . . 254
B.7.3 Barrier Command . . . 254
B.7.4 Match on VLAN Priority Bits . . . 254
B.7.5 Selective Flow Expirations . . . 254
B.7.6 Flow Mod Behavior . . . 255
B.7.7 Flow Expiration Duration . . . 255
B.7.8 Notification for Flow Deletes . . . 255
B.7.9 Rewrite DSCP in IP ToS header . . . 255
B.7.10 Port Enumeration now starts at 1 . . . 255
B.7.11 Other changes to the Specification . . . 255

 © 2015; The Open Networking Foundation

B.8 OpenFlow version 1.0 . . . 256
B.8.1 Slicing . . . 256
B.8.2 Flow cookies . . . 256
B.8.3 User-specifiable datapath description . . . 256
B.8.4 Match on IP fields in ARP packets . . . 256
B.8.5 Match on IP ToS/DSCP bits . . . 256
B.8.6 Querying port stats for individual ports . . . 256
B.8.7 Improved flow duration resolution in stats/expiry messages . . . 257
B.8.8 Other changes to the Specification . . . 257
B.9 OpenFlow version 1.1 . . . 257
B.9.1 Multiple Tables . . . 257
B.9.2 Groups . . . 258
B.9.3 Tags : MPLS & VLAN . . . 258
B.9.4 Virtual ports . . . 259
B.9.5 Controller connection failure . . . 259
B.9.6 Other changes . . . 259
B.10 OpenFlow version 1.2 . . . 260
B.10.1 Extensible match support . . . 260
B.10.2 Extensible 'set_field' packet rewriting support . . . 260
B.10.3 Extensible context expression in 'packet-in' . . . 260
B.10.4 Extensible Error messages via experimenter error type . . . 261
B.10.5 IPv6 support added . . . 261
B.10.6 Simplified behaviour of flow-mod request . . . 261
B.10.7 Removed packet parsing specification . . . 261
B.10.8 Controller role change mechanism . . . 261
B.10.9 Other changes . . . 262
B.11 OpenFlow version 1.3 . . . 262
B.11.1 Refactor capabilities negotiation . . . 262
B.11.2 More flexible table miss support . . . 263
B.11.3 IPv6 Extension Header handling support . . . 263
B.11.4 Per flow meters . . . 263
B.11.5 Per connection event filtering . . . 264
B.11.6 Auxiliary connections . . . 264
B.11.7 MPLS BoS matching . . . 264
B.11.8 Provider Backbone Bridging tagging . . . 265
B.11.9 Rework tag order . . . 265
B.11.10 Tunnel-ID metadata . . . 265
B.11.11 Cookies in packet-in . . . 265
B.11.12 Duration for stats . . . 265
B.11.13 On demand flow counters . . . 266
B.11.14 Other changes . . . 266
B.12 OpenFlow version 1.3.1 . . . 266
B.12.1 Improved version negotiation . . . 266
B.12.2 Other changes . . . 266
B.13 OpenFlow version 1.3.2 . . . 267
B.13.1 Changes . . . 267
B.13.2 Clarifications . . . 267

 © 2015; The Open Networking Foundation

B.14 OpenFlow version 1.3.3 . . . 268
B.14.1 Changes . . . 268
B.14.2 Clarifications . . . 268
B.15 OpenFlow version 1.3.4 . . . 269
B.15.1 Changes . . . 269
B.15.2 Clarifications . . . 269
B.16 OpenFlow version 1.3.5 . . . 270
B.16.1 Changes . . . 270
B.16.2 Clarifications . . . 271
B.17 OpenFlow version 1.4.0 . . . 272
B.17.1 More extensible wire protocol . . . 272
B.17.2 More descriptive reasons for packet-in . . . 272
B.17.3 Optical port properties . . . 273
B.17.4 Flow-removed reason for meter delete . . . 273
B.17.5 Flow monitoring . . . 273
B.17.6 Role status events . . . 274
B.17.7 Eviction . . . 274
B.17.8 Vacancy events . . . 274
B.17.9 Bundles . . . 274
B.17.10Synchronised tables . . . 275
B.17.11Group and Meter change notifications . . . 275
B.17.12Error code for bad priority . . . 275
B.17.13Error code for Set-async-config . . . 275
B.17.14PBB UCA header field . . . 276
B.17.15Error code for duplicate instruction . . . 276
B.17.16Error code for multipart timeout . . . 276
B.17.17Change default TCP port to 6653 . . . 276
B.18 OpenFlow version 1.4.1 . . . 276
B.18.1 Changes . . . 276
B.18.2 Clarifications . . . 277
B.19 OpenFlow version 1.5.0 . . . 277
B.19.1 Egress Tables . . . 277
B.19.2 Packet Type aware pipeline . . . 278
B.19.3 Extensible Flow Entry Statistics . . . 278
B.19.4 Flow Entry Statistics Trigger . . . 279
B.19.5 Copy-Field action to copy between two OXM fields . . . 279
B.19.6 Packet Register pipeline fields . . . 279
B.19.7 TCP flags matching . . . 279
B.19.8 Group command for selective bucket operation . . . 279
B.19.9 Allow set-field action to set metadata field . . . 280
B.19.10Allow wildcard to be used in set-field action . . . 280
B.19.11Scheduled Bundles . . . 280
B.19.12Controller connection status . . . 280
B.19.13Meter action . . . 281
B.19.14Enable setting all pipeline fields in packet-out . . . 281
B.19.15Port properties for pipeline fields . . . 281
B.19.16Port property for recirculation . . . 281

 © 2015; The Open Networking Foundation

B.19.17Clarify and improve barrier . . . 281
B.19.18Always generate port status on port config change . . . 282
B.19.19Make all Experimenter OXM-IDs 64 bits . . . 282
B.19.20Unified requests for group, port and queue multiparts . . . 282
B.19.21Rename some type for consistency . . . 282
B.19.22Specification reorganisation . . . 283
B.20 OpenFlow version 1.5.1 . . . 283
B.20.1 Changes . . . 283
B.20.2 Clarifications . . . 283

C Credits **283**

List of Tables

1 Main components of a flow entry in a flow table. . . . 22
2 Push/pop tag actions. . . . 29
3 Change-TTL actions. . . . 30
4 Existing fields that may be copied into new fields on a push action. . . . 31
5 List of counters. . . . 32
6 Main components of a group entry in the group table. . . . 33
7 Main components of a meter entry in the meter table. . . . 35
8 Main components of a meter band in a meter entry. . . . 36
9 Common Canonical Header Types. . . . 73
10 OXM TLV header fields. . . . 75
11 OXM mask and value. . . . 77
12 Required match fields. . . . 80
13 Header match fields details. . . . 81
14 Match combinations for VLAN tags. . . . 82
15 Pipeline match fields details. . . . 83
16 Packet Types. . . . 85
17 OXS TLV header fields. . . . 88
18 Stat fields details. . . . 89

List of Figures

1 Main components of an OpenFlow switch. . . . 11
2 Packet flow through the processing pipeline. . . . 19
3 Simplified flowchart detailing packet flow through an OpenFlow switch. . . . 23
4 Matching and Instruction execution in a flow table. . . . 25
5 Meters and Hierarchical DSCP metering. . . . 35
6 Scheduled Bundle Procedure . . . 60
7 Discarding a Scheduled Commit . . . 61
8 Scheduling Tolerance . . . 62
9 OXM TLV header layout. . . . 75
10 OXS TLV header layout. . . . 88

 © 2015; The Open Networking Foundation

1 Introduction

This document describes the requirements of an OpenFlow Logical Switch. Additional information describing OpenFlow and Software Defined Networking is available on the Open Networking Foundation website (`https://www.opennetworking.org/`). This specification covers the components and the basic functions of the switch, and the OpenFlow switch protocol to manage an OpenFlow switch from a remote OpenFlow controller.

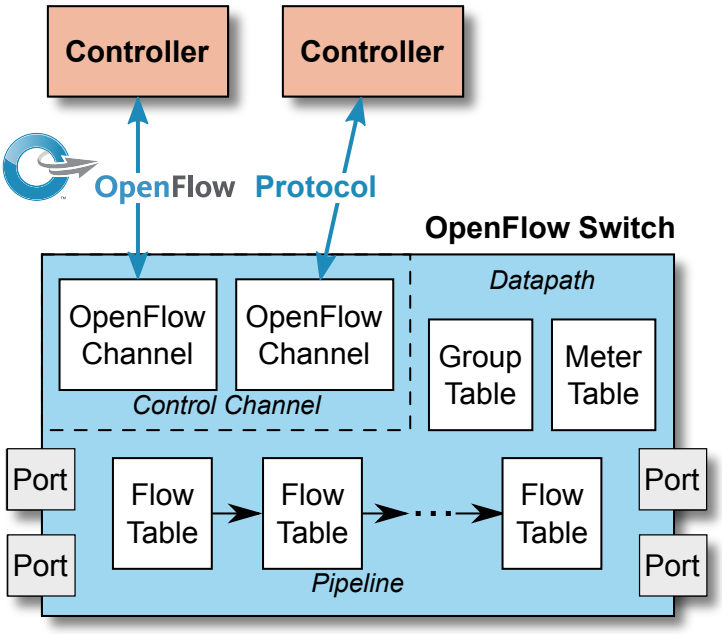


Figure 1: Main components of an OpenFlow switch.

2 Switch Components

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

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

Matching starts at the first flow table and may continue to additional flow tables of the pipeline (see 5.1). Flow entries match packets in priority order, with the first matching entry in each table being used (see 5.3). If a matching entry is found, the instructions associated with the specific flow entry are executed (see 5.5). If no match is found in a flow table, the outcome depends on configuration of the

 © 2015; The Open Networking Foundation

table-miss flow entry: for example, the packet may be forwarded to the controllers over the OpenFlow channel, dropped, or may continue to the next flow table (see 5.4).

Instructions associated with each flow entry either contain actions or modify pipeline processing (see 5.5). Actions included in instructions describe packet forwarding, packet modification and group table processing. Pipeline processing instructions allow packets to be sent to subsequent tables for further processing and allow information, in the form of metadata, to be communicated between tables. Table pipeline processing stops when the instruction set associated with a matching flow entry does not specify a next table; at this point the packet is usually modified and forwarded (see 5.6).

Flow entries may forward to a *port*. This is usually a physical port, but it may also be a logical port defined by the switch or a reserved port defined by this specification (see 4.1). Reserved ports may specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as "normal" switch processing (see 4.5), while switch-defined logical ports may specify link aggregation groups, tunnels or loopback interfaces (see 4.4).

Actions associated with flow entries may also direct packets to a group, which specifies additional processing (see 5.10). Groups represent sets of actions for flooding, as well as more complex forwarding semantics (e.g. multipath, fast reroute, and link aggregation). As a general layer of indirection, groups also enable multiple flow entries to forward to a single identifier (e.g. IP forwarding to a common next hop). This abstraction allows common output actions across flow entries to be changed efficiently.

The group table contains group entries; each group entry contains a list of *action buckets* with specific semantics dependent on group type (see 5.10.1). The actions in one or more action buckets are applied to packets sent to the group.

Switch designers are free to implement the internals in any way convenient, provided that correct match and instruction semantics are preserved. For example, while a flow entry may use an all group to forward to multiple ports, a switch designer may choose to implement this as a single bitmask within the hardware forwarding table. Another example is matching; the pipeline exposed by an OpenFlow switch may be physically implemented with a different number of hardware tables.

3 Glossary

This section describes key OpenFlow specification terms. Most terms are specific to this specification.

- **Action**: an operation that acts on a packet. An action may forward the packet to a port, modify the packet (such as decrementing the TTL field) or change its state (such as associating it with a queue). Most actions include parameters, for example a set-field action includes a field type and field value. Actions may be specified as part of the instruction set associated with a flow entry or in an action bucket associated with a group entry. Actions may be accumulated in the Action Set of the packet or applied immediately to the packet (see 5.8).
- **List of Actions**: an ordered list of actions that may be included in a flow entry in the *Apply-Actions* instruction or in a packet-out message, and that are executed immediately in the list order (see 5.7). Actions in a list can be duplicated, their effects are cumulative.

 © 2015; The Open Networking Foundation

- **Set of Actions**: a set of actions included in a flow entry in the *Write-Actions* instruction that are added to the action set, or in a group action-bucket that are executed in action-set order (see 5.6). Actions in a set can occur only once.
- **Action Bucket**: a set of actions in a group. The group will select one or more buckets for each packet.
- **Action Set**: a set of actions associated with the packet that are accumulated while the packet is processed by each table and that are executed in specified order when the instruction set terminates pipeline processing (see 5.6).
- **Byte**: an 8-bit octet.
- **Connection**: a network connection that carries OpenFlow messages between a switch and a controller, it may be implemented using various network transport protocols (see 6.3). An OpenFlow channel has a main connection, and optionally a number of auxiliary connections (see 6.3.8).
- **Control Channel**: The aggregation of components of an OpenFlow logical switch that manages communication with controllers. The control channel includes one OpenFlow channel per OpenFlow controller.
- **Controller**: see OpenFlow controller.
- **Counter**: counters are the main element of OpenFlow statistics and accumulated at various specific points of the pipeline, such as on a port or on a flow entry (see 5.9). Counters typically count the number of packets and bytes passing through an OpenFlow element, however other counters types are also defined.
- **Datapath**: the aggregation of components of an OpenFlow logical switch that are directly involved in traffic processing and forwarding. The datapath includes the pipeline of flow tables, the group table and the ports.
- **Flow Entry**: an element in a flow table used to match and process packets. It contains a set of match fields for matching packets, a priority for matching precedence, a set of counters to track packets, and a set of instructions to apply (see 5.2).
- **Flow Table**: a stage of the pipeline. It contains flow entries.
- **Forwarding**: Deciding the output port or set of output ports for a packet, and transfering that packet to those output ports.
- **Group**: a list of action buckets and some means of choosing one or more of those buckets to apply on a per-packet basis (see 5.10).
- **Header**: control information embedded in a packet used by a switch to identify the packet and to inform the switch on how to process and forward the packet. The header typically includes various header fields to identify the source and destination of the packet, and how to interpret other headers and the payload.
- **Header Field**: a value from the packet header. The packet header is parsed to extract its header fields which are matched against corresponding match fields.
- **Hybrid**: integrate both OpenFlow operation and *normal* Ethernet switching operation (see 5.1).

 © 2015; The Open Networking Foundation

- **Instruction**: instructions are attached to a flow entry and describe the OpenFlow processing that happens when a packet matches the flow entry. An instruction either modifies pipeline processing, such as directing the packet to another flow table, or contains a *set* of actions to add to the action set, or contains a *list* of actions to apply immediately to the packet (see 5.5).
- **Instruction Set**: a set of instructions attached to a flow entry in a flow table.
- **Match Field**: a part of a flow entry against which a packet is matched. Match fields can match the various packet header fields (see 7.2.3.8), the packet ingress port, the metadata value and other pipeline fields (see 7.2.3.9). A match field may be wildcarded (match any value) and in some cases bitmasked (match subset of bits).
- **Matching**: comparing the set of header fields and pipeline fields of a packet to the match fields of a flow entry (see 5.3).
- **Metadata**: a maskable register that is used to carry information from one table to the next.
- **Message**: OpenFlow protocol unit sent over an OpenFlow connection. May be a request, a reply, a control message or a status event.
- **Meter**: a switch element that can measure and control the rate of packets. The meter triggers a meter band if the packet rate or byte rate passing through the meter exceeds a predefined threshold (see 5.11). If the meter band drops the packet, it is called a **Rate Limiter**.
- **OpenFlow Channel**: interface between an OpenFlow switch and an OpenFlow controller, used by the controller to manage the switch.
- **OpenFlow Controller**: an entity interacting with the OpenFlow switch using the OpenFlow switch protocol. In most case, an OpenFlow Controller is software which controls many OpenFlow Logical Switches.
- **OpenFlow Logical Switch**: A set of OpenFlow resources that can be managed as a single entity, includes a datapath and a control channel.
- **OpenFlow Protocol**: The protocol defined by this specification. Also called OpenFlow Switch Protocol.
- **OpenFlow Switch**: See OpenFlow Logical Switch.
- **Packet**: a series of bytes comprising a header, a payload and optionally a trailer, in that order, and treated as a unit for purposes of processing and forwarding. The default packet type is Ethernet, other packet types are also supported (see 7.2.3.11).
- **Pipeline**: the set of linked flow tables that provide matching, forwarding, and packet modification in an OpenFlow switch (see 5.1).
- **Pipeline fields**: set of values attached to the packet during pipeline processing which are not header fields. Include the ingress port, the metadata value, the Tunnel-ID value and others (see 7.2.3.9).
- **Port**: where packets enter and exit the OpenFlow pipeline (see 4.1). May be a physical port, a logical port, or a reserved port defined by the OpenFlow switch protocol.
- **Queue**: Schedule packets according to their priority on an output port to provide Quality-of-Service (QoS).

 © 2015; The Open Networking Foundation

- **Switch**: See OpenFlow Logical Switch.
- **Tag**: a header that can be inserted or removed from a packet via push and pop actions.
- **Outermost Tag**: the tag that appears closest to the beginning of a packet.

4 OpenFlow Ports

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

4.1 OpenFlow Ports

OpenFlow ports are the network interfaces for passing packets between OpenFlow processing and the rest of the network. OpenFlow switches connect logically to each other via their OpenFlow ports, a packet can be forwarded from one OpenFlow switch to another OpenFlow switch only via an output OpenFlow port on the first switch and an ingress OpenFlow port on the second switch.

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

OpenFlow packets are received on an **ingress port** and processed by the OpenFlow pipeline (see 5.1) which may forward them to an **output port**. The packet ingress port is a property of the packet throughout the OpenFlow pipeline and represents the OpenFlow port on which the packet was received into the OpenFlow switch. The ingress port can be used when matching packets (see 5.3). The OpenFlow pipeline can decide to send the packet on an output port using the output action (see 5.8), which defines how the packet goes back to the network.

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

4.2 Standard Ports

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

Standard ports can be used as ingress and output ports, they can be used in groups (see 5.10), they have port counters (see 5.9) and they have state and configuration (see 7.2.1).

 © 2015; The Open Networking Foundation

4.3 Physical Ports

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

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

4.4 Logical Ports

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

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

The only differences between *physical ports* and *logical ports* is that a packet associated with a logical port may have an extra pipeline field called *Tunnel-ID* associated with it (see 7.2.3.9) and when a packet received on a logical port is sent to the controller, both its logical port and its underlying physical port are reported to the controller (see 7.4.1).

4.5 Reserved Ports

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

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

- *Required:* **ALL:** Represents all ports the switch can use for forwarding a specific packet. Can be used only as an output port. In that case a copy of the packet starts egress processing on all standard ports, excluding the packet ingress port and ports that are configured `OFPPC_NO_FWD`.
- *Required:* **CONTROLLER:** Represents the control channel with the OpenFlow controllers. Can be used as an ingress port or as an output port. When used as an output port, encapsulates the packet in a *packet-in* message and sends it using the OpenFlow switch protocol (see 7.4.1). When used as an ingress port, this identifies a packet originating from the controller.
- *Required:* **TABLE:** Represents the start of the OpenFlow pipeline (see 5.1). This port is only valid in an output action in the list of actions of a *packet-out* message (see 7.3.6), and submits the packet to the first flow table so that the packet can be processed through the regular OpenFlow pipeline.
- *Required:* **IN_PORT:** Represents the packet ingress port. Can be used only as an output port, sends the packet out through its ingress port.

 © 2015; The Open Networking Foundation

- *Required:* **ANY:** Special value used in some OpenFlow requests when no port is specified (i.e. port is wildcarded). Some OpenFlow requests contain a reference to a specific port that the request only applies to. Using *ANY* as the port number in these requests allows that request instance to apply to any and all ports. Can neither be used as an ingress port nor as an output port.
- *Required:* **UNSET:** Special value to specify that the output port has not been set in the *Action-Set*. Only used when trying to match the output port in the action set using the `OXM_OF_ACTSET_OUTPUT` match field (see 7.2.3.7). Can neither be used as an ingress port nor as an output port.
- *Optional:* **LOCAL:** Represents the switch's local networking stack and its management stack. Can be used as an ingress port or as an output port. The local port enables remote entities to interact with the switch and its network services via the OpenFlow network, rather than via a separate control network. With an appropriate set of flow entries, it can be used to implement an in-band controller connection (this is outside the scope of this specification).
- *Optional:* **NORMAL:** Represents forwarding using the traditional non-OpenFlow pipeline of the switch (see 5.1). Can be used only as an output port and processes the packet using the normal pipeline. In general will bridge or route the packet, however the actual result is implementation dependent. If the switch cannot forward packets from the OpenFlow pipeline to the normal pipeline, it must indicate that it does not support this action.
- *Optional:* **FLOOD:** Represents flooding using the traditional non-OpenFlow pipeline of the switch (see 5.1). Can be used only as an output port, actual result is implementation dependent. In general will send the packet out all standard ports, but not to the ingress port, nor ports that are in `OFPPS_BLOCKED` state. The switch may also use the packet VLAN ID or other criteria to select which ports to use for flooding.

OpenFlow-only switches do not support the **NORMAL** port and **FLOOD** port, while *OpenFlow-hybrid* switches may support them (see 5.1). Forwarding packets to the **FLOOD** port depends on the switch implementation and configuration, while forwarding using a **group** of type *all* enables the controller to more flexibly implement flooding (see 5.10.1).

4.6 Port changes

A switch configuration, for example using the OpenFlow Configuration Protocol, may add or remove ports from the OpenFlow switch at any time. The switch may change the port state based on the underlying port mechanism, for example if the link is going down (see 7.2.1). The controller or a switch configuration may change the port configuration (see 7.2.1). Any such changes to port state or configuration must be communicated to the OpenFlow controller (see 7.4.3).

Port addition, modification or removal never changes the content of the flow tables, in particular flow entries referencing those ports are not modified or removed (flow entries may reference ports via the match or actions). Packets forwarded to non-existent ports are just dropped (see 5.6). Similarly, Port addition, modification and removal never changes the content of the group table, however the behaviour of some groups may change through liveness checking (see 6.7).

If a port is deleted and its port number is later reused for a different physical or logical port, any remaining flow entries or group entries still referencing that port number may be effectively re-targeted

 © 2015; The Open Networking Foundation

to the new port, possibly with undesirable results. Therefore, when a port is deleted it is left to the controller to clean up any flow entries or group entries referencing that port, if needed.

4.7 Port recirculation

Logical ports can optionally be used to insert a network service or complex processing in the OpenFlow switch (see 4.4). Most often, packets sent to logical ports never return to the same OpenFlow switch, they are either consumed by the logical port or eventually sent over a physical port. In other cases, packets sent to a logical port are recirculated back to the OpenFlow switch after the logical port processing.

Packet recirculation via logical ports is optional, and OpenFlow supports multiple types of port recirculation. The simplest recirculation is when a packet sent on a logical port returns back into the switch via the same logical port. This could be used for a loopback or unidirectional packet processing. Recirculation can also happen between a port pair, in which a packet sent on a logical port returns back into the switch via the other logical port of the pair. This could be used to represent tunnel endpoints or bidirectional packet processing. A port property describes the recirculation relationship between ports (see 7.2.1.2).

A switch should protect itself from packet infinite loops when using port recirculation. This mechanism is implementation-specific and outside the scope of this specification. For example, the switch could attach an internal recirculation count to each packet, which is incremented for each recirculation, and the switch drops packets for which the counter is above a switch-defined threshold. Controllers are strongly encouraged to avoid generating combinations of flow entries that may yield recirculation loops.

Because of the wide range of possible processing, very little can be assumed about the packets recirculated back into the switch. Recirculated packets go back to the first flow table of the pipeline (see 5.1) and can be identified by their new input port. The packet headers may have changed, so the match fields are not guaranteed to be the same. The logical port may do packet fragmentation and reassembly, so the packets may not match one to one and may have different sizes.

The *Tunnel-ID* field and some other pipeline fields associated with the packet may optionally be preserved through the recirculation and available for matching when returning to the switch, the pipeline fields that are preserved are indicated via the port match field property (see 7.2.1.2). If a pipeline field is present in both the `OFPPDPT_PIPELINE_OUTPUT` property of the output port and the `OFPPDPT_PIPELINE_INPUT` property of the return port, then this pipeline field is preserved with the packet (its value must remain the same).

5 OpenFlow Tables

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

 © 2015; The Open Networking Foundation

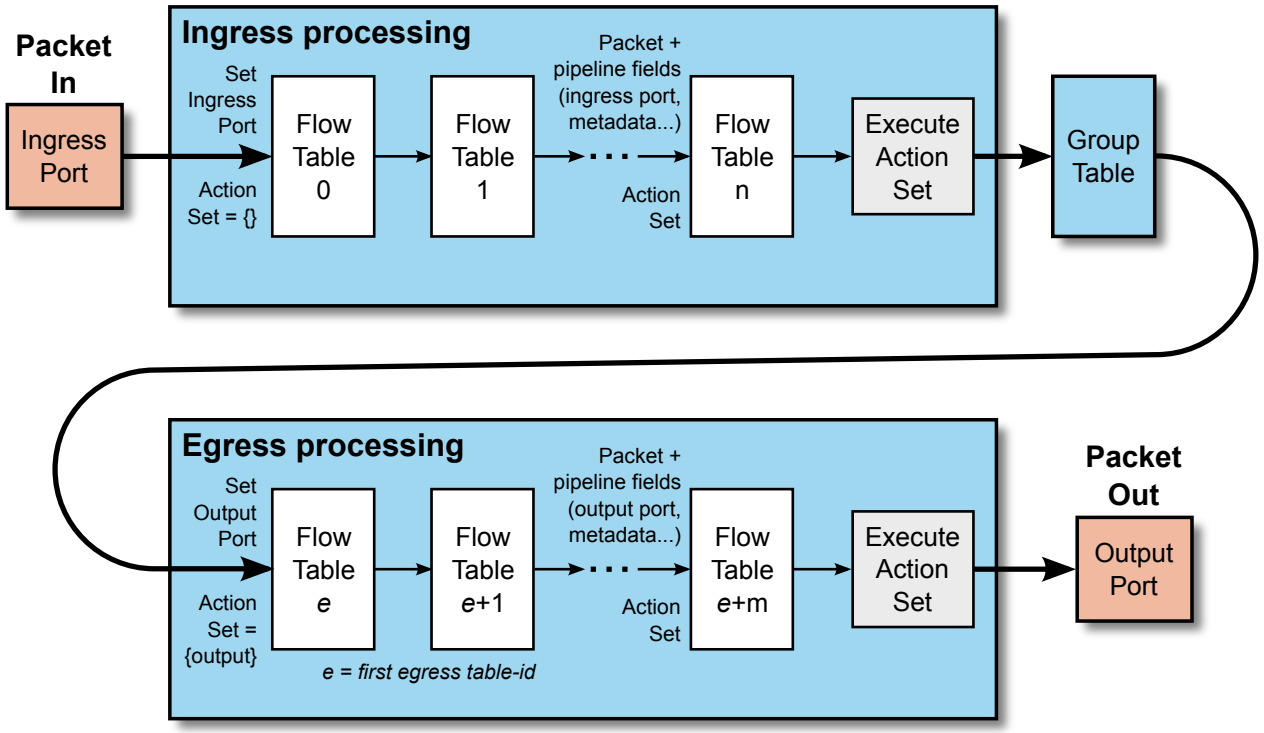


Figure 2: Packet flow through the processing pipeline.

5.1 Pipeline Processing

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

OpenFlow-hybrid switches support both OpenFlow operation and *normal* Ethernet switching operation, i.e. traditional L2 Ethernet switching, VLAN isolation, L3 routing (IPv4 routing, IPv6 routing...), ACL and QoS processing. Those switches should provide a classification mechanism outside of OpenFlow that routes traffic to *either* the OpenFlow pipeline *or* the normal pipeline. For example, a switch may use the VLAN tag or input port of the packet to decide whether to process the packet using one pipeline or the other, or it may direct all packets to the OpenFlow pipeline. This classification mechanism is outside the scope of this specification. An OpenFlow-hybrid switch may also allow a packet to go from the OpenFlow pipeline to the normal pipeline through the *NORMAL* and *FLOOD* reserved ports (see 4.5).

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

The flow tables of an OpenFlow switch are numbered in the order they can be traversed by packets,

 © 2015; The Open Networking Foundation

starting at 0. Pipeline processing happens in two stages, *ingress processing* and *egress processing*. The separation of the two stages is indicated by the first egress table (see 7.3.2), all tables with a number lower than the first egress table must be used as ingress tables, and no table with a number higher than or equal to the first egress table can be used as an ingress table.

Pipeline processing always starts with ingress processing at the first flow table: the packet must be first matched against flow entries of flow table 0 (see Figure 3). Other ingress flow tables may be used depending on the outcome of the match in the first table. If the outcome of ingress processing is to forward the packet to an output port, the OpenFlow switch may perform egress processing in the context of that output port. Egress processing is optional, a switch may not support any egress tables or may not be configured to use them. If no valid egress table is configured as the first egress table (see 7.3.2), the packet must be processed by the output port, and in most cases the packet is forwarded out of the switch. If a valid egress table is configured as the first egress table (see 7.3.2), the packet must be matched against flow entries of that flow table, and other egress flow tables may be used depending on the outcome of the match in that flow table.

When processed by a flow table, the packet is matched against the flow entries of the flow table to select a flow entry (see 5.3). If a flow entry is found, the instruction set included in that flow entry is executed. These instructions may explicitly direct the packet to another flow table (using the Goto-Table Instruction, see 5.5), where the same process is repeated again. A flow entry can only direct a packet to a flow table number which is greater than its own flow table number, in other words pipeline processing can only go forward and not backward. Obviously, the flow entries of the last table of a pipeline stage can not include the Goto-Table instruction. If the matching flow entry does not direct packets to another flow table, the current stage of pipeline processing stops at this table, the packet is processed with its associated action set and usually forwarded (see 5.6).

If a packet does not match a flow entry in a flow table, this is a table miss. The behavior on a table miss depends on the table configuration (see 5.4). The instructions included in the table-miss flow entry in the flow table can flexibly specify how to process unmatched packets, useful options include dropping them, passing them to another table or sending them to the controllers over the control channel via packet-in messages (see 6.1.2).

There are few cases where a packet is not fully processed by a flow entry and pipeline processing stops without processing the packet's action set or directing it to another table. If no table-miss flow entry is present, the packet is dropped (see 5.4). If an invalid TTL is found, the packet may be sent to the controller (see 5.8).

The OpenFlow pipeline and various OpenFlow operations process packets of a specific type in conformance with the specifications defined for that packet type, unless the present specification or the OpenFlow configuration specify otherwise. For example, the Ethernet header definition used by OpenFlow must conform to IEEE specifications, and the TCP/IP header definition used by OpenFlow must conform to RFC specifications. Additionally, packet reordering in an OpenFlow switch must conform to the requirements of IEEE specifications, provided that the packets are processed by the same flow entries, group buckets and meter bands.

 © 2015; The Open Networking Foundation

5.1.1 Pipeline Consistency

The OpenFlow pipeline is an abstraction that is mapped to the actual hardware of the switch. In some cases, the OpenFlow switch is virtualised on the hardware, for example to support multiple OpenFlow switch instances or in the case of an hybrid switch. Even if the OpenFlow switch is not virtualised, the hardware typically will not correspond to the OpenFlow pipeline, for example if a flow table only supports Ethernet packets, non-Ethernet packets would have to be mapped to Ethernet. In another example some switches may carry VLAN information in some internal metadata while for the OpenFlow pipeline it is logically part of the packet. Some OpenFlow switches may define logical ports implementing complex encapsulations that extensively modify the packet headers. The consequence is that a packet on a link or in hardware may be mapped differently in the OpenFlow pipeline.

However, the OpenFlow pipeline expects that the mapping to the hardware is consistent, and that the OpenFlow pipeline behaves consistently. In particular, this is what is expected :

- **Tables consistency**: the packet must match in the same way in all the OpenFlow flow tables, and the only difference in matching must be due to the flow table content and explicit OpenFlow processing done by those flow tables. In particular, headers can not be transparently removed, added or changed between tables, unless explicitly specified by OpenFlow processing.
- **Flow entry consistency**: the way the actions of a flow entry apply to a packet must be consistent with the flow entry match. In particular, if a match field in the flow entry matches a specific packet header field, the corresponding set-field action in the flow entry must modify the same header field, unless explicit OpenFlow processing has modified the packet.
- **Group consistency**: the application of groups must be consistent with flow tables. In particular, actions of a group bucket must apply to the packet the same way as if they were in a flow table, the only difference must be due to explicit OpenFlow processing.
- **Packet-in consistency**: the packet embedded in the packet-in messages must be consistent with the OpenFlow flow tables. In particular, if the packet-in was generated directly by a flow entry, the packet received by the controller must match the flow entry that sent it to the controller.
- **Packet-out consistency**: the packet generated as the result of a packet-out request must be consistent with the OpenFlow flow tables and the packet-in process. In particular, if a packet received via a packet-in is sent directly without modifications out a port via a packet-out, the packet on that port must be identical, as if the packet had been sent to that port instead of encapsulated in a packet-in. Similarly, if a packet-out is directed at flow tables, the flow entries must match the encapsulated packet as expected by the OpenFlow matching process.
- **Port consistency**: the ingress and egress processing of an OpenFlow port must be consistent with each other. In particular, if an OpenFlow packet is output on a port and generates a physical packet on a switch physical link, then if the reception by the switch of the same physical packet on the same link generates an OpenFlow packet on the same port, the OpenFlow packet must be identical.

 © 2015; The Open Networking Foundation

5.2 Flow Tables and flow entries

A flow table consists of flow entries.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Table 1: Main components of a flow entry in a flow table.

Each flow table entry (see Table 1) contains :

- **match fields**: to match against packets. These consist of the ingress port and packet headers, and optionally other pipeline fields such as metadata specified by a previous table.
- **priority**: matching precedence of the flow entry.
- **counters**: updated when packets are matched.
- **instructions**: to modify the action set or pipeline processing.
- **timeouts**: maximum amount of time or idle time before flow is expired by the switch.
- **cookie**: opaque data value chosen by the controller. May be used by the controller to filter flow entries affected by flow statistics, flow modification and flow deletion requests. Not used when processing packets.
- **flags**: flags alter the way flow entries are managed, for example the flag `OFPFF_SEND_FLOW_REM` triggers flow removed messages for that flow entry.

A flow table entry is identified by its match fields and priority: the match fields and priority taken together identify a unique flow entry in a specific flow table. The flow entry that wildcards all fields (all fields omitted) and has priority equal to 0 is called the table-miss flow entry (see 5.4).

A flow entry instruction may contain actions to be performed on the packet at some point of the pipeline (see 5.8). The *set-field* action may specify some header fields to rewrite. Each flow table may not support every match field, every instruction, every action or every set-field defined by this specification, and different flow tables of the switch may not support the same subset. The table features request enables the controller to discover what each table supports (see 7.3.5.18).

5.3 Matching

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

Packet header fields are extracted from the packet, and packet pipeline fields are retrieved. Packet header fields used for table lookups depend on the packet type, and typically include various protocol header fields, such as Ethernet source address or IPv4 destination address (see 7.2.3). In addition to packet headers, matches can also be performed against the ingress port, the metadata field and other pipeline fields. Metadata may be used to pass information between tables in a switch. The packet header fields and pipeline fields represent the packet in its current state, if actions applied in a previous

 © 2015; The Open Networking Foundation

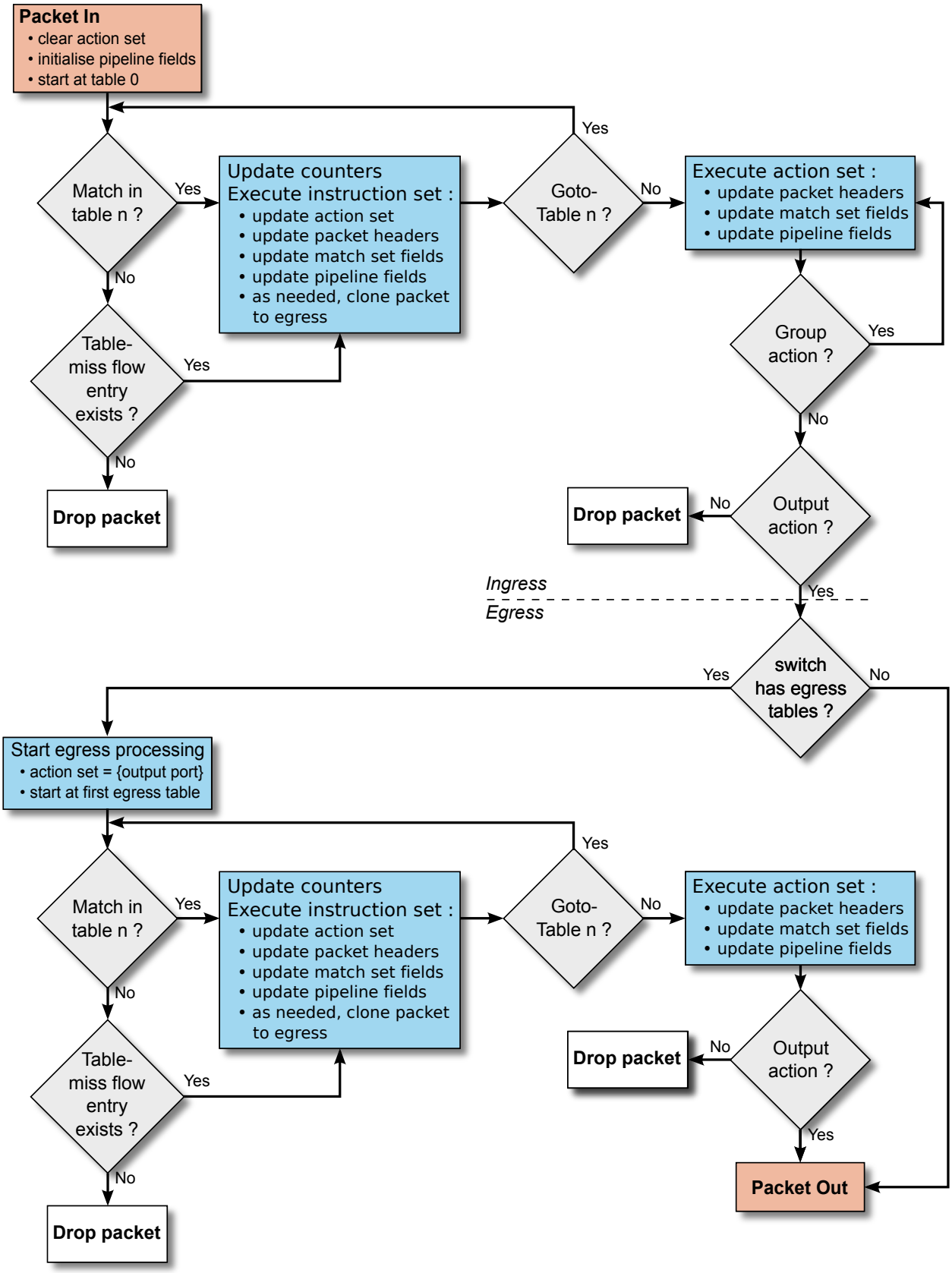


Figure 3: Simplified flowchart detailing packet flow through an OpenFlow switch.

 © 2015; The Open Networking Foundation

table using the *Apply-Actions* instruction changed the packet headers or pipeline fields, those changes are reflected in the packet header fields and pipeline fields.

A packet matches a flow entry if all the match fields of the flow entry are matching the corresponding header fields and pipeline fields from the packet. If a match field is omitted in the flow entry (i.e. value ANY), it matches all possible values in the header field or pipeline field of the packet (see 7.2.3.4). If the match field is present and does not include a mask, the match field is matching the corresponding header field or pipeline field from the packet if it has the same value. If the switch supports arbitrary bitmasks on specific match fields, these masks can more precisely specify matches, the match field is matching if it has the same value for the bits which are set in the mask (see 7.2.3.5).

The packet is matched against flow entries in the flow table and *only* the highest priority flow entry that matches the packet must be selected. The counters associated with the selected flow entry must be updated (see 5.9) and the instruction set included in the selected flow entry must be executed (see 5.5). If there are multiple matching flow entries with the same highest priority, the selected flow entry is explicitly undefined. This case can only arise when a controller writer never sets the `OFPFF_CHECK_OVERLAP` bit on flow mod messages and adds overlapping entries.

IP fragments must be reassembled before pipeline processing if the switch configuration contains the `OFPC_FRAG_REASM` flag (see 7.3.2).

This version of the specification does *not* define the expected behavior when a switch receives a malformed or corrupted packet either on an OpenFlow port (see 4.1) or in a *Packet-Out* message (see 6.1.1).

5.4 Table-miss

Every flow table must support a table-miss flow entry to process table misses. The table-miss flow entry specifies how to process packets unmatched by other flow entries in the flow table (see 5.1), and may, for example, send packets to the controller, drop packets or direct packets to a subsequent table.

The table-miss flow entry is identified by its match and its priority (see 5.2), it wildcards all match fields (all fields omitted) and has the lowest priority (0). The match of the table-miss flow entry may fall outside the normal range of matches supported by a flow table, for example an exact match table would not support wildcards for other flow entries but must support the table-miss flow entry wildcarding all fields. The table-miss flow entry may not have the same capability as regular flow entry (see 7.3.5.18). The table-miss flow entry must support at least sending packets to the controller using the CONTROLLER reserved port (see 4.5) and dropping packets using the Clear-Actions instruction (see 5.5). Implementations are encouraged to support directing packets to a subsequent table when possible for compatibility with earlier versions of this specification.

The table-miss flow entry behaves in most ways like any other flow entry: it does not exist by default in a flow table, the controller may add it or remove it at any time (see 6.4), it may expire (see 6.5), it may be evicted (see 6.4) and it may be read by the controller (see 7.3.5.2). The table-miss flow entry matches packets in the table as expected from its set of match fields and priority (see 5.3): it matches packets unmatched by other flow entries in the flow table. The table-miss flow entry instructions are applied to packets matching the table-miss flow entry (see 5.5). If the table-miss flow entry directly sends packets to the controller using the CONTROLLER reserved port (see 4.5), the packet-in reason must identify a table-miss (see 7.4.1).

 © 2015; The Open Networking Foundation

If the table-miss flow entry does not exist, by default packets unmatched by flow entries are dropped (discarded). A switch configuration, for example using the OpenFlow Configuration Protocol, may override this default and specify another behaviour. A flow entry that uses the lowest priority (0) and has a match that does *not* wildcards all match fields can be used if the flow table supports it, however this is not a table-miss flow entry. Using such flow entry would make sense only if a table-miss flow entry is not used, because if a table-miss flow entry exists they would overlap and matching is then undefined. For this reason, it is recommended that the controller does not create non-table-miss flow entries that use the lowest priority (0).

5.5 Instructions

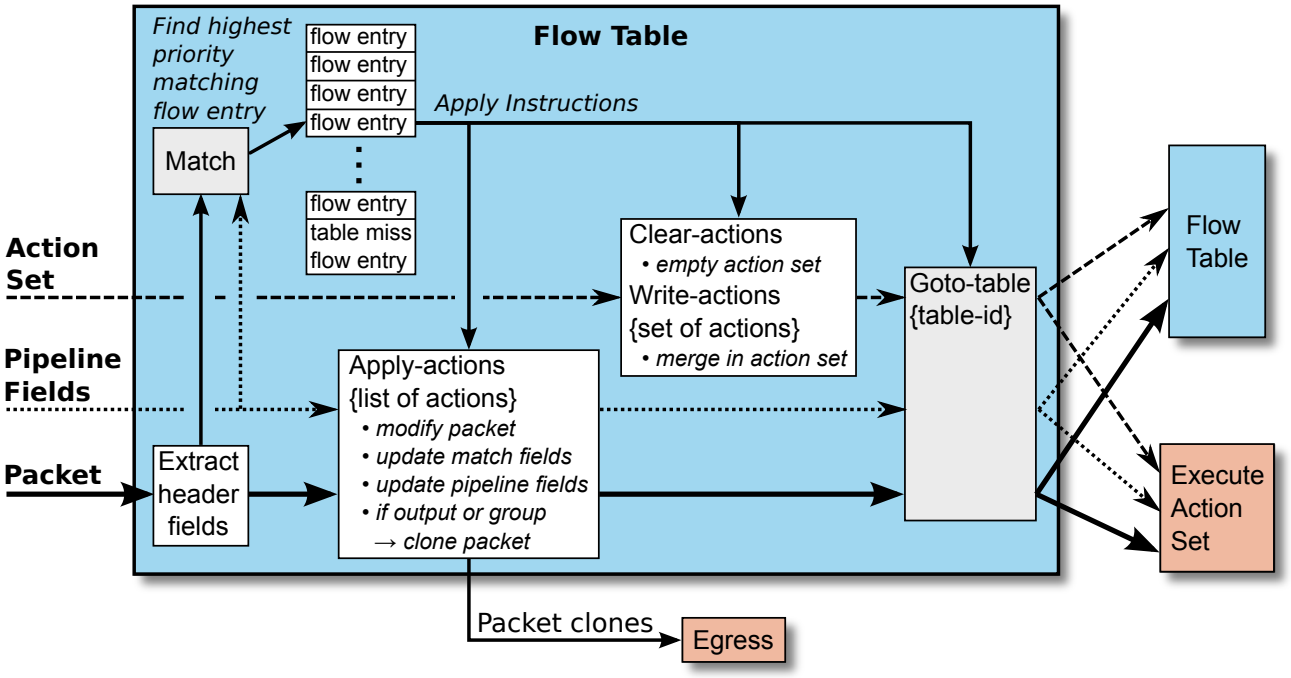


Figure 4: Matching and Instruction execution in a flow table.

Each flow entry contains a set of instructions that are executed when a packet matches the entry. These instructions result in changes to the packet, action set and/or pipeline processing (see Figure 4).

A switch is not required to support all instruction types, just those marked *"Required Instruction"* below. The controller can also query the switch about which of the *"Optional Instruction"* types it supports (see 7.3.5.18).

- *Optional Instruction:* **Apply-Actions *action(s)***: Applies the specific action(s) immediately, without any change to the Action Set. This instruction may be used to modify the packet between two tables or to execute multiple actions of the same type. The actions are specified as a list of actions (see 5.7).

 © 2015; The Open Networking Foundation

- *Required Instruction:* **Clear-Actions**: Clears all the actions in the action set immediately. Support of this instruction is required only for *table-miss flow entries* (see 5.4), and is optional for other flow entries.
- *Required Instruction:* **Write-Actions *action(s)***: Merges the specified set of action(s) into the current action set (see 5.6). If an action of the given type exists in the current set, overwrite it, otherwise add it. If a *set-field* action with a given field type exists in the current set, overwrite it, otherwise add it. This instruction must be supported in all flow tables.
- *Optional Instruction:* **Write-Metadata *metadata / mask***: Writes the masked metadata value into the metadata field. The mask specifies which bits of the metadata register should be modified (i.e. new_metadata = old_metadata & ~mask | value & mask).
- *Optional Instruction:* **Stat-Trigger *stat thresholds***: Generate an event to the controller if some of the flow statistics cross one of the stat threshold values.
- *Required Instruction:* **Goto-Table *next-table-id***: Indicates the next table in the processing pipeline. The table-id must be greater than the current table-id. This instruction must be supported in all flow tables except the last one, OpenFlow switches with only a single flow table are not required to implement this instruction. The flow entries of the last table of the pipeline can not include this instruction (see 5.1).

The instruction set associated with a flow entry contains a maximum of one instruction of each type. The *experimenter* instructions are identified by their experimenter-id and experimenter-type, therefore the instruction set may contain a maximum of one *experimenter* instruction for each combination of experimenter-id and experimenter-type. The instructions of the set execute in the order specified by this above list. In practice, the only constraints are that the *Clear-Actions* instruction is executed before the *Write-Actions* instruction, the *Apply-Actions* is executed before the *Write-Metadata* and that *Goto-Table* is executed last.

A switch must reject a flow entry if it is unable to execute the instructions or part of the instructions associated with the flow entry. In this case, the switch must return the error message associated with the issue (see 7.5.4.4).

5.6 Action Set

An action set is associated with each packet. This set is empty by default. A flow entry can modify the action set using a *Write-Action* instruction or a *Clear-Action* instruction associated with a particular match. The action set is carried between flow tables. When the instruction set of a flow entry does not contain a *Goto-Table* instruction, pipeline processing stops and the actions in the action set of the packet are executed (see Figure 3).

An action set contains a maximum of one action of each type. The *set-field* actions are identified by their destination field types, therefore the action set contains a maximum of one *set-field* action for each field type (i.e. multiple fields can be set, but each field can only be set once). The effect of *copy-field* actions in the action set is undefined due to race conditions, and therefore its implementation discouraged. The *experimenter* actions are identified by their experimenter-id and experimenter-type, therefore the action set may contain a maximum of one *experimenter* action for each combination of experimenter-id and experimenter-type. When an action of a specific type is added in the action set, if an action of the same

 © 2015; The Open Networking Foundation

type exists, it is overwritten by the later action. If multiple actions of the same type are required, e.g. pushing multiple MPLS labels or popping multiple MPLS labels, the *Apply-Actions* instruction should be used (see 5.7).

The action set for egress processing has one restriction, both the *output* action and the *group* action can not be added to the egress action-set. The action set for egress processing is initialised at the beginning of egress processing with an *output* action for the current output port, whereas the action set for ingress processing starts empty.

The actions in an action set are applied in the order specified below, regardless of the order that they were added to the set. If an action set contains a group action, the actions in the appropriate action bucket(s) of the group are also applied in the order specified below. The switch may support arbitrary action execution order through the list of actions of the *Apply-Actions* instruction.

1. **copy TTL inwards**: apply copy TTL inward actions to the packet
2. **pop**: apply all tag pop actions to the packet
3. **push-MPLS**: apply MPLS tag push action to the packet
4. **push-PBB**: apply PBB tag push action to the packet
5. **push-VLAN**: apply VLAN tag push action to the packet
6. **copy TTL outwards**: apply copy TTL outwards action to the packet
7. **decrement TTL**: apply decrement TTL action to the packet
8. **set**: apply all set-field actions to the packet
9. **qos**: apply all QoS actions, such as meter and set_queue to the packet
10. **group**: if a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list
11. **output**: if no group action is specified, forward the packet on the port specified by the output action

The *output* action in the action set is executed last. If both an *output* action and a *group* action are specified in an action set, the output action is ignored and the group action takes precedence. If no output action and no group action were specified in an action set, the packet is dropped. If no group action is specified and the output action references a non-existent port, the packet is dropped. The execution of groups is recursive if the switch supports it; a group bucket may specify another group, in which case the execution of actions traverses all the groups specified by the group configuration.

The *output* action is processed differently in ingress and egress (see 5.1). When the ingress action set contains an *output* action or a *group* action forwarding the packet to a port, the packet must start egress processing on that port (see 5.1). If the *output* action references the *ALL* reserved port, a clone (copy) of the packet start egress processing for each relevant port (see 4.5). When the egress action set contains an *output* action, the packet must exit egress processing and must be processed by the port, and in most cases it is forwarded out of the switch.

 © 2015; The Open Networking Foundation

5.7 List of Actions

The *Apply-Actions* instruction and the *Packet-out* message include a list of actions. The semantics of the list of actions is identical to the OpenFlow 1.0 specification. The actions of a list of actions are executed in the order specified by the list, and are applied immediately to the packet.

The execution of a list of actions starts with the first action in the list and each action is executed on the packet in sequence. The effect of those actions is cumulative, if the list of actions contains two Push VLAN actions, two VLAN headers are added to the packet.

If the list of actions contains an *output* action, a clone (copy) of the packet is forwarded in its current state to the desired port where it starts egress processing. If the *output* action references the *ALL* reserved port, a clone of the packet start egress processing for each relevant port (see 4.5). If the output action references an non-existent port, the clone of the packet is dropped. If the list of actions contains a *group* action, a clone of the packet in its current state is processed by the relevant group buckets (see 5.10). The set of pipeline fields is cloned with the packet. Any modifications done to a clone of the packet or its pipeline fields generated by the output or group action, for example changing a header field or the metadata field in a group bucket or an egress table, applies only to that clone and does not apply to the original packet or other clones.

After the execution of the list of actions in an *Apply-Actions* instruction, pipeline execution continues on the modified packet (see 5.1). The action set of the packet is unchanged by the execution of the list of actions.

5.8 Actions

A switch is not required to support all action types, just those marked *"Required Action"* below. Required actions must be supported in all flow tables. The controller can also query the switch about which of the *"Optional Action"* it supports (see 7.3.5.18).

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

Required Action: **Group *group_id***. Process the packet through the specified group (see 5.10). The exact interpretation depends on group type.

Required Action: **Drop**. There is no explicit action to represent drops. Instead, packets whose action sets have no output action and no group action must be dropped. This result could come from empty instruction sets or empty action buckets in the processing pipeline (see 5.6), or after executing a *Clear-Actions* instruction (see 5.5).

Optional Action: **Set-Queue *queue_id***. The set-queue action sets the queue id for a packet. When the packet is forwarded to a port using the output action, the queue id determines which queue attached to this port is used for scheduling and forwarding the packet. Forwarding behavior is dictated by the configuration of the queue and is used to provide basic Quality-of-Service (QoS) support (see section 7.3.5.8).

Optional Action: **Meter *meter_id***. Direct packet to the specified meter (see 5.11). As the result of the metering, the packet may be dropped (depending on meter configuration and state). If the

 © 2015; The Open Networking Foundation

switch supports meters, it must support this action as the first action in a *list of actions*, for backward compatibility with earlier versions of the specification. Optionally, a switch supporting meters may also support the meter action in other positions in a *list of actions*, multiple meter actions in a *list of actions* and the meter action in the *action set* (see section 7.3.5.14).

Optional Action: **Push-Tag/Pop-Tag *ethertype***. Switches may support the ability to push/pop tags as shown in Table 2. To aid integration with existing networks, we suggest that the ability to push/pop VLAN tags be supported.

Newly pushed tags should *always* be inserted as the outermost tag in the outermost valid location for that tag (see 7.2.6.6). When multiple push actions are added to the action set of the packet, they apply to the packet in the order defined by the action set rules, first MPLS, then PBB, than VLAN (see 5.6). When multiple push actions are included in a list of actions, they apply to the packet in the list order (see 5.7).

Note: Refer to section 5.8.1 for information on default field values.

Action	**Associated Data**	**Description**
Push VLAN header	Ethertype	Push a new VLAN header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8100 and 0x88a8 should be used.
Pop VLAN header	-	Pop the outer-most VLAN header from the packet.
Push MPLS header	Ethertype	Push a new MPLS shim header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8847 and 0x8848 should be used.
Pop MPLS header	Ethertype	Pop the outer-most MPLS tag or shim header from the packet. The Ethertype is used as the Ethertype for the resulting packet (Ethertype for the MPLS payload).
Push PBB header	Ethertype	Push a new PBB service instance header (I-TAG TCI) onto the packet (see 7.2.6.6). The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x88E7 should be used.
Pop PBB header	-	Pop the outer-most PBB service instance header (I-TAG TCI) from the packet (see 7.2.6.6).

Table 2: Push/pop tag actions.

Optional Action: **Set-Field *field_type value***. The various Set-Field actions are identified by their field type and modify the values of respective header fields in the packet (see 7.2.3.7). While not strictly required, the support of rewriting various header fields using Set-Field actions greatly increases the usefulness of an OpenFlow implementation. To aid integration with existing networks, we suggest that VLAN modification actions be supported. Set-Field actions should *always* be applied to the outermost-possible header (e.g. a "Set VLAN ID" action always sets the ID of the outermost VLAN tag), unless the field type specifies otherwise.

Optional Action: **Copy-Field *src_field_type dst_field_type***. The Copy-Field action may copy data between any header or pipeline fields. It is typically used to copy data from a header field to a packet register pipeline field or from a packet register pipeline field to a header field, and in some cases from a header field to another header field. A switch may not support all combinations of copies between header or pipeline fields.

 © 2015; The Open Networking Foundation

Optional Action: **Change-TTL *ttl***. The various Change-TTL actions modify the values of the IPv4 TTL, IPv6 Hop Limit or MPLS TTL in the packet. While not strictly required, the actions shown in Table 3 greatly increase the usefulness of an OpenFlow implementation for implementing routing functions. Change-TTL actions should *always* be applied to the outermost-possible header.

Action	Associated Data	Description
Set MPLS TTL	8 bits: New MPLS TTL	Replace the existing MPLS TTL. Only applies to packets with an existing MPLS shim header.
Decrement MPLS TTL	-	Decrement the MPLS TTL. Only applies to packets with an existing MPLS shim header.
Set IP TTL	8 bits: New IP TTL	Replace the existing IPv4 TTL or IPv6 Hop Limit and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Decrement IP TTL	-	Decrement the IPv4 TTL or IPv6 Hop Limit field and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Copy TTL outwards	-	Copy the TTL from next-to-outermost to outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or IP-to-MPLS.
Copy TTL inwards	-	Copy the TTL from outermost to next-to-outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or MPLS-to-IP.

Table 3: Change-TTL actions.

The OpenFlow switch checks for packets with invalid IP TTL or MPLS TTL and rejects them. Checking for invalid TTL does not need to be done for every packet, but it must be done at a minimum every time a *decrement TTL* action is applied to a packet. The asynchronous configuration of the switch may be changed (see 6.1.1) to send packets with invalid TTL to the controllers over the control channel via a packet-in message (see 6.1.2).

5.8.1 Default values for fields on push

When a new tag is added via a push action (see 5.8), the push action itself does not specify most of the values for the header fields part of the tag. Most of those are set separately via "set-field" actions. The way those header fields are initialised during the push action depends on if those fields are defined as OpenFlow match fields (see 7.2.3.7) and the state of the packet. Header fields part of tag push operation and defined as OpenFlow match fields are listed in Table 4.

When executing a push action, for all fields specified in Table 4 part of the added header, the value from the corresponding field in the existing outer headers of the packet should be copied in the new field. If the corresponding outer header field does not exist in the packet, the new field should be set to zero.

Header fields which are not defined as OpenFlow match fields should be initialized to appropriate protocol values : those header fields should be set in compliance with the specification governing that field, taking into account the packet headers and the switch configuration. For example, the DEI bit in the VLAN header should be set to zero in most cases.

 © 2015; The Open Networking Foundation

New Fields		Existing Field(s)
VLAN ID	←	VLAN ID
VLAN priority	←	VLAN priority
MPLS label	←	MPLS label
MPLS traffic class	←	MPLS traffic class
MPLS TTL	←	{ MPLS TTL IP TTL
PBB I-SID	←	PBB I-SID
PBB I-PCP	←	VLAN PCP
PBB C-DA	←	ETH DST
PBB C-SA	←	ETH SRC

Table 4: Existing fields that may be copied into new fields on a push action.

Fields in new headers may be overridden by specifying a "set-field" action for the appropriate field(s) after the push operation. The execution order of actions in the action set is designed to facilitate that process (see 5.6).

5.9 Counters

Counters are maintained for each flow table, flow entry, port, queue, group, group bucket, meter and meter band. OpenFlow-compliant counters may be implemented in software and maintained by polling hardware counters with more limited ranges. Table 5 contains the set of counters defined by the OpenFlow specification. A switch is not required to support all counters, just those marked *"Required"* in Table 5.

Duration refers to the amount of time the flow entry, a port, a group, a queue or a meter has been installed in the switch, and must be tracked with second precision. The Receive Errors field is the total of all receive and collision errors defined in Table 5, as well as any others not called out in the table.

Packet related counters for an OpenFlow object must count every packet using that object, even if the object is having no effect on the packet or if the packet is ultimately dropped or sent to the controller. For example, the switch should maintain the packet related counters of the following :

- a flow entry with only a goto-table instruction and without actions
- a group outputing to a non-existent port
- a flow entry triggering a TTL exception
- a port which is down

Counters are unsigned and wrap around with no overflow indicator. Counters must use the full bit range defined for the counter before rolling over, for example if a counter is defined as 64 bits, it can not use only the lower 32 bits. If a specific numeric counter is not available in the switch, its value must be set to the maximum field value (the unsigned equivalent of -1).

 © 2015; The Open Networking Foundation

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

Table 5: List of counters.

 © 2015; The Open Networking Foundation

5.10 Group Table

A group table consists of group entries. The ability for a flow entry to point to a *group* enables OpenFlow to represent additional methods of forwarding (e.g. select and all).

Group Identifier	Group Type	Counters	Action Buckets

Table 6: Main components of a group entry in the group table.

Each group entry (see Table 6) is identified by its group identifier and contains:

- **group identifier**: a 32 bit unsigned integer uniquely identifying the group on the OpenFlow switch.
- **group type**: to determine group semantics (see Section 5.10.1).
- **counters**: updated when packets are processed by a group.
- **action buckets**: an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters. The actions in a bucket are always applied as an action set (see 5.6).

A group entry may consist of zero or more buckets, except for the group of type *indirect* that always has one bucket. A group with no buckets effectively drops the packet (see 5.6).

A bucket typically contains actions that modify the packet and an output action that forwards it to a port. A bucket may also include a group action which invokes another group if the switch supports group chaining (see 6.7), in this case packet processing continues in the group invoked (see 5.6). A bucket with no actions is valid, a bucket with no output or group action effectively drops the clone of the packet associated with that bucket.

5.10.1 Group Types

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

- *Required:* **indirect**: Execute the one defined bucket in this group. This group supports only a single bucket. Allows multiple flow entries or groups to point to a common group identifier, supporting faster, more efficient convergence (e.g. next hops for IP forwarding). This group type is effectively identical to an all group with one bucket. This group is the simplest type of group, and therefore switches will typically support a greater number of them than other group types.
- *Required:* **all**: Execute all buckets in the group. This group is used for multicast or broadcast forwarding. The packet is effectively cloned for each bucket; one packet is processed for each bucket of the group. If a bucket directs a packet explicitly out the ingress port, this packet clone is dropped. If the controller writer wants to forward out the ingress port, the group must include an extra bucket which includes an output action to the `OFPP_IN_PORT` reserved port.

 © 2015; The Open Networking Foundation

- *Optional:* **select**: Execute one bucket in the group. Packets are processed by a single bucket in the group, based on a switch-computed selection algorithm (e.g. hash on some user-configured tuple or simple round robin). All configuration and state for the selection algorithm are external to OpenFlow. The selection algorithm should implement equal load sharing and can optionally be based on bucket weights. When a port specified in a bucket in a select group goes down, the switch may restrict bucket selection to the remaining set (those with forwarding actions to live ports) instead of dropping packets destined to that port. This behavior may reduce the disruption of a downed link or switch.
- *Optional:* **fast failover**: Execute the first live bucket. Each action bucket is associated with a specific port and/or group that controls its liveness. The buckets are evaluated in the order defined by the group, and the first bucket which is associated with a live port/group is selected. This group type enables the switch to change forwarding without requiring a round trip to the controller. If no buckets are live, packets are dropped. This group type must implement a *liveness mechanism* (see 6.7).

5.10.2 Group Liveness Monitoring

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

A group bucket may explicitly monitor the liveness of a port or another group. The rules for determining liveness include:

- A port is considered live if it exists in the datapath and has the `OFPPS_LIVE` flag set in its port state. Port liveness may be managed by code outside of the OpenFlow portion of a switch, defined outside of the OpenFlow specification, such as Spanning Tree or a KeepAlive mechanism. The port must not be considered live (and the `OFPPS_LIVE` flag must be unset) if one of the port liveness mechanisms of the switch enabled on that OpenFlow port considers the port not live, or if the port config bit `OFPPC_PORT_DOWN` indicates the port is down, or if the port state bit `OFPPS_LINK_DOWN` indicates the link is down.
- A bucket is considered live if either `watch_port` is not `OFPP_ANY` and the port watched is live, or if `watch_group` is not `OFPG_ANY` and the group watched is live. In other words, the bucket is considered not live if `watch_port` is `OFPP_ANY` or the port watched is not live, and if `watch_group` is `OFPG_ANY` or the group watched is not live.
- A group is considered live if a least one of its buckets is live.

The controller can infer the liveness state of the group by monitoring the states of the various ports.

5.11 Meter Table

A meter table consists of meter entries, defining per-flow meters. Per-flow meters enable OpenFlow to implement rate-limiting, a simple QoS operation constraining a set of flows to a chosen bandwidth. Per-flow meters can also enable OpenFlow to implement more complex QoS policing operations, such as

 © 2015; The Open Networking Foundation

DSCP based metering that can classify a set of packets in multiple categories based on its rate. Meters are fully independent of per-port queues (see 5.8), however in many cases those two features can be combined to implement complex work conserving QoS frameworks, such as DiffServ.

A meter measures the rate of packets assigned to it and enables controlling the rate of those packets. Meters are attached directly to flow entries (as opposed to queues which are attached to ports). If the switch supports it, any flow entry can specify a meter action in a list of actions (see 5.8): the meter measures and controls the rate of the aggregate of all flow entries to which it is attached.

Meter Identifier	Meter Bands	Counters

Table 7: Main components of a meter entry in the meter table.

Each meter entry (see Table 7) is identified by its meter identifier and contains:

- **meter identifier**: a 32 bit unsigned integer uniquely identifying the meter
- **meter bands**: an unordered list of meter bands, where each meter band specifies the rate of the band and the way to process the packet
- **counters**: updated when packets are processed by a meter

Different flow entries in the same flow table may use the same meter, different meters or no meter at all. By using different meters in a flow table, disjoint set of flow entries can be metered independently. Packets may go through multiple meters when using meters in successive flow tables, at each flow table the matching flow entry may direct it to one meter. Another way to use multiple meters, if the switch supports it, is to use multiple meter actions per flow entry. This can be used to perform hierarchical metering, where various set of traffic flows are first metered independently and then together (see Figure 5).

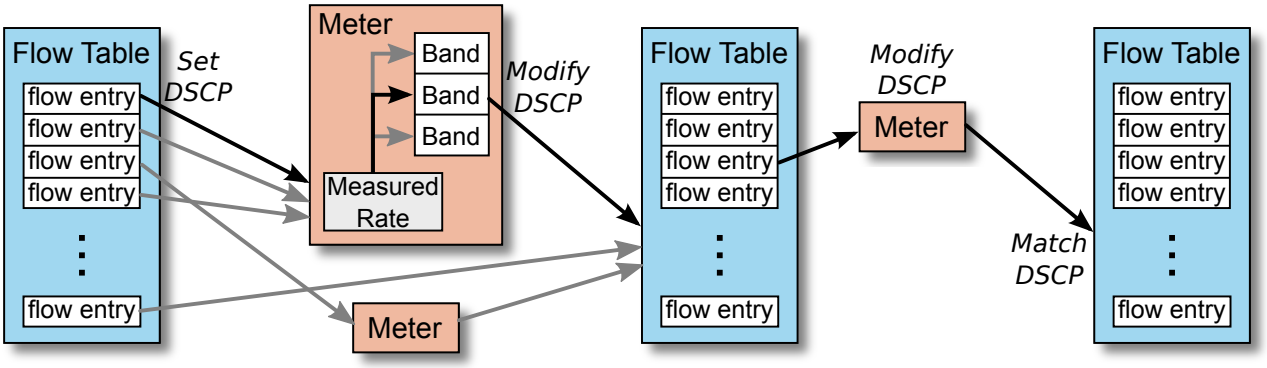


Figure 5: Meters and Hierarchical DSCP metering.

5.11.1 Meter Bands

Each meter may have one or more meter bands. The bands are used to define the behaviour of the meters on packets for various ranges of the meter measured rate. The meter *measured rate* is calculated

 © 2015; The Open Networking Foundation

using all packet from all flow entries directing packets to that meter. For each packet the meter selects one of the meter band, based on the measured rate, the band rate values and the meter configuration. Once the band is chosen, the meter applies the processing specified by that meter band to the packet. A packet is only processed by a single meter band in a meter.

Each meter band specifies a target rate for that band and a way packets should be processed if that rate is exceeded. The default meter band is always included in the meter and can not be set, it is equivalent to a band with target rate 0 that does nothing, it just lets packet through without doing anything.

Band Type	Rate	Burst	Counters	Type specific arguments

Table 8: Main components of a meter band in a meter entry.

Each meter band (see Table 8) is identified by its target rate and contains:

- **band type**: defines how packets are processed
- **rate**: target rate for that band - used by the meter to select the meter band, usually the lowest rate at which the band can apply
- **burst**: defines the granularity of the meter band
- **counters**: updated when packets are processed by a meter band
- **type specific arguments**: some band types have optional arguments

There is no band type *"Required"* by this specification. The controller can query the switch about which of the *"Optional"* meter band types it supports.

- *Optional:* **drop**: drop (discard) the packet. Can be used to define a rate limiter band.
- *Optional:* **dscp remark**: increase the drop precedence of the DSCP field in the IP header of the packet. Can be used to define a simple DiffServ policer.

The band selection process, including computation of *measured rate*, is implementation specific, and this specification does not specify how each individual packet is mapped into each band. The meters bands are ranked by increasing target rate, with the default band having rank 0 and the first configurable band having rank 1. The outcome of the band selection process must obey all the following constraints :

- A packet is processed only by a single meter band.
- Packets must be processed by a meter band only if the meter *measured rate* exceeds the band target rate.
- For any meter band that is processing packets, the amount of the traffic processed by all meter bands with lower rank must be equal to the band target rate. For example, if packets are processed by the the first configurable meter band, the amount of traffic processed by the default band must be equal to its target rate.

 © 2015; The Open Networking Foundation

A simplified model is that for every packet, the meter applies the meter band with the highest target rate that is lower than the current measured rate.

In practice, for short time windows, the meter may not obey perfectly those constraints due to packet granularity, the rate measurement approximations and various implementation constraints, so this describes only the long term behaviour. The *measured rate* calculation is usually done with a token bucket or a sliding window. In particular with a token bucket a short burst of packet may temporarily exceed a band target rate without triggering that band. The *burst* parameter defines the granularity of the band : for burst of packets longer than that value the comparison of the actual traffic rate and the band target rate must be accurate, for shorter bursts it may not be accurate.

5.12 Ingress and egress processing differences

Flow tables can be used for ingress or egress processing (see 5.1). Ingress processing is the main processing that happens when the packet enters the OpenFlow switch, and may involve one or more flow tables. Egress processing is the processing that happens after the determination of the output port, it happens in the context of the output port and may involve zero or more flow tables.

There are few differences between flow tables used for ingress and egress processing, the flow entries have the same components (see 5.2), flow table matching is the same (see 5.3) and execution of instructions is the same (see 5.5). Table miss processing is the same (see 5.4), and therefore the controller is strongly advised to set a table-miss flow entry in each egress table to avoid dropped packets. In some cases, a flow table can be reconfigured for ingress or egress processing by changing the first egress table (see 7.3.2).

At the beginning of ingress processing, the action set is empty. Flow tables used for ingress processing can only direct packets via the *Goto-Table* instruction to ingress flow tables, and can not direct packets to egress flow tables using the *Goto-Table* instruction. Ingress flow tables usually don't support the *Action Set Output* match field, but may support it in some cases.

At the beginning of egress processing, the action set contains only an *output* action for the current output port. Valid values for that output action are any physical or logical port, or the *CONTROLLER* or *LOCAL* reserved ports (see 4.5), other reserved ports must be substituted with their actual value. The values of the pipeline fields are preserved, except for the *Action Set Output* match field ; for example the initial value of the Metadata match field is the value it had when the packet was sent to egress processing. Flow tables used for egress processing can only direct packets via the *Goto-Table* instruction to egress flow tables. They must support the *Action Set Output* match field, to allow flows to be based on the output port context. They must forbid the use of an *output* action or *group* action in *write-action* instruction, so that the packet output port can not be changed. Those restrictions must be advertised in the flow table features (see 7.3.5.18).

The egress tables may optionally support the *output* action or *group* action in *apply-action* instruction. Those actions behave like they do in an ingress table, they forward clones of the packet to the specified ports, and those clones must start egress processing from the first egress table (see 5.7). If supported, this can be used for example for selective egress mirroring. A switch should protect itself from packet infinite loops when using *output* action or *group* action in egress tables, this mechanism is implementation specific and outside the scope of this specification.

 © 2015; The Open Networking Foundation

6 OpenFlow Channel and Control Channel

The OpenFlow channel is the interface that connects each OpenFlow Logical Switch to an OpenFlow controller. Through this interface, the controller configures and manages the switch, receives events from the switch, and sends packets out the switch. The Control Channel of the switch may support a single OpenFlow channel with a single controller, or multiple OpenFlow channels enabling multiple controllers to share management of the switch.

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

6.1 OpenFlow Switch Protocol Overview

The OpenFlow switch 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 about network events and changes to the switch state. Symmetric messages are initiated by either the switch or the controller and sent without solicitation. The message types used by OpenFlow are described below.

6.1.1 Controller-to-Switch

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

Features: The controller may request the identity and the basic capabilities of a switch by sending a features request; the switch must respond with a features reply that specifies the identity and basic capabilities of the switch. This is commonly performed upon establishment of the OpenFlow channel.

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

Modify-State: Modify-State messages are sent by the controller to manage state on the switches. Their primary purpose is to add, delete and modify flow/group entries and insert/remove action buckets of group in the OpenFlow tables and to set switch port properties.

Read-State: Read-State messages are used by the controller to collect various information from the switch, such as current configuration, statistics and capabilities. Most Read-State requests and replies are implemented using multipart message sequences (see 7.3.5).

Packet-out: These are used by the controller to send packets out of a specified port on the switch, and to forward packets received via Packet-in messages. Packet-out messages must contain a full packet or a buffer ID referencing a packet stored in the switch. The message must also contain a list of actions to be applied in the order they are specified; an empty list of actions drops the packet.

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

 © 2015; The Open Networking Foundation

Role-Request: Role-Request messages are used by the controller to set the role of its OpenFlow channel, set its Controller ID, or query these. This is mostly useful when the switch connects to multiple controllers (see 6.3.7).

Asynchronous-Configuration: The Asynchronous-Configuration messages are used by the controller to set an additional filter on the asynchronous messages that it wants to receive on its OpenFlow channel, or to query that filter. This is mostly useful when the switch connects to multiple controllers (see 6.3.7) and commonly performed upon establishment of the OpenFlow channel.

6.1.2 Asynchronous

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

Packet-in: Transfer the control of a packet to the controller. For all packets forwarded to the **CONTROLLER** reserved port using a flow entry or the table-miss flow entry, a packet-in event is always sent to controllers (see 5.8). Other processing, such as TTL checking, may also generate packet-in events to send packets to the controller.

Packet-in events can be configured to buffer packets. For packet-in generated by an output action in a flow entries or group bucket, it can be specified individually in the output action itself (see 7.2.6.1), for other packet-in it can be configured in the switch configuration (see 7.3.2). If the packet-in event is configured to buffer packets and the switch has sufficient memory to buffer them, the packet-in event contains only some fraction of the packet header and a buffer ID to be used by a controller when it is ready for the switch to forward the packet. Switches that do not support internal buffering, are configured to not buffer packets for the packet-in event, or have run out of internal buffering, must send the full packet to controllers as part of the event. Buffered packets will usually be processed via a **Packet-out** or **Flow-mod** message from a controller, or automatically expired after some time.

If the packet is buffered, the number of bytes of the original packet to include in the packet-in can be configured. By default, it is 128 bytes. For packet-in events generated by an output action in a flow entry or a group bucket, it can be specified individually in the output action itself (see 7.2.6.1), for other packet-in events it can be configured in the switch configuration (see 7.3.2).

Flow-Removed: Inform the controller about the removal of a flow entry from a flow table. Flow-Removed messages are only sent for flow entries with the `OFPFF_SEND_FLOW_REM` flag set. They are generated as the result of a controller flow delete request, the switch flow expiry process when one of the flow timeouts is exceeded, or other reasons (see 6.5).

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

Role-status: Inform the controller of a change of its role. When a new controller elects itself master, the switch is expected to send role-status messages to the former master controller (see 6.3.7).

 © 2015; The Open Networking Foundation

Controller-Status: Inform the controller when the status of an OpenFlow channel changes. The switch sends these messages to all controllers when the status of the OpenFlow channel to any switch changes. This can assist failover processing if controllers lose the ability to communicate among themselves.

Flow-monitor: Inform the controller of a change in a flow table. A controller may define a set of monitors to track changes in flow tables (see 7.3.5.19).

6.1.3 Symmetric

Symmetric messages are sent without solicitation, in either direction.

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

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

Error: Error messages are used by the switch or the controller to notify problems to the other side of the connection. They are mostly used by the switch to indicate a failure of a request initiated by the controller.

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

6.2 Message Handling

The OpenFlow switch protocol provides reliable message delivery and processing, but does *not* automatically provide acknowledgements or ensure ordered message processing. The OpenFlow message handling behaviour described in this section is provided on the main connection and auxiliary connections using reliable transport, however it is not supported on auxiliary connections using unreliable transport (see 6.3.8).

Message Delivery: Messages are guaranteed delivery, unless the OpenFlow channel fails entirely, in which case the controller should not assume anything about the switch state (e.g., the switch may have gone into "fail standalone mode").

Message Processing: Switches must process every message received from a controller in full, possibly generating a reply. If a switch cannot completely process a message received from a controller, it must send back an error message. For packet-out messages, fully processing the message does not guarantee that the included packet actually exits the switch. The included packet may be silently dropped after OpenFlow processing due to congestion at the switch, QoS policy, or if sent to a blocked or invalid port.

In addition, switches must send to the controller all asynchronous messages generated by OpenFlow state changes, such as flow-removed, port-status or packet-in messages, so that the controller view of the switch is consistent with its actual state. Those messages may get filtered out based on the *Asynchronous Configuration* (see 6.1.1). Moreover, conditions that would trigger an OpenFlow state change may get filtered prior to causing such change. For example, packets received on data ports that

 © 2015; The Open Networking Foundation

should be forwarded to the controller may get dropped due to congestion or QoS policy within the switch and generate no packet-in messages. These drops may occur for packets with an explicit output action to the controller. These drops may also occur when a packet fails to match any entries in a table and that table's default action is to send to the controller. The policing of packets destined to the controller is advised, to prevent denial of service of the controller connection, this can be done via a queue on the controller port (see 7.3.5.8) or using per-flow meters (see 5.11).

Controllers are free to ignore messages they receive, but must respond to echo messages to prevent the switch from terminating the connection.

Message Grouping: The controller can group together related messages using the optional *Bundle* messages (see 6.9). The set of messages in the bundle are applied as a unit, they are processed together, and if any errors arise, none of the messages is applied. If the bundle is configured as ordered, the set of messages in the bundle is also applied in order. The processing of a bundle is independent of the processing of other messages and other bundles.

Message Ordering: Ordering of messages can be ensured through the use of *barrier* messages (see 7.3.7). In the absence of barrier messages, switches may arbitrarily reorder messages to maximize performance; hence, controllers should not depend on a specific processing order. In particular, flow entries may be inserted in tables in an order different than that of flow mod messages received by the switch. Messages must not be reordered across a barrier message and the barrier message must be processed only when all prior messages have been processed. More precisely:

1. messages before a barrier must be fully processed before the barrier, including sending any resulting replies or errors
2. the barrier must then be processed, state is committed to the datapath and a barrier reply sent
3. messages after the barrier may then begin processing

If two messages from the controller depend on each other, they must either be separated by a barrier message or be put in the same ordered bundle. Examples of such message dependencies include a group mod add with a flow mod add referencing the group, a port mod with a packet-out forwarding to the port, or a flow mod add with a following packet-out to `OFPP_TABLE`.

6.3 OpenFlow Channel Connections

The OpenFlow channel is used to exchange OpenFlow messages between an OpenFlow switch and an OpenFlow controller. A typical OpenFlow controller manages multiple OpenFlow channels, each one to a different OpenFlow switch. An OpenFlow switch may have one OpenFlow channel to a single controller, or multiple channels for reliability, each to a different controller (see 6.3.7).

An OpenFlow controller typically manages an OpenFlow switch remotely over one or more networks. The specification of the networks used for the OpenFlow channels is outside the scope of the present specification. It may be a separate dedicated network (out-of-band controller connection), or the OpenFlow channel may use the network managed by the OpenFlow switch (in-band controller connection). The only requirement is that it should provide TCP/IP connectivity.

The OpenFlow channel is usually instantiated as a single network connection between the switch and the controller, using TLS or plain TCP (see 6.3.6). Alternatively, the OpenFlow channel may be composed

 © 2015; The Open Networking Foundation

of multiple network connections to exploit parallelism (see 6.3.8). The OpenFlow switch must be able to create an OpenFlow channel by initiating a connection to an OpenFlow controller (see 6.3.3). Some switch implementations may optionally allow an OpenFlow controller to connect to the OpenFlow switch, in this case the switch usually should restrict itself to secured connections (see 6.3.6) to prevent unauthorised connections.

6.3.1 Connection URI

The switch identifies a controller connection by a unique **Connection URI**. This URI must conform to the syntax for URI defined in RFC 3986, in particular with respect to character encoding. The Connection URI may have one of the following forms :

- *protocol*:*name-or-address*:*port*
- *protocol*:*name-or-address*

The *protocol* field defines the transport used for OpenFlow messages. Acceptable values for *protocol* are `tls` or `tcp` for main connections (see 6.3.6). Acceptable values for auxiliary connections (see 6.3.8) are `tls`, `dtls`, `tcp`, or `udp`.

The *name-or-address* field is the hostname or IP address of the controller. If the hostname is locally configured, it is recommended that the URI uses the IP address. If the hostname is available via DNS the URI may use either, so long as it is consistent. When the address is IPv6, the address should be enclosed in square brackets, as recommended in RFC 2732.

The *port* field is the transport port used on the controller. If the *port* field is not specified, it must be equivalent to setting this field to the default OpenFlow transport port 6653.

If the switch implementation supports a transport for the OpenFlow connection that is not defined by this specification, it must use a *protocol* that properly identifies that transport type. In this case, the rest of the Connection URI is protocol defined.

Some examples of Connection URIs are given below.

- `tcp:198.168.10.98:6653`
- `tcp:198.168.10.98`
- `tls:test.opennetworking.org:6653`
- `tls:[3ffe:2a00:100:7031::1]:6653`

The Connection URI always represents the switch's view of the connection information. If the address or port is translated by the network, this may not match the controller's view of the address or port. Controllers can request a unique ID not subject to network translation (see 6.3.7).

 © 2015; The Open Networking Foundation

6.3.2 Alternate connection transports

The current specification only defines OpenFlow main connections over `tls` and `tcp` (see 6.3.1). Auxiliary connections are optional, the current specification only defines auxiliary connections over `tls`, `tcp`, `dtls` and `udp` (see 6.3.8). Auxiliary connections on reliable transport (`tls` or `tcp`) can use the full OpenFlow protocol, however auxiliary connections on unreliable transport (`dtls` or `udp`) can only use a small subset of the OpenFlow protocol.

The full OpenFlow protocol can be used over an alternate transport protocol instead of `tls` or `tcp` : a main connection or a reliable auxiliary connection can be defined on an alternate transport protocol only if this transport protocol provide the following functionalities :

- **Multiplexing**: An OpenFlow switch must be able to connect to multiple controllers, or open multiple connections to the same controller, therefore the transport protocol must be able to identify and demultiplex the various OpenFlow connections.
- **Reliability**: The OpenFlow protocol does not have any error detection and recovery mechanisms, and assumes messages are never lost, therefore the transport protocol must be reliable.
- **Ordered delivery**: The OpenFlow Barrier message expects the order of OpenFlow messages on a connection to be preserved, therefore the transport protocol must deliver messages in order.
- **Segmentation and reassembly**: The size of an OpenFlow message can be up to 64kB, which exceeds most link MTU sizes, and OpenFlow does not offer a mechanism to fragment most messages, therefore the transport protocol must be able to perform fragmentation and reassembly of OpenFlow messages as needed.
- **Flow control**: The OpenFlow protocol assumes that the switch can slow down or stop the controller sending messages by stopping to read messages from the underlying connection, therefore the transport protocol must implement some flow control.
- **Security**: The OpenFlow protocol does not implement security by itself, therefore the transport protocol or a protocol underneath it must provide security if it is needed.

If the transport protocol can not meet these requirements, an adaptation layer must be used, for example layering TCP/IP over that transport. Beyond those requirements, the mapping of OpenFlow messages to the alternate transport protocol is defined by the adaptation layer or the transport protocol and outside the scope of this specification.

An unreliable auxiliary connection can be defined on an alternate transport protocol only if this transport protocol provides multiplexing (see above) and is restricted to the OpenFlow subset defined for unreliable transports (see 6.3.8).

6.3.3 Connection Setup

The switch must be able to establish communication with a controller at a user-configurable (but otherwise fixed) Connection URI, using either a user-specified transport port or the default OpenFlow transport port 6653. If the switch is configured with the Connection URI of the controller to connect to, the switch initiates a standard TLS or TCP connection to the controller as specified by the fields of the Connection URI. The switch must enable such connections to be established and maintained. For

 © 2015; The Open Networking Foundation

out-of-band connections, the switch must make sure that traffic to and from the OpenFlow channel is not run through the OpenFlow pipeline. For in-band connections, the switch must set up the proper set of flow entries for the connection in the OpenFlow pipeline.

Optionally, the switch may allow the controller to initiate the connection. In this case, the switch should accept incoming standard TLS or TCP connections from the controller, using either a user-specified transport port or the default OpenFlow transport port 6653. Connections initiated by the switch and the controller behave the same once the transport connection is established.

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 switch protocol version supported by the sender (see 7.1.1). This Hello message may optionally include some OpenFlow elements to help connection setup (see 7.5.1). Upon receipt of this message, the recipient must calculate the OpenFlow switch protocol version to be used. If both the Hello message sent and the Hello message received contained a `OFPHET_VERSIONBITMAP` hello element, and if those bitmaps have some common bits set, the negotiated version must be the highest version set in both bitmaps. Otherwise, the negotiated version must be the smaller of the version number that was sent and the one that was received in the `version` fields.

If the negotiated version is supported by the recipient, then the connection proceeds. Otherwise, the recipient must reply with a *Hello Failed* error message (see 7.5.4.1), and then terminate the connection.

After the switch and the controller have exchanged `OFPT_HELLO` messages and successfully negotiated a common version number, the connection setup is done and standard OpenFlow messages can be exchanged over the connection. One of the first things that the controller should do is to send a `OFPT_FEATURES_REQUEST` message to get the *Datapath ID* of the switch (see 7.3.1).

6.3.4 Connection Maintenance

OpenFlow connection maintenance is mostly done by the underlying TLS or TCP connection mechanisms, this insures that OpenFlow connections are supported on a wide range of networks and conditions. The main mechanisms to detect connection interruption and terminate the connection must be TCP timeouts and TLS session timeouts (when available).

Each connection must be maintained separately, if the connection with a specific controller or switch is terminated or broken, this should not cause connections to other controllers or switches to be terminated (see 6.3.7). Auxiliary connections must be terminated when the corresponding main connection is terminated or broken (see 6.3.8), on the other hand if an auxiliary connection is terminated or broken this should not impact the main connection or other auxiliary connections.

When a connection is terminated due to network conditions or timeout, if a switch or controller was the originator of the connection, it should attempt to reconnect to the other party until a new connection is established or until the Connection URI of the other party is removed from its configuration. Reconnection attempts, when done, should be done at increasing intervals to avoid overwhelming both the network and the other party, the initial interval should be greater than the underlying full TCP connection setup timeout.

OpenFlow messages are processed out of order (see 6.2), and the processing of some requests by the switch may take a long time, therefore a controller must never terminate a connection because a request

 © 2015; The Open Networking Foundation

is taking too much time. An exception to that rule is for *echo replies*, a controller or a switch may terminate a connection if the reply for an echo request it sent takes too much time, however there must be a way to disable that feature and the timeout should be large enough to accommodate a wide variety of conditions.

A controller or switch may terminate the connection when receiving an OpenFlow error message, for example if no common OpenFlow version can be found (see 6.3.3), or if an essential feature is not supported by the other party. In this case, the party that terminated the connection should not attempt to automatically reconnect.

Flow control is also done using the underlying TCP mechanisms (when available). For connections based on TCP, if the switch or controller can not process incoming OpenFlow messages fast enough, it should stop servicing that connection (stop reading messages from the socket) to induce TCP flow control to stop the sender. Controllers should monitor their sending queue and avoid it getting too large. For auxiliary connections based on UDP, if the switch or controller can not process incoming OpenFlow messages fast enough, it should drop excess messages.

6.3.5 Connection Interruption

In the case that a switch loses contact with a controller, it must send a Controller Status message to all remaining connected controllers (if any) indicating the role and control channel status of the affected controller. If the switch loses contact with multiple controllers, it must send multiple Controller Status messages, one for each affected controller. This allows remaining controllers to take action even if they have also lost communication with one another. When the OpenFlow channel is reestablished, the switch must send an updated Controller-Status message to all controllers.

In the case that a switch loses contact with all controllers, as a result of echo request timeouts, TLS session timeouts, or other disconnections, the switch must *immediately* enter either "fail secure mode" or "fail standalone mode", depending upon the switch implementation and configuration. In "fail secure mode", the only change to switch behavior is that packets and messages destined to the controllers are dropped. Flow entries should continue to expire according to their timeouts in "fail secure mode". In "fail standalone mode", the switch processes all packets using the `OFPP_NORMAL` reserved port; in other words, the switch acts as a legacy Ethernet switch or router. When in "fail standalone mode", the switch is free to use flow tables in any way it wishes, the switch may delete, add or modify any flow entry. The "fail standalone mode" is usually only available on Hybrid switches (see 5.1).

When the OpenFlow channel is reestablished, the flow entries present in the flow tables at that time are preserved and normal OpenFlow operation resumes. If desired, the controller has then the option of reading all flow entries with a *flow-stats* request (see 7.3.5.2), to re-synchronise its state with the switch state. Alternatively, the controller then has the option of deleting all flow entries with a *flow-mod* request (see 6.4), to start from a clean state on the switch.

The first time a switch starts up, it will operate in either "fail secure mode" or "fail standalone mode" mode, until it is successfully connected to a controller. Configuration of the default set of flow entries to be used at startup is outside the scope of the OpenFlow switch protocol.

 © 2015; The Open Networking Foundation

6.3.6 Encryption

The default security mechanism of the OpenFlow protocol is TLS (Transport Layer Security). The switch and controller may communicate through a TLS connection to provide authentication and encryption of the connection. The switch and the controller should use a secure version of TLS ; at the time of publication we recommend using TLS version 1.2 or greater.

If a valid Connection URI specifies `tls` as the protocol, a TLS connection is initiated by the switch on startup to the controller, which is listening either on a user-specified TCP port or on the default TCP port 6653. Optionally, the TLS connection is initiated by the controller to the switch, which is listening either on a user-specified TCP port or on the default TCP port 6653.

The switch and controller mutually authenticate by exchanging certificates signed by a site-specific private key (recommended). Each switch must be user-configurable with its own certificate and a site-specific public certificate which is used for authenticating the controller. Additionally, the switch may optionally support the configuration of multiple CA certificates as well as maintain Certificate Revocation Lists (CRLs) when establishing connections with multiple controllers. It is recommended to configure and manage all related security credentials (cipher settings, keys, and certificates) using a switch management protocol, such as the OpenFlow Configuration protocol.

Alternatively, for certain deployments, the switch can also support self-signed controller certificates or use pre-shared key exchange to authenticate the entities (not recommended).

The switch and controller may optionally communicate using plain TCP. Plain TCP can be used for non-encrypted communication (not recommended), or to implement an alternate security mechanism, such as using IPsec, VPN or a separate physical network, the detail of such configuration is outside the specification. If a valid Connection URI specifies `tcp` as the protocol, the TCP connection is initiated by the switch on startup to the controller, which is listening either on a user-specified TCP port or on the default TCP port 6653. Optionally, the TCP connection is initiated by the controller to the switch, which is listening either on a user-specified TCP port or on the default TCP port 6653. When using plain TCP, it is recommended to use alternative security measures to prevent eavesdropping, controller impersonation or other attacks on the OpenFlow channel.

6.3.7 Multiple Controllers

The switch may establish communication with a single controller, or may establish communication with multiple controllers. Having multiple controllers improves reliability, as the switch can continue to operate in OpenFlow mode if one controller or controller connection fails. The hand-over between controllers is initiated by the controllers themselves, which enables fast recovery from failure and also controller load balancing. The controllers coordinate the management of the switch amongst themselves via mechanisms outside the scope of the present specification, and the goal of the multiple controller functionality is only to help synchronise controller handoffs performed by the controllers. The multiple controller functionality only addresses controller fail-over and load balancing, and doesn't address virtualisation which can be done outside the OpenFlow switch protocol.

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

 © 2015; The Open Networking Foundation

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

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

A controller can request its role to be changed to `OFPCR_ROLE_SLAVE`. In this role, the controller has read-only access to the switch. By default, the controller does not receive switch asynchronous messages, apart from Port-status messages. The controller is denied the ability to execute all controller-to-switch commands that send packets or modify the state of the switch. For example, `OFPT_PACKET_OUT`, `OFPT_FLOW_MOD`, `OFPT_GROUP_MOD`, `OFPT_PORT_MOD`, `OFPT_TABLE_MOD` requests, and `OFPMP_TABLE_FEATURES` multiparts requests with a non-empty body must be rejected. If the controller sends one of those commands, the switch must reply with an *Is Slave* error message (see 7.5.4.2). Other controller-to-switch messages, such as `OFPT_ROLE_REQUEST`, `OFPT_SET_ASYNC` and `OFPT_MULTIPART_REQUEST` that only query data, should be processed normally.

A controller can request its role to be changed to `OFPCR_ROLE_MASTER`. This role is similar to `OFPCR_ROLE_EQUAL` and has full access to the switch, the difference is that the switch ensures it is the only controller in this role. When a controller changes its role to `OFPCR_ROLE_MASTER`, the switch changes the current controller with the role `OFPCR_ROLE_MASTER` to have the role `OFPCR_ROLE_SLAVE`, but does not affect controllers with role `OFPCR_ROLE_EQUAL`. When the switch performs such role changes, if a controller role is changed from `OFPCR_ROLE_MASTER` to `OFPCR_ROLE_SLAVE`, the switch must generate a controller role status event for this controller informing it of its new state (in many cases that controller is no longer reachable, and the switch may not be able to transmit that event).

Each controller may send an `OFPT_ROLE_REQUEST` message to communicate its role to the switch (see 7.3.8), and the switch must remember the role of each controller connection. A controller may change its role at any time, provided the `generation_id` in the message is current (see below).

The role request message offers a lightweight mechanism to help the controller master election process, the controllers configure their role and usually still need to coordinate among themselves. The switch can not change the state of a controller on its own, controller state is always changed as a result of a request from one of the controllers. Any Slave controller or Equal controller can elect itself Master. A switch may be *simultaneously* connected to multiple controllers in Equal state, multiple controllers in Slave state, and at most one controller in Master state. The controller in Master state (if any) and all the controllers in Equal state can fully change the switch state, there is no mechanism to enforce partitioning of the switch between those controllers. If the controller in Master role needs to be the only controller able to make changes on the switch, then no controllers should be in Equal state and all other controllers should be in Slave state.

Controllers may also use the `OFPT_ROLE_REQUEST` message to set the ID number the switch uses in Controller-Status messages. This is particularly useful when the OpenFlow connection traverses a NAT boundary that changes IP identifiers, since the URI in the Controller-Status message always represents the switch's view of the connection (see 6.3.1).

 © 2015; The Open Networking Foundation

A controller can also control which types of switch asynchronous messages are sent over its OpenFlow channel, and change the defaults described above. This is done via an *Asynchronous Configuration* message (see 6.1.1), listing all reasons for each message type that need to be enabled or filtered out (see 7.3.10) for the specific OpenFlow channel. Using this feature, different controllers can receive different notifications, a controller in master mode can selectively disable notifications it does not care about, and a controller in slave mode can enable notifications it wants to monitor.

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

On receiving a `OFPT_ROLE_REQUEST` with role equal to `OFPCR_ROLE_MASTER` or `OFPCR_ROLE_SLAVE` the switch must compare the `generation_id` in the message against the largest generation id seen so far. A message with a `generation_id` smaller than a previously seen generation id must be considered stale and discarded, and the switch must reply with a *Stale* error message (see 7.5.4.12).

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

On switch startup:

```
generation_is_defined = false;
```

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

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

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

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

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

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

 © 2015; The Open Networking Foundation

6.3.8 Auxiliary Connections

By default, the *OpenFlow channel* between an OpenFlow switch and an OpenFlow controller is a single network connection. The OpenFlow channel may also be composed of a **main connection** and multiple **auxiliary connections**. Auxiliary connections are created by the OpenFlow switch and are helpful to improve the switch processing performance and exploit the parallelism of most switch implementations. Auxiliary connections are always initiated by the switch, but may be configured by the controller.

Each connection from the switch to the controller is identified by the switch *Datapath ID* and an *Auxiliary ID* (see 7.3.1). The main connection must have its Auxiliary ID set to zero, whereas an auxiliary connection must have a non-zero Auxiliary ID and the same Datapath ID. Auxiliary connections must use the same source IP address as the main connection, but can use a different transport, for example TLS, TCP, DTLS or UDP, depending on the switch configuration. The auxiliary connection should have the same destination IP address and same transport destination port as the main connection, unless the switch configuration specifies otherwise. The controller must recognise incoming connections with non-zero Auxiliary ID as auxiliary connections and bind them to the main connection with the same Datapath ID.

The switch must not initiate auxiliary connection before having completed the connection setup over the main connection (see 6.3.3), it must setup and maintain auxiliary connections with the controller only while the corresponding main connection is alive. The connection setup for auxiliary connections is the same as for the main connection (see 6.3.3). If the switch detects that the main connection to a controller is broken, it must immediately close all its auxiliary connections to that controller, to enable the controller to properly resolve Datapath ID conflicts.

Both the OpenFlow switch and the OpenFlow controller must accept any OpenFlow message types and sub-types on all connections : the main connection or an auxiliary connection can not be restricted to a specific message type or sub-type. However, the processing performance of different message types or sub-types on different connections may be different. The switch may service auxiliary connections with different priorities, for example one auxiliary connection may be dedicated to high priority requests and always processed by the switch before other auxiliary connections. A switch configuration, for example using the OpenFlow Configuration Protocol, may optionally configure the priority of auxiliary connections.

A reply to an OpenFlow request must be made on the same connection it came in. There is no synchronisation between connections, and messages sent on different connections may be processed in any order. A barrier message applies only to the connection where it is used (see 6.2). Auxiliary connections using DTLS or UDP may lose or reorder messages, OpenFlow does not provide ordering or delivery guarantees on those connections (see 6.2). If messages must be processed in sequence, they must be sent over the same connection, use a connection that does not reorder packets, and use barrier messages.

The controller is free to use the various switch connections for sending OpenFlow messages at its entire discretion, however to maximise performance on most switches the following guidelines are suggested:

- All OpenFlow controller requests which are not Packet-out (flow-mod, statistic request...) should be sent over the main connection.
- Connection maintenance messages (hello, echo request, features request) should be sent on the main connection and on each auxiliary connection as needed.

 © 2015; The Open Networking Foundation

- All Packet-Out messages containing a packet from a Packet-In message should be sent on the connection where the Packet-In came from.
- All other Packet-Out messages should be spread across the various auxiliary connections using a mechanism keeping the packets of a same flow mapped to the same connection.
- If the desired auxiliary connection is not available, the controller should use the main connection.

The switch is free to use the various controller connections for sending OpenFlow messages as it wishes, however the following guidelines are suggested :

- All OpenFlow messages which are not Packet-in should be sent over the main connection.
- All Packet-In messages spread across the various auxiliary connection using a mechanism keeping the packets of a same flow mapped to the same connection.

Auxiliary connections on **unreliable transports** (UDP, DTLS) have additional restrictions and rules that don't apply to auxiliary connection on other transports (TCP, TLS). The only message types supported on unreliable auxiliary connections are `OFPT_HELLO`, `OFPT_ERROR`, `OFPT_ECHO_REQUEST`, `OFPT_ECHO_REPLY`, `OFPT_FEATURES_REQUEST`, `OFPT_FEATURES_REPLY`, `OFPT_PACKET_IN`, `OFPT_PACKET_OUT` and `OFPT_EXPERIMENTER`, other message types are not supported by the specification. Each UDP packet must contain an integral number of OpenFlow messages ; many OpenFlow messages may be sent in the same UDP packet and an OpenFlow message may not be split across UDP packets. Deployments using UDP and DTLS connections are advised to configure the network and switches to avoid IP fragmentation of UDP packets.

On unreliable auxiliary connections, *Hello messages* are sent at connection initiation to setup the connection (see 6.3.3). If an OpenFlow device receives another message on an unreliable auxiliary connection prior to receiving a *Hello message*, the device must either assume the connection is setup properly and use the version number from that message, or it must return an Error message with `OFPET_BAD_REQUEST` type and `OFPBRC_BAD_VERSION` code. If an OpenFlow device receives an error message with `OFPET_BAD_REQUEST` type and `OFPBRC_BAD_VERSION` code on an unreliable auxiliary connection, it must either send a new *Hello message* or terminate the unreliable auxiliary connection (the connection may be retried at a later time). If no message was ever received on an auxiliary connection after some implementation chosen amount of time lower than 5 seconds, the device must either send a new *Hello* message or terminate the unreliable auxiliary connection. If after sending a *Feature Request* message, the controller does not receive a *Feature Reply* message after some implementation chosen amount of time lower than 5 seconds, the device must either send a new *Feature Request* message or terminate the unreliable auxiliary connection. If after receiving a message, a device does not receive any other message after some implementation chosen amount of time lower than 30 seconds, the device must terminate the unreliable auxiliary connection. If a device receives a message for an unreliable auxiliary connection already terminated, it must assume it is a new connection.

OpenFlow devices using unreliable auxiliary connections should follow recommendations in RFC 5405 when possible. Unreliable auxiliary connections can be considered as an encapsulation tunnel, as most OpenFlow messages are prohibited apart from `OFPT_PACKET_IN` and `OFPT_PACKET_OUT` messages. If some of the packets encapsulated in the `OFPT_PACKET_IN` messages are not TCP packet or do not belong to a protocol having proper congestion control, the controller should configure the switch to have those messages processed by a Meter or a Queue to manage congestion.

 © 2015; The Open Networking Foundation

6.4 Flow Table Modification Messages

Flow table modification messages can have the following types:

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

For **add** requests (OFPFC_ADD) with the OFPFF_CHECK_OVERLAP flag set, the switch must first check for any overlapping flow entries in the requested table. Two flow entries overlap if a single packet may match both, and both flow entries have the same priority, but the two flow entries don't have the exact same match. If an overlap conflict exists between an existing flow entry and the **add** request, the switch must refuse the addition and respond with an *Overlap* error message (see 7.5.4.6).

For non-overlapping **add** requests, or those with no overlap checking, the switch must insert the flow entry in the requested table. If a flow entry with identical match fields and priority already resides in the requested table, then that entry, including its duration, must be cleared from the table, and the new flow entry added. If the OFPFF_RESET_COUNTS flag is set, the flow entry counters must be cleared, otherwise they should be copied from the replaced flow entry. No flow-removed message is generated for the flow entry eliminated as part of an **add** request; if the controller wants a flow-removed message it should explicitly send a **delete** request for the old flow entry prior to adding the new one.

For **modify** requests (OFPFC_MODIFY or OFPFC_MODIFY_STRICT), if a matching entry exists in the table, the `instructions` field of this entry is replaced with the value from the request, whereas its `cookie`, `idle_timeout`, `hard_timeout`, `importance`, `flags`, counters and duration fields are left unchanged. If the OFPFF_RESET_COUNTS flag is set, the flow entry counters must be cleared. For **modify** requests, if no flow entry currently residing in the requested table matches the request, no error is recorded, and no flow table modification occurs.

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

Modify and **delete** flow_mod commands have *non-strict* versions (OFPFC_MODIFY and OFPFC_DELETE) and *strict* versions (OFPFC_MODIFY_STRICT or OFPFC_DELETE_STRICT). In the *strict* versions, the set of match fields, all match fields, including their masks, and the priority, are strictly matched against the entry, and only an identical flow entry is modified or removed. For example, if a message to remove entries is sent that has no match fields included, the OFPFC_DELETE command would delete all flow entries from the tables, while the OFPFC_DELETE_STRICT command would only delete a flow entry that applies to all packets at the specified priority.

For *non-strict* **modify** and **delete** commands, all flow entries that match the flow_mod description are modified or removed. Those commands can not apply partially : if one flow entry can not be

 © 2015; The Open Networking Foundation

modified or deleted, then none of the matching flow entries is modified or removed and an error is returned. In the *non-strict* versions, a match will occur when a flow entry exactly matches or is more specific than the description in the flow_mod command; in the flow_mod the missing match fields are wildcarded, field masks are active, and other flow_mod fields such as priority are ignored. For example, if an `OFPFC_DELETE` command says to delete all flow entries with a destination port of 80, then a flow entry that wildcards all match fields will not be deleted. However, an `OFPFC_DELETE` command that wildcards all match fields will delete an entry that matches all port 80 traffic. This same interpretation of mixed wildcard and exact match fields also applies to individual and aggregate flows stats requests.

Delete commands can be optionally filtered by destination group or output port. If the `out_port` field contains a value other than `OFPP_ANY`, it introduces a constraint when matching. This constraint is that each matching flow entry must contain an *output* action directed at the specified port in the actions associated with that flow entry. This constraint is limited to only the actions directly associated with the flow entry. In other words, the switch must not recurse through the action buckets of pointed-to groups, which may have matching *output* actions. The `out_group`, if different from `OFPG_ANY`, introduces a similar constraint on the *group* action. These fields are ignored by `OFPFC_ADD`, `OFPFC_MODIFY` and `OFPFC_MODIFY_STRICT` messages.

Modify and **delete** commands can also be filtered by cookie value, if the `cookie_mask` field contains a value other than 0. This constraint is that the bits specified by the `cookie_mask` in both the `cookie` field of the flow mod and a flow entry's `cookie` value must be equal. In other words, $(flow_entry.cookie \,\&\, flow_mod.cookie_mask) == (flow_mod.cookie \,\&\, flow_mod.cookie_mask)$.

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

If the switch supports eviction, a flow table may be configured to perform eviction when flow entries are added to that table (see 7.3.4.1). If eviction is enabled on a flow table, and if that flow table is full, an **add** request will use the eviction mechanism to try to make space for the new flow entry. The eviction process is switch defined. If eviction is successful, the evicted flow entries are removed from the flow table (see 6.5) and new flow entry from the request is inserted in the flow table. If eviction is not successful, no flow entry is removed and the **add** request returns a *Table Full* error message.

If a switch cannot find any space in the requested table in which to add the incoming flow entry, the switch must return a *Table Full* error message (see 7.5.4.6).

The switch must validate the content *Flow-mod* message and must return the appropriate **error** messages for instruction errors (see 7.5.4.4), match errors (see 7.5.4.5), action errors (see 7.5.4.3), meter errors (see 7.5.4.13) and other flow-mod errors (see 7.5.4.6).

6.5 Flow Removal

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

The switch **flow expiry** mechanism is run by the switch independently of the controller and is based on the state and configuration of flow entries. Each flow entry has an `idle_timeout` and a `hard_timeout` associated with it (see 7.3.4.2). If the `hard_timeout` field is non-zero, the switch must note the flow entry's arrival time, as it may need to evict the entry later. A non-zero `hard_timeout` field causes

 © 2015; The Open Networking Foundation

the flow entry to be removed after the given number of seconds, regardless of how many packets it has matched. If the `idle_timeout` field is non-zero, the switch must note the arrival time of the last packet associated with the flow, as it may need to evict the entry later. A non-zero `idle_timeout` field causes the flow entry to be removed when it has matched no packets in the given number of seconds. The switch must implement flow expiry and remove flow entries from the flow table when one of their timeouts is exceeded.

The controller may actively remove flow entries from flow tables by sending **delete** flow table modification messages (`OFPFC_DELETE` or `OFPFC_DELETE_STRICT` - see 6.4). Flow entries may also be removed as the result of removal of a group (see 6.7) or a meter (see 6.8) by the controller. On the other hand, when a port is added, modified or removed, flow entries are never removed or modified, the controller must explicitly remove those flow entries if needed (see 4.6).

Flow entries may be **evicted** from flow tables when the switch needs to reclaim resources (see 6.4). Flow entry eviction occurs only on flow tables where it is explicitly enabled (see 7.3.3). Flow entry eviction is an optional feature, and the mechanism used to select which flow entries to evict is switch defined and may depend on flow entry parameters, resource mappings in the switch and other internal switch constraints.

When a flow entry is removed, either by the controller, the flow expiry mechanism, or the optional switch eviction mechanism, the switch must check the flow entry's `OFPFF_SEND_FLOW_REM` flag. If this flag is set, the switch must send a flow removed message to the controller (see 7.4.2). Each flow removed message contains a complete description of the flow entry, the reason for removal (expiry, delete or eviction), the flow entry duration at the time of removal, and the flow statistics at the time of removal.

6.6 Flow Table Synchronisation

A flow table may optionally be *synchronised* with another flow table. When a flow table is synchronised, its content is automatically updated by the switch to reflect changes in the flow table it is synchronised with. This mechanism enables a switch to perform multiple matches on different views of the same data at different points of the OpenFlow pipeline.

Flow tables can be synchronised unidirectionally (one flow table synchronises from another flow table) or bidirectionally (two flow tables synchronise from each other), a switch may also define a flow table that synchronise from multiple tables or that synchronise on itself. The synchronisation of flow tables is described in the table features message, each synchronised flow table includes a table property identifying the source flow tables it synchronises from (see 7.3.5.18). For a synchronised flow table, when a flow entry is added, modified or removed in the source flow table, a corresponding flow entry must be automatically added, modified or removed in the synchronised flow table by the switch. If a flow entry is added in the source flow table with the `OFPFF_CHECK_OVERLAP` flag set, and if the corresponding flow entry would overlap a flow entry already residing in the synchronised flow table, the switch must refuse the full flow-mod request and respond with a *Can't Sync* error message (see 7.5.4.6). If a flow entry is added in the source flow table, and the corresponding flow entry has identical match and priority to an older flow entry already residing in the synchronised flow table, the existing flow entry must be replaced by the synchronised flow entry, no flow removed message is generated for the old flow entry. If a flow entry is modified or removed in the source flow table, and if the corresponding flow entry no longer exists in the synchronised flow table, no flow entry should be modified, removed or created in the synchronised flow table.

 © 2015; The Open Networking Foundation

The flow entry created in the synchronised table may not be identical to the corresponding flow entry in the source flow table, as they are often different views on the same data. The translation between those synchronised flow entries is not specified by the OpenFlow protocol and depends on the switch implementation and configuration. For example, synchronised flow entries may have a different instruction set and actions, and the match fields may be transposed (source and destination inverted). If the flows tables are synchronised bidirectionally, the translation must be bijective, each translation must be the exact reverse of the other translation, so that a translated flow entry will always translate back to the original source flow entry. The counters of a flow entry must be kept independently of counters in the source flow table, those counters are not synchronised. In general, synchronised flow entries are created as permanent flow entries (both expiry timers set to zero) so that their lifetime is properly synchronised with the flow entries in the source flow table.

The switch may optionally allow the controller to modify or delete those flow entries automatically added in the synchronised table, and may also allow the controller to create new entries in the synchronised table. This enables the controller to customise those synchronised flow entries beyond the switch automatic translation. These changes may be lost if the source flow entry is modified. If flow table synchronisation is bidirectional (both flow table synchronised on each other), changes done by the controller on a flow entry created by the synchronisation mechanism need to be reflected back on the other synchronised flow table (like it would be for a non-synchronised flow entry in that flow table).

The switch may have restrictions on flow table synchronisation. If the switch can't synchronise a flow mod request generated by the controller in the source flow table, the switch must reject the flow mod and immediately return a *Can't Sync* error message (see 7.5.4.6). If the controller attempts to add, modify or delete a flow entry in a synchronised flow table and the switch can't support such changes, the switch must reject the flow mod and immediately return an *Is Sync* error message (see 7.5.4.6).

The flow table synchronisation feature could in theory be used to describe complex synchronisation interactions, such as table synchronising from multiple tables or table synchronising from synchronised tables. This specification recommends to only describe simple and common synchronisation configurations, such as those involving only two tables. In particular, synchronisation loops should be avoided.

An example of the use of synchronised tables is to map a Ethernet learning/forwarding hardware table into OpenFlow as a set of two synchronised flow tables. There are multiple ways synchronised flow tables can be used for Ethernet learning/forwarding, this specification makes no recommendations on which way to use. In this illustrative and simplistic example, a first flow table expresses the learning lookup and matches on Ethernet source address and input port (and optionally other fields such as VLAN ID). A second flow table expresses the forwarding lookup and matches on Ethernet destination address and sets the output port. The two tables synchronise from each other, with the Ethernet addresses transposed in the match. Flow entries in the first table are set to expire based on the learning ageing timer, flow entries in the second table are set without expiry timer and are removed automatically when the corresponding flow entry in the first table expires. The controller can add a flow entry in either of those two flow tables, and then modify the instructions of the flow entry automatically added in the other flow table. Another example of using synchronised tables is mapping RPF checks for multicast packets as a flow table.

 © 2015; The Open Networking Foundation

6.7 Group Table Modification Messages

The Group table modification message includes a command and the various components of the group needed for the command (see 7.3.4.3). The command in a Group-Mod message can be one of following :

```
/* Group commands */
enum ofp_group_mod_command {
    OFPGC_ADD    = 0,       /* New group. */
    OFPGC_MODIFY = 1,       /* Modify all matching groups. */
    OFPGC_DELETE = 2,       /* Delete all matching groups. */
    OFPGC_INSERT_BUCKET = 3,/* Insert action buckets to the already available
                               list of action buckets in a matching group */
    /* OFPGC_??? = 4, */    /* Reserved for future use. */
    OFPGC_REMOVE_BUCKET = 5,/* Remove all action buckets or any specific action
                               bucket from matching group */
};
```

The message includes in many cases action buckets (see 5.10). The set of actions for each bucket must be validated using the same rules as those for flow mods (see 7.5.4.3), with additional group-specific checks (see 7.5.4.7).

A group-mod request with an **add** command (`OFPGC_ADD`) adds a new group entry in the group table. If the specified group already exist, the switch must return a *Group Exist* error message (see 7.5.4.7).

For **modify** requests (`OFPGC_MODIFY`), if a group entry with the specified group identifier already resides in the group table, then the configuration of that group entry, including its type and action buckets, must be removed (cleared), and the group entry must use the new configuration included in the request. For that entry, group top-level statistics are preserved (continued), group bucket statistics are reset (cleared). If the specified group does not already exist, the switch must refuse the group mod and send a *Unknown Group* error message (see 7.5.4.7).

For **delete** requests (`OFPGC_DELETE`), if no group entry with the specified group identifier currently exists in the group table, no error is recorded, and no group table modification occurs. Otherwise, the group is removed, and all flow entries containing this group in a Group action are also removed. The group type need not be specified for the **delete** request. **Delete** also differs from an **add** or **modify** with no buckets specified in that future attempts to **add** the group identifier will not result in a group exists error. If one wishes to effectively delete a group yet leave in flow entries using it, that group can be cleared by sending a **modify** with no buckets specified.

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

For **insert bucket** requests (`OFPGC_INSERT_BUCKET`), if a group entry with the specified group identifier already resides in the group table, new action buckets are inserted at a specified position in the current action bucket list. For that group, group top-level statistics and group bucket statistics for existing buckets are preserved (continued).

In a request of type `OFPGC_INSERT_BUCKET`, the `command_bucket_id` field is used to specify the position in the current action bucket list to insert new action buckets. The following values for the `command_bucket_id` field in an `OFPGC_INSERT_BUCKET` request are supported :

 © 2015; The Open Networking Foundation

- If command_bucket_id is OFPG_BUCKET_FIRST, the new action buckets will be inserted at the beginning of the list just before the current action buckets, that is, the entire current action bucket list will be pushed and appended to the new action buckets. If the current action bucket list is empty, the new actions buckets just replace the current action bucket list.
- If command_bucket_id is OFPG_BUCKET_LAST, the new action buckets will be inserted after the current bucket list, that is, new action buckets are appended to the current action bucket list. If the current action bucket list is empty, the new actions buckets just replace the current action bucket list.
- If command_bucket_id is a valid identifier in the current action bucket list, the new action buckets will be inserted after command_bucket_id value, that is, new action buckets will be inserted between the bucket identifier value mentioned in command_bucket_id and the next bucket of the list. In case the command_bucket_id identifies the last bucket, the command_bucket_id will be treated as OFPG_BUCKET_LAST.

If command_bucket_id field value is not one of the reserved values above or does not exist in the current action bucket list, the switch must return a *Unknown Bucket* error message (see 7.5.4.7).

For **remove bucket** requests (OFPGC_REMOVE_BUCKET), if a group entry with the specified group identifier already resides in the group table, removes all buckets or the specified bucket from the current action bucket list. For that group, group top-level statistics and group bucket statistics for remaining buckets are preserved (continued).

In a request of type OFPGC_REMOVE_BUCKET, the command_bucket_id field is used to specify the identifier of the bucket to remove from current action bucket list. The following values for the command_bucket_id field in anOFPGC_REMOVE_BUCKET request are supported :

- If command_bucket_id is OFPG_BUCKET_ALL, removes all action buckets from the group and thereby the current action bucket list of the group becomes empty.
- If command_bucket_id is OFPG_BUCKET_FIRST, removes first action bucket from the current action bucket list.
- If command_bucket_id is OFPG_BUCKET_LAST, removes last action bucket from the current action bucket list.
- If command_bucket_id is a valid identifier in the current action bucket list, removes specified action bucket from the list.

Groups may be **chained** if the switch supports it, when at least one group forwards to another group, or in more complex configuration. For example, a fast reroute group may have two buckets, where each points to a select group.

A switch may support checking that no loop is created while chaining groups : if a group mod is sent such that a forwarding loop would be created, the switch must reject the group mod and must send a *Loop* error message (see 7.5.4.7). If the switch does not support such checking, the forwarding behavior is undefined.

A switch may support checking that groups forwarded to by other groups are not removed : If a switch cannot delete a group because it is referenced by another group, it must refuse to delete the group entry and must send a *Chained Group* error message (see 7.5.4.7). If the switch does not support such checking, the forwarding behavior is undefined.

 © 2015; The Open Networking Foundation

6.8 Meter Modification Messages

Meter modification messages can have the following commands :

```
/* Meter commands */
enum ofp_meter_mod_command {
    OFPMC_ADD    = 0,       /* New meter. */
    OFPMC_MODIFY = 1,       /* Modify specified meter. */
    OFPMC_DELETE = 2,       /* Delete specified meter. */
};
```

A meter-mod request with an **add** command (`OFPMC_ADD`) adds a new meter entry in the switch. If the specified meter already exists, the switch must return a *Meter Exist* error message (see 7.5.4.13).

For **modify** requests (`OFPMC_MODIFY`), if a meter entry with the specified meter identifier already exists, then the configuration of that meter entry, including its flags and bands, must be removed (cleared), and the meter entry must use the new configuration included in the request. For that entry, meter top-level statistics are preserved (continued), meter band statistics are reset (cleared). If a meter entry with the specified meter identifier does not already exist then the switch must refuse the meter mod and send a *Unknown Meter* error message (see 7.5.4.13).

For **delete** requests (`OFPMC_DELETE`), if no meter entry with the specified meter identifier currently exists, no error is recorded, and no meter modification occurs. Otherwise, the meter is removed, and all flow entries that include the meter in their instruction set are also removed. Only the meter identifier needs to be specified for the **delete** request, other fields such as `bands` can be omitted.

To delete all meters with a single message, specify `OFPM_ALL` as the meter value. Virtual meters can never be deleted and are not removed when deleting all meters.

6.9 Bundle Messages

6.9.1 Bundle overview

A bundle is a sequence of OpenFlow modification requests from the controller that is applied as a single OpenFlow operation. If all modifications in the bundle succeed, all of the modifications are retained, but if any errors arise, none of the modifications are retained.

The first goal of bundles is to group related state changes on a switch so that all changes are applied together or that none of them are applied. The second goal is to better synchronise changes across a set of OpenFlow switches : bundles can be prepared and pre-validated on each switch and then applied by the controller approximately at the same time.

A bundle is specified as all messages conveyed in Bundle Add messages with the same `bundle_id` on a specific controller connection (see 7.3.9.7). Messages conveyed in a Bundle Add message are formatted like regular OpenFlow messages and have the same semantics. The messages included in a bundle are pre-validated as they are stored in the bundle, minimising the probability of errors when the bundle is applied. The application of the message included in the Bundle Add message is postponed to when the bundle is committed (see 7.3.9.9).

 © 2015; The Open Networking Foundation

Bundle is an optional feature and may not be supported by all OpenFlow switches. A switch is not required to accept arbitrary messages in a bundle, a switch may not accept some message types in bundles, and a switch may not allow all combinations of message types to be bundled together (see 7.3.9.7). For example, a switch should not allow to embed a bundle message within a Bundle Add message. At a minimum, if a switch supports bundles, it must be able to support a bundle of multiple flow-mods and port-mods in any order.

6.9.2 Bundle example usage

The controller can issue the following sequence of messages to apply a sequence of modifications together.

1. `OFPBCT_OPEN_REQUEST` `bundle_id`
2. `OFPT_BUNDLE_ADD_MESSAGE` `bundle_id` *modification 1*
3. `OFPT_BUNDLE_ADD_MESSAGE` `bundle_id` ...
4. `OFPT_BUNDLE_ADD_MESSAGE` `bundle_id` *modification n*
5. `OFPBCT_CLOSE_REQUEST` `bundle_id`
6. `OFPBCT_COMMIT_REQUEST` `bundle_id`

The switch is expected to behave as follows. When a bundle is opened, modifications are saved into a temporary staging area without taking effect. When the bundle is committed, the changes in the staging area are applied to the state (e.g. tables) used by the switch. If an error occurs in one modification, no change is applied to the state.

6.9.3 Bundle error processing

The OpenFlow messages part of a bundle must be pre-validated before they are stored in the bundle (see 7.3.9.7). For each message sent by the controller, the switch must verify that the syntax of the message is valid and that all features in the message are supported features, and immediately return an error message if this message can not be validated. The switch may optionally verify resource availability and may commit resources at this time and generate errors. Messages generating errors when added to the bundle are not stored in the bundle and the bundle is unmodified.

When the bundle is committed, most errors will have been already detected and reported. One of the message included in the bundle may still fail during commit, for example due to resource availability. In this case, no message part of the bundle is applied and the switch must generate the error message corresponding to the failure (see 7.3.9.9). Messages of a bundle should have unique `xid` to help matching errors to messages. If none of the messages part of the bundle generate an error message, the switch informs the controller of the successful application of the bundle.

 © 2015; The Open Networking Foundation

6.9.4 Bundle atomic modifications

Committing the bundle must be controller atomic, i.e. a controller querying the switch must never see the intermediate state, it must see either the state of the switch with none or with all of the modifications contained in the bundle having been applied. In particular, if a bundle fails, controllers should not receive any notification resulting from the partial application of the bundle.

If the flag `OFPBF_ORDERED` is specified (see 7.3.9.3), the messages part of the bundle must be applied strictly in order, as if separated by `OFPT_BARRIER_REQUEST` messages (however no `OFPT_BARRIER_REPLY` is generated). If this flag is not specified, messages don't need to be applied in order.

If the flag `OFPBF_ATOMIC` is specified (see 7.3.9.3), committing the bundle must also be packet atomic, i.e. a given packet from an input port or packet-out request should either be processed with none or with all of the modifications having been applied. Whether this flag is supported would depend on the switch hardware and software architecture. Packets and messages can temporarily be enqueued while changes are applied. As the resulting increase in forwarding / processing latency may be unacceptable, double buffering techniques are often employed.

If the flag `OFPBF_ATOMIC` is not specified, committing the bundle does not need to be packet atomic. Packet may be processed by intermediate state resulting from partial application of the bundle, even if the bundle commit ultimately fails. The various OpenFlow counters would also reflect the partial application of the bundle in this case.

6.9.5 Bundle parallelism

The switch must support exchanging echo request and echo reply messages during the creation and population of the bundle, the switch must reply to an echo request without waiting for the end of the bundle. Echo request and echo reply messages can not be included in a bundle. Similarly, asynchronous messages generated by the switch are not impacted by the bundle, the switch must send status events without waiting for the end of the bundle.

If the switch supports multiple controller channels or auxiliary connections, the switch must maintain a separate bundle staging area for each controller-switch connection. This permits multiple bundles to be incrementally populated in parallel on a given switch. The `bundle_id` namespace is specific to each controller connection, so that controllers don't need to coordinate.

A switch may also optionally support populating multiple bundle in parallel on the same controller connection, by multiplexing bundle messages with different `bundle_id`. The controller may create and send multiple bundles, each identified by a unique `bundle_id`, and then can apply any of them in any order by specifying its `bundle_id` in a commit message. Inversely, a switch may not allow to create another bundle or accept any regular OpenFlow messages between opening a bundle and either committing it or discarding it.

In some implementations, when a switch starts storing a bundle, it may lock some of the objects referenced by the bundle. If a message on the same or another controller connection tries to modify an object locked by a bundle, the switch must reject that message and return an error. A switch is not required to lock objects referenced by a bundle. If a switch does not lock objects referenced by a bundle, the application of the bundle may fail if those objects have been modified via other controller connections.

 © 2015; The Open Networking Foundation

6.9.6 Scheduled Bundles

A bundle commit request may include an *execution time*, specifying *when* the bundle should be committed. A switch that receives a scheduled bundle, commits the bundle as close as possible to the execution time that was specified in the commit message.

6.9.6.1 The Scheduled Bundle Procedure

The scheduled bundle procedure is illustrated in Figure 6:

1. The controller starts the bundle procedure by sending an `OFPBCT_OPEN_REQUEST`, and receives a reply from the switch.
2. The controller then sends a set of N `OFPT_BUNDLE_ADD_MESSAGE` messages, for some $N \geq 1$.
3. The controller may then send an `OFPBCT_CLOSE_REQUEST`. The close request is optional, and thus the controller may skip this step.
4. The controller sends an `OFPBCT_COMMIT_REQUEST`. The `OFPBCT_COMMIT_REQUEST` includes the `OFPBF_TIME` flag, which indicates whether this is a *scheduled commit*. A scheduled commit request includes the time property, `ofp_bundle_prop_time`, which contains the scheduled time at which the switch is expected to apply the bundle.
5. After receiving the commit message, the switch applies the bundle at the scheduled time, T_s, and sends a `OFPBCT_COMMIT_REPLY` to the controller.

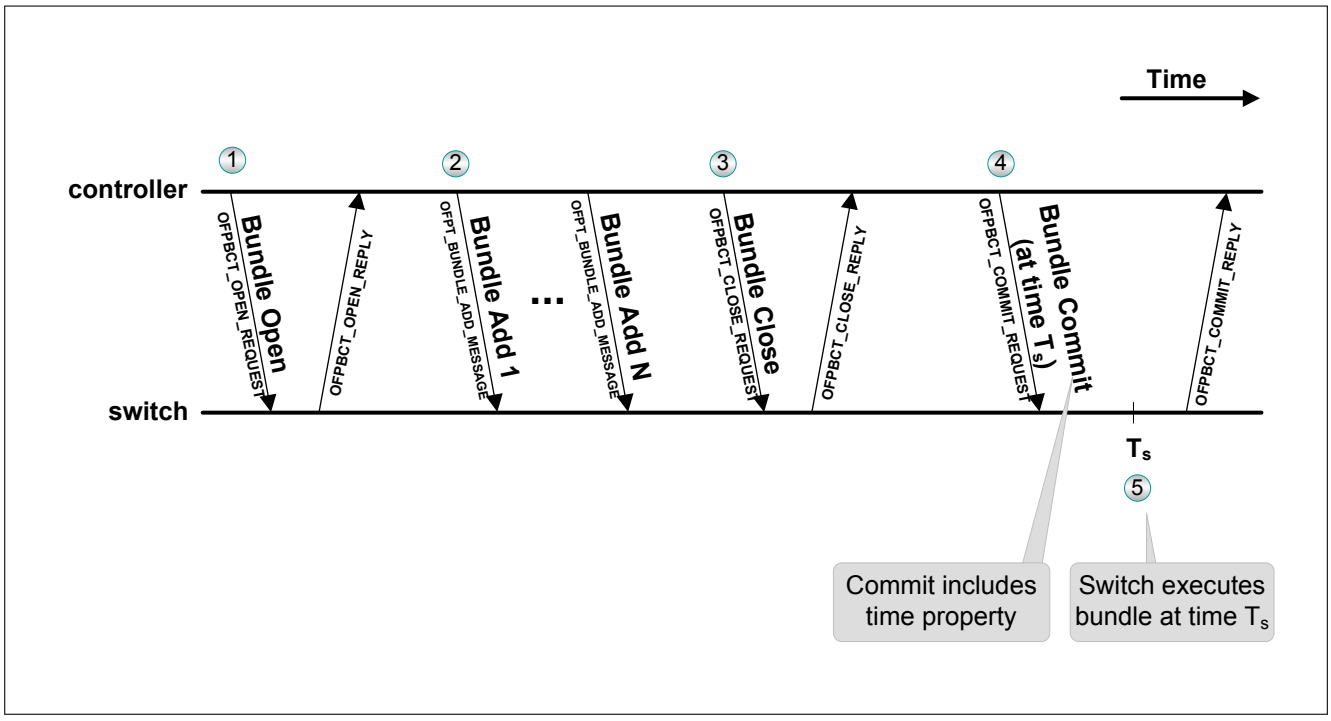


Figure 6: Scheduled Bundle Procedure

 © 2015; The Open Networking Foundation

6.9.6.2 Discarding Scheduled Bundles

The controller may cancel a scheduled bundle by sending an `OFPT_BUNDLE_CONTROL` message with type `OFPBCT_DISCARD_REQUEST`. An example is shown in Figure 7; if the switch receives the scheduled commit message and is not able to schedule the bundle, it responds to the controller with an error message. This indication may be used for implementing a coordinated update where either all the switches successfully schedule the operation, or the bundle is discarded; when a controller receives a scheduling error message from one of the switches it can send a discard message (step 5' in in Figure 7) to other switches that need to commit a bundle at the same time, and abort the bundle.

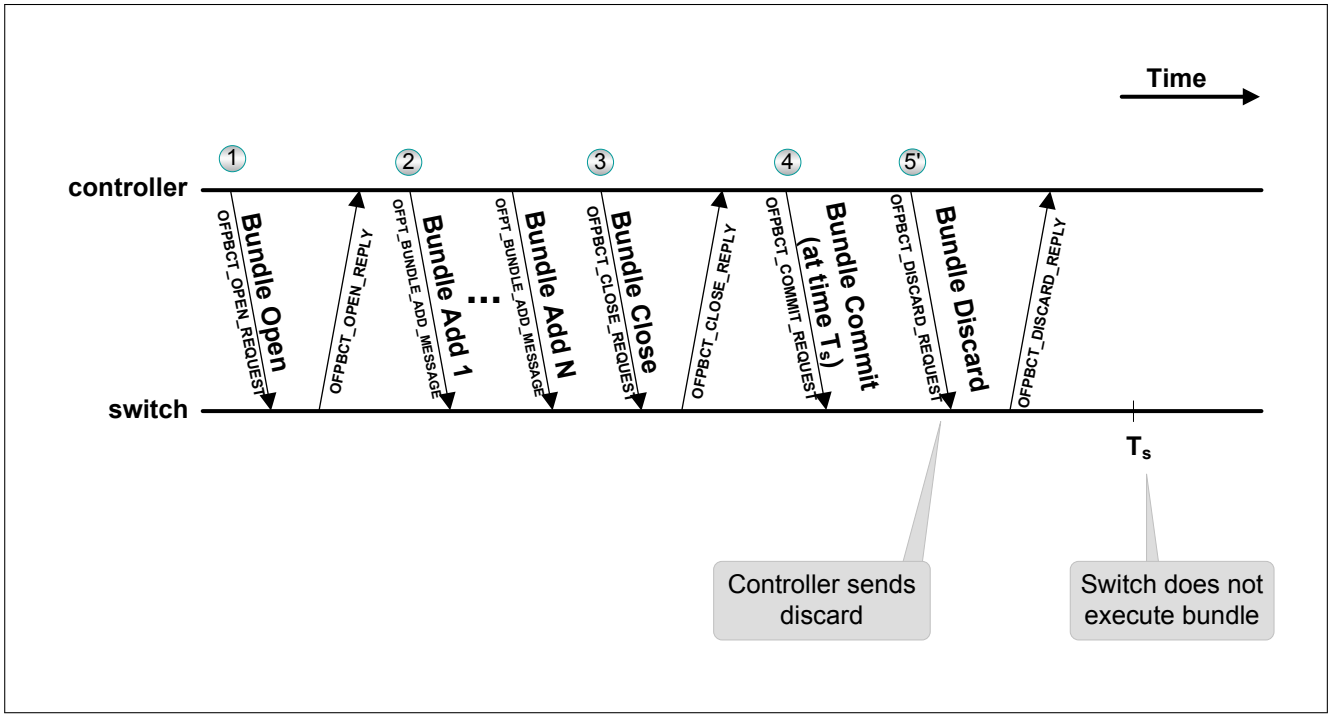


Figure 7: Discarding a Scheduled Commit

6.9.6.3 Timekeeping and Synchronization

Every switch that supports scheduled bundles must maintain a clock. It is assumed that clocks are synchronized by a method that is outside the scope of this specification, e.g., the Network Time Protocol (NTP) or the Precision Time Protocol (PTP).

Two factors affect how accurately a switch can commit a scheduled bundle; one factor is the accuracy of the clock synchronization method used to synchronize the switches' clocks, and the second factor is the switch's ability to execute real-time operations, which greatly depends on how it is implemented.

This specification does not define any requirements pertaining to the degree of accuracy of performing scheduled operations. However, every switch that supports the scheduled bundles must be able to report

 © 2015; The Open Networking Foundation

its estimated scheduling accuracy to the controller. The controller can retrieve this information from the switch using the `OFPMP_BUNDLE_FEATURES` message, defined in Section 7.3.5.20.

Since a switch does not commit a bundle instantaneously, the processing time of required operations should not be overlooked; the scheduled time and execution time always refer to the *start time* of the relevant operation.

6.9.6.4 Scheduling Tolerance

When a switch receives a scheduled commit request, it must verify that the scheduled time, T_s, is not too far in the past or in the future. As illustrated in Figure 8, the switch verifies that T_s is within the *scheduling tolerance* range.

The lower bound on T_s verifies the freshness of the bundle so as to avoid acting upon old and possibly irrelevant messages. Similarly, the upper bound on T_s guarantees that the switch does not take a long-term commitment to execute a bundle that may become obsolete by the time it is scheduled to be invoked.

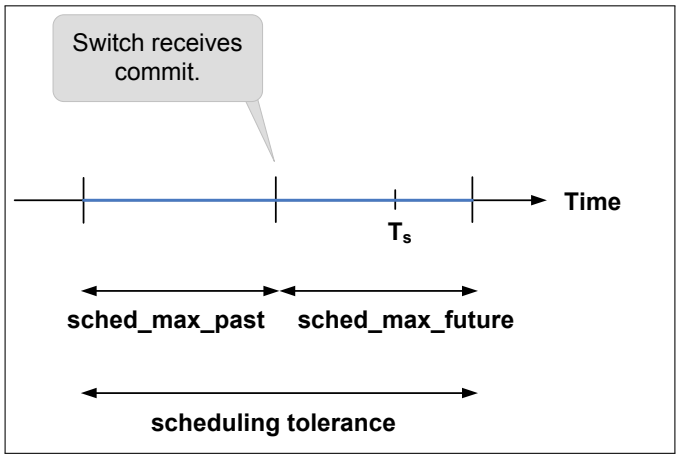


Figure 8: Scheduling Tolerance

The scheduling tolerance is determined by two parameters, `sched_max_future` and `sched_max_past`. The default value of these two parameters is 1 second. The controller may set these fields to a different value using the bundle features request, as described in Section 7.3.5.20.

If the scheduled time, T_s is within the scheduling tolerance range, the scheduled bundle is committed; if T_s occurs in the past and within the scheduling tolerance, the switch commits the bundle as soon as possible. If T_s is a future time, the switch commits the bundle as close as possible to T_s. If T_s is not within the scheduling tolerance range, the switch responds to the controller with an error message.

7 The OpenFlow Switch Protocol

The heart of the OpenFlow switch specification is the set of structures used for OpenFlow Switch Protocol messages.

 © 2015; The Open Networking Foundation

7.1 Protocol basic format

The OpenFlow protocol is implemented using OpenFlow messages transmitted over the OpenFlow channel (see 6). Each message type is described by a specific structure, which starts with the common OpenFlow header (see 7.1.1), and the message structure may include other structures which may be common to multiple message types (see 7.2). Each structure defines the order in which information is included in the message and may contain other structures, values, enumerations or bitmasks (see 7.1.3).

The structures, defines, and enumerations described below are derived from the file `openflow.h` which is included in this document (see Appendix A). Most structures are packed with padding and 8-byte aligned, as checked by the assertion statements (see 7.1.2). All OpenFlow messages are sent in big-endian format.

OpenFlow is a simple binary protocol, therefore invalid OpenFlow messages generated by OpenFlow implementations will in general result in the message having the wrong length or some fields having invalid values (see 7.1.3). It is recommended to monitor those conditions to detect non-compliant implementations.

7.1.1 OpenFlow Header

Each OpenFlow message begins with the OpenFlow header:

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

The `version` specifies the OpenFlow switch protocol version being used. The most significant bit in the version field is reserved and must be set to 0. The 7 lower bits indicate the revision number of the protocol. The version of the protocol described by the current specification is 1.5.1, and its *ofp_version* is 0x06 .

The `length` field indicates the total length of the message, so no additional framing is used to distinguish one frame from the next. The `type` can have the following values:

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

 © 2015; The Open Networking Foundation

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

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

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

    /* Multipart messages. */
    OFPT_MULTIPART_REQUEST      = 18, /* Controller/switch message */
    OFPT_MULTIPART_REPLY        = 19, /* Controller/switch message */

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

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

    /* Asynchronous message configuration. */
    OFPT_GET_ASYNC_REQUEST  = 26, /* Controller/switch message */
    OFPT_GET_ASYNC_REPLY    = 27, /* Controller/switch message */
    OFPT_SET_ASYNC          = 28, /* Controller/switch message */

    /* Meters and rate limiters configuration messages. */
    OFPT_METER_MOD          = 29, /* Controller/switch message */

    /* Controller role change event messages. */
    OFPT_ROLE_STATUS        = 30, /* Async message */

    /* Asynchronous messages. */
    OFPT_TABLE_STATUS       = 31, /* Async message */

    /* Request forwarding by the switch. */
    OFPT_REQUESTFORWARD     = 32, /* Async message */

    /* Bundle operations (multiple messages as a single operation). */
    OFPT_BUNDLE_CONTROL     = 33, /* Controller/switch message */
    OFPT_BUNDLE_ADD_MESSAGE = 34, /* Controller/switch message */

    /* Controller Status async message. */
    OFPT_CONTROLLER_STATUS  = 35, /* Async message */

};
```

 © 2015; The Open Networking Foundation

7.1.2 Padding

Most OpenFlow messages contain padding fields. Those are included in the various message types and in various common structures. Most of those padding fields can be identified by the fact that their names start with `pad`. The goal of padding fields is to align multi-byte entities on natural processor boundaries.

All common structures included in messages are aligned on 64 bit boundaries. Various other types are aligned as needed, for example 32 bits integers are aligned on 32 bit boundaries. An exception to the padding rules are OXM match fields which are never padded (see 7.2.3.2). In general, the end of OpenFlow messages is not padded, unless explicitly specified. On the other hand, common structures are almost always padded at the end.

The padding fields should be set to zero. An OpenFlow implementation must accept any values set in padding fields, and must ignore the content of padding fields.

7.1.3 Reserved and unsupported values and bit positions

Most OpenFlow messages contain enumerations, they are used for example to describe a type, a command or a reason. The specification defines all the values used by this version of the protocol, all other values are reserved, unless explicitly specified. Deprecated values are also reserved. Reserved values should not be used in OpenFlow messages. The specification may also define that the support for some values is optional, so an implementation may not support those optional values. If an OpenFlow implementation receives a request containing a reserved value or an optional value it does not support, it must reject the request and return an appropriate error message. If an OpenFlow implementation receives a reply or asynchronous message containing a reserved value or an optional value it does not support, it should ignore the object containing the unknown value and log an error.

Some messages contain bitmaps (arrays of bits), they are used for example to encode configuration flags, status bits or capabilities. The specification defines all the bit positions used by this version of the protocol, all other bit positions are reserved, unless explicitly specified. Deprecated bit positions are also reserved. Reserved bit positions should be set to zero in OpenFlow messages. The specification may also define that the support for some bit positions is optional, so an implementation may not support those optional bit positions. If an OpenFlow implementation receives a request containing a reserved bit position or an optional bit position it does not support set to 1, it must reject the request and return an appropriate error message. If an OpenFlow implementation receives a reply or asynchronous message containing a reserved bit position or an optional bit position it does not support set to 1, it should ignore the bit position and log an error.

Some messages contain TLVs (Type, Length, Value). These are used for example to encode properties, actions, match fields or optional attributes in a structure. The specification defines all the TLV types used by this version of the protocol, all other TLV types are reserved, unless explicitly specified. Deprecated TLV types are also reserved. Reserved TLV types should not be used in OpenFlow messages. The specification may also define that the support for some TLV types is optional, so an implementation may not support those optional TLV types. If an OpenFlow implementation receives a request containing a reserved TLV type or an optional TLV type it does not support, it must reject the request and return an appropriate error message. If an OpenFlow implementation receives a reply or asynchronous message

 © 2015; The Open Networking Foundation

containing a reserved TLV type or an optional TLV type it does not support, it should ignore the TLV and log an error.

7.2 Common Structures

This section describes structures used by multiple message types.

7.2.1 Port Structures

The OpenFlow pipeline receives and sends packets on ports. The switch may define physical and logical ports, and the OpenFlow specification defines some reserved ports (see 4.1).

Each port on the switch is uniquely identified by a port number, a 32 bit number. Reserved ports have port numbers defined by the specification. Port numbers for physical and logical ports are assigned by the switch and can be any numbers starting at 1 and ending at `OFPP_MAX`. The port numbers use the following conventions:

```
/* Port numbering. Ports are numbered starting from 1. */
enum ofp_port_no {
    /* Maximum number of physical and logical switch ports. */
    OFPP_MAX        = 0xffffff00,

    /* Reserved OpenFlow Port (fake output "ports"). */
    OFPP_UNSET      = 0xfffffff7,  /* Output port not set in action-set.
                                      used only in OXM_OF_ACTSET_OUTPUT. */
    OFPP_IN_PORT    = 0xfffffff8,  /* Send the packet out the input port.  This
                                      reserved port must be explicitly used
                                      in order to send back out of the input
                                      port. */
    OFPP_TABLE      = 0xfffffff9,  /* Submit the packet to the first flow table
                                      NB: This destination port can only be
                                      used in packet-out messages. */
    OFPP_NORMAL     = 0xfffffffa,  /* Forward using non-OpenFlow pipeline. */
    OFPP_FLOOD      = 0xfffffffb,  /* Flood using non-OpenFlow pipeline. */
    OFPP_ALL        = 0xfffffffc,  /* All standard ports except input port. */
    OFPP_CONTROLLER = 0xfffffffd,  /* Send to controller. */
    OFPP_LOCAL      = 0xfffffffe,  /* Local openflow "port". */
    OFPP_ANY        = 0xffffffff   /* Special value used in some requests when
                                      no port is specified (i.e. wildcarded). */
};
```

7.2.1.1 Port Description Structures

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

```
/* Description of a port */
struct ofp_port {
    uint32_t port_no;
    uint16_t length;
```

 © 2015; The Open Networking Foundation

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

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

    /* Port description property list - 0 or more properties */
    struct ofp_port_desc_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_port) == 40);
```

The `port_no` field is the port number and it uniquely identifies a port within a switch. The `hw_addr` field typically is the MAC address for the port; `OFP_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 is a bitmap that describes port administrative settings, and may include a combination of the following flags :

```
/* Flags to indicate behavior of the physical port.  These flags are
 * used in ofp_port to describe the current configuration.  They are
 * used in the ofp_port_mod message to configure the port's behavior.
 */
enum ofp_port_config {
    OFPPC_PORT_DOWN    = 1 << 0,  /* Port is administratively down. */

    OFPPC_NO_RECV      = 1 << 2,  /* Drop all packets received by port. */
    OFPPC_NO_FWD       = 1 << 5,  /* Drop packets forwarded to port. */
    OFPPC_NO_PACKET_IN = 1 << 6   /* Do not send packet-in msgs for port. */
};
```

The `OFPPC_PORT_DOWN` bit indicates that the port has been administratively brought down and should not be used by OpenFlow to send traffic. The `OFPPC_NO_RECV` bit indicates that packets received on that port should be ignored. The `OFPPC_NO_FWD` bit indicates that OpenFlow should not send packets to that port. The `OFPPC_NO_PACKET_IN` bit indicates that packets on that port that generate a table miss should never trigger a packet-in message to the controller.

In general, the port config bits are set by one of the controllers and not changed by the switch. Those bits may be useful for the controller to implement protocols such as STP or BFD. The port config bits may also be changed by a switch configuration, for example using the OpenFlow Configuration Protocol. If the port config bits are changed on the switch, the switch must send an `OFPT_PORT_STATUS` message to notify the controller of the change.

The `state` field is a bitmap that describes the port internal state, and may include a combination of the following flags :

```
/* Current state of the physical port.  These are not configurable from
 * the controller.
 */
enum ofp_port_state {
```

 © 2015; The Open Networking Foundation

```
    OFPPS_LINK_DOWN    = 1 << 0,  /* No physical link present. */
    OFPPS_BLOCKED      = 1 << 1,  /* Port is blocked */
    OFPPS_LIVE         = 1 << 2,  /* Live for Fast Failover Group. */
};
```

The port state bits represent the state of the physical link or switch protocols outside of OpenFlow. The `OFPPS_LINK_DOWN` bit indicates the the physical link is not present. The `OFPPS_BLOCKED` bit indicates that a switch protocol outside of OpenFlow, such as 802.1D Spanning Tree, is preventing the use of that port with `OFPP_FLOOD`.

All port state bits are read-only and cannot be changed by the controller. When the port flags are changed, the switch must send an `OFPT_PORT_STATUS` message to notify the controller of the change.

The `properties` field is a list of port description properties, describing various configuration and state of the port.

7.2.1.2 Port Description Properties

The list of port description property types that are currently defined are:

```
/* Port description property types.
 */
enum ofp_port_desc_prop_type {
    OFPPDPT_ETHERNET          = 0,      /* Ethernet property. */
    OFPPDPT_OPTICAL           = 1,      /* Optical property. */
    OFPPDPT_PIPELINE_INPUT    = 2,      /* Ingress pipeline fields. */
    OFPPDPT_PIPELINE_OUTPUT   = 3,      /* Egress pipeline fields. */
    OFPPDPT_RECIRCULATE       = 4,      /* Recirculation property. */
    OFPPDPT_EXPERIMENTER      = 0xFFFF, /* Experimenter property. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all port description properties. */
struct ofp_port_desc_prop_header {
    uint16_t         type;    /* One of OFPPDPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_header) == 4);
```

The `OFPPDPT_ETHERNET` property uses the following structure and fields:

```
/* Ethernet port description property. */
struct ofp_port_desc_prop_ethernet {
    uint16_t         type;    /* OFPPDPT_ETHERNET. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint8_t          pad[4];  /* Align to 64 bits. */
    /* Bitmaps of OFPPF_* that describe features.  All bits zeroed if
     * unsupported or unavailable. */
    uint32_t curr;          /* Current features. */
```

 © 2015; The Open Networking Foundation

```
    uint32_t advertised;    /* Features being advertised by the port. */
    uint32_t supported;     /* Features supported by the port. */
    uint32_t peer;          /* Features advertised by peer. */

    uint32_t curr_speed;    /* Current port bitrate in kbps. */
    uint32_t max_speed;     /* Max port bitrate in kbps */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_ethernet) == 32);
```

The `curr`, `advertised`, `supported`, and `peer` fields are bitmaps that indicate link modes (speed and duplexity), link type (copper/fiber) and link features (autonegotiation and pause). Port features may include a combination of the following flags :

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

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

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

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

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

The `OFPPDPT_OPTICAL` property uses the following structure and fields:

```
/* Optical port description property. */
struct ofp_port_desc_prop_optical {
    uint16_t         type;    /* OFPPDPT_3OPTICAL. */
    uint16_t         length;  /* Length in bytes of this property. */
```

 © 2015; The Open Networking Foundation

```
    uint8_t          pad[4];  /* Align to 64 bits. */

    uint32_t supported;     /* Features supported by the port. */
    uint32_t tx_min_freq_lmda;   /* Minimum TX Frequency/Wavelength */
    uint32_t tx_max_freq_lmda;   /* Maximum TX Frequency/Wavelength */
    uint32_t tx_grid_freq_lmda;  /* TX Grid Spacing Frequency/Wavelength */
    uint32_t rx_min_freq_lmda;   /* Minimum RX Frequency/Wavelength */
    uint32_t rx_max_freq_lmda;   /* Maximum RX Frequency/Wavelength */
    uint32_t rx_grid_freq_lmda;  /* RX Grid Spacing Frequency/Wavelength */
    uint16_t tx_pwr_min;         /* Minimum TX power */
    uint16_t tx_pwr_max;         /* Maximum TX power */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_optical) == 40);
```

The minimum, maximum, and grid spacing are specified for both transmit and receive optical ports as either a frequency in MHz or wavelength (lambda) as nm * 100. For ports that are not tunable, the minimum and maximum values will be identical and specify the fixed value. The `tx_pwr_min` and `tx_pwr_max` are dBm * 10. The optical port features may include a combination of the following flags :

```
/* Features of optical ports available in switch. */
enum ofp_optical_port_features {
    OFPOPF_RX_TUNE   = 1 << 0,  /* Receiver is tunable */
    OFPOPF_TX_TUNE   = 1 << 1,  /* Transmit is tunable */
    OFPOPF_TX_PWR    = 1 << 2,  /* Power is configurable */
    OFPOPF_USE_FREQ  = 1 << 3,  /* Use Frequency, not wavelength */
};
```

The `OFPOPF_RX_TUNE` indicates the port receive function is able to tune, the `OFPOPF_TX_TUNE` indicates the port transmit function is able to tune. The `OFPOPF_TX_PWR` indicates the transmit power can be set. The `OFPOPF_USE_FREQ` indicates frequency must be used instead of wavelength for description of the port tuning. It is highly important that the same tuning domain (frequency or wavelength) is used as indicated, since conversion between wavelength and frequency is often prone to error due to the range of the values.

The `OFPPDPT_PIPELINE_INPUT` and `OFPPDPT_PIPELINE_OUTPUT` properties uses the following structure and fields:

```
/* Ingress or egress pipeline fields. */
struct ofp_port_desc_prop_oxm {
    uint16_t         type;    /* One of OFPPDPT_PIPELINE_INPUT or
                                 OFPPDPT_PIPELINE_OUTPUT. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the oxm_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         oxm_ids[0];   /* Array of OXM headers */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_oxm) == 4);
```

 © 2015; The Open Networking Foundation

The oxm_ids is a list of OXM types supported by the port (see 7.2.3.2). The elements of that list are 32-bit OXM headers for non-experimenter OXM fields or 64-bit OXM headers for experimenter OXM fields, those OXM fields don't include any payload. The oxm_length field in OXM headers must be the length value defined for the OXM field, i.e. the payload length if the OXM field had a payload. For experimenter OXM fields with variable payload size, the oxm_length field must be the maximum length of the payload.

The OFPPDPT_PIPELINE_INPUT property indicates the list of pipeline fields that are provided for packets received on the port (see 7.2.3.7). The port sets those pipeline fields with useful values on the packets it receives based on internal processing. If OXM_OF_IN_PORT is the only pipeline field supported, this property does not need to be included.

The OFPPDPT_PIPELINE_OUTPUT property indicates the list of pipeline fields that are consumed by the port for packets that are output on that port (see 7.2.3.7). The port uses the values set in those pipeline fields for internal processing when forwarding packet. If no pipeline field is supported, this property does not need to be included.

The OFPPDPT_RECIRCULATE property uses the following structure and fields:

```
/* Recirculate port description property. */
struct ofp_port_desc_prop_recirculate {
    uint16_t     type;          /* OFPPDPT_RECIRCULATE. */
    uint16_t     length;        /* Length in bytes of the property,
                                   including this header, excluding padding. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the port numbers, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t     port_nos[0];    /* List of recirculated input port numbers.
                                    0 or more. The number of port numbers
                                    is inferred from the length field in
                                    the header. */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_recirculate) == 4);
```

The port_nos field is the list of input ports where a packet output on the present port can be recirculated back into the OpenFlow pipeline (see 4.7). This property must be used only when the port recirculates packets back to the OpenFlow pipeline, other ports must not used this property. For a loopback or unidirectional service returning on the same port, this field would only include the present port number. For an encapsulation/decapsulation service or other bidirectional services, this field would include the other port number representing the other side of the service. If the recirculation can return via an arbitrary port number, the list would be empty.

The OFPPDPT_EXPERIMENTER property uses the following structure and fields:

```
/* Experimenter port description property. */
struct ofp_port_desc_prop_experimenter {
    uint16_t         type;    /* OFPPDPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
```

 © 2015; The Open Networking Foundation

```
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_experimenter) == 12);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

7.2.2 Header Types Structure

Headers types are used to identify the various header sections of the packet header. A header type is defined using the `ofp_header_type` structure:

```
/* Header type structure. */
struct ofp_header_type {
    uint16_t    namespace;    /* One of OFPHTN_*. */
    uint16_t    ns_type;      /* Type within namespace. */
};
OFP_ASSERT(sizeof(struct ofp_header_type) == 4);
```

The `namespace` field identifies the namespace used for the header type. Most namespaces are well known packet type lists managed outside the OpenFlow protocol.

The `ns_type` field is the packet type within the chosen namespace. The definition of that field depends on the namespace.

The following namespaces are defined :

```
enum ofp_header_type_namespaces {
    OFPHTN_ONF              = 0,  /* ONF namespace. */
    OFPHTN_ETHERTYPE        = 1,  /* ns_type is an Ethertype. */
    OFPHTN_IP_PROTO         = 2,  /* ns_type is a IP protocol number. */
    OFPHTN_UDP_TCP_PORT     = 3,  /* ns_type is a TCP or UDP port. */
    OFPHTN_IPV4_OPTION      = 4,  /* ns_type is an IPv4 option number. */
};
```

- `OFPHTN_ONF` is the ONF namespace. For this namespace, `ns_type` is one of the values defined below.
- `OFPHTN_ETHERTYPE` is the Ethertype namespace. For this namespace, `ns_type` is an Ethertype, as defined by the IEEE Standards Association.
- `OFPHTN_IP_PROTO` is the IP protocol namespace. For this namespace, `ns_type` is a IP protocol number, as defined by the IETF. Only the lower 8 bits of `ns_type` are significant, the high order bits must be set to zero.

 © 2015; The Open Networking Foundation

- OFPHTN_UDP_TCP_PORT is the TCP UDP port namespace. For this namespace, ns_type is a TCP or UDP port number, as defined by the IETF.
- OFPHTN_IPV4_OPTION is the IPv4 option namespace. For this namespace, ns_type is a IPv4 option number, as defined by the IETF. Only the lower 8 bits of ns_type are significant, the high order bits must be set to zero.

The following values for ns_type in the ONF namespace are defined :

```
enum ofp_header_type_onf {
    OFPHTO_ETHERNET         = 0,  /* Ethernet (DIX or IEEE 802.3) - default. */
    OFPHTO_NO_HEADER        = 1,  /* No header, ex. circuit switch. */
    OFPHTO_OXM_EXPERIMENTER = 0xFFFF, /* Use Experimenter OXM. */
};
```

- OFPHTO_ETHERNET is a basic Ethernet header, as defined by the combination of the Ethernet DIX standard and the IEEE 802.3 standard. It contains a source address, a destination address and an Ethertype.
- OFPHTO_NO_HEADER is an empty header. It is used when no packet header can be used, for example for circuit switches.
- OFPHTO_OXM_EXPERIMENTER reference to an OXM experimenter value. If a header type equal to OFPHTO_OXM_EXPERIMENTER is used in a list of OXM values, the header type is defined by the first OXM experimenter value following it (see 7.2.3.12). This allows experimenters to define any arbitrary header types.

Most parts of the packet headers can be identified by their **header type**, i.e. the tuple (namespace, ns_type). A specific part of the packet headers may have multiple header types, for example one in each namespace. The **canonical header type** of common headers is defined in Table 9. For headers not listed in that table, the **canonical header type** is the header type with the smallest namespace number, and within a single namespace the header type with the smallest ns_type number.

namespace	ns_type	Header definition	non-canonical
0	0	Ethernet header: IEEE Std 802.3.	(1, 0x6558)
0	1	No header.	
1	0x800	IPv4 header: RFC 791.	(2, 0x4)
1	0x8100	VLAN tag: IEEE Std 802.1Q.	(1, 0x88a8)
1	0x86dd	IPv6 header: RFC 2460.	(2, 0x29)
1	0x8847	MPLS shim header: RFC 3032.	(1, 0x8848), (2, 0x89)
1	0x88E7	PBB I-TAG: IEEE std 802.1Q.	
2	6	Basic TCP header: RFC 793.	
2	17	UDP header: RFC 768.	
3	4789	VxLAN header: RFC 7348.	

Table 9: Common Canonical Header Types.

 © 2015; The Open Networking Foundation

7.2.3 Flow Match Structures

The OpenFlow match structure is used for matching packets (see 5.3) or matching flow entries in the table (see 5.2). It is composed of a flow match header and a sequence of zero or more flow match fields encoded using OXM TLVs.

7.2.3.1 Flow Match Header

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

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

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

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

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

 © 2015; The Open Networking Foundation

7.2.3.2 Flow Match Field Structures

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

An OXM TLV's header is interpreted as a 32-bit word in network byte order (see figure 9).

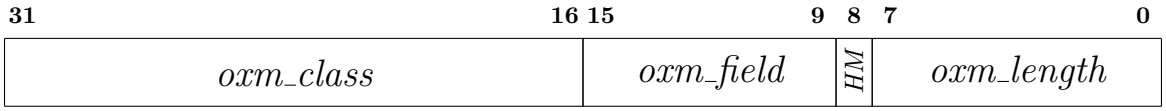


Figure 9: OXM TLV header layout.

The OXM TLV's header fields are defined in Table 10

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

Table 10: OXM TLV header fields.

The `oxm_class` is an OXM match class that contains related match types, and is described in section 7.2.3.3. `oxm_field` is a class-specific value, identifying one of the match types within the match class. The combination of `oxm_class` and `oxm_field` (the most-significant 23 bits of the header) are collectively `oxm_type`. The `oxm_type` normally designates a protocol header field, such as the Ethernet type, but it can also refer to a packet pipeline field, such as the switch port on which a packet arrived.

`oxm_hasmask` defines if the OXM TLV contains a bitmask, more details are explained in section 7.2.3.5.

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

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

7.2.3.3 OXM classes

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

 © 2015; The Open Networking Foundation

The following OXM classes are defined:

```
/* OXM Class IDs.
 * The high order bit differentiate reserved classes from member classes.
 * Classes 0x0000 to 0x7FFF are member classes, allocated by ONF.
 * Classes 0x8000 to 0xFFFE are reserved classes, reserved for standardisation.
 */
enum ofp_oxm_class {
    OFPXMC_NXM_0          = 0x0000,    /* Backward compatibility with NXM */
    OFPXMC_NXM_1          = 0x0001,    /* Backward compatibility with NXM */
    OFPXMC_OPENFLOW_BASIC = 0x8000,    /* Basic class for OpenFlow */
    OFPXMC_PACKET_REGS    = 0x8001,    /* Packet registers (pipeline fields). */
    OFPXMC_EXPERIMENTER   = 0xFFFF,    /* Experimenter class */
};
```

The class `OFPXMC_OPENFLOW_BASIC` contains the basic set of OpenFlow match fields (see 7.2.3.7). The optional class `OFPXMC_PACKET_REGS` contains packet registers (see 7.2.3.10). The optional class `OFPXMC_EXPERIMENTER` is used for experimenter matches (see 7.2.3.12), it differs from other classes because it includes an experimenter header between the OXM TLV header and the value in the payload. Other ONF reserved classes are reserved for future uses such as modularisation of the specification. The first two ONF member classes `OFPXMC_NXM_0` and `OFPXMC_NXM_1` are reserved for backward compatibility with the Nicira Extensible Match (NXM) specification.

7.2.3.4 Flow Matching

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

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

- If `oxm_hasmask` is 0, the OXM TLV's body contains a value for the field, called `oxm_value`. The OXM TLV match matches only packets in which the corresponding field equals `oxm_value`.
- If `oxm_hasmask` is 1, then the `oxm_entry`'s body contains a value for the field (`oxm_value`), followed by a bitmask of the same length as the value, called `oxm_mask`. Each 1-bit in `oxm_mask` constrains the OXM TLV to match only packets in which the corresponding bit of the field equals the corresponding bit in `oxm_value`. Each 0-bit in `oxm_mask` places no constraint on the corresponding bit in the field.

When using masking, it is an error for a 0-bit in `oxm_mask` to have a corresponding 1-bit in `oxm_value` (see 7.5.4.5).

The following table summarizes the constraint that a pair of corresponding `oxm_mask` and `oxm_value` bits place upon the corresponding field bit when using masking. Omitting `oxm_mask` is equivalent to supplying an `oxm_mask` that is all 1-bits.

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

 © 2015; The Open Networking Foundation

	oxm_value	
oxm_mask	0	1
0	no constraint	error
1	must be 0	must be 1

Table 11: OXM mask and value.

7.2.3.5 Flow Match Field Masking

When `oxm_hasmask` is 1, the OXM TLV contains a bitmask that follows the field value, it has the same size as the field value and is encoded in the same way. For fields not in the experimenter class, this means that `oxm_length` is effectively doubled, so `oxm_length` is always even when `oxm_hasmask` is 1. For fields in the experimenter class, the experimenter header is not duplicated, the length increase corresponds only to the size of the mask, and the resulting `oxm_length` depends on the experimenter header used.

The masks are defined such that a 0 in a given bit position indicates a "don't care" match for the same bit in the corresponding field, whereas a 1 means match the bit exactly. An all-zero-bits `oxm_mask` is equivalent to omitting the OXM TLV entirely. An all-one-bits `oxm_mask` is equivalent to specifying 0 for `oxm_hasmask` and omitting `oxm_mask`.

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

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

These restrictions are detailed in specifications for individual fields. A switch may accept an `oxm_hasmask` or `oxm_mask` value that the specification disallows, but only if the switch correctly implements support for that `oxm_hasmask` or `oxm_mask` value. A switch must reject an attempt to set up a flow entry that contains a `oxm_hasmask` or `oxm_mask` value that it does not support (see 6.4).

7.2.3.6 Flow Match Field Prerequisite

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

For example:

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

 © 2015; The Open Networking Foundation

matching on the TCP source port is allowed only if the Ethernet type is IP and the IP protocol is TCP.

- An OXM TLV for `oxm_type`=OXM_OF_MPLS_LABEL is allowed only if it is preceded by an entry with `oxm_type`=OXM_OF_ETH_TYPE, `oxm_hasmask`=0, `oxm_value`=0x8847 or 0x8848.
- An OXM TLV for `oxm_type`=OXM_OF_VLAN_PCP is allowed only if it is preceded by an entry with `oxm_type`=OXM_OF_VLAN_VID, `oxm_value`!=OFPVID_NONE.

The prerequisite of a match field is another match field type and match field value that this match field depends on. Most match fields have prerequisites, these restrictions are noted in specifications for individual fields (see 7.2.3.8). The prerequisites are cumulative, a match field inherits all the restrictions of its prerequisites (see examples above), and all the chains of prerequisites must be present in the match.

A switch may implement relaxed versions of these restrictions. For example, a switch may accept no prerequisite at all. A switch must reject an attempt to set up a flow entry that violates its restrictions (see 7.5.4.5), and must deal with inconsistent matches created by the lack of prerequisites (for example matching both a TCP source port and a UDP destination port).

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

An OXM TLV that has prerequisite restrictions must appear after the OXM TLVs for its prerequisites. Ordering of OXM TLVs within an OpenFlow match is not otherwise constrained (apart from the Packet Type Match Field).

Any given `oxm_type` may appear in an OpenFlow match at most once, otherwise the switch must generate an error (see 7.5.4.5). A switch may implement a relaxed version of this rule and may allow in some cases an `oxm_type` to appear multiple times in an OpenFlow match, however the behaviour of matching is then implementation-defined.

If a flow table implements a specific OXM TLV, this flow table must accept valid matches containing the prerequisites of this OXM TLV, even if the flow table does not support matching all possible values for the match fields specified by those prerequisites. For example, if a flow table matches the IPv4 source address, this flow table must accept matching the Ethertype exactly to IPv4, however this flow table does not need to support matching Ethertype to any other value.

7.2.3.7 Flow Match Fields

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

```
/* OXM Flow match field types for OpenFlow basic class. */
enum oxm_ofb_match_fields {
    OFPXMT_OFB_IN_PORT        = 0,  /* Switch input port. */
    OFPXMT_OFB_IN_PHY_PORT    = 1,  /* Switch physical input port. */
    OFPXMT_OFB_METADATA       = 2,  /* Metadata passed between tables. */
    OFPXMT_OFB_ETH_DST        = 3,  /* Ethernet destination address. */
```

 © 2015; The Open Networking Foundation

```
    OFPXMT_OFB_ETH_SRC        = 4,  /* Ethernet source address. */
    OFPXMT_OFB_ETH_TYPE       = 5,  /* Ethernet frame type. */
    OFPXMT_OFB_VLAN_VID       = 6,  /* VLAN id. */
    OFPXMT_OFB_VLAN_PCP       = 7,  /* VLAN priority. */
    OFPXMT_OFB_IP_DSCP        = 8,  /* IP DSCP (6 bits in ToS field). */
    OFPXMT_OFB_IP_ECN         = 9,  /* IP ECN (2 bits in ToS field). */
    OFPXMT_OFB_IP_PROTO       = 10, /* IP protocol. */
    OFPXMT_OFB_IPV4_SRC       = 11, /* IPv4 source address. */
    OFPXMT_OFB_IPV4_DST       = 12, /* IPv4 destination address. */
    OFPXMT_OFB_TCP_SRC        = 13, /* TCP source port. */
    OFPXMT_OFB_TCP_DST        = 14, /* TCP destination port. */
    OFPXMT_OFB_UDP_SRC        = 15, /* UDP source port. */
    OFPXMT_OFB_UDP_DST        = 16, /* UDP destination port. */
    OFPXMT_OFB_SCTP_SRC       = 17, /* SCTP source port. */
    OFPXMT_OFB_SCTP_DST       = 18, /* SCTP destination port. */
    OFPXMT_OFB_ICMPV4_TYPE    = 19, /* ICMP type. */
    OFPXMT_OFB_ICMPV4_CODE    = 20, /* ICMP code. */
    OFPXMT_OFB_ARP_OP         = 21, /* ARP opcode. */
    OFPXMT_OFB_ARP_SPA        = 22, /* ARP source IPv4 address. */
    OFPXMT_OFB_ARP_TPA        = 23, /* ARP target IPv4 address. */
    OFPXMT_OFB_ARP_SHA        = 24, /* ARP source hardware address. */
    OFPXMT_OFB_ARP_THA        = 25, /* ARP target hardware address. */
    OFPXMT_OFB_IPV6_SRC       = 26, /* IPv6 source address. */
    OFPXMT_OFB_IPV6_DST       = 27, /* IPv6 destination address. */
    OFPXMT_OFB_IPV6_FLABEL    = 28, /* IPv6 Flow Label */
    OFPXMT_OFB_ICMPV6_TYPE    = 29, /* ICMPv6 type. */
    OFPXMT_OFB_ICMPV6_CODE    = 30, /* ICMPv6 code. */
    OFPXMT_OFB_IPV6_ND_TARGET = 31, /* Target address for ND. */
    OFPXMT_OFB_IPV6_ND_SLL    = 32, /* Source link-layer for ND. */
    OFPXMT_OFB_IPV6_ND_TLL    = 33, /* Target link-layer for ND. */
    OFPXMT_OFB_MPLS_LABEL     = 34, /* MPLS label. */
    OFPXMT_OFB_MPLS_TC        = 35, /* MPLS TC. */
    OFPXMT_OFP_MPLS_BOS       = 36, /* MPLS BoS bit. */
    OFPXMT_OFB_PBB_ISID       = 37, /* PBB I-SID. */
    OFPXMT_OFB_TUNNEL_ID      = 38, /* Logical Port Metadata. */
    OFPXMT_OFB_IPV6_EXTHDR    = 39, /* IPv6 Extension Header pseudo-field */
    OFPXMT_OFB_PBB_UCA        = 41, /* PBB UCA header field. */
    OFPXMT_OFB_TCP_FLAGS      = 42, /* TCP flags. */
    OFPXMT_OFB_ACTSET_OUTPUT  = 43, /* Output port from action set metadata. */
    OFPXMT_OFB_PACKET_TYPE    = 44, /* Packet type value. */
};
```

A switch that supports Ethernet packet types must support the required match fields listed in Table 12 in its pipeline. Each required match field must be supported in at least one flow table of the switch : that flow table must enable matching that field and the match field prerequisites must be met in that table (see 7.2.3.6). The required fields don't need to be implemented in all flow tables, and don't need to be implemented in the same flow table. Flow tables can support non-required and experimenter match fields. The controller can query the switch about which match fields are supported in each flow table (see 7.3.5.18).

Match fields come in two types, header match fields (see 7.2.3.8) and pipeline match fields (see 7.2.3.9).

 © 2015; The Open Networking Foundation

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

Table 12: Required match fields.

7.2.3.8 Header Match Fields

Header match fields are match fields matching values extracted from the packet headers. Most header match fields map directly to a specific field in the packet header defined by a datapath protocol.

All header match fields have different size, prerequisites and masking capability, as specified in Table 13. If not explicitly specified in the field description, each field type refers to the outermost occurrence of the field in the packet headers.

Field	Bits	Bytes	Mask	Pre-requisites	Description
OXM_OF_ETH_DST	48	6	Yes	PACKET_TYPE=(0,0) or None	Ethernet destination MAC address.
OXM_OF_ETH_SRC	48	6	Yes	PACKET_TYPE=(0,0) or None	Ethernet source MAC address.
OXM_OF_ETH_TYPE	16	2	No	PACKET_TYPE=(0,0) or None	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_VLAN_VID	12+1	2	Yes	PACKET_TYPE=(0,0) or None	VLAN-ID from 802.1Q header. The CFI bit indicates the presence of a valid VLAN-ID, see below.
OXM_OF_VLAN_PCP	3	1	No	VLAN_VID!=NONE	VLAN-PCP from 802.1Q header.
OXM_OF_IP_DSCP	6	1	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd or PACKET_TYPE=(1,0x800) or PACKET_TYPE=(1,0x86dd)	Diff Serv Code Point (DSCP). Part of the IPv4 ToS field or the IPv6 Traffic Class field.

Table 13 – continued on next page

 © 2015; The Open Networking Foundation

Table 13 – concluded from previous page

Field	Bits	Bytes	Mask	Pre-requisites	Description
OXM_OF_IP_ECN	2	1	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd or PACKET_TYPE=(1,0x800) or PACKET_TYPE=(1,0x86dd)	ECN bits of the IP header. Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_PROTO	8	1	No	ETH_TYPE=0x0800 or ETH_TYPE=0x86dd or PACKET_TYPE=(1,0x800) or PACKET_TYPE=(1,0x86dd)	IPv4 or IPv6 protocol number.
OXM_OF_IPV4_SRC	32	4	Yes	ETH_TYPE=0x0800 or PACKET_TYPE=(1,0x800)	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	32	4	Yes	ETH_TYPE=0x0800 or PACKET_TYPE=(1,0x800)	IPv4 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_TCP_SRC	16	2	No	IP_PROTO=6	TCP source port
OXM_OF_TCP_DST	16	2	No	IP_PROTO=6	TCP destination port
OXM_OF_TCP_FLAGS	12	2	Yes	IP_PROTO=6	TCP flags
OXM_OF_UDP_SRC	16	2	No	IP_PROTO=17	UDP source port
OXM_OF_UDP_DST	16	2	No	IP_PROTO=17	UDP destination port
OXM_OF_SCTP_SRC	16	2	No	IP_PROTO=132	SCTP source port
OXM_OF_SCTP_DST	16	2	No	IP_PROTO=132	SCTP destination port
OXM_OF_ICMPV4_TYPE	8	1	No	IP_PROTO=1	ICMP type
OXM_OF_ICMPV4_CODE	8	1	No	IP_PROTO=1	ICMP code
OXM_OF_ARP_OP	16	2	No	ETH_TYPE=0x0806	ARP opcode
OXM_OF_ARP_SPA	32	4	Yes	ETH_TYPE=0x0806	Source IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_TPA	32	4	Yes	ETH_TYPE=0x0806	Target IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_SHA	48	6	Yes	ETH_TYPE=0x0806	Source Ethernet address in the ARP payload.
OXM_OF_ARP_THA	48	6	Yes	ETH_TYPE=0x0806	Target Ethernet address in the ARP payload.
OXM_OF_IPV6_SRC	128	16	Yes	ETH_TYPE=0x86dd or PACKET_TYPE=(1,0x86dd)	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	128	16	Yes	ETH_TYPE=0x86dd or PACKET_TYPE=(1,0x86dd)	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_FLABEL	20	4	Yes	ETH_TYPE=0x86dd or PACKET_TYPE=(1,0x86dd)	IPv6 flow label.
OXM_OF_ICMPV6_TYPE	8	1	No	IP_PROTO=58	ICMPv6 type
OXM_OF_ICMPV6_CODE	8	1	No	IP_PROTO=58	ICMPv6 code
OXM_OF_IPV6_ND_TARGET	128	16	No	ICMPV6_TYPE=135 or ICMPV6_TYPE=136	The target address in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_SLL	48	6	No	ICMPV6_TYPE=135	The source link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_TLL	48	6	No	ICMPV6_TYPE=136	The target link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_MPLS_LABEL	20	4	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The LABEL in the first MPLS shim header.
OXM_OF_MPLS_TC	3	1	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The TC in the first MPLS shim header.
OXM_OF_MPLS_BOS	1	1	No	ETH_TYPE=0x8847 or ETH_TYPE=0x8848	The BoS bit (Bottom of Stack bit) in the first MPLS shim header.
OXM_OF_PBB_ISID	24	3	Yes	ETH_TYPE=0x88E7	The I-SID in the first PBB service instance tag.
OXM_OF_IPV6_EXTHDR	9	2	Yes	ETH_TYPE=0x86dd or PACKET_TYPE=(1,0x86dd)	IPv6 Extension Header pseudo-field.
OXM_OF_PBB_UCA	1	1	No	ETH_TYPE=0x88E7	The UCA field in the first PBB service instance tag.

Table 13: Header match fields details.

© 2015; The Open Networking Foundation

Omitting the OFPXMT_OFB_VLAN_VID field specifies that a flow entry should match packets regardless of whether they contain the corresponding tag. Special values are defined below for the VLAN tag to allow matching of packets with any tag, independent of the tag's value, and to support matching packets without a VLAN tag. The special values defined for OFPXMT_OFB_VLAN_VID are:

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

The OFPXMT_OFB_VLAN_PCP field must be rejected when the OFPXMT_OFB_VLAN_VID field is wildcarded (not present) or when the value of OFPXMT_OFB_VLAN_VID is set to OFPVID_NONE. Table 14 summarizes the combinations of wildcard bits and field values for particular VLAN tag matches.

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

Table 14: Match combinations for VLAN tags.

The field OXM_OF_IPV6_EXTHDR is a pseudo header match field that indicates the presence of various IPv6 extension headers in the packet header. The IPv6 extension header bits are combined together in the fields OXM_OF_IPV6_EXTHDR, and those bits can have the following values:

```
/* Bit definitions for IPv6 Extension Header pseudo-field. */
enum ofp_ipv6exthdr_flags {
    OFPIEH_NONEXT = 1 << 0,     /* "No next header" encountered. */
    OFPIEH_ESP    = 1 << 1,     /* Encrypted Sec Payload header present. */
    OFPIEH_AUTH   = 1 << 2,     /* Authentication header present. */
    OFPIEH_DEST   = 1 << 3,     /* 1 or 2 dest headers present. */
    OFPIEH_FRAG   = 1 << 4,     /* Fragment header present. */
    OFPIEH_ROUTER = 1 << 5,     /* Router header present. */
    OFPIEH_HOP    = 1 << 6,     /* Hop-by-hop header present. */
    OFPIEH_UNREP  = 1 << 7,     /* Unexpected repeats encountered. */
    OFPIEH_UNSEQ  = 1 << 8,     /* Unexpected sequencing encountered. */
};
```

- OFPIEH_HOP is set to 1 if a hop-by-hop IPv6 extension header is present as the first extension header in the packet.
- OFPIEH_ROUTER is set to 1 if a router IPv6 extension header is present.
- OFPIEH_FRAG is set to 1 if a fragmentation IPv6 extension header is present.
- OFPIEH_DEST is set to 1 if one or more Destination options IPv6 extension headers are present. It is normal to have either one or two of these in one IPv6 packet (see RFC 2460).

 © 2015; The Open Networking Foundation

- OFPIEH_AUTH is set to 1 if an Authentication IPv6 extension header is present.
- OFPIEH_ESP is set to 1 if an Encrypted Security Payload IPv6 extension header is present.
- OFPIEH_NONEXT is set to 1 if a No Next Header IPv6 extension header is present.
- OFPIEH_UNSEQ is set to 1 if IPv6 extension headers were not in the order preferred (but not required) by RFC 2460.
- OFPIEH_UNREP is set to 1 if more than one of a given IPv6 extension header is unexpectedly encountered. (Two destination options headers may be expected and would not cause this bit to be set.)

The field OXM_OF_TCP_FLAGS matches the flag bits in a TCP header. Considering bit 0 the least significant bit, it specifically contains:

- Bits 0–5: The original TCP flags defined in RFC 793.
- Bits 6–8: Additional TCP flags defined by RFCs 3168 and 3540.
- Bits 9–11: Reserved bits not yet standardized by RFCs.
- Bits 12–15: Not part of the OXM field, forced to zero. (The TCP header uses the corresponding bits for other purposes, so they will never be standardized as flags.)

7.2.3.9 Pipeline Match Fields

Pipeline match fields are match fields matching values attached to the packet for pipeline processing and not associated with packet headers.

All pipeline match fields have different size, prerequisites and masking capability, as specified in Table 15.

Field	Bits	Bytes	Mask	Pre-requisites	Description
OXM_OF_IN_PORT	32	4	No	None	Ingress port. Numerical representation of incoming port, starting at 1. This may be a physical or switch-defined logical port.
OXM_OF_IN_PHY_PORT	32	4	No	IN_PORT present	Physical port. In ofp_packet_in messages, underlying physical port when packet received on a logical port.
OXM_OF_METADATA	64	8	Yes	None	Table metadata. Used to pass information between tables.
OXM_OF_TUNNEL_ID	64	8	Yes	None	Metadata associated with a logical port.
OXM_OF_ACTSET_OUTPUT	32	4	No	None	Output port from action set Metadata or OFPP_UNSET.
OXM_OF_PACKET_TYPE	16+16	4	No	None	Packet type - canonical header type of outermost header.

Table 15: Pipeline match fields details.

The ingress port field OXM_OF_IN_PORT is used to match the OpenFlow port on which the packet was received in the OpenFlow datapath. It can either be a physical port, a logical port, the OFPP_LOCAL reserved port or the OFPP_CONTROLLER reserved port (see 4.1). The ingress port must be a valid standard

 © 2015; The Open Networking Foundation

OpenFlow port or the OFPP_CONTROLLER reserved port, and for standard ports the port configuration must allow it to receive packets (both OFPPC_PORT_DOWN and OFPPC_NO_RECV config bits cleared).

The physical port field OXM_OF_IN_PHY_PORT is used in *Packet-in* messages to identify a physical port underneath a logical port (see 7.4.1). A switch must support the physical port field in *Packet-in* messages only if the physical port is an OpenFlow port of that switch, and its value different from the ingress port. In addition, some switches may optionally allow to match the physical port field in flow entries, in this case when the physical port is the same as the ingress port or is not an OpenFlow port, the value of this field must be the ingress port. The physical port must be a valid OpenFlow port, and must be included in the port description request (see 7.3.5.6). The physical port does not need to have any specific configuration, for example it can be down or receiving packets may be disabled.

When a packet is received directly on a physical port and not processed by a logical port, OXM_OF_IN_PORT and OXM_OF_IN_PHY_PORT have the same value, the OpenFlow port_no of this physical port (see 4.1). When a packet is received on a logical port by way of a physical port which is not an OpenFlow port, OXM_OF_IN_PORT and OXM_OF_IN_PHY_PORT have the same value, the OpenFlow port_no of this logical port. When a packet is received on a logical port by way of a physical port which is an OpenFlow port, OXM_OF_IN_PORT is the logical port's port_no and OXM_OF_IN_PHY_PORT is the physical port's port_no. For example, consider a packet received on a tunnel interface defined over a link aggregation group (LAG) with two physical port members. If the tunnel interface is the logical port bound to OpenFlow, then OXM_OF_IN_PORT is the tunnel's port_no and OXM_OF_IN_PHY_PORT is the physical port_no member of the LAG on which the tunnel is configured.

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

The Tunnel ID field OXM_OF_TUNNEL_ID carries optional encapsulation metadata associated with a logical port. When a packet is received on a logical port that supports Tunnel ID, the Tunnel ID field associated with the packet is set with the encapsulation metadata and can be matched by flow entries. If the logical port does not provide such data or if the packet was received on a physical port, its value is zero. When a packet is sent on a logical port that supports the tunnel-id field, it will use the value of that field for internal encapsulation processing. For example, that field may be set by a flow entry using a *set-field* action.

If a logical port supports the Tunnel ID field, either provides it on input, consumes it on output, or both, it should report that support in the corresponding port description property (see 7.2.1.2).

The mapping of the optional encapsulation metadata in the Tunnel ID field is defined by the logical port implementation, it is dependent on the type of logical port and it is implementation specific. We recommend that for a packet received via a GRE tunnel including a (32-bit) key, the key is stored in the lower 32-bits and the high bits are zeroed. We recommend that for a MPLS logical port, the lower 20 bits represent the MPLS Label. We recommend that for a VxLAN logical port, the lower 24 bits represent the VNI.

The Action Set Output field OXM_OF_ACTSET_OUTPUT can be used to match the output action embedded in the action set of the packet, and in most cases can be used to match packets based on their output port. All flow tables used as egress tables are required to support this field, flow tables used as ingress tables may optionally support that field, depending on switch capabilities. If the action set contains an output action and no group action, this field equals the output port from that action. Otherwise, if

 © 2015; The Open Networking Foundation

the action set does not contain an output action, or contains a group action, the field equals its initial value, `OFPP_UNSET`.

7.2.3.10 Packet registers

The packet register fields `OXM_OF_PKT_REG(N)` are used to store temporary values and information alongside the packet through pipeline processing. Each packet register is 64 bits wide and maskable. In most cases, the packet registers can not be matched in tables, i.e. they usually can not be used in the flow entry match structure. They can be used with the *set-field* and *copy-field* actions (see 7.2.6.7 and 7.2.6.8).

A switch may optionally implement any number of packet registers, up to the limit of the OXM format (i.e. 128). Each packet register is identified by its OXM type field. If a switch supports packet registers and uses contiguous numbering starting at zero, only the supported packet register with the greatest field number needs to be listed in the table feature properties, otherwise all supported packet registers must be individually listed (see 7.3.5.18.2). A switch that does not support packet register fields must not include any packet register entry in the table feature properties.

7.2.3.11 Packet Type Match Field

The *Packet Type Match Field* is a pipeline field with OXM type `OXM_OF_PACKET_TYPE`, its value identifies a packet type in the datapath. The value of common packet types is defined in Table 16. For other packet types, the value must be the *canonical header type* of the packet outermost header (see 7.2.2). The Packet Type Match Field must appear as the first OXM TLV, and only matches packets of that specific type. Most Packet Type values are also prerequisites for other match fields. If a Packet Type Match Field is not present, the packet type must be Ethernet.

The Packet Types defined by the specification are listed in Table 16. New Packet Types can be defined using other canonical header types or the Experimenter mechanism (see 7.2.2). Most switches will support only a single Packet Type, support for multiple Packet Types is currently outside this specification (but may be added in a future version).

namespace	ns_type	Match description	Packet-in and packet-out format
0	0	Ethernet packet (default type).	Ethernet header and Ethernet payload.
1	0x800	IPv4 packet (with no header in front).	IPv4 header and IPv4 payload.
1	0x86dd	IPv6 packet (with no header in front).	IPv6 header and IPv6 payload.
0	1	No packet (for example circuit switch).	Empty.
0	0xFFFF	Experimenter defined.	Experimenter defined.

Table 16: Packet Types.

 © 2015; The Open Networking Foundation

An OpenFlow switch that supports a specific packet type advertises the corresponding Packet Type Match Field in the *table feature properties* (see 7.3.5.18.2). *Flow-mod* messages must include the proper Packet Type Match Field in the match, unless the packet type is Ethernet (see 7.3.4.2). *Packet-in* messages must include the Packet Type Match Field in the match, unless the packet type is Ethernet (see 7.4.1). *Packet-out* messages must include the Packet Type Match Field in the match, unless the packet type is Ethernet (see 7.3.6).

The Packet Type (0, 0) is the packet type for Ethernet packets. For this packet type, the match may include, for example, the Ethernet destination address, Ethernet source address, Ethertype, and other match fields based on the Ethertype. For this packet type, the packet-in and packet-out data field includes the Ethernet headers and the Ethernet payload of an Ethernet frame, but does not include the preamble, the start of frame, the Ethernet FCS/CRC and the interframe gap. This packet type would typically be used in Ethernet switches.

The Packet Type (1, 0x800) is the packet type for IPv4 packets. For this packet type, the match may include, for example, IPv4 destination and source addresses, IP protocol and other match fields based on the protocol. For this packet type, the packet-in and packet-out data field includes the IPv4 header and the IPv4 payload of the IPv4 datagram. For example, this packet type may be used in IP routers.

The Packet Type (1, 0x86dd) is the packet type for IPv6 packets. For this packet type, the match may include, for example, IPv6 destination and source addresses, IP protocol and other match fields based on the protocol. For this packet type, the packet-in and packet-out data field includes the IPv6 header and the IPv6 payload of the IPv6 datagram. For example, this packet type may be used in IP routers.

The Packet Type (0, 1) is the packet type for tables that don't match packets. For this packet type, the match only includes pipeline match fields, pseudo-header fields and does not include any real header fields. For example, this packet type may be used in circuit switches.

The Packet Type (0, 0xFFFF) is the packet type for Experimenter packet types. The actual packet type is defined by the first OXM experimenter value following it (see 7.2.3.12). This allows experimenters to define arbitrary packet types.

7.2.3.12 Experimenter Flow Match Fields

Support for experimenter-specific flow match fields is optional. Experimenter-specific flow match fields may be defined using the `oxm_class`=`OFPXMC_EXPERIMENTER`.

```
/* Header for OXM experimenter match fields.
 * The experimenter class should not use OXM_HEADER() macros for defining
 * fields due to this extra header. */
struct ofp_oxm_experimenter_header {
    uint32_t oxm_header;        /* oxm_class = OFPXMC_EXPERIMENTER */
    uint32_t experimenter;      /* Experimenter ID. */
};
OFP_ASSERT(sizeof(struct ofp_oxm_experimenter_header) == 8);
```

The `oxm_class` field in the OXM header must be set to `OFPXMC_EXPERIMENTER`.

 © 2015; The Open Networking Foundation

The `oxm_field` field in the OXM header is the experimenter type, which is experimenter managed (see 7.2.8). Together, the experimenter identifier and the experimenter type identifies the experimenter match field.

The field `oxm_length` in the OXM header counts all of the bytes that follow `oxm_header`. For experimenter OXMs, `oxm_length` notably includes the `experimenter` field itself.

The `experimenter` field is encoded in the first four bytes of the OXM TLV's body. It contains the experimenter identifier, which takes the same form as in the typical experimenter structure (see 7.2.8).

The rest of the OXM TLV body is experimenter-defined and does not need to be padded or aligned.

7.2.4 Flow Stats Structures

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

7.2.4.1 Flow Stats Header

The flow stats header is described by the `ofp_stats` structure:

```
/* Flow statistics structure - list of statistic fields. */
struct ofp_stats {
    uint16_t reserved;         /* Reserved for future use, currently zeroed. */
    uint16_t length;           /* Length of ofp_stats (excluding padding) */
    /* Followed by:
     *   - Exactly (length - 4) (possibly 0) bytes containing OXS TLVs, then
     *   - Exactly ((length + 7)/8*8 - length) (between 0 and 7) bytes of
     *     all-zero bytes
     * In summary, ofp_stats is padded as needed, to make its overall size
     * a multiple of 8, to preserve alignement in structures using it.
     */
    uint8_t oxs_fields[0];     /* 0 or more OXS stat fields */
    uint8_t pad[4];            /* Zero bytes - see above for sizing */
};
OFP_ASSERT(sizeof(struct ofp_stats) == 8);
```

The `length` field is set to the actual length of `ofp_stats` structure including all stat fields. The payload of the OpenFlow stats is a set of OXS Flow stat fields.

7.2.4.2 Flow Stat Field Structures

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

An OXS TLV's header is interpreted as a 32-bit word in network byte order (see figure 10).

The OXS TLV's header fields are defined in Table 17

 © 2015; The Open Networking Foundation

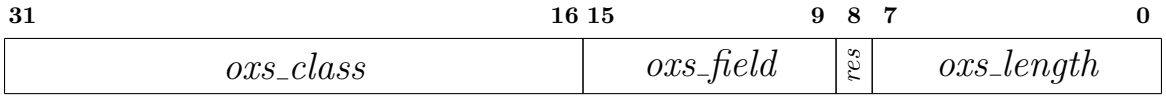


Figure 10: OXS TLV header layout.

Name		Width	Usage
oxs_type	oxs_class	16	Stat class: member class or reserved class
	oxs_field	7	Stat field within the class
	oxs_reserved	1	Reserved for future use
	oxs_length	8	Length of OXS payload

Table 17: OXS TLV header fields.

The oxs_class is a OXS stat class that contains related stat types, and is described in section 7.2.4.3. oxs_field is a class-specific value, identifying one of the stat types within the stat class. The combination of oxs_class and oxs_field (the most-significant 23 bits of the header) are collectively oxs_type. The oxs_type normally designates a statistic field, such as flow byte count.

oxs_reserved is reserved for future use and must be set to zero.

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

For a given oxs_class, oxs_field, and oxs_reserved value, oxs_length is a constant. It is included only to allow software to minimally parse OXS TLVs of unknown types. (Similarly, for a given oxs_class, oxs_field, and oxs_length values, oxs_reserved is a constant.)

7.2.4.3 OXS classes

The stat types are structured using OXS stat classes. Those follow the same convention as the OXM match classes (see 7.2.3.3). The following OXS classes are defined:

```
/* OXS Class IDs.
 * The high order bit differentiate reserved classes from member classes.
 * Classes 0x0000 to 0x7FFF are member classes, allocated by ONF.
 * Classes 0x8000 to 0xFFFE are reserved classes, reserved for standardisation.
 */
enum ofp_oxs_class {
    OFPXSC_OPENFLOW_BASIC = 0x8002,    /* Basic stats class for OpenFlow */
    OFPXSC_EXPERIMENTER   = 0xFFFF,    /* Experimenter class */
};
```

The class OFPXSC_OPENFLOW_BASIC contains the basic set of OpenFlow stat fields (see 7.2.4.4). The optional class OFPXSC_EXPERIMENTER is used for experimenter stat fields (see 7.2.4.5).

 © 2015; The Open Networking Foundation

7.2.4.4 Flow Stat Fields

The specification defines a default set of stat fields with `oxs_class=OFPXSC_OPENFLOW_BASIC` which can have the following values:

```
/* OXS flow stat field types for OpenFlow basic class. */
enum oxs_ofb_stat_fields {
    OFPXST_OFB_DURATION        = 0,  /* Time flow entry has been alive. */
    OFPXST_OFB_IDLE_TIME       = 1,  /* Time flow entry has been idle. */
    OFPXST_OFB_FLOW_COUNT      = 3,  /* Number of aggregated flow entries. */
    OFPXST_OFB_PACKET_COUNT    = 4,  /* Number of packets in flow entry. */
    OFPXST_OFB_BYTE_COUNT      = 5,  /* Number of bytes in flow entry. */
};
```

Each flow table of the switch must support the required stat fields listed in Table 18. Optional stat fields must be included in the `ofp_stats` structure only if they are supported by the flow table ; if the flow table does not support them, they can't be included in `ofp_stats` structure.

All stat fields have different size, as specified in Table 18.

Field	Bits	Flow entry	Aggregate	Description
OXS_OF_DURATION	2*32	Required	Undefined	Time flow entry has been alive.
OXS_OF_IDLE_TIME	2*32	Optional	Undefined	Time flow entry has been idle.
OXS_OF_FLOW_COUNT	32	Undefined	Required	Number of aggregated flow entries.
OXS_OF_PACKET_COUNT	64	Optional	Optional	Number of packets matched by a flow entry.
OXS_OF_BYTE_COUNT	64	Optional	Optional	Number of bytes matched by a flow entry.

Table 18: Stat fields details.

The `OXS_OF_DURATION` field indicates the elapsed time the flow entry has been installed in the switch. The first 32 bit value in the payload is `duration_sec` and the second 32 bit value is `duration_nsec`. The total duration in nanoseconds can be computed as $duration_sec \times 10^9 + duration_nsec$. Implementations are required to provide second precision; higher precision is encouraged where available. This field is not defined for `OFPMP_AGGREGATE` replies and should not be used in that case.

The `OXS_OF_IDLE_TIME` field indicates the elapsed time for which the flow entry has not matched any packets, and is the value that would be used by the switch to compare against the flow entry `idle_timeout` (see 7.3.4.2). The encoding of this time value is identical to the `OXS_OF_DURATION` field. This field is not defined for `OFPMP_AGGREGATE` replies and should not be used in that case.

The `OXS_OF_FLOW_COUNT` field is only used in `OFPMP_AGGREGATE` replies and indicates how many flow entries match the request (see 7.3.5.4). This field is not defined for flow entries (i.e. outside of `OFPMP_AGGREGATE` replies) and should not be used in those cases.

The `OXS_OF_PACKET_COUNT` field is the number of packets matched by a flow entry. In `OFPMP_AGGREGATE` replies, it is the number of packets matched by all the flow entries matching the request.

The `OXS_OF_BYTE_COUNT` field is the number of bytes in packets matched by a flow entry. In `OFPMP_AGGREGATE` replies, it is the number of bytes matched by all the flow entries matching the request.

 © 2015; The Open Networking Foundation

7.2.4.5 Experimenter Flow Stat Fields

Support for experimenter-specific flow stat fields is optional. Experimenter-specific flow stat fields may be defined using the `oxs_class`=`OFPXSC_EXPERIMENTER`.

```
/* Header for OXS experimenter stat fields. */
struct ofp_oxs_experimenter_header {
    uint32_t oxs_header;        /* oxs_class = OFPXSC_EXPERIMENTER */
    uint32_t experimenter;      /* Experimenter ID. */
};
OFP_ASSERT(sizeof(struct ofp_oxs_experimenter_header) == 8);
```

The `oxs_class` field in the OXS header must be set to `OFPXMC_EXPERIMENTER`.

The `oxs_field` field in the OXS header is the experimenter type, which is experimenter managed (see 7.2.8). Together, the experimenter identifier and the experimenter type identify the experimenter stat field.

The `experimenter` field is encoded in the first four bytes of the OXS TLV's body. It contains the experimenter identifier, which takes the same form as in the typical experimenter structure (see 7.2.8).

The rest of the OXS TLV body is experimenter-defined and does not need to be padded or aligned.

7.2.5 Flow Instruction Structures

Flow instructions associated with a flow table entry are executed when a flow matches the entry. The list of instructions that are currently defined are:

```
enum ofp_instruction_type {
    OFPIT_GOTO_TABLE = 1,       /* Setup the next table in the lookup
                                   pipeline */
    OFPIT_WRITE_METADATA = 2,   /* Setup the metadata field for use later in
                                   pipeline */
    OFPIT_WRITE_ACTIONS = 3,    /* Write the action(s) onto the datapath action
                                   set */
    OFPIT_APPLY_ACTIONS = 4,    /* Applies the action(s) immediately */
    OFPIT_CLEAR_ACTIONS = 5,    /* Clears all actions from the datapath
                                   action set */
    OFPIT_DEPRECATED = 6,       /* Deprecated (was apply meter) */
    OFPIT_STAT_TRIGGER = 7,     /* Statistics triggers */

    OFPIT_EXPERIMENTER = 0xFFFF  /* Experimenter instruction */
};
```

The instruction set is described in section 5.5. Flow tables may support a subset of instruction types. An instruction definition contains the instruction type, length, and any associated data:

 © 2015; The Open Networking Foundation

```
/* Instruction header that is common to all instructions.  The length includes
 * the header and any padding used to make the instruction 64-bit aligned.
 * NB: The length of an instruction *must* always be a multiple of eight. */
struct ofp_instruction_header {
    uint16_t type;                /* One of OFPIT_*. */
    uint16_t len;                 /* Length of this struct in bytes. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_header) == 4);
```

The OFPIT_GOTO_TABLE instruction uses the following structure and fields:

```
/* Instruction structure for OFPIT_GOTO_TABLE */
struct ofp_instruction_goto_table {
    uint16_t type;                /* OFPIT_GOTO_TABLE */
    uint16_t len;                 /* Length is 8. */
    uint8_t table_id;             /* Set next table in the lookup pipeline */
    uint8_t pad[3];               /* Pad to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_goto_table) == 8);
```

table_id indicates the next table in the packet processing pipeline.

The OFPIT_WRITE_METADATA instruction uses the following structure and fields:

```
/* Instruction structure for OFPIT_WRITE_METADATA */
struct ofp_instruction_write_metadata {
    uint16_t type;                /* OFPIT_WRITE_METADATA */
    uint16_t len;                 /* Length is 24. */
    uint8_t pad[4];               /* Align to 64-bits */
    uint64_t metadata;            /* Metadata value to write */
    uint64_t metadata_mask;       /* Metadata write bitmask */
};
OFP_ASSERT(sizeof(struct ofp_instruction_write_metadata) == 24);
```

Metadata for the next table lookup can be written using the metadata and the metadata_mask in order to set specific bits on the match field. If this instruction is not specified, the metadata is passed, unchanged.

The OFPIT_WRITE_ACTIONS, OFPIT_APPLY_ACTIONS, and OFPIT_CLEAR_ACTIONS instructions use the following structure and fields:

```
/* Instruction structure for OFPIT_WRITE/APPLY/CLEAR_ACTIONS */
struct ofp_instruction_actions {
    uint16_t type;              /* One of OFPIT_*_ACTIONS */
    uint16_t len;               /* Length is padded to 64 bits. */
    uint8_t pad[4];             /* Align to 64-bits */
    struct ofp_action_header actions[0];  /* 0 or more actions associated with
                                             OFPIT_WRITE_ACTIONS and
                                             OFPIT_APPLY_ACTIONS */
};
OFP_ASSERT(sizeof(struct ofp_instruction_actions) == 8);
```

 © 2015; The Open Networking Foundation

For the Apply-Actions instruction, the `actions` field is treated as a list and the actions are applied to the packet *in-order* (see 5.7).

For the Write-Actions instruction, the `actions` field is treated as a set and the actions are merged into the current action set (see 5.6). If the set of actions contains two actions of the same type or two set-field actions of the same type, the switch can either return an error (see 7.5.4.3), or the switch can merge the set of actions in the action set *in-order*, with the later action of the set of actions overwriting earlier actions of the same type (see 5.6).

For the Clear-Actions instruction, the structure does not contain any actions.

The `OFPIT_STAT_TRIGGER` instruction uses the following structure and fields:

```
/* Instruction structure for OFPIT_STAT_TRIGGER */
struct ofp_instruction_stat_trigger {
    uint16_t type;                /* OFPIT_STAT_TRIGGER */
    uint16_t len;                 /* Length is padded to 64 bits. */
    uint32_t flags;               /* Bitmap of OFPSTF_* flags. */
    struct ofp_stats thresholds;  /* Threshold list. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_stat_trigger) == 16);
```

The `flags` field is a bitmap that defines the behaviour of the statistic trigger. It may include a combination of the following flags:

```
enum ofp_stat_trigger_flags {
    OFPSTF_PERIODIC    = 1 << 0,  /* Trigger for all multiples of thresholds. */
    OFPSTF_ONLY_FIRST  = 1 << 1,  /* Trigger on only first reach threshold. */
};
```

When the `OFPSTF_PERIODIC` flag is set, the trigger will apply not only on the values in the thresholds, but also on all multiples of those values. It allows for example to have a trigger every 100 packets for the lifetime of the flow.

When the `OFPSTF_ONLY_FIRST` is set, only the first threshold that is crossed is considered, and other thresholds are ignored. It allows the controller to receive only a single trigger event for multiple thresholds.

The `thresholds` is a list of statistic field thresholds. When one of the statistic field values of the flow entry crosses its threshold a *stat trigger* event is sent to the controller. The flow entry is unmodified by this operation. For example, if the threshold list includes a byte count of 10000, when the flow entry byte count goes from 9000 to 10500, a message must be generated.

When a *stat trigger* event must be sent to the controller, the switch must add a flow stats entry with reason `OFPFSR_STAT_TRIGGER` in a flow stats multipart message (see 7.3.5.3), and that message must be sent to the controller. A flow stats multipart message may contain multiple unrelated *stat trigger* events to reduce overhead on the control channel. The switch may drop or delay any *stat trigger* event due to congestion and backup on the control channel or to optimise usage of the control channel. A controller is strongly advised to use this instruction in such a way that the event rate is manageable by the switch.

An `OFPIT_EXPERIMENTER` instruction uses the following structure and fields:

 © 2015; The Open Networking Foundation

```
/* Instruction structure for experimental instructions */
struct ofp_instruction_experimenter_header {
    uint16_t type;              /* OFPIT_EXPERIMENTER. */
    uint16_t len;               /* Length is padded to 64 bits. */
    uint32_t experimenter;      /* Experimenter ID. */
    /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_experimenter_header) == 8);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8). The rest of the instruction body is experimenter defined, the whole instruction need to be padded to 64 bits.

7.2.6 Action Structures

A number of actions may be associated with flow entries, groups or packets. The currently defined action types are:

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

    OFPAT_PUSH_VLAN    = 17, /* Push a new VLAN tag */
    OFPAT_POP_VLAN     = 18, /* Pop the outer VLAN tag */
    OFPAT_PUSH_MPLS    = 19, /* Push a new MPLS tag */
    OFPAT_POP_MPLS     = 20, /* Pop the outer MPLS tag */
    OFPAT_SET_QUEUE    = 21, /* Set queue id when outputting to a port */
    OFPAT_GROUP        = 22, /* Apply group. */
    OFPAT_SET_NW_TTL   = 23, /* IP TTL. */
    OFPAT_DEC_NW_TTL   = 24, /* Decrement IP TTL. */
    OFPAT_SET_FIELD    = 25, /* Set a header field using OXM TLV format. */
    OFPAT_PUSH_PBB     = 26, /* Push a new PBB service tag (I-TAG) */
    OFPAT_POP_PBB      = 27, /* Pop the outer PBB service tag (I-TAG) */
    OFPAT_COPY_FIELD   = 28, /* Copy value between header and register. */
    OFPAT_METER        = 29, /* Apply meter (rate limiter) */
    OFPAT_EXPERIMENTER = 0xffff
};
```

Output, group, and set-queue actions are described in Section 5.8, tag push/pop actions are described in Table 2, and Set-Field actions are described from their OXM types in Table 13.

Actions are used in flow entry instructions (see 7.2.5), in group buckets (see 7.3.4.3) and in packet-out messages (see 7.3.6). The use of an action that modifies the packet assumes that a corresponding set of headers exists in the packet. The effect of an action on a packet that does not have the corresponding set of headers is undefined (switch and field dependent). For example, using a *pop VLAN* action on a packet that does not have a VLAN tag, or using a *set TTL* on a packet without an IP or a MPLS header,

 © 2015; The Open Networking Foundation

are undefined. The switch may optionally reject flow entries for which an action is inconsistent with the match structure and prior actions of the flow entry (see 6.4). Controllers are strongly encouraged to avoid generating combinations of table entries that may yield inconsistent actions.

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 this struct in bytes. */
};
OFP_ASSERT(sizeof(struct ofp_action_header) == 4);
```

7.2.6.1 Output Action Structures

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

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

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

```
enum ofp_controller_max_len {
        OFPCML_MAX       = 0xffe5, /* maximum max_len value which can be used
                                      to request a specific byte length. */
        OFPCML_NO_BUFFER = 0xffff  /* indicates that no buffering should be
                                      applied and the whole packet is to be
                                      sent to the controller. */
};
```

 © 2015; The Open Networking Foundation

7.2.6.2 Group Action Structures

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

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

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

7.2.6.3 Set-Queue Action Structure

The *Set-Queue* action sets the queue id that will be used to map a flow entry to an already-configured queue on a port, regardless of the IP DSCP and VLAN PCP bits. The packet should not change as a result of a Set-Queue action. If the switch needs to set the DSCP/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/DSCP bits. In that case, it cannot map an arbitrary flow entry to a specific queue, therefore the Set-Queue action is not supported. The user can still use these queues and map flow entries to them by setting the relevant fields (IP DSCP, VLAN PCP), however the mapping from DSCP or PCP to the queues is implementation specific.

A *Set Queue* action uses the following structure and fields:

```
/* Action structure for OFPAT_SET_QUEUE. */
struct ofp_action_set_queue {
    uint16_t type;            /* OFPAT_SET_QUEUE. */
    uint16_t len;             /* Len is 8. */
    uint32_t queue_id;        /* Queue id for the packets. */
};
OFP_ASSERT(sizeof(struct ofp_action_set_queue) == 8);
```

7.2.6.4 Meter Action Structure

The *Meter* action uses the following structure and fields:

```
/* Action structure for OFPAT_METER */
struct ofp_action_meter {
    uint16_t type;                /* OFPAT_METER */
    uint16_t len;                 /* Length is 8. */
    uint32_t meter_id;            /* Meter instance. */
};
OFP_ASSERT(sizeof(struct ofp_action_meter) == 8);
```

`meter_id` indicates which meter to apply on the packet.

 © 2015; The Open Networking Foundation

7.2.6.5 TTL Actions Structures

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

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

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

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

```
/* Action structure for OFPAT_COPY_TTL_OUT, OFPAT_COPY_TTL_IN,
 * OFPAT_DEC_MPLS_TTL, OFPAT_DEC_NW_TTL, OFPAT_POP_VLAN and OFPAT_POP_PBB. */
struct ofp_action_generic {
    uint16_t type;                  /* One of OFPAT_*. */
    uint16_t len;                   /* Length is 8. */
    uint8_t pad[4];                 /* Pad to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_action_generic) == 8);
```

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

```
/* Action structure for OFPAT_SET_NW_TTL. */
struct ofp_action_nw_ttl {
    uint16_t type;                  /* OFPAT_SET_NW_TTL. */
    uint16_t len;                   /* Length is 8. */
    uint8_t nw_ttl;                 /* IP TTL */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_nw_ttl) == 8);
```

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

A *Decrement Network TTL* action takes no arguments and consists only of a generic `ofp_action_generic`. This action decrements the TTL in the IP header if one is present.

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

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

 © 2015; The Open Networking Foundation

7.2.6.6 Push and Pop Actions Structures

The *Push VLAN header*, *Push MPLS header* and *Push PBB header* actions use the following structure and fields:

```
/* Action structure for OFPAT_PUSH_VLAN/MPLS/PBB. */
struct ofp_action_push {
    uint16_t type;                  /* OFPAT_PUSH_VLAN/MPLS/PBB. */
    uint16_t len;                   /* Length is 8. */
    uint16_t ethertype;             /* Ethertype */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_push) == 8);
```

The `ethertype` field indicates the Ethertype of the new tag. It is used when pushing a new VLAN tag, new MPLS header or PBB service header to identify the type of the new header, and it must be valid for that tag type.

The Push tag actions *always* insert a new tag header in the outermost valid location for that tag, as defined by the specifications governing that tag.

The *Push VLAN header* action pushes a new VLAN tag onto the packet (C-TAG or S-TAG). When a new VLAN tag is pushed, it should be the outermost tag inserted, immediately after the Ethernet header and before other tags. The `ethertype` field must be 0x8100 or 0x88a8.

The *Push MPLS header* action pushes a new MPLS shim header onto the packet. When a new MPLS tag is pushed on an IP packet, it is the outermost MPLS tag, inserted as a shim header immediately before any MPLS tags or immediately before the IP header, whichever comes first. The `ethertype` field must be 0x8847 or 0x8848.

The *Push PBB header* action logically pushes a new PBB service instance header onto the packet (I-TAG TCI), and copies the original Ethernet addresses of the packet into the customer addresses (C-DA and C-SA) of the tag. The PBB service instance header should be the outermost tag inserted, immediately after the Ethernet header and before other tags. The `ethertype` field must be 0x88E7. The customer addresses of the I-TAG are in the location of the original Ethernet addresses of the encapsulated packet, therefore this action can be seen as adding both the backbone MAC-in-MAC header and the I-SID field to the front of the packet. The *Push PBB header* action does not add a backbone VLAN header (B-TAG) to the packet, it can be added via the *Push VLAN header* action after the push PBB header operation. After this operation, regular set-field actions can be used to modify the outer Ethernet addresses (B-DA and B-SA).

A *Pop VLAN header* action takes no arguments and consists only of a generic `ofp_action_generic`. The action pops the outermost VLAN tag from the packet.

A *Pop PBB header* action takes no arguments and consists only of a generic `ofp_action_generic`. The action logically pops the outer-most PBB service instance header from the packet (I-TAG TCI) and copies the customer addresses (C-DA and C-SA) in the Ethernet addresses of the packet. This action can be seen as removing the backbone MAC-in-MAC header and the I-SID field from the front of the packet. The *Pop PBB header* action does not remove the backbone VLAN header (B-TAG) from the packet, it should be removed prior to this operation via the *Pop VLAN header* action.

 © 2015; The Open Networking Foundation

A *Pop MPLS header* action uses the following structure and fields:

```
/* Action structure for OFPAT_POP_MPLS. */
struct ofp_action_pop_mpls {
    uint16_t type;                  /* OFPAT_POP_MPLS. */
    uint16_t len;                   /* Length is 8. */
    uint16_t ethertype;             /* Ethertype */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_pop_mpls) == 8);
```

The `ethertype` indicates the Ethertype of the MPLS payload. The `ethertype` is used as the Ethertype for the resulting packet regardless of whether the "bottom of stack (BoS)" bit was set in the removed MPLS shim. It is recommended that flow entries using this action match both the MPLS label and the MPLS BoS fields to avoid applying the wrong Ethertype to the MPLS payload.

The MPLS specification does not allow setting an arbitrary Ethertype to MPLS payload when BoS is not equal to 1, and the controller is responsible in complying with this requirement and only set 0x8847 or 0x8848 as the Ethertype for those MPLS payloads. The switch can optionally enforce this MPLS requirement: in this case the switch should reject any flow entry matching a wildcard BoS and any flow entry matching BoS to 0 with the wrong `ethertype` in the Pop MPLS header action, and in both cases should return a *Match Inconsistent* error message (see 7.5.4.3).

7.2.6.7 Set-Field Action Structure

The *Set Field* actions use the following structure and fields:

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

The `field` contains a header field or pipeline field described using a single OXM TLV structure (see 7.2.3). *Set-Field* actions are defined by `oxm_type`, the type of the OXM TLV, and modify the corresponding header field or pipeline field of the packet with the value of `oxm_value`, the payload of the OXM TLV, and optionally using a bitmask `oxm_mask` if `oxm_hasmask` is set.

The type of a *set-field* action is one of the valid OXM header types. The list of possible OXM types are described in Section 7.2.3.7 and Table 13. All header match fields are valid in the set-field action, except for `OXM_OF_IPV6_EXTHDR`. The pipeline fields `OXM_OF_METADATA`, `OXM_OF_TUNNEL_ID` and all `OXM_OF_PKT_REG(N)` are valid in the set-field action, other pipeline fields, `OXM_OF_IN_PORT` and

 © 2015; The Open Networking Foundation

OXM_OF_IN_PHY_PORT are not valid in the set-field action. The *set-field* action can include an Experimenter OXM field, the validity of Experimenter Set-Field actions is defined by the Experimenter OXM type itself.

The value of oxm_value indicates the value to write in the packet header or pipeline field. When masking is not used, the value in the payload of the OXM TLV must be valid, in particular bits of the payload not defined by the OXM type must be set to zero. When masking is used, the bits of oxm_value that correspond to a 0-bit in oxm_mask must be set to 0, and the remaining bits must form a valid modification of the field. In an OXM_OF_VLAN_VID set-field action, the OFPVID_PRESENT bit must be a 1-bit in oxm_value and in oxm_mask.

The value of oxm_hasmask is used to indicate that the action contains a bitmask for the value. This feature is optional and can only be used if the switch indicated its availability (see 7.3.5.18.2), otherwise an error must be generated (see 7.5.4.3). If oxm_hasmask is 0, the full header field is set to oxm_value. If oxm_hasmask is 1, then the oxm_entry's body contains a value for the field (oxm_value), followed by a bitmask of the same length as the value, called oxm_mask. For each 1-bit in oxm_mask, the corresponding bit of header field is set with the corresponding bit from oxm_value. For each 0-bit in oxm_mask, the corresponding bit of header field is preserved.

When using masking, it is an error for a 0-bit in oxm_mask to have a corresponding 1-bit in oxm_value (see 7.5.4.3).

When the *Set-Field* action changes a header field, it overwrites the actual packet header field in the packet specified by the OXM type. The OXM field refers to the outermost-possible occurrence in the header, unless the field type explicitly specifies otherwise, and therefore in general the set-field actions apply to the outermost-possible header (e.g. a "Set VLAN ID" set-field action always sets the ID of the outermost VLAN tag). When a header field is overwritten with the *set-field* action, the switch must recalculate all affected checksum fields before sending the packet out on an output port or in a packet-in message.

The OXM prerequisites (see 7.2.3.6) corresponding to the field to be set must be included in the flow entry, otherwise an error must be generated (see 7.5.4.3). Each prerequisite either must be included in the match of the flow entry or must be met through an action occurring before the set-field action (for example pushing a tag). The use of a *set-field* action assumes that the corresponding header field exists in the packet, the effect of a *set-field* action on a packet that does not have the corresponding header field is undefined (switch and field dependent). In particular, when setting the VID on a packet without a VLAN header, a switch may or may not automatically add a new VLAN tag, and the controller must explicitly use a *Push VLAN header* action to be compatible with all switch implementations. Controllers are strongly encouraged to avoid generating combinations of table entries that may yield inconsistent set-field actions.

7.2.6.8 Copy-Field Action Structure

The *Copy Field* action uses the following structure and fields:

```
/* Action structure for OFPAT_COPY_FIELD. */
struct ofp_action_copy_field {
    uint16_t type;                  /* OFPAT_COPY_FIELD. */
    uint16_t len;                   /* Length is padded to 64 bits. */
```

 © 2015; The Open Networking Foundation

```
    uint16_t n_bits;                /* Number of bits to copy. */
    uint16_t src_offset;            /* Starting bit offset in source. */
    uint16_t dst_offset;            /* Starting bit offset in destination. */
    uint8_t pad[2];                 /* Align to 32 bits. */
    /* Followed by:
     *   - Exactly 8, 12 or 16 bytes containing the oxm_ids, then
     *   - Enough all-zero bytes (either 0 or 4) to make the action a whole
     *     multiple of 8 bytes in length */
    uint32_t         oxm_ids[0];   /* Source and destination OXM headers */
};
OFP_ASSERT(sizeof(struct ofp_action_copy_field) == 12);
```

Copies `src_oxm_id[src_offset:src_offset+n_bits]` to `dst_oxm_id[dst_offset:dst_offset+n_bits]`, where a[b:c] denotes the bits within 'a' numbered 'b' through 'c' (not including bit 'c'). Bit numbering starts at 0 for the least-significant bit, 1 for the next most significant bit, and so on.

The `n_bits` field contains the number of bits to copy from `src_oxm_id` into `dst_oxm_id`.

The `src_offset` field indicates the bit offset in `src_oxm_id` where bits should be read.

The `dst_offset` field indicates the bit offset in `dst_oxm_id` where bits should be written.

The `oxm_ids` field is a list containing two OXM types. The first element of this list, `src_oxm_id`, identifies the header or pipeline field where the value is copied from. The second element of this list, `dst_oxm_id`, identifies the header or pipeline field where the value is copied to. The elements of that list are 32-bit OXM headers for non-experimenter OXM fields or 64-bit OXM headers for experimenter OXM fields, those OXM fields don't include any payload. The value of `oxm_hasmask` must be zero and no value or mask is included. If the *Copy Field* action is supported, it must support `src_oxm_id` or `dst_oxm_id` being one of the packet registers supported by the switch. Support for other pipeline fields and header fields is optional, for those fields this action must obey the same rules as the *Set Field* action, in particular with respect to prerequisites.

The switch must reject *Copy Field* actions for which `src_offset+n_bits` is greater than the width of `src_oxm_id` or `dst_offset+n_bits` is greater than the width of `dst_oxm_id` with a *Bad Set Argument* error message (see 7.5.4.3). This action must behaves properly when `src_oxm_id` overlaps with `dst_oxm_id`, that is, it behaves as if `src_oxm_id` were copied out to a temporary buffer, then the temporary buffer copied to `dst_oxm_id`, if this is not possible the switch must reject the *Copy Field* action with a *Bad Set Type* error message.

The effect of *copy-field* actions in the action set is undefined due to race conditions, and therefore its implementation in the action set is discouraged (see 5.6).

7.2.6.9 Experimenter Action Structure

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

```
/* Action header for OFPAT_EXPERIMENTER.
 * The rest of the body is experimenter-defined. */
struct ofp_action_experimenter_header {
    uint16_t type;                  /* OFPAT_EXPERIMENTER. */
    uint16_t len;                   /* Length is a multiple of 8. */
```

 © 2015; The Open Networking Foundation

```
    uint32_t experimenter;          /* Experimenter ID. */
};
OFP_ASSERT(sizeof(struct ofp_action_experimenter_header) == 8);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8). The rest of the instruction body is experimenter defined, the whole action need to be padded to 64 bits.

7.2.7 Controller Status Structure

The OpenFlow switch uses the controller status structure to inform the controller about the status of the control channel it maintains with each controller. This information is primarily useful in a multi-controller deployment.

The controller status structure always represents the switch's view of the status and looks like this:

```
/* Body of OFPMP_CONTROLLER_STATUS reply message and body of async
 * OFPT_CONTROLLER_STATUS message */
struct ofp_controller_status {
   uint16_t length;            /* Length of this entry. */
   uint16_t short_id;          /* ID number which identifies the controller. */
   uint32_t role;              /* Controller's role. One of OFPCR_ROLE_*. */
   uint8_t reason;             /* One of OFPCSR_* reason codes. */
   uint8_t channel_status;     /* Status of control channel.
                                * One of OFPCT_STATUS_*. */
   uint8_t pad[6];             /* Align to 64-bits. */

    /* Controller Status Property list.  The Connection URI property is
       required; other properties are optional. */
    struct ofp_controller_status_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_controller_status) == 16);
```

The `short_id` is the identifier for the controller as set by that controller in a `OFPT_ROLE_REQUEST` message. If the switch has not requested an identifier, the `short_id` should be set to `OFPCID_UNDEFINED`. `OFPCID_UNDEFINED` is 0.

The `role` is the role the controller currently has (see 6.3.7).

The `reason` indicates why the switch is sending the controller status. It is one of the following:

```
/* Why is the controller status being reported? */
enum ofp_controller_status_reason {
    OFPCSR_REQUEST            = 0,  /* Controller requested status. */
    OFPCSR_CHANNEL_STATUS     = 1,  /* Oper status of channel changed. */
    OFPCSR_ROLE               = 2,  /* Controller role changed. */
    OFPCSR_CONTROLLER_ADDED   = 3,  /* New controller added. */
    OFPCSR_CONTROLLER_REMOVED = 4,  /* Controller removed from config. */
    OFPCSR_SHORT_ID           = 5,  /* Controller ID changed. */
    OFPCSR_EXPERIMENTER       = 6,  /* Experimenter data changed. */
};
```

 © 2015; The Open Networking Foundation

The channel_status is the switch's view of the operational status of the control channel for the indicated controller. It takes its value from:

```
/* Control channel status. */
enum ofp_control_channel_status {
    OFPCT_STATUS_UP     = 0,    /* Control channel is operational. */
    OFPCT_STATUS_DOWN   = 1,    /* Control channel is not operational. */
};
```

The properties field is a list of controller status properties, describing additional status information. The list of controller status property types that are currently defined are:

```
/* Controller status property types.
 */
enum ofp_controller_status_prop_type {
    OFPCSPT_URI                     = 0,      /* Connection URI property. */
    OFPCSPT_EXPERIMENTER            = 0xFFFF, /* Experimenter property. */
};
```

The OFPCSPT_URI property is required and uses the following structure and fields:

```
/* Connection URI controller status property */
struct ofp_controller_status_prop_uri {
    uint16_t         type;    /* OFPCSPT_URI. */
    uint16_t         length;  /* Length in bytes of this property. */

    /* Followed by:
     *   - Exactly (length - 4) bytes containing Connection URI, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    char             uri[0];
};
OFP_ASSERT(sizeof(struct ofp_controller_status_prop_uri) == 4);
```

The uri is the Connection URI of the controller and has the form *protocol:name-or-address:port*. For example: tls:192.168.34.23:6653. See 6.3.1 for more detail.

The optional OFPCSPT_EXPERIMENTER property uses the following structure and fields:

```
/* Experimenter controller status property */
struct ofp_controller_status_prop_experimenter {
    uint16_t         type;    /* OFPCSPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_controller_status_prop_experimenter) == 12);
```

 © 2015; The Open Networking Foundation

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

7.2.8 Experimenter Structure

Experimenter extensions 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. Many OpenFlow object types offer Experimenter extensions, such as basic messages (see 7.5.5), OXM matches (see 7.2.3.2), instructions (see 7.2.5), actions (see 7.2.6.9), queues (see 7.3.5.8), meters (see 7.3.4.5) and error (see 7.5.4.19).

A typical experimenter structure looks like this :

```
/* Typical Experimenter structure. */
struct ofp_experimenter_structure {
    uint32_t experimenter;      /* Experimenter ID:
                                 * - MSB 0: low-order bytes are IEEE OUI.
                                 * - MSB != 0: defined by ONF. */
    uint32_t exp_type;          /* Experimenter defined. */
    uint8_t  experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_experimenter_structure) == 8);
```

The actual Experimenter extension included in some objects may not match exactly this structure due to various constraints.

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

The `exp_type` field is a 32-bit value present in some Experimenter extensions, it defines the experimenter type. This experimenter type is managed by the experimenter and can have any value and function the experimenter wishes.

The `experimenter_data` field represents the rest of the body of the Experimenter extension, it is uninterpreted by standard OpenFlow processing and is arbitrarily defined by the corresponding experimenter. The length of that body must fit within the OpenFlow object and the OpenFlow object length must be properly set to accommodate this data. Some OpenFlow objects require the whole Experimenter extension to be padded to 64 bits.

The actual Experimenter extensions are outside the scope of this specification, they are always optional, and a switch is not required to support any Experimenter extension. If a switch does not understand an experimenter extension, it must reject the entire message and send an appropriate error message. The error message to use depends on the OpenFlow object (see 7.5.4). If a switch rejects a specific Experimenter extension, the controller should no longer use that specific Experimenter extension in subsequent messages.

 © 2015; The Open Networking Foundation

7.3 Controller-to-Switch Messages

7.3.1 Handshake

The OFPT_FEATURES_REQUEST message is used by the controller to identify the switch and read its basic capabilities. Upon session establishment (see 6.3.3), the controller should send an OFPT_FEATURES_REQUEST message. This message does not contain a body beyond the OpenFlow header. The switch must respond 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 auxiliary_id;   /* Identify auxiliary connections */
    uint8_t pad[2];         /* Align to 64-bits. */

    /* Features. */
    uint32_t capabilities;  /* Bitmap of support "ofp_capabilities". */
    uint32_t reserved;
};
OFP_ASSERT(sizeof(struct ofp_switch_features) == 32);
```

The datapath_id field uniquely identifies a datapath. The lower 48 bits are intended for the switch MAC address, while the top 16 bits are up to the implementer. An example use of the top 16 bits would be a VLAN ID to distinguish multiple virtual switch instances on a single physical switch. This field should be treated as an opaque bit string by controllers.

The n_buffers field specifies the maximum number of packets the switch can buffer when sending packets to the controller using *packet-in* messages (see 6.1.2).

The n_tables field describes the number of tables supported by the switch, each of which can have a different set of supported match fields, actions and number of entries. When the controller and switch first communicate, the controller will find out how many tables the switch supports from the Features Reply. If it wishes to understand the size, types, and order in which tables are consulted, the controller sends a OFPMP_TABLE_FEATURES multipart request (see 7.3.5.18). A switch must return these tables in the order the packets traverse the tables.

The auxiliary_id field identifies the type of connection from the switch to the controller, the main connection has this field set to zero, an auxiliary connection has this field set to a non-zero value (see 6.3.8).

The capabilities field is a bitmap that uses a combination of the following flags :

 © 2015; The Open Networking Foundation

```
/* Capabilities supported by the datapath. */
enum ofp_capabilities {
    OFPC_FLOW_STATS     = 1 << 0,  /* Flow statistics. */
    OFPC_TABLE_STATS    = 1 << 1,  /* Table statistics. */
    OFPC_PORT_STATS     = 1 << 2,  /* Port statistics. */
    OFPC_GROUP_STATS    = 1 << 3,  /* Group statistics. */
    OFPC_IP_REASM       = 1 << 5,  /* Can reassemble IP fragments. */
    OFPC_QUEUE_STATS    = 1 << 6,  /* Queue statistics. */
    OFPC_PORT_BLOCKED   = 1 << 8,  /* Switch will block looping ports. */
    OFPC_BUNDLES        = 1 << 9,  /* Switch supports bundles. */
    OFPC_FLOW_MONITORING = 1 << 10,  /* Switch supports flow monitoring. */
};
```

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

7.3.2 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;             /* Bitmap of OFPC_* flags. */
    uint16_t miss_send_len;     /* Max bytes of packet that datapath
                                   should send to the controller. See
                                   ofp_controller_max_len for valid values.
                                   */
};
OFP_ASSERT(sizeof(struct ofp_switch_config) == 12);
```

The flags field is a bitmap that uses a combination of the following configuration flags :

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

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

 © 2015; The Open Networking Foundation

packet should not match any entry that has that field set. Support for "normal" is mandatory, support for "drop" and "reasm" is optional.

The `miss_send_len` field defines the number of bytes of each packet sent to the controller by the OpenFlow pipeline when not using an output action to the `OFPP_CONTROLLER` logical port, for example sending packets with invalid TTL if this message reason is enabled. If this field equals 0, the switch must send zero bytes of the packet in the `ofp_packet_in` message. If the value is set to `OFPCML_NO_BUFFER` the complete packet must be included in the message, and should not be buffered.

7.3.3 Flow Table Configuration

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

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

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

Flow entries are modified in the flow table using the `OFPT_FLOW_MOD` request (see 7.3.4.2). Table statistics can be queried using the `OFPMP_TABLE_STATS` multipart requests (see 7.3.5.16). The controller can configure the dynamic state in a flow table with the `OFPT_TABLE_MOD` request (see 7.3.4.2). The flow table static features can be modified using the `OFPMP_TABLE_FEATURES` multipart request (see 7.3.5.18).

7.3.4 Modify State Messages

7.3.4.1 Modify Flow Table Message

The controller can configure the dynamic state in a flow table with the `OFPT_TABLE_MOD` request. The `OFPT_TABLE_MOD` uses the following structure and fields:

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

    /* Table Mod Property list */
    struct ofp_table_mod_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_table_mod) == 16);
```

 © 2015; The Open Networking Foundation

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

The `config` field is a bitmap that configures the behaviour of the table. The following values are defined for that field :

```
/* Flags to configure the table. */
enum ofp_table_config {
    OFPTC_DEPRECATED_MASK    = 3,       /* Deprecated bits */
    OFPTC_EVICTION           = 1 << 2,  /* Authorise table to evict flows. */
    OFPTC_VACANCY_EVENTS     = 1 << 3,  /* Enable vacancy events. */
};
```

The flag `OFPTC_EVICTION` controls flow entry eviction in that flow table (see 6.5). If this flag is set, the switch can evict flow entries from that flow table. If this flag is unset, the switch cannot evict flow entries from that table. Flow entry eviction is optional and as a consequence a switch may not support setting this flag.

The flag `OFPTC_VACANCY_EVENTS` controls vacancy events in that table (see 7.4.5). If this flag is set, the switch must generate vacancy events for that table. If this flag is unset, the switch must not generate those events. Parameters for vacancy events may be specified using the `OFPTMPT_VACANCY` property (see below), if this property is not included in the table-mod message, previously set values of the parameters must be used.

The `properties` field is a list of table mod properties, describing dynamic parameters of table configuration.

The table mod property types currently defined are:

```
/* Table Mod property types.
 */
enum ofp_table_mod_prop_type {
    OFPTMPT_EVICTION               = 0x2,    /* Eviction property. */
    OFPTMPT_VACANCY                = 0x3,    /* Vacancy property. */
    OFPTMPT_EXPERIMENTER           = 0xFFFF, /* Experimenter property. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all Table Mod Properties */
struct ofp_table_mod_prop_header {
    uint16_t         type;    /* One of OFPTMPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_table_mod_prop_header) == 4);
```

The `OFPTMPT_EVICTION` property usually cannot be modified using a `OFP_TABLE_MOD` request, because the eviction mechanism is switch defined, and this property is usually only used in a `OFPMP_TABLE_DESC` multipart reply to inform the controller about the eviction mechanism used. It uses the following structure and fields:

 © 2015; The Open Networking Foundation

```
/* Eviction table mod Property. Mostly used in OFPMP_TABLE_DESC replies. */
struct ofp_table_mod_prop_eviction {
    uint16_t         type;    /* OFPTMPT_EVICTION. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         flags;   /* Bitmap of OFPTMPEF_* flags */
};
OFP_ASSERT(sizeof(struct ofp_table_mod_prop_eviction) == 8);
```

The `flags` field is a bitmap that defines the type of eviction the switch is implementing. It may include a combination of the following flags :

```
/* Eviction flags. */
enum ofp_table_mod_prop_eviction_flag {
    OFPTMPEF_OTHER           = 1 << 0,     /* Using other factors. */
    OFPTMPEF_IMPORTANCE      = 1 << 1,     /* Using flow entry importance. */
    OFPTMPEF_LIFETIME        = 1 << 2,     /* Using flow entry lifetime. */
};
```

If the flag `OFPTMPEF_IMPORTANCE` is set, the eviction process will use the flow entry `importance` field (see 7.3.4.2) to perform eviction. If the flag `OFPTMPEF_IMPORTANCE` is the only flag set, eviction will be performed strictly in order of `importance`, that is flow entry with lower importance will always be evicted before flow entry with higher importance. If a new flow entry to insert is less important than all existing flow entries, no eviction takes place. The eviction mechanism is switch defined, therefore it is not possible to predict in which order flow entries of same importance will get evicted.

If the flag `OFPTMPEF_LIFETIME` is set, the eviction process will use the flow entry remaining lifetime, the shortest time to expiration (see 6.5), to perform eviction. If the flag `OFPTMPEF_LIFETIME` is the only flag set, eviction will be performed stricly in order of remaining lifetime, and permanent flow entries are never removed.

If the flag `OFPTMPEF_OTHER` is set, the eviction process will use other factors, such as internal constraints, to perform eviction. When this flag is set, the controller cannot predict in which order flow entries will get evicted. This flag may be combined with other flags to signify that, for example, eviction is mostly done by the switch according to importance or lifetime but not strictly.

The `OFPTMPT_VACANCY` property specifies the parameters for vacancy events and uses the following structure and fields:

```
/* Vacancy table mod property */
struct ofp_table_mod_prop_vacancy {
    uint16_t         type;   /* OFPTMPT_VACANCY. */
    uint16_t         length; /* Length in bytes of this property. */
    uint8_t vacancy_down;    /* Vacancy threshold when space decreases (%). */
    uint8_t vacancy_up;      /* Vacancy threshold when space increases (%). */
    uint8_t vacancy;         /* Current vacancy (%) - only in ofp_table_desc. */
    uint8_t pad[1];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_table_mod_prop_vacancy) == 8);
```

 © 2015; The Open Networking Foundation

The fields vacancy_down and vacancy_up are the threshold for generating vacancy events that should be configured on this flow table, expressed as a percent.

When the remaining space in the flow table decreases to less than vacancy_down, and if vacancy down events are enabled, a vacancy down event must be generated to the controller using the OFPT_TABLE_STATUS message type with reason OFPTR_VACANCY_DOWN (see 7.4.5). Further vacancy down events are disabled until a vacancy up event is generated.

When the remaining space in the flow table increases to more than vacancy_up, and if vacancy up events are enabled, a vacancy up event must be generated to the controller using the OFPT_TABLE_STATUS message type with reason OFPTR_VACANCY_UP. Further vacancy up events are disabled until a vacancy down event is generated.

When vacancy events are enabled on the table using the OFPTC_VACANCY_EVENTS flag, either the vacancy up or vacancy down event is enabled. When enabling events, if the current vacancy is less than vacancy_up, vacancy up events must be enabled, and vacancy down events must be disabled. When enabling events, if the current vacancy is greater or equal to vacancy_up, vacancy down events must be enabled, and vacancy up events must be disabled. When vacancy events are disabled on the table using the OFPTC_VACANCY_EVENTS flag, both vacancy up and vacancy down events must be disabled.

If in an OFP_TABLE_MOD message the value of vacancy_down is greater than the value of vacancy_up, the switch should reject the OFP_TABLE_MOD message and send an ofp_error_msg with OFPET_BAD_PROPERTY type and OFPBPC_BAD_VALUE code.

The vacancy field is only used when this property in included in an OFPMP_TABLE_DESC multipart reply or an OFPT_TABLE_STATUS message and represent the current vacancy of the table, expressed as a percent. In OFP_TABLE_MOD requests, this field must be set to 0.

The OFPTMPT_EXPERIMENTER property uses the following structure and fields:

```
/* Experimenter table mod property */
struct ofp_table_mod_prop_experimenter {
    uint16_t         type;    /* OFPTMPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_table_mod_prop_experimenter) == 12);
```

The experimenter field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

 © 2015; The Open Networking Foundation

7.3.4.2 Modify Flow Entry Message

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

```
/* Flow setup and teardown (controller -> datapath). */
struct ofp_flow_mod {
    struct ofp_header header;
    uint64_t cookie;              /* Opaque controller-issued identifier. */
    uint64_t cookie_mask;         /* Mask used to restrict the cookie bits
                                     that must match when the command is
                                     OFPFC_MODIFY* or OFPFC_DELETE*. A value
                                     of 0 indicates no restriction. */
    uint8_t table_id;             /* ID of the table to put the flow in.
                                     For OFPFC_DELETE_* commands, OFPTT_ALL
                                     can also be used to delete matching
                                     flows from all tables. */
    uint8_t command;              /* One of OFPFC_*. */
    uint16_t idle_timeout;        /* Idle time before discarding (seconds). */
    uint16_t hard_timeout;        /* Max time before discarding (seconds). */
    uint16_t priority;            /* Priority level of flow entry. */
    uint32_t buffer_id;           /* Buffered packet to apply to, or
                                     OFP_NO_BUFFER.
                                     Not meaningful for OFPFC_DELETE*. */
    uint32_t out_port;            /* For OFPFC_DELETE* commands, require
                                     matching entries to include this as an
                                     output port.  A value of OFPP_ANY
                                     indicates no restriction. */
    uint32_t out_group;           /* For OFPFC_DELETE* commands, require
                                     matching entries to include this as an
                                     output group.  A value of OFPG_ANY
                                     indicates no restriction. */
    uint16_t flags;               /* Bitmap of OFPFF_* flags. */
    uint16_t importance;          /* Eviction precedence (optional). */
    struct ofp_match match;       /* Fields to match. Variable size. */
    /* The variable size and padded match is always followed by instructions. */
    //struct ofp_instruction_header instructions[0];
                          /* Instruction set - 0 or more. The length
                                     of the instruction set is inferred from
                                     the length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_flow_mod) == 56);
```

The cookie field is an opaque data value chosen by the controller. This value appears in flow removed messages (see 7.4.2) and flow description multipart messages (see 7.3.5.2), and can also be used to filter flow statistics, flow modification and flow deletion (see 6.4). It is not used by the packet processing pipeline, and thus does not need to reside in hardware. The value -1 (0xffffffffffffffff) is reserved and must not be used. When a flow entry is inserted in a table through an OFPFC_ADD message, its cookie field is set to the provided value. When a flow entry is modified (OFPFC_MODIFY or OFPFC_MODIFY_STRICT messages), its cookie field is unchanged.

If the cookie_mask field is non-zero, it is used with the cookie field to restrict flow matching while modifying or deleting flow entries. This field is ignored by OFPFC_ADD messages. The cookie_mask field's behavior is explained in Section 6.4.

 © 2015; The Open Networking Foundation

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

The command field must be one of the following:

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

The differences between OFPFC_MODIFY and OFPFC_MODIFY_STRICT, and between OFPFC_DELETE and OFPFC_DELETE_STRICT are explained in Section 6.4. An OpenFlow switch is required to implement all commands defined above.

The idle_timeout and hard_timeout fields control how quickly flow entries expire (see 6.5). When a flow entry is inserted in a table, its idle_timeout and hard_timeout fields are set with the values from the message. When a flow entry is modified (OFPFC_MODIFY or OFPFC_MODIFY_STRICT messages), the idle_timeout and hard_timeout fields are ignored.

The flow entry timeout values are used by the switch flow expiry mechanism. 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 entry 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 accuracy of the flow expiry process is not defined by the specification and is switch implementation dependent.

The priority field indicates the priority of the flow entry within the specified flow table. Higher numbers indicate higher priorities when matching packets (see 5.3). This field is used only for OFPFC_ADD messages when matching and adding flow entries, and for OFPFC_MODIFY_STRICT or OFPFC_DELETE_STRICT messages when matching flow entries. This field is not used for OFPFC_MODIFY or OFPFC_DELETE (non-strict) messages.

The buffer_id refers to a packet buffered at the switch and sent to the controller by a *packet-in* message. If no buffered packet is associated with the flow mod, it must be set to OFP_NO_BUFFER. A flow mod that includes a valid buffer_id removes the corresponding packet from the buffer and processes it through the entire OpenFlow pipeline after the flow is inserted, starting at the first flow table. This is effectively equivalent to sending a two-message sequence of a flow mod and a packet-out forwarding to the OFPP_TABLE logical port (see 7.3.6), with the requirement that the switch must fully process the flow mod before the packet out. These semantics apply regardless of the table to which the flow mod refers, or the instructions contained in the flow mod. This field is ignored by OFPFC_DELETE and OFPFC_DELETE_STRICT flow mod messages.

 © 2015; The Open Networking Foundation

The out_port and out_group fields optionally filter the scope of OFPFC_DELETE and OFPFC_DELETE_STRICT messages by output port and group. If either out_port or out_group contains a value other than OFPP_ANY or OFPG_ANY respectively, it introduces a constraint when matching. This constraint is that the flow entry must contain an output action directed at that port or group. Other constraints such as ofp_match structs and priorities are still used; this is purely an *additional* constraint. Note that to disable output filtering, both out_port and out_group must be set to OFPP_ANY and OFPG_ANY respectively. These fields are ignored by OFPFC_ADD, OFPFC_MODIFY or OFPFC_MODIFY_STRICT messages.

The flags field is a bitmap that may include a combination of the following flags:

```
enum ofp_flow_mod_flags {
    OFPFF_SEND_FLOW_REM = 1 << 0,  /* Send flow removed message when flow
                                    * expires or is deleted. */
    OFPFF_CHECK_OVERLAP = 1 << 1,  /* Check for overlapping entries first. */
    OFPFF_RESET_COUNTS  = 1 << 2,  /* Reset flow packet and byte counts. */
    OFPFF_NO_PKT_COUNTS = 1 << 3,  /* Don't keep track of packet count. */
    OFPFF_NO_BYT_COUNTS = 1 << 4,  /* Don't keep track of byte count. */
};
```

When the OFPFF_SEND_FLOW_REM flag is set, the switch must send a flow removed message when the flow entry expires or is deleted (see 6.5). An OpenFlow switch is required to honor this flag and implement the associated behaviour.

When the OFPFF_CHECK_OVERLAP flag is set, the switch must check that there are no overlapping flow entries with the same priority prior to inserting it in the flow table (see 6.4). A flow entry would overlap if it does not have the same match, it has the same priority and a packet could match both entries. If there is one, the flow mod fails and an error message is returned (see 7.5.4.6). If a switch does not support this flag, it must return an ofp_error_msg with OFPET_FLOW_MOD_FAILED type and OFPFMFC_BAD_FLAGS code.

When the OFPFF_RESET_COUNTS flag is set, the switch must clear the counters (byte and packet count) of the matching flow entries, if this flag is unset the counters are preserved. An OpenFlow switch is required to honor this flag and implement the associated behaviour.

When the OFPFF_NO_PKT_COUNTS flag is set, the switch does not need to keep track of the flow packet count. When the OFPFF_NO_BYT_COUNTS flag is set, the switch does not need to keep track of the flow byte count. Setting those flags may decrease the processing load on some OpenFlow switches, however those counters may not be available in flow statistics and flow removed messages for this flow entry. An OpenFlow switch is not required to honor these two flags, it may silently ignore them and may keep track of a flow count and return it despite the corresponding flag being set. If a switch does not keep track of a flow count, the corresponding counter is not available and must be set to the maximum field value (see 5.9).

When a flow entry is inserted in a table, its flags field is set with the values from the message. When a flow entry is matched and modified (OFPFC_MODIFY or OFPFC_MODIFY_STRICT messages), the flags of the flow entry are not changed, only OFPFF_RESET_COUNTS is used and other flags are ignored.

The match field contains the match structure and set of match fields defining how the flow entry matches packets (see 7.2.3). The combination of match field and priority field uniquely identifies a flow entry in the table (see 5.2).

 © 2015; The Open Networking Foundation

The importance field is the importance of the flow entry. This field may be optionally used by the flow entry eviction mechanism (see 6.5). When a flow entry is inserted in a table, its importance field is set with the values from the message. When a flow entry is modified (OFPFC_MODIFY or OFPFC_MODIFY_STRICT messages), the importance field is ignored.

The instructions field contains the instruction set for the flow entry when adding or modifying entries (see 7.2.5). If the instruction set is not valid or supported, the switch must generate an error (see 7.5.4.4).

7.3.4.3 Modify Group Entry Message

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

```
/* Group setup and teardown (controller -> datapath). */
struct ofp_group_mod {
    struct ofp_header header;
    uint16_t command;             /* One of OFPGC_*. */
    uint8_t type;                 /* One of OFPGT_*. */
    uint8_t pad;                  /* Pad to 64 bits. */
    uint32_t group_id;            /* Group identifier. */
    uint16_t bucket_array_len;    /* Length of action buckets data. */
    uint8_t pad2[2];              /* Pad to 64 bits. */
    uint32_t command_bucket_id;   /* Bucket Id used as part of
                                     OFPGC_INSERT_BUCKET and OFPGC_REMOVE_BUCKET
                                     commands execution.*/
    /* Followed by:
     *   - Exactly 'bucket_array_len' bytes containing an array of
     *     struct ofp_bucket.
     *   - Zero or more bytes of group properties to fill out the overall
     *     length in header.length. */
    struct ofp_bucket buckets[0]; /* The length of the bucket array is
                                     bucket_array_len bytes. */
    //struct ofp_group_prop_header properties[0];
};
OFP_ASSERT (sizeof(struct ofp_group_mod) == 24);
```

The semantics of the type and group fields are explained in Section 6.7.

The command field must be one of the following:

```
/* Group commands */
enum ofp_group_mod_command {
    OFPGC_ADD    = 0,       /* New group. */
    OFPGC_MODIFY = 1,       /* Modify all matching groups. */
    OFPGC_DELETE = 2,       /* Delete all matching groups. */
    OFPGC_INSERT_BUCKET = 3,/* Insert action buckets to the already available
                               list of action buckets in a matching group */
    /* OFPGC_??? = 4, */    /* Reserved for future use. */
    OFPGC_REMOVE_BUCKET = 5,/* Remove all action buckets or any specific action
                               bucket from matching group */
};
```

 © 2015; The Open Networking Foundation

The command OFPGC_ADD creates a new group based on the content of the request (and returns an error if a group with the same group_id already exists). The command OFPGC_MODIFY replaces the definition of an existing group with the content of the request. The command OFPGC_DELETE destroys the group specified by group_id, or all groups if group_id is OFPG_ALL. The command OFPGC_INSERT_BUCKET adds buckets to an existing group. The command OFPGC_REMOVE_BUCKET removes buckets from an existing group.

The type field specifies how the group behaves and which group buckets will be used to process packets (see 5.10.1), it must be one of the following values:

```
/* Group types.  Values in the range [128, 255] are reserved for experimental
 * use. */
enum ofp_group_type {
    OFPGT_ALL      = 0, /* All (multicast/broadcast) group.  */
    OFPGT_SELECT   = 1, /* Select group. */
    OFPGT_INDIRECT = 2, /* Indirect group. */
    OFPGT_FF       = 3, /* Fast failover group. */
};
```

The group_id field uniquely identifies a group within a switch, a group can have any Group ID lower than OFPG_MAX as long as it is unique on the switch. The following special group identifiers are defined:

```
/* Group numbering. Groups can use any number up to OFPG_MAX. */
enum ofp_group {
    /* Last usable group number. */
    OFPG_MAX        = 0xffffff00,

    /* Fake groups. */
    OFPG_ALL        = 0xfffffffc,  /* Represents all groups for group delete
                                      commands. */
    OFPG_ANY        = 0xffffffff   /* Special wildcard: no group specified. */
};
```

The bucket_array_len field is the size in bytes of the array of buckets in the buckets field.

The command_bucket_id field is either the Bucket ID of one of the buckets already existing in the group, or one of the following special bucket identifiers:

```
/* Bucket Id can be any value between 0 and OFPG_BUCKET_MAX */
enum ofp_group_bucket {
    OFPG_BUCKET_MAX   = 0xffffff00, /* Last usable bucket ID. */
    OFPG_BUCKET_FIRST = 0xfffffffd, /* First bucket ID in the list of action
                                       buckets of a group. This is applicable
                                       for OFPGC_INSERT_BUCKET and
                                       OFPGC_REMOVE_BUCKET commands. */
    OFPG_BUCKET_LAST  = 0xfffffffe, /* Last bucket ID in the list of action
                                       buckets of a group. This is applicable
                                       for OFPGC_INSERT_BUCKET and
                                       OFPGC_REMOVE_BUCKET commands. */
    OFPG_BUCKET_ALL   = 0xffffffff  /* All action buckets in a group,
                                       This is applicable for
                                       only OFPGC_REMOVE_BUCKET command. */
};
```

 © 2015; The Open Networking Foundation

The interpretation of the command_bucket_id field and which Bucket IDs are valid for that field depends on the command field. For OFPGC_INSERT_BUCKET, it defines the place in the bucket list to insert the new buckets. For OFPGC_REMOVE_BUCKET, it defines the buckets to be removed. The commands OFPGC_ADD, OFPGC_MODIFY and OFPGC_DELETE don't use the command_bucket_id field, for those commands it must be set to OFPG_BUCKET_ALL.

The buckets field is a list of action buckets (see 5.10). Group buckets are described in the next section (see 7.3.4.3.1).

The properties field is a list of group properties. Group properties are described in a following section (see 7.3.4.3.2).

7.3.4.3.1 Group Buckets

The buckets field in the OFPT_GROUP_MOD message is a list of buckets. An *Indirect* group must contain exactly one bucket (see 5.10.1), therefore the buckets field must contain a single bucket and any command that would make such a group have multiple buckets must return an error (see 7.5.4.7). Other group types may have multiple buckets in the buckets field. For *Fast Failover* group, the bucket order does define the bucket priorities (see 5.10.1), and the bucket order can be changed by modifying the group (for example using a OFPT_GROUP_MOD message with command OFPGC_MODIFY).

Group buckets use the following structure :

```
/* Bucket for use in groups. */
struct ofp_bucket {
    uint16_t len;                   /* Length of the bucket in bytes, including
                                       this header and any padding to make it
                                       64-bit aligned. */
    uint16_t action_array_len;      /* Length of all actions in bytes. */
    uint32_t bucket_id;             /* Bucket Id used to identify bucket*/
    /* Followed by:
     *   - Exactly 'action_array_len' bytes containing an array of
     *     struct ofp_action_*.
     *   - Zero or more bytes of group bucket properties to fill out the
     *     overall length in header.length. */
    struct ofp_action_header actions[0]; /* The length of the action array is
                                            action_array_len bytes. */
    //struct ofp_group_bucket_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_bucket) == 8);
```

The action_array_len field is the size in bytes of the set of actions in the actions field.

The bucket_id field uniquely identifies an action bucket of a group within a switch. Each bucket of the group must have an unique Bucket ID ; if a group mod would make the group have a duplicate Bucket ID, the switch must refuse the group mod and send a *Bucket Exist* error message (see 7.5.4.7). The bucket_id field must be lower than OFPG_BUCKET_MAX (it can not be one of the special bucket values).

The actions field is the set of actions associated with the bucket. When the bucket is selected for a packet, its actions are applied to the packet as an action set (see 5.6).

The properties field is a list of group bucket properties.

 © 2015; The Open Networking Foundation

The list of group bucket property types that are currently defined are:

```
/* Group bucket property types.  */
enum ofp_group_bucket_prop_type {
    OFPGBPT_WEIGHT                  = 0,  /* Select groups only. */
    OFPGBPT_WATCH_PORT              = 1,  /* Fast failover groups only. */
    OFPGBPT_WATCH_GROUP             = 2,  /* Fast failover groups only. */
    OFPGBPT_EXPERIMENTER      = 0xFFFF,  /* Experimenter defined. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all group bucket properties. */
struct ofp_group_bucket_prop_header {
    uint16_t         type;    /* One of OFPGBPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_group_bucket_prop_header) == 4);
```

The OFPGBPT_WEIGHT property uses the following structure and fields:

```
/* Group bucket weight property, for select groups only. */
struct ofp_group_bucket_prop_weight {
    uint16_t         type;    /* OFPGBPT_WEIGHT. */
    uint16_t         length;  /* 8. */
    uint16_t         weight;  /* Relative weight of bucket. */
    uint8_t          pad[2];  /* Pad to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_group_bucket_prop_weight) == 8);
```

The weight field is only defined for select groups, and its support is optional. For other group types, this property must be omitted. In select groups, the weight field is used to support unequal load sharing. If the switch does not support unequal load sharing, this field must be set to 1. The bucket's share of the traffic processed by the group is defined by the individual bucket's weight divided by the sum of the bucket weights in the group. If its weight is set to zero, the bucket is not used by the select group. When a port goes down, the change in traffic distribution is undefined. The precision by which a switch's packet distribution should match bucket weights is undefined.

The OFPGBPT_WATCH_PORT and OFPGBPT_WATCH_GROUP properties use the following structure and fields:

```
/* Group bucket watch port or watch group property, for fast failover groups
 * only. */
struct ofp_group_bucket_prop_watch {
    uint16_t         type;    /* OFPGBPT_WATCH_PORT or OFPGBPT_WATCH_GROUP. */
    uint16_t         length;  /* 8. */
    uint32_t         watch;   /* The port or the group.  */
};
OFP_ASSERT(sizeof(struct ofp_group_bucket_prop_watch) == 8);
```

 © 2015; The Open Networking Foundation

The watch field is respectively the port number or group identifier that is being watched by the bucket.

The OFPGBPT_WATCH_PORT and OFPGBPT_WATCH_GROUP properties are only required for *fast failover* groups, and may be optionally implemented for other group types. These properties indicate the port and/or group whose liveness controls whether this bucket is a candidate for forwarding (see 6.7). If watch_port is OFPP_ANY, no port is being watched. If watch_group is OFPG_ANY, no group is being watched. For fast failover groups, the first bucket defined is the highest-priority bucket, and only the highest-priority live bucket is used (see 5.10.1).

The OFPGBPT_EXPERIMENTER property uses the following structure and fields:

```
/* Experimenter group bucket property */
struct ofp_group_bucket_prop_experimenter {
    uint16_t         type;         /* OFPGBPT_EXPERIMENTER. */
    uint16_t         length;       /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_group_bucket_prop_experimenter) == 12);
```

The experimenter field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

7.3.4.3.2 Group Properties

The properties field in the OFPT_GROUP_MOD message is a list of group properties.

The list of group property types that are currently defined are:

```
/* Group property types.  */
enum ofp_group_prop_type {
    OFPGPT_EXPERIMENTER      = 0xFFFF,  /* Experimenter defined. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all group properties. */
struct ofp_group_prop_header {
    uint16_t         type;    /* One of OFPGPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_group_prop_header) == 4);
```

The OFPGPT_EXPERIMENTER property uses the following structure and fields:

 © 2015; The Open Networking Foundation

```
/* Experimenter group property */
struct ofp_group_prop_experimenter {
    uint16_t         type;         /* OFPGPT_EXPERIMENTER. */
    uint16_t         length;       /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_group_prop_experimenter) == 12);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

7.3.4.4 Port Modification Message

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

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

    /* Port mod property list - 0 or more properties */
    struct ofp_port_mod_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_port_mod) == 32);
```

The `config` field describes port administrative settings (see 7.2.1).

The `mask` field is used to select bits in the `config` field to change.

The `properties` field is a list of port description properties, enabling the configuration of additional properties of the port. All port properties do not need to be included in the `OFPT_PORT_MOD` message, properties that are omitted are unchanged.

The list of port description property types that are currently defined are:

 © 2015; The Open Networking Foundation

```
/* Port mod property types.
 */
enum ofp_port_mod_prop_type {
    OFPPMPT_ETHERNET          = 0,      /* Ethernet property. */
    OFPPMPT_OPTICAL           = 1,      /* Optical property. */
    OFPPMPT_EXPERIMENTER      = 0xFFFF, /* Experimenter property. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all port mod properties. */
struct ofp_port_mod_prop_header {
    uint16_t         type;    /* One of OFPPMPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_port_mod_prop_header) == 4);
```

The `OFPPMPT_ETHERNET` property uses the following structure and fields:

```
/* Ethernet port mod property. */
struct ofp_port_mod_prop_ethernet {
    uint16_t      type;       /* OFPPMPT_ETHERNET. */
    uint16_t      length;     /* Length in bytes of this property. */
    uint32_t      advertise;  /* Bitmap of OFPPF_*.  Zero all bits to prevent
                                 any action taking place. */
};
OFP_ASSERT(sizeof(struct ofp_port_mod_prop_ethernet) == 8);
```

The `advertise` field describe Ethernet features advertised on the link (7.2.1). It has no mask; all port features change together.

The `OFPPMPT_OPTICAL` property uses the following structure and fields:

```
struct ofp_port_mod_prop_optical {
    uint16_t      type;       /* OFPPMPT_OPTICAL. */
    uint16_t      length;     /* Length in bytes of this property. */
    uint32_t      configure;  /* Bitmap of OFPOPF_*. */
    uint32_t      freq_lmda;  /* The "center" frequency */
    int32_t       fl_offset;  /* signed frequency offset */
    uint32_t      grid_span;  /* The size of the grid for this port */
    uint32_t      tx_pwr;     /* tx power setting */
};
OFP_ASSERT(sizeof(struct ofp_port_mod_prop_optical) == 24);
```

The `configure` field describes optical features to change for this port. Any of `OFPOPF_*` may be set. Frequency is specified in MHz, wavelength (lambda) as nm * 100. The `OFPOPF_USE_FREQ` must match the advertised port feature. The `tx_pwr` is dBm * 10.

The tuned frequency is the sum of `freq_lmda` and `fl_offset`. This is for convenience of the software, since some tuning options, including Flex Grid, may be based upon a center frequency and an offset. The ”center” frequency is often used for passive filters, again making this convenient for the software.

 © 2015; The Open Networking Foundation

The grid_span is the amount of bandwidth consumed by this port, useful for Flex Grid as well as other tuning information.

The OFPPMPT_EXPERIMENTER property uses the following structure and fields:

```
/* Experimenter port mod property. */
struct ofp_port_mod_prop_experimenter {
    uint16_t         type;    /* OFPPMPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_port_mod_prop_experimenter) == 12);
```

The experimenter field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

7.3.4.5 Meter Modification Message

Modifications to a meter from the controller are done with the OFPT_METER_MOD message:

```
/* Meter configuration. OFPT_METER_MOD. */
struct ofp_meter_mod {
    struct ofp_header header;
    uint16_t            command;        /* One of OFPMC_*. */
    uint16_t            flags;          /* Bitmap of OFPMF_* flags. */
    uint32_t            meter_id;       /* Meter instance. */
    struct ofp_meter_band_header bands[0]; /* The band list length is
                                              inferred from the length field
                                              in the header. */
};
OFP_ASSERT(sizeof(struct ofp_meter_mod) == 16);
```

The meter_id field uniquely identifies a meter within a switch. Meters are defined starting with meter_id=1 up to the maximum number of meters that the switch can support. The OpenFlow switch protocol also defines some additional virtual meters that can not be associated with flows:

```
/* Meter numbering. Flow meters can use any number up to OFPM_MAX. */
enum ofp_meter {
    /* Last usable meter. */
    OFPM_MAX        = 0xffff0000,

    /* Virtual meters. */
    OFPM_SLOWPATH   = 0xfffffffd,  /* Meter for slow datapath. */
```

 © 2015; The Open Networking Foundation

```
    OFPM_CONTROLLER = 0xfffffffe,  /* Meter for controller connection. */
    OFPM_ALL        = 0xffffffff,  /* Represents all meters for stat requests
                                      commands. */
};
```

Virtual meters are provided to support existing implementations of OpenFlow. New implementations are encouraged to use regular per-flow meters (see 5.11) or priority queues (see 7.3.5.8) instead.

- `OFPM_CONTROLLER`: Virtual meter controlling all packets sent to the controllers via *Packet-in* messages, either using the **CONTROLLER** reserved port or in other processing (see 6.1.2). Can be used to limit the amount of traffic sent to the controllers.
- `OFPM_SLOWPATH`: Virtual meter controlling all packets processed by the slow datapath of the switch. Many switch implementations have a fast and slow datapath, for example a hardware switch may have a slow software datapath, or a software switch may have a slow userspace datapath.

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

```
/* Meter commands */
enum ofp_meter_mod_command {
    OFPMC_ADD    = 0,       /* New meter. */
    OFPMC_MODIFY = 1,       /* Modify specified meter. */
    OFPMC_DELETE = 2,       /* Delete specified meter. */
};
```

The `flags` field is a bitmap that may include a combination of following flags:

```
/* Meter configuration flags */
enum ofp_meter_flags {
    OFPMF_KBPS        = 1 << 0,  /* Rate value in kb/s (kilo-bit per second). */
    OFPMF_PKTPS       = 1 << 1,  /* Rate value in packet/sec. */
    OFPMF_BURST       = 1 << 2,  /* Do burst size. */
    OFPMF_STATS       = 1 << 3,  /* Collect statistics. */
};
```

The flag `OFPMF_PKTPS` specifies that the meter bands `rate` and `burst_size` should be interpreted as packets per second values. If the flag `OFPMF_PKTPS` is not set, the flag `OFPMF_KBPS` specifies that the meter bands `rate` and `burst_size` should be interpreted as kilo-bit per second values. The flag `OFPMF_STATS` specifies that the meter must collect the meter statistics it supports, if the flag is not set statistics don't need to be collected.

The flag `OFPMF_BURST` specifies that the meter bands `burst_size` value must be used. If the flag `OFPMF_BURST` is not set the `burst_size` values from meter bands are ignored, and if the meter implementation uses a burst value, this burst value must be set to an implementation defined optimal value.

The `bands` field is a list of rate bands. It can contain any number of bands, and each band type can be repeated when it makes sense. The meter default band (rate 0) can not be included nor configured in that list and does nothing to the packets. Only a single band is used for each packet, the band selection process is based on the band rates and the meter measured rate (see 5.11.1).

All the meter bands are defined using the same common header:

 © 2015; The Open Networking Foundation

```
/* Common header for all meter bands */
struct ofp_meter_band_header {
    uint16_t        type;    /* One of OFPMBT_*. */
    uint16_t        len;     /* Length in bytes of this band. */
    uint32_t        rate;    /* Rate for this band. */
    uint32_t        burst_size; /* Size of bursts. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_header) == 12);
```

The `rate` field indicates the rate value above which the corresponding band may apply to packets (see 5.11.1). The rate value is in kilobits per second, unless the `flags` field includes `OFPMF_PKTPS`, in which case the rate is in packets per second.

The `burst_size` field is used only if the `flags` field includes `OFPMF_BURST`. If the `OFPMF_BURST` flag is not set, the `burst_size` field must be set to 0. It defines the granularity of the meter band, for all packet or byte bursts whose length is greater than burst value, the meter rate will always be strictly enforced. The burst value is in kilobits, unless the `flags` field includes `OFPMF_PKTPS`, in which case the burst value is in packets.

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

```
/* Meter band types */
enum ofp_meter_band_type {
    OFPMBT_DROP            = 1,      /* Drop packet. */
    OFPMBT_DSCP_REMARK     = 2,      /* Remark DSCP in the IP header. */
    OFPMBT_EXPERIMENTER    = 0xFFFF  /* Experimenter meter band. */
};
```

An OpenFlow switch may not support all band types, and may not allow the use of all its supported band types on all meters, i.e. some meters may be specialised.

The band `OFPMBT_DROP` defines a simple rate limiter that drops packets that exceed the band rate value, and uses the following structure:

```
/* OFPMBT_DROP band - drop packets */
struct ofp_meter_band_drop {
    uint16_t        type;    /* OFPMBT_DROP. */
    uint16_t        len;     /* Length is 16. */
    uint32_t        rate;    /* Rate for dropping packets. */
    uint32_t        burst_size; /* Size of bursts. */
    uint8_t         pad[4];
};
OFP_ASSERT(sizeof(struct ofp_meter_band_drop) == 16);
```

The band `OFPMBT_DSCP_REMARK` defines a simple DiffServ policer that remarks the drop precedence of the DSCP field in the IP header of the packets that exceed the band rate value, and uses the following structure:

 © 2015; The Open Networking Foundation

```
/* OFPMBT_DSCP_REMARK band - Remark DSCP in the IP header */
struct ofp_meter_band_dscp_remark {
    uint16_t        type;    /* OFPMBT_DSCP_REMARK. */
    uint16_t        len;     /* Length is 16. */
    uint32_t        rate;    /* Rate for remarking packets. */
    uint32_t        burst_size; /* Size of bursts. */
    uint8_t         prec_level; /* Number of drop precedence level to add. */
    uint8_t         pad[3];
};
OFP_ASSERT(sizeof(struct ofp_meter_band_dscp_remark) == 16);
```

The `prec_level` field indicates by which amount the drop precedence of the packet should be increased if the band is exceeded. This band increases the encoded drop precedence by this amount, not the raw DSCP value ; it always results in a valid DSCP value, and DSCP values that do not encode a drop precedence are not modified.

The band `OFPMBT_EXPERIMENTER` is experimenter defined and uses the following structure:

```
/* OFPMBT_EXPERIMENTER band - Experimenter type.
 * The rest of the band is experimenter-defined. */
struct ofp_meter_band_experimenter {
    uint16_t        type;    /* OFPMBT_EXPERIMENTER. */
    uint16_t        len;     /* Length in bytes of this band. */
    uint32_t        rate;    /* Rate for this band. */
    uint32_t        burst_size;   /* Size of bursts. */
    uint32_t        experimenter; /* Experimenter ID. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_experimenter) == 16);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8). The rest of the band body is experimenter defined.

7.3.5 Multipart Messages

Multipart messages are used to encode requests or replies that potentially carry a large amount of data and would not always fit in a single OpenFlow message, which is limited to 64KB. The request or reply is encoded as a sequence of multipart messages on the same connection with a specific multipart type, and re-assembled by the receiver. Each sequence of multipart messages carries a single multipart request or reply. Multipart messages are primarily used to request statistics or state information from the switch.

The request is carried in one or more `OFPT_MULTIPART_REQUEST` messages:

```
struct ofp_multipart_request {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPMP_* constants. */
    uint16_t flags;             /* OFPMPF_REQ_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the request. 0 or more bytes. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_request) == 16);
```

 © 2015; The Open Networking Foundation

The switch responds with one or more OFPT_MULTIPART_REPLY messages:

```
struct ofp_multipart_reply {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPMP_* constants. */
    uint16_t flags;             /* OFPMPF_REPLY_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the reply. 0 or more bytes. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_reply) == 16);
```

The body field contains one segment of the request or reply. Every multipart request or reply is defined either as a single structure or as an array of 0 or more structures of the same type. If the multipart request or reply is defined as a single structure, it must use a single multipart message and the whole request or reply must be included in the body. If the multipart request or reply is defined as an array of structures, the body field must contain an integral number of objects, and no object can be split across two messages. To ease implementation, a multipart request or reply defined as an array may use messages with no additional entries (i.e. an empty body) at any point of the multipart sequence.

The flags field is a bitmap that controls the segmentation/reassembly process. In a multipart request message, it may include a combination of the following flags :

```
enum ofp_multipart_request_flags {
    OFPMPF_REQ_MORE  = 1 << 0  /* More requests to follow. */
};
```

In a multipart reply message, it may include a combination of the following flags :

```
enum ofp_multipart_reply_flags {
    OFPMPF_REPLY_MORE  = 1 << 0  /* More replies to follow. */
};
```

The OFPMPF_REQ_MORE bit and the OFPMPF_REPLY_MORE bit indicate that more requests/replies will follow the current one. If OFPMPF_REQ_MORE or OFPMPF_REPLY_MORE is set in a multipart message, then another multipart message of the same multipart sequence must always follow that message. A request or reply that spans multiple messages (has one or more messages with the *more* flag set), must use the same multipart type and transaction id (xid) for all messages in the message sequence.

Messages from a multipart request or reply may be interleaved with other OpenFlow message types, including other multipart requests or replies, but must have a distinct transaction ID on the connection if multiple unanswered multipart requests or replies are in flight simultaneously. Transaction ids of replies must always match the request that prompted them.

If a multipart request spans multiple messages and grows to a size that the switch is unable to buffer, the switch must respond with an error message of type OFPET_BAD_REQUEST and code OFPBRC_MULTIPART_BUFFER_OVERFLOW. If a multipart request contains a type that is not supported, the switch must respond with an error message of type OFPET_BAD_REQUEST and code OFPBRC_BAD_MULTIPART. If the switch receives a multipart message with the same xid as a multipart sequence received earlier on the same connection which is not terminated (did not receive a messages

 © 2015; The Open Networking Foundation

without the *more* flag set), and with a different multipart `type`, the switch must respond with an error message of type `OFPET_BAD_REQUEST` and code `OFPBRC_BAD_MULTIPART`. If a multipart request sequence contains more than one multipart request or other data beyond a single request, the switch must respond with an error message of type `OFPET_BAD_REQUEST` and code `OFPBRC_BAD_LEN`.

If a switch receives a multipart request message sequence that does not include a message with `OFPMPF_REQ_MORE` flag set to zero, after a switch defined amount of time greater than 100 milliseconds from the last message, the switch must discard the incomplete multipart request, and may generate an error messsage of type `OFPET_BAD_REQUEST` and code `OFPBRC_MULTIPART_REQUEST_TIMEOUT` to the controller. If a controller receives a multipart reply message sequence that does not include a message with `OFPMPF_REPLY_MORE` flag set to zero, after a controller defined amount of time greater than 1 second from the last message, the controller must discard the incomplete multipart reply, and may generate an error messsage of type `OFPET_BAD_REQUEST` and code `OFPBRC_MULTIPART_REPLY_TIMEOUT` to the switch. The controller may retry the failed switch operation as needed as a new multipart request.

In all types of multipart replies containing statistics, if a specific numeric counter is not available in the switch, if the counter is described using OXS (see 7.2.4.2), it must be omitted, otherwise its value must be set to the maximum field value (the unsigned equivalent of -1). Counters are unsigned and wrap around with no overflow indicator. Counters must use the full bit range defined for the counter before rolling over, for example if a counter is defined as 64 bits, it can not use only the lower 32 bits.

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_multipart_type {
    /* Description of this OpenFlow switch.
     * The request body is empty.
     * The reply body is struct ofp_desc. */
    OFPMP_DESC = 0,

    /* Individual flow descriptions and statistics.
     * The request body is struct ofp_flow_stats_request.
     * The reply body is an array of struct ofp_flow_desc. */
    OFPMP_FLOW_DESC = 1,

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

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

    /* Port statistics.
     * The request body is struct ofp_port_multipart_request.
     * The reply body is an array of struct ofp_port_stats. */
    OFPMP_PORT_STATS = 4,

    /* Queue statistics for a port
     * The request body is struct ofp_queue_multipart_request.
     * The reply body is an array of struct ofp_queue_stats */
```

 © 2015; The Open Networking Foundation

```
    OFPMP_QUEUE_STATS = 5,

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

    /* Group description.
     * The request body is struct ofp_group_multipart_request.
     * The reply body is an array of struct ofp_group_desc. */
    OFPMP_GROUP_DESC = 7,

    /* Group features.
     * The request body is empty.
     * The reply body is struct ofp_group_features. */
    OFPMP_GROUP_FEATURES = 8,

    /* Meter statistics.
     * The request body is struct ofp_meter_multipart_request.
     * The reply body is an array of struct ofp_meter_stats. */
    OFPMP_METER_STATS = 9,

    /* Meter configuration.
     * The request body is struct ofp_meter_multipart_request.
     * The reply body is an array of struct ofp_meter_desc. */
    OFPMP_METER_DESC = 10,

    /* Meter features.
     * The request body is empty.
     * The reply body is struct ofp_meter_features. */
    OFPMP_METER_FEATURES = 11,

    /* Table features.
     * The request body is either empty or contains an array of
     * struct ofp_table_features containing the controller's
     * desired view of the switch. If the switch is unable to
     * set the specified view an error is returned.
     * The reply body is an array of struct ofp_table_features. */
    OFPMP_TABLE_FEATURES = 12,

    /* Port description.
     * The request body is struct ofp_port_multipart_request.
     * The reply body is an array of struct ofp_port. */
    OFPMP_PORT_DESC = 13,

    /* Table description.
     * The request body is empty.
     * The reply body is an array of struct ofp_table_desc. */
    OFPMP_TABLE_DESC = 14,

    /* Queue description.
     * The request body is struct ofp_queue_multipart_request.
     * The reply body is an array of struct ofp_queue_desc. */
    OFPMP_QUEUE_DESC = 15,

    /* Flow monitors. Reply may be an asynchronous message.
     * The request body is an array of struct ofp_flow_monitor_request.
```

 © 2015; The Open Networking Foundation

```
     * The reply body is an array of struct ofp_flow_update_header. */
    OFPMP_FLOW_MONITOR = 16,

    /* Individual flow statistics (without description).
     * The request body is struct ofp_flow_stats_request.
     * The reply body is an array of struct ofp_flow_stats. */
    OFPMP_FLOW_STATS = 17,

    /* Controller status.
     * The request body is empty.
     * The reply body is an array of struct ofp_controller_status. */
    OFPMP_CONTROLLER_STATUS = 18,

    /* Bundle features.
    * The request body is ofp_bundle_features_request.
    * The reply body is struct ofp_bundle_features. */
    OFPMP_BUNDLE_FEATURES = 19,

    /* Experimenter extension.
     * The request and reply bodies begin with
     * struct ofp_experimenter_multipart_header.
     * The request and reply bodies are otherwise experimenter-defined. */
    OFPMP_EXPERIMENTER = 0xffff
};
```

7.3.5.1 Description

Information about the switch manufacturer, hardware revision, software revision, serial number, and a description field is available from the `OFPMP_DESC` multipart request type:

```
/* Body of reply to OFPMP_DESC request.  Each entry is a NULL-terminated
 * ASCII string. */
struct ofp_desc {
    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) == 1056);
```

Each entry is ASCII formatted and padded on the right with null bytes (\0). `DESC_STR_LEN` is 256 and `SERIAL_NUM_LEN` is 32. The `dp_desc` field is a free-form string to describe the datapath for debugging purposes, e.g., "switch3 in room 3120". As such, it is not guaranteed to be unique and should not be used as the primary identifier for the datapath—use the `datapath_id` field from the switch features instead (see 7.3.1).

7.3.5.2 Individual Flow Descriptions

Information about individual flow entries is requested with the `OFPMP_FLOW_DESC` multipart request type:

 © 2015; The Open Networking Foundation

```
/* Body for ofp_multipart_request of type OFPMP_FLOW_DESC & OFPMP_FLOW_STATS. */
struct ofp_flow_stats_request {
    uint8_t table_id;         /* ID of table to read (from ofp_table_desc),
                                 OFPTT_ALL for all tables. */
    uint8_t pad[3];           /* Align to 32 bits. */
    uint32_t out_port;        /* Require matching entries to include this
                                 as an output port.  A value of OFPP_ANY
                                 indicates no restriction. */
    uint32_t out_group;       /* Require matching entries to include this
                                 as an output group.  A value of OFPG_ANY
                                 indicates no restriction. */
    uint8_t pad2[4];          /* Align to 64 bits. */
    uint64_t cookie;          /* Require matching entries to contain this
                                 cookie value */
    uint64_t cookie_mask;     /* Mask used to restrict the cookie bits that
                                 must match. A value of 0 indicates
                                 no restriction. */
    struct ofp_match match;   /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 40);
```

The `match` field contains a description of the flow entries that should be matched and may contain wildcarded and masked fields. This field's matching behavior is described in Section 6.4.

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

The `out_port` and `out_group` fields optionally filter by output port and group. If either `out_port` or `out_group` contain a value other than `OFPP_ANY` and `OFPG_ANY` respectively, it introduces a constraint when matching. This constraint is that the flow entry must contain an output action directed at that port or group. Other constraints such as `match` field are still used ; this is purely an *additional* constraint. Note that to disable output filtering, both `out_port` and `out_group` must be set to `OFPP_ANY` and `OFPG_ANY` respectively.

The usage of the `cookie` and `cookie_mask` fields is defined in Section 6.4.

The `body` of the reply to an `OFPMP_FLOW_DESC` multipart request consists of an array of the following:

```
/* Body of reply to OFPMP_FLOW_DESC request. */
struct ofp_flow_desc {
    uint16_t length;          /* Length of this entry. */
    uint8_t pad2[2];          /* Align to 64-bits. */
    uint8_t table_id;         /* ID of table flow came from. */
    uint8_t pad;
    uint16_t priority;        /* Priority of the entry. */
    uint16_t idle_timeout;    /* Number of seconds idle before expiration. */
    uint16_t hard_timeout;    /* Number of seconds before expiration. */
    uint16_t flags;           /* Bitmap of OFPFF_* flags. */
    uint16_t importance;      /* Eviction precedence. */
    uint64_t cookie;          /* Opaque controller-issued identifier. */
    struct ofp_match match;   /* Description of fields. Variable size. */
    //struct ofp_stats stats; /* Statistics list. Variable size. */
    //struct ofp_instruction_header instructions[0];
                              /* Instruction set - 0 or more. */
};
OFP_ASSERT(sizeof(struct ofp_flow_desc) == 32);
```

 © 2015; The Open Networking Foundation

The fields consist of those provided in the `flow_mod` that created the flow entry (see 7.3.4.2), plus the `table_id` into which the entry was inserted.

The `stats` field contains a list of OXS fields (see 7.2.4.2). The field `OXS_OF_DURATION` is mandatory and tracks how long the flow entry has been installed in the switch. The fields `OXS_OF_PACKET_COUNT` and `OXS_OF_BYTE_COUNT` are optional and count all packets processed by the flow entry.

7.3.5.3 Individual Flow Statistics

Statistics about individual flow entries can also be requested with the `OFPMP_FLOW_STATS` multipart request type:

```
/* Body for ofp_multipart_request of type OFPMP_FLOW_DESC & OFPMP_FLOW_STATS. */
struct ofp_flow_stats_request {
    uint8_t table_id;         /* ID of table to read (from ofp_table_desc),
                                 OFPTT_ALL for all tables. */
    uint8_t pad[3];           /* Align to 32 bits. */
    uint32_t out_port;        /* Require matching entries to include this
                                 as an output port.  A value of OFPP_ANY
                                 indicates no restriction. */
    uint32_t out_group;       /* Require matching entries to include this
                                 as an output group.  A value of OFPG_ANY
                                 indicates no restriction. */
    uint8_t pad2[4];          /* Align to 64 bits. */
    uint64_t cookie;          /* Require matching entries to contain this
                                 cookie value */
    uint64_t cookie_mask;     /* Mask used to restrict the cookie bits that
                                 must match. A value of 0 indicates
                                 no restriction. */
    struct ofp_match match;   /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 40);
```

The fields in this message have the same meanings as in the individual flow desc request type (`OFPMP_FLOW_DESC` - see 7.3.5.2).

The `body` of the reply to a `OFPMP_FLOW_STATS` multipart request consists of an array of the following:

```
/* Body of reply to OFPMP_FLOW_STATS request
 * and body for OFPIT_STAT_TRIGGER generated status. */
struct ofp_flow_stats {
    uint16_t length;          /* Length of this entry. */
    uint8_t pad2[2];          /* Align to 64-bits. */
    uint8_t table_id;         /* ID of table flow came from. */
    uint8_t reason;           /* One of OFPFSR_*. */
    uint16_t priority;        /* Priority of the entry. */
    struct ofp_match match;   /* Description of fields. Variable size. */
    //struct ofp_stats stats; /* Statistics list. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 16);
```

 © 2015; The Open Networking Foundation

The fields in this message have the same meanings as in the individual flow desc reply type (OFPMP_FLOW_DESC - see 7.3.5.2).

The reason field is one of the following:

```
/* Reason for generating flow stats. */
enum ofp_flow_stats_reason {
    OFPFSR_STATS_REQUEST = 0,   /* Reply to a OFPMP_FLOW_STATS request. */
    OFPFSR_STAT_TRIGGER  = 1,   /* Status generated by OFPIT_STAT_TRIGGER. */
};
```

The reason value OFPRR_STATS_REQUEST means that the message is a reply to a OFPMP_FLOW_STATS multipart request. The reason value OFPRR_STAT_TRIGGER means that the message is an event (status) generated when one of the flow statistic threshold was crossed in a *stat trigger* instruction (see 7.2.5).

7.3.5.4 Aggregate Flow Statistics

Aggregate information about multiple flow entries is requested with the OFPMP_AGGREGATE_STATS multipart request type:

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

The fields in this message have the same meanings as in the individual flow desc request type (OFPMP_FLOW_DESC - see 7.3.5.2).

The body of the reply consists of the following:

```
/* Body of reply to OFPMP_AGGREGATE_STATS request. */
struct ofp_aggregate_stats_reply {
    struct ofp_stats stats;   /* Aggregated statistics list. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_reply) == 8);
```

 © 2015; The Open Networking Foundation

The stats field contains a list of OXS fields (see 7.2.4.2). The field OXS_OF_FLOW_COUNT is mandatory and is equal to the number of flow entries that matched the request. The fields OXS_OF_PACKET_COUNT and OXS_OF_BYTE_COUNT are optional and count all packets processed by all the flow entries matching the request.

7.3.5.5 Port Statistics

Information about port statistics is requested with the OFPMP_PORT_STATS multipart request type:

```
/* Body for ofp_multipart_request of types OFPMP_PORT_STATS and
 * OFPMP_PORT_DESC. */
struct ofp_port_multipart_request {
    uint32_t port_no;        /* OFPMP_PORT message must request statistics
                              * either for a single port (specified in
                              * port_no) or for all ports (if port_no ==
                              * OFPP_ANY). */
    uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_port_multipart_request) == 8);
```

The port_no field optionally filters the stats request to the given port. To request all port statistics, port_no must be set to OFPP_ANY.

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

```
/* Body of reply to OFPMP_PORT_STATS request. If a counter is unsupported,
 * set the field to all ones. */
struct ofp_port_stats {
    uint16_t length;         /* Length of this entry. */
    uint8_t pad[2];          /* Align to 64 bits. */
    uint32_t port_no;
    uint32_t duration_sec;   /* Time port has been alive in seconds. */
    uint32_t duration_nsec;  /* Time port has been alive in nanoseconds beyond
                                duration_sec. */
    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 in properties. */
    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.) */

    /* Port description property list - 0 or more properties */
    struct ofp_port_stats_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_port_stats) == 80);
```

 © 2015; The Open Networking Foundation

The `duration_sec` and `duration_nsec` fields indicate the elapsed time the port has been configured into the OpenFlow pipeline. The total duration in nanoseconds can be computed as $duration_sec \times 10^9 + duration_nsec$. Implementations are required to provide second precision; higher precision is encouraged where available.

The `rx_packets`, `tx_packets`, `rx_bytes` and `tx_bytes` are basic port counters measuring packets going through the port.

The `properties` field is a list of port statistics properties, which include port type specific statistics.

The list of port description property types that are currently defined are:

```
/* Port stats property types.
 */
enum ofp_port_stats_prop_type {
    OFPPSPT_ETHERNET          = 0,      /* Ethernet property. */
    OFPPSPT_OPTICAL           = 1,      /* Optical property. */
    OFPPSPT_EXPERIMENTER      = 0xFFFF, /* Experimenter property. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all port stats properties. */
struct ofp_port_stats_prop_header {
    uint16_t         type;    /* One of OFPPSPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats_prop_header) == 4);
```

The `OFPPSPT_ETHERNET` property uses the following structure and fields:

```
/* Ethernet port stats property. */
struct ofp_port_stats_prop_ethernet {
    uint16_t         type;    /* OFPPSPT_ETHERNET. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint8_t          pad[4];  /* Align to 64 bits. */

    uint64_t rx_frame_err;   /* Number of frame alignment errors. */
    uint64_t rx_over_err;    /* Number of packets with RX overrun. */
    uint64_t rx_crc_err;     /* Number of CRC errors. */
    uint64_t collisions;     /* Number of collisions. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats_prop_ethernet) == 40);
```

The `OFPPSPT_OPTICAL` property uses the following structure and fields:

```
/* Optical port stats property. */
struct ofp_port_stats_prop_optical {
    uint16_t         type;    /* OFPPSPT_OPTICAL. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint8_t          pad[4];  /* Align to 64 bits. */
```

 © 2015; The Open Networking Foundation

```
    uint32_t flags;          /* Features enabled by the port. */
    uint32_t tx_freq_lmda;   /* Current TX Frequency/Wavelength */
    uint32_t tx_offset;      /* TX Offset */
    uint32_t tx_grid_span;   /* TX Grid Spacing */
    uint32_t rx_freq_lmda;   /* Current RX Frequency/Wavelength */
    uint32_t rx_offset;      /* RX Offset */
    uint32_t rx_grid_span;   /* RX Grid Spacing */
    uint16_t tx_pwr;         /* Current TX power */
    uint16_t rx_pwr;         /* Current RX power */
    uint16_t bias_current;   /* TX Bias Current */
    uint16_t temperature;    /* TX Laser Temperature */
};
OFP_ASSERT(sizeof(struct ofp_port_stats_prop_optical) == 44);
```

The `flags` is a bitmap that indicates which of the stats values are valid and may include a combination of the following flags :

```
/* Flags is one of OFPOSF_ below */
enum ofp_port_stats_optical_flags  {
    OFPOSF_RX_TUNE   = 1 << 0,  /* Receiver tune info valid */
    OFPOSF_TX_TUNE   = 1 << 1,  /* Transmit tune info valid */
    OFPOSF_TX_PWR    = 1 << 2,  /* TX Power is valid */
    OFPOSF_RX_PWR    = 1 << 4,  /* RX power is valid */
    OFPOSF_TX_BIAS   = 1 << 5,  /* Transmit bias is valid */
    OFPOSF_TX_TEMP   = 1 << 6,  /* TX Temp is valid */
};
```

The port tuning values are represented as described by the Port Description (see 7.2.1). Frequency values are in MHz, wavelength (lambda) in nm * 100. The `tx_pwr` and `rx_pwr` are dBm * 10. The `bias_current` is mA * 10 and is useful as a laser efficiency reference. The laser temperature is degrees C * 10.

The `OFPPSPT_EXPERIMENTER` property uses the following structure and fields:

```
/* Experimenter port stats property. */
struct ofp_port_stats_prop_experimenter {
    uint16_t         type;    /* OFPPSPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_port_stats_prop_experimenter) == 12);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

 © 2015; The Open Networking Foundation

7.3.5.6 Port Description

The port description request `OFPMP_PORT_DESCRIPTION` enables the controller to get a description of all the standard ports of the OpenFlow switch (see 4.2). The request body consists of the following structure:

```
/* Body for ofp_multipart_request of types OFPMP_PORT_STATS and
 * OFPMP_PORT_DESC. */
struct ofp_port_multipart_request {
    uint32_t port_no;        /* OFPMP_PORT message must request statistics
                              * either for a single port (specified in
                              * port_no) or for all ports (if port_no ==
                              * OFPP_ANY). */
    uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_port_multipart_request) == 8);
```

The `port_no` field specifies a valid port on the switch or can be set to `OFPP_ANY` to refer to all ports on the switch.

The reply body consists of an array of the following:

```
/* Description of a port */
struct ofp_port {
    uint32_t port_no;
    uint16_t length;
    uint8_t pad[2];
    uint8_t hw_addr[OFP_ETH_ALEN];
    uint8_t pad2[2];                  /* Align to 64 bits. */
    char name[OFP_MAX_PORT_NAME_LEN]; /* Null-terminated */

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

    /* Port description property list - 0 or more properties */
    struct ofp_port_desc_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_port) == 40);
```

This structure is the common port structure describing ports (see section 7.2.1), and includes port number, port config and port status. All port properties relevant to the port type must be included in the port description. For example, a physical port of type Ethernet must include an Ethernet property in its description, whereas a logical port does not need to include it.

The port description reply must include all the standard ports defined in the OpenFlow switch or attached to it, regardless of their state or configuration. It is recommended that the list of standard ports should not change dynamically, but only as the result of a network topology configuration or a switch configuration, for example using the OpenFlow Configuration Protocol (see 4.6).

 © 2015; The Open Networking Foundation

7.3.5.7 Queue Statistics

The OFPMP_QUEUE_STATS multipart request message provides queue statistics for one or more ports and one or more queues. The request body contains a port_no field identifying the OpenFlow port for which statistics are requested, or OFPP_ANY to refer to all ports. The queue_id field identifies one of the priority queues, or OFPQ_ALL to refer to all queues configured at the specified port. OFPQ_ALL is 0xffffffff.

```
/* Body for ofp_multipart_request of types OFPMP_QUEUE_DESC and
 * OFPMP_QUEUE_STATS. */
struct ofp_queue_multipart_request {
    uint32_t port_no;        /* All ports if OFPP_ANY. */
    uint32_t queue_id;       /* All queues if OFPQ_ALL. */
};
OFP_ASSERT(sizeof(struct ofp_queue_multipart_request) == 8);
```

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

```
/* Body of reply to OFPMP_QUEUE_STATS request. */
struct ofp_queue_stats {
    uint16_t length;         /* Length of this entry. */
    uint8_t pad[6];          /* Align to 64 bits. */
    uint32_t port_no;        /* Port the queue is attached to. */
    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. */
    uint32_t duration_sec;   /* Time queue has been alive in seconds. */
    uint32_t duration_nsec;  /* Time queue has been alive in nanoseconds beyond
                                duration_sec. */

    struct ofp_queue_stats_prop_header properties[0]; /* List of properties. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats) == 48);
```

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

The properties field is a list of queue statistics properties.

The list of queue statistics property types that are currently defined are:

```
enum ofp_queue_stats_prop_type {
    OFPQSPT_EXPERIMENTER  = 0xffff  /* Experimenter defined property. */
};
```

A property definition contains the property type, length, and any associated data:

 © 2015; The Open Networking Foundation

```
/* Common header for all queue properties */
struct ofp_queue_stats_prop_header {
    uint16_t         type;    /* One of OFPQSPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats_prop_header) == 4);
```

The OFPQSPT_EXPERIMENTER properties uses the following structure and fields:

```
/* Experimenter queue property description. */
struct ofp_queue_stats_prop_experimenter {
    uint16_t         type;    /* OFPQSPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_queue_stats_prop_experimenter) == 12);
```

The experimenter field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

7.3.5.8 Queue Descriptions

An OpenFlow switch provides limited Quality-of-Service support (QoS) through a simple queuing mechanism.

A switch can optionally have one or more queues attached to a specific output port, and those queues can be used to schedule packets exiting the datapath on that output port. Each queue on the switch is uniquely identified by a *port number* and a *queue ID*. Two queues on different ports can have the same *queue ID*. Packets are directed to one of the queues based on the packet output port and the packet queue id, set using the *Output* action and *Set Queue* action respectively.

Packets mapped to a specific queue will be treated according to that queue's configuration (e.g. min rate). Queue processing happens logically after all pipeline processing. Packet scheduling using queues is not defined by this specification and is switch dependent, in particular no priority between *queue IDs* is assumed.

Queue configuration takes place outside the OpenFlow switch 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 OFPMP_QUEUE_DESC multipart request.

The OFPMP_QUEUE_DESC multipart request message provides queue descriptions for one or more ports and one or more queues. The request body contains a port_no field identifying the OpenFlow port for which descriptions are requested, or OFPP_ANY to refer to all ports. The queue_id field identifies one

 © 2015; The Open Networking Foundation

of the priority queues, or OFPQ_ALL to refer to all queues configured at the specified port. OFPQ_ALL is 0xffffffff.

```
/* Body for ofp_multipart_request of types OFPMP_QUEUE_DESC and
 * OFPMP_QUEUE_STATS. */
struct ofp_queue_multipart_request {
    uint32_t port_no;        /* All ports if OFPP_ANY. */
    uint32_t queue_id;       /* All queues if OFPQ_ALL. */
};
OFP_ASSERT(sizeof(struct ofp_queue_multipart_request) == 8);
```

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

```
/* Body of reply to OFPMP_QUEUE_DESC request. */
struct ofp_queue_desc {
    uint32_t port_no;      /* Port this queue is attached to. */
    uint32_t queue_id;     /* id for the specific queue. */
    uint16_t len;          /* Length in bytes of this queue desc. */
    uint8_t pad[6];        /* 64-bit alignment. */

    struct ofp_queue_desc_prop_header properties[0]; /* List of properties. */
};
OFP_ASSERT(sizeof(struct ofp_queue_desc) == 16);
```

The properties field is a list of queue description properties.

The list of queue description property types that are currently defined are:

```
enum ofp_queue_desc_prop_type {
    OFPQDPT_MIN_RATE      = 1,      /* Minimum datarate guaranteed. */
    OFPQDPT_MAX_RATE      = 2,      /* Maximum datarate. */
    OFPQDPT_EXPERIMENTER  = 0xffff  /* Experimenter defined property. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all queue properties */
struct ofp_queue_desc_prop_header {
    uint16_t         type;    /* One of OFPQDPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_queue_desc_prop_header) == 4);
```

The OFPQDPT_MIN_RATE properties uses the following structure and fields:

```
/* Min-Rate queue property description. */
struct ofp_queue_desc_prop_min_rate {
    uint16_t         type;    /* OFPQDPT_MIN_RATE. */
    uint16_t         length;  /* Length is 8. */
    uint16_t rate;        /* In 1/10 of a percent; >1000 -> disabled. */
    uint8_t pad[2];       /* 64-bit alignment */
};
OFP_ASSERT(sizeof(struct ofp_queue_desc_prop_min_rate) == 8);
```

 © 2015; The Open Networking Foundation

The rate field is the minimum rate guaranteed to the queue expressed as a fraction of the current speed of the output port the queue is attached to (see 7.2.1.2), encoded in 1/1000th increments. If rate is not configured, it is set to OFPQ_MIN_RATE_UNCFG, which is 0xffff.

The OFPQDPT_MAX_RATE properties uses the following structure and fields:

```
/* Max-Rate queue property description. */
struct ofp_queue_desc_prop_max_rate {
    uint16_t         type;    /* OFPQDPT_MAX_RATE. */
    uint16_t         length;  /* Length is 8. */
    uint16_t rate;        /* In 1/10 of a percent; >1000 -> disabled. */
    uint8_t pad[2];       /* 64-bit alignment */
};
OFP_ASSERT(sizeof(struct ofp_queue_desc_prop_max_rate) == 8);
```

The rate field is the maximum rate that the queue can use expressed as a fraction of the current speed of the output port the queue is attached to (see 7.2.1.2), encoded in 1/1000th increments. If rate is not configured, it is set to OFPQ_MAX_RATE_UNCFG, which is 0xffff.

The OFPQDPT_EXPERIMENTER properties uses the following structure and fields:

```
/* Experimenter queue property description. */
struct ofp_queue_desc_prop_experimenter {
    uint16_t         type;    /* OFPQDPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_queue_desc_prop_experimenter) == 12);
```

The experimenter field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

7.3.5.9 Group Statistics

The OFPMP_GROUP_STATS multipart request message provides statistics for one or more groups. The request body consists of the following structure:

```
/* Body of OFPMP_GROUP_STATS and OFPMP_GROUP_DESC requests. */
struct ofp_group_multipart_request {
    uint32_t group_id;       /* All groups if OFPG_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_group_multipart_request) == 8);
```

 © 2015; The Open Networking Foundation

The group_id field specifies a valid group on the switch or can be set to OFPG_ALL to refer to all groups on the switch.

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

```
/* Body of reply to OFPMP_GROUP_STATS request. */
struct ofp_group_stats {
    uint16_t length;         /* Length of this entry. */
    uint8_t pad[2];          /* Align to 64 bits. */
    uint32_t group_id;       /* Group identifier. */
    uint32_t ref_count;      /* Number of flows or groups that directly forward
                                to this group. */
    uint8_t pad2[4];         /* Align to 64 bits. */
    uint64_t packet_count;   /* Number of packets processed by group. */
    uint64_t byte_count;     /* Number of bytes processed by group. */
    uint32_t duration_sec;   /* Time group has been alive in seconds. */
    uint32_t duration_nsec;  /* Time group has been alive in nanoseconds beyond
                                duration_sec. */
    struct ofp_bucket_counter bucket_stats[0]; /* One counter set per bucket. */
};
OFP_ASSERT(sizeof(struct ofp_group_stats) == 40);
```

The fields include the group_id referencing the group, the ref_count field counting the number of flow entries or groups referencing directly the group, the packet_count, and the byte_count fields counting all packets processed by the group.

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

The bucket_stats field consists of an array of ofp_bucket_counter structs:

```
/* Used in group stats replies. */
struct ofp_bucket_counter {
    uint64_t packet_count;   /* Number of packets processed by bucket. */
    uint64_t byte_count;     /* Number of bytes processed by bucket. */
};
OFP_ASSERT(sizeof(struct ofp_bucket_counter) == 16);
```

7.3.5.10 Group Description

The OFPMP_GROUP_DESC multipart request message provides a way to list the set of groups on a switch, along with their corresponding bucket actions. The request body consists of the following structure:

```
/* Body of OFPMP_GROUP_STATS and OFPMP_GROUP_DESC requests. */
struct ofp_group_multipart_request {
    uint32_t group_id;       /* All groups if OFPG_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_group_multipart_request) == 8);
```

 © 2015; The Open Networking Foundation

The group_id field specifies a valid group on the switch or can be set to OFPG_ALL to refer to all groups on the switch.

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

```
/* Body of reply to OFPMP_GROUP_DESC request. */
struct ofp_group_desc {
    uint16_t length;              /* Length of this entry. */
    uint8_t type;                 /* One of OFPGT_*. */
    uint8_t pad;                  /* Pad to 64 bits. */
    uint32_t group_id;            /* Group identifier. */
    uint16_t bucket_array_len;    /* Length of action buckets data. */
    uint8_t pad2[6];              /* Pad to 64 bits. */
    /* Followed by:
     *   - Exactly 'bucket_array_len' bytes containing an array of
     *     struct ofp_bucket.
     *   - Zero or more bytes of group properties to fill out the overall
     *     length in the length field. */
    struct ofp_bucket buckets[0];   /* List of buckets - 0 or more. */
    //struct ofp_group_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_group_desc) == 16);
```

Fields for group description are the same as those used with the ofp_group_mod struct (see 7.3.4.3).

7.3.5.11 Group Features

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

```
/* Body of reply to OFPMP_GROUP_FEATURES request. Group features. */
struct ofp_group_features {
    uint32_t  types;           /* Bitmap of (1 << OFPGT_*) values supported. */
    uint32_t  capabilities;    /* Bitmap of OFPGFC_* capability supported. */
    uint32_t  max_groups[4];   /* Maximum number of groups for each type. */
    uint32_t  actions[4];      /* Bitmaps of (1 << OFPAT_*) values supported. */
};
OFP_ASSERT(sizeof(struct ofp_group_features) == 40);
```

The types field is a bitmap of group types supported by the switch. The bitmask uses the values from ofp_group_type as the number of bits to shift left for an associated group type. Experimenter types should *not* be reported via this bitmask. For example, OFPGT_ALL would use the mask 0x00000001.

The capabilities field is a bitmap that uses a combination of the following flags :

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

 © 2015; The Open Networking Foundation

The max_groups field is the maximum number of groups for each type of group.

The actions field is a set of bitmaps of actions supported by each group type. The first bitmap applies to the OFPGT_ALL group type. The bitmask uses the values from ofp_action_type as the number of bits to shift left for an associated action. Experimenter actions should *not* be reported via this bitmask. For example, OFPAT_OUTPUT would use the mask 0x00000001.

7.3.5.12 Meter Statistics

The OFPMP_METER_STATS statistics request message provides statistics for one or more meters. The request body consists of a meter_id field, which can be set to OFPM_ALL to refer to all meters on the switch.

```
/* Body of OFPMP_METER_STATS and OFPMP_METER_DESC requests. */
struct ofp_meter_multipart_request {
    uint32_t meter_id;       /* Meter instance, or OFPM_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_meter_multipart_request) == 8);
```

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

```
/* Body of reply to OFPMP_METER_STATS request. Meter statistics. */
struct ofp_meter_stats {
    uint32_t        meter_id;         /* Meter instance. */
    uint16_t        len;              /* Length in bytes of this stats. */
    uint8_t         pad[6];
    uint32_t        ref_count;        /* Number of flows or groups that
                                         directly reference this meter. */
    uint64_t        packet_in_count;  /* Number of packets in input. */
    uint64_t        byte_in_count;    /* Number of bytes in input. */
    uint32_t   duration_sec;  /* Time meter has been alive in seconds. */
    uint32_t   duration_nsec; /* Time meter has been alive in nanoseconds beyond
                                 duration_sec. */
    struct ofp_meter_band_stats band_stats[0]; /* The band_stats length is
                                         inferred from the length field. */
};
OFP_ASSERT(sizeof(struct ofp_meter_stats) == 40);
```

The packet_in_count and the byte_in_count fields count all packets processed by the meter. The ref_count field counts the number of flow entries or groups directly referencing the meter.

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

The band_stats field consists of an array of ofp_meter_band_stats structs:

 © 2015; The Open Networking Foundation

```
/* Statistics for each meter band */
struct ofp_meter_band_stats {
    uint64_t        packet_band_count;   /* Number of packets in band. */
    uint64_t        byte_band_count;     /* Number of bytes in band. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_stats) == 16);
```

The `packet_band_count` and `byte_band_count` fields count all packets processed by the band.

The order of the band statistics must be the same as in the `OFPMP_METER_DESC` multipart reply. The meter default band (rate 0) can not be included in that list.

7.3.5.13 Meter Descriptions

The `OFPMP_METER_DESC` multipart request message provides configuration for one or more meters. The request body consists of a `meter_id` field, which can be set to `OFPM_ALL` to refer to all meters on the switch.

```
/* Body of OFPMP_METER_STATS and OFPMP_METER_DESC requests. */
struct ofp_meter_multipart_request {
    uint32_t meter_id;       /* Meter instance, or OFPM_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_meter_multipart_request) == 8);
```

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

```
/* Body of reply to OFPMP_METER_DESC request. Meter configuration. */
struct ofp_meter_desc {
    uint16_t        length;           /* Length of this entry. */
    uint16_t        flags;            /* All OFPMF_* that apply. */
    uint32_t        meter_id;         /* Meter instance. */
    struct ofp_meter_band_header bands[0]; /* The bands length is
                                          inferred from the length field. */
};
OFP_ASSERT(sizeof(struct ofp_meter_desc) == 8);
```

The fields are the same fields used for configuring the meter (see 7.3.4.5). The meter default band (rate 0) can not be included in the list of bands for each meter.

If the `OFPMF_BURST` flag of the meter is unset, and if the meter uses internal burst values, the meter should set the `burst_size` field of each band with the corresponding internal burst value.

 © 2015; The Open Networking Foundation

7.3.5.14 Meter Features

The OFPMP_METER_FEATURES multipart request message provides the set of features of the metering subsystem. The request body is empty, and the body of the reply consists of the following structure:

```
/* Body of reply to OFPMP_METER_FEATURES request. Meter features. */
struct ofp_meter_features {
    uint32_t    max_meter;    /* Maximum number of meters. */
    uint32_t    band_types;   /* Bitmaps of (1 << OFPMBT_*) values supported. */
    uint32_t    capabilities; /* Bitmaps of "ofp_meter_flags". */
    uint8_t     max_bands;    /* Maximum bands per meters */
    uint8_t     max_color;    /* Maximum color value */
    uint8_t     pad[2];
    uint32_t    features;     /* Bitmaps of "ofp_meter_feature_flags". */
    uint8_t     pad2[4];
};
OFP_ASSERT(sizeof(struct ofp_meter_features) == 24);
```

The band_types field is a bitmap of band types supported by the switch, the switch may have other constraints on how band types may be combined in a specific meter. The bitmask uses the values from ofp_meter_band_type as the number of bits to shift left for an associated band type. Experimenter types should *not* be reported via this bitmask. For example, OFPMBT_DROP would use the mask 0x00000002.

The max_bands field is the maximum number of bands supported for each meter. If the switch only supports simple rate limiters, this value would be 1. If the switch also supports classical two rate policers, this value would be 2.

The capabilities field is a bitmap that uses a combination of the following flags:

```
/* Meter configuration flags */
enum ofp_meter_flags {
    OFPMF_KBPS        = 1 << 0,  /* Rate value in kb/s (kilo-bit per second). */
    OFPMF_PKTPS       = 1 << 1,  /* Rate value in packet/sec. */
    OFPMF_BURST       = 1 << 2,  /* Do burst size. */
    OFPMF_STATS       = 1 << 3,  /* Collect statistics. */
};
```

The features field is a bitmap that uses a combination of the following flags:

```
/* Meter feature flags */
enum ofp_meter_feature_flags {
    OFPMFF_ACTION_SET   = 1 << 0, /* Support meter action in action set. */
    OFPMFF_ANY_POSITION = 1 << 1, /* Support any position in action list. */
    OFPMFF_MULTI_LIST   = 1 << 2, /* Support multiple actions in action list. */
};
```

 © 2015; The Open Networking Foundation

7.3.5.15 Controller Status Multipart

The OFPMP_CONTROLLER_STATUS multipart message allows a controller to request the status, the roles and the control channels of other controllers configured on the switch. The request body is empty. The reply body consists of a list of controller status structures (7.2.7), one for every configured controller.

The reason field in every controller status structure must be set to OFPCSR_REQUEST in the multipart response.

7.3.5.16 Table Statistics

Information about tables is requested with the OFPMP_TABLE_STATS multipart 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 OFPMP_TABLE_STATS 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. */
    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) == 24);
```

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

7.3.5.17 Table Description

The OFPMP_TABLE_DESC multipart request message provides a way to list the current configuration of the tables on a switch, which is set using the OFPT_TABLE_MOD message. The request body is empty, while the reply body is an array of the following structure:

```
/* Body of reply to OFPMP_TABLE_DESC request. */
struct ofp_table_desc {
    uint16_t length;         /* Length is padded to 64 bits. */
    uint8_t table_id;        /* Identifier of table.  Lower numbered tables
                                are consulted first. */
    uint8_t pad[1];          /* Align to 32-bits. */
    uint32_t config;         /* Bitmap of OFPTC_* values. */

    /* Table Mod Property list - 0 or more. */
    struct ofp_table_mod_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_table_desc) == 8);
```

Fields for table description are the same as those used with the ofp_table_mod struct (see 7.3.4.1).

 © 2015; The Open Networking Foundation

7.3.5.18 Table Features

The OFPMP_TABLE_FEATURES multipart type allows a controller to both query for the capabilities of existing tables, and to optionally ask the switch to reconfigure its tables to match a supplied configuration. In general, the table feature capabilities represent all possible features of a table, however some features may be mutually exclusive and the current capabilities structures do not allow representing such exclusions.

7.3.5.18.1 Table Features request and reply

If the OFPMP_TABLE_FEATURES request body is *empty* the switch will return an array of ofp_table_features structures containing the capabilities of the currently configured flow tables. The flow tables and the pipeline are unchanged by this operation.

If the request body contains an array of one or more ofp_table_features structures, the switch will attempt to change its flow tables to match the requested flow table configuration change. Support for such requests is optional, and is discouraged when another protocol is used to configure tables, such as the OpenFlow Configuration Protocol. If those requests are not supported, the switch must return an ofp_error_msg with OFPET_BAD_REQUEST type and OFPBRC_BAD_LEN code. If those requests are disabled, the switch must return an ofp_error_msg with OFPET_TABLE_FEATURES_FAILED type and OFPTFFC_EPERM code.

A request containing a set of ofp_table_features structures can enable, disable, modify flow tables, or it can replace the full pipeline, depending on the *command* in the request. The following *commands* are supported :

```
/* Table Features request commands */
enum ofp_table_features_command {
    OFPTFC_REPLACE = 0,       /* Replace full pipeline. */
    OFPTFC_MODIFY  = 1,       /* Modify flow tables capabilities. */
    OFPTFC_ENABLE  = 2,       /* Enable flow tables in the pipeline. */
    OFPTFC_DISABLE = 3,       /* Disable flow tables in pipeline. */
};
```

The command OFPTFC_REPLACE configures the entire pipeline, and the set of flow tables in the pipeline must match the set in the request, or an error must be returned (see 7.5.4.14). The capability of all flow tables included in the request must be updated. In particular, if the requested configuration does not contain an ofp_table_features structure for one or more flow tables that the switch supports, these flow tables are to be removed from the pipeline if the configuration is successfully set.

The command OFPTFC_MODIFY configures only the set of flow tables specified in the request, other flow tables not included in the request are unchanged and their presence in the pipeline unchanged. The capability of the flow tables included in the request must be updated. This command can be used to modify the capabilities of a single flow table. If a flow table in the request was not included in the pipeline, it remains excluded.

The command OFPTFC_ENABLE inserts in the pipeline the flow tables specified in the request. The flow tables are inserted in the pipeline based in their *table-id* number. The capability of the flow tables included in the request are unchanged and must not be updated.

 © 2015; The Open Networking Foundation

The command OFPTFC_DISABLE removes from the pipeline the flow tables specified in the request. The capability of the flow tables included in the request are unchanged and must not be updated.

A successful configuration change will configure all flow tables in the request, that is, either all the flow tables specified in the request are modified or none. If the command modifies the features of flow tables, the new capabilities for each flow table must be either a superset of, or equal to the requested capabilities. If a table feature included in the request has an empty list of properties, the list of properties for that flow table is unchanged and only the other features of that flow table are updated. If the flow table configuration is successful, flow entries from flow tables that have been removed or flow tables that had their capabilities change between the prior and new configuration are removed from the flow table, however no ofp_flow_removed messages are sent. The switch then replies with the new configuration.

The switch must be able to set the requested configuration in its entirety, or return an error message (see 7.5.4.14). A switch may not support all commands listed above (see 7.5.4.14). For commands other than OFPTFC_REPLACE, the switch may support a limited number of flow tables in each request (see 7.5.4.14).

Requests and replies containing ofp_table_features are expected to meet the following minimum requirements:

- The table_id field value specified in each ofp_table_features structure should be unique amongst all ofp_table_features structures in the message.
- The properties field included in each ofp_table_features structure must either be empty or must contain exactly one of each of the ofp_table_feature_prop_type properties, with two exceptions. First, properties with the _MISS suffix may be omitted if it is the same as the corresponding property for regular flow entries. Second, properties of type OFPTFPT_EXPERIMENTER and OFPTFPT_EXPERIMENTER_MISS may be omitted or included many times. Ordering is unspecified, but implementers are encouraged to use the ordering listed in the specification (see 7.3.5.18.2).

A switch receiving a request that does not meet these requirements should return an error of type OFPET_TABLE_FEATURES_FAILED with the appropriate error code (see 7.5.4.14).

The following structure describes the body of the table features request and reply:

```
/* Body for ofp_multipart_request of type OFPMP_TABLE_FEATURES./
 * Body of reply to OFPMP_TABLE_FEATURES request. */
struct ofp_table_features {
    uint16_t length;         /* Length is padded to 64 bits. */
    uint8_t table_id;        /* Identifier of table.  Lower numbered tables
                                are consulted first. */
    uint8_t command;         /* One of OFPTFC_*. */
    uint32_t features;       /* Bitmap of OFPTFF_* values. */
    char name[OFP_MAX_TABLE_NAME_LEN];
    uint64_t metadata_match; /* Bits of metadata table can match. */
    uint64_t metadata_write; /* Bits of metadata table can write. */
    uint32_t capabilities;   /* Bitmap of OFPTC_* values. */
    uint32_t max_entries;    /* Max number of entries supported. */

    /* Table Feature Property list */
    struct ofp_table_feature_prop_header properties[0]; /* List of properties */
};
```

 © 2015; The Open Networking Foundation

```
OFP_ASSERT(sizeof(struct ofp_table_features) == 64);
```

The OFPMP_TABLE_FEATURES reply contains an array of one or more ofp_table_features structures, one structure for each flow table supported by the switch. The entries are always returned in the order that packets traverse the flow tables.

The command field is the configuration operation requested (see above). Only the command field from the first ofp_table_features entry is used, and it applies to the whole request. The command fields from subsequent ofp_table_features entries are ignored. In table features replies, this field is reserved and should be set to 0 (see 7.1.3).

The name field is a null-terminated string containing a human-readable name for the flow table. The value of OFP_MAX_TABLE_NAME_LEN is 32.

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

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

The capabilities field is the set of configuration flags that are supported in a flow table configuration message (see 7.3.4.1).

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

All fields in ofp_table_features may be requested to be changed by the controller with the exception of the max_entries field, this is read only and returned by the switch.

The features field is a bitmap that defines how the flow table can be used and what are its basic features. It may include a combination of the following flags :

```
/* Flags of features supported by the table. */
enum ofp_table_feature_flag {
    OFPTFF_INGRESS_TABLE     = 1 << 0, /* Can be configured as ingress table. */
    OFPTFF_EGRESS_TABLE      = 1 << 1, /* Can be configured as egress table. */
    OFPTFF_FIRST_EGRESS      = 1 << 4, /* Is the first egress table. */
};
```

 © 2015; The Open Networking Foundation

The flag OFPTFF_INGRESS_TABLE, if set, indicates that the flow table can be configured as an ingress flow table (see 5.12). Flow table 0 must have this flag set, i.e. flow table 0 can always be used as an ingress flow table. This flag usually can't be changed via a table features request.

The flag OFPTFF_EGRESS_TABLE, if set, indicates that the flow table can be configured as an egress flow table (see 5.12). Flow table 0 must have this flag unset, i.e. table 0 can never be used as an egress flow table. This flag usually can't be changed via a table feature request.

The flag OFPTFF_FIRST_EGRESS, if set, indicates that this flow table is the first egress table : when a a packet is output to a port, egress processing will start with this flow table (see 5.12). In a table feature reply, only a single flow table can have this flag set. This flag can be set on a flow table only if the flow table is capable of being an egress table, i.e. has the OFPTFF_EGRESS_TABLE flag set. When the flag OFPTFF_FIRST_EGRESS is set on a flow table to make it the first egress table, the flow table previously used as a first egress table, if any, must have its corresponding flag clear. If multiple flow tables have this flag set in a request, only the last one is set. If no flow table has the OFPTFF_FIRST_EGRESS flag set, no egress table is used : when a packet is output to a port, it is directly processed by the output port.

The properties field is a list of table feature properties, describing various capabilities of the flow table.

7.3.5.18.2 Table Features properties

The list of table feature property types that are currently defined are:

```
/* Table Feature property types.
 * Low order bit cleared indicates a property for a regular Flow Entry.
 * Low order bit set indicates a property for the Table-Miss Flow Entry.
 */
enum ofp_table_feature_prop_type {
    OFPTFPT_INSTRUCTIONS           = 0,  /* Instructions property. */
    OFPTFPT_INSTRUCTIONS_MISS      = 1,  /* Instructions for table-miss. */
    OFPTFPT_NEXT_TABLES            = 2,  /* Next Table property. */
    OFPTFPT_NEXT_TABLES_MISS       = 3,  /* Next Table for table-miss. */
    OFPTFPT_WRITE_ACTIONS          = 4,  /* Write Actions property. */
    OFPTFPT_WRITE_ACTIONS_MISS     = 5,  /* Write Actions for table-miss. */
    OFPTFPT_APPLY_ACTIONS          = 6,  /* Apply Actions property. */
    OFPTFPT_APPLY_ACTIONS_MISS     = 7,  /* Apply Actions for table-miss. */
    OFPTFPT_MATCH                  = 8,  /* Match property. */
    OFPTFPT_WILDCARDS              = 10, /* Wildcards property. */
    OFPTFPT_WRITE_SETFIELD         = 12, /* Write Set-Field property. */
    OFPTFPT_WRITE_SETFIELD_MISS    = 13, /* Write Set-Field for table-miss. */
    OFPTFPT_APPLY_SETFIELD         = 14, /* Apply Set-Field property. */
    OFPTFPT_APPLY_SETFIELD_MISS    = 15, /* Apply Set-Field for table-miss. */
    OFPTFPT_TABLE_SYNC_FROM        = 16, /* Table synchronisation property. */
    OFPTFPT_WRITE_COPYFIELD        = 18, /* Write Copy-Field property. */
    OFPTFPT_WRITE_COPYFIELD_MISS   = 19, /* Write Copy-Field for table-miss. */
    OFPTFPT_APPLY_COPYFIELD        = 20, /* Apply Copy-Field property. */
    OFPTFPT_APPLY_COPYFIELD_MISS   = 21, /* Apply Copy-Field for table-miss. */
    OFPTFPT_PACKET_TYPES           = 22, /* Packet types property. */
    OFPTFPT_EXPERIMENTER           = 0xFFFE, /* Experimenter property. */
    OFPTFPT_EXPERIMENTER_MISS      = 0xFFFF, /* Experimenter for table-miss. */
};
```

 © 2015; The Open Networking Foundation

The properties with the _MISS suffix describe the capabilities for the table-miss flow entry (see 5.4), whereas other properties describe the capabilities for regular flow entry. If a specific property does not have any capability (for example no Set-Field support), a property with an empty list must be included in the property list. When a property of the table-miss flow entry is the same as the corresponding property for regular flow entries (i.e. both properties have the same list of capabilities), this table-miss property can be omited from the property list.

A property definition contains the property type, length, and any associated data:

```
/* Common header for all Table Feature Properties */
struct ofp_table_feature_prop_header {
    uint16_t         type;    /* One of OFPTFPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_header) == 4);
```

The OFPTFPT_INSTRUCTIONS and OFPTFPT_INSTRUCTIONS_MISS properties use the following structure and fields:

```
/* Instructions property */
struct ofp_table_feature_prop_instructions {
    uint16_t         type;    /* One of OFPTFPT_INSTRUCTIONS,
                                 OFPTFPT_INSTRUCTIONS_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the instruction ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    struct ofp_instruction_id  instruction_ids[0];   /* List of instructions */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_instructions) == 4);
```

The instruction_ids field is the list of instructions supported by this table (see 5.5). The elements of that list are variable in size to enable expressing experimenter instructions. Non-experimenter instructions are 4 bytes, and they use the following structure:

```
/* Instruction ID */
struct ofp_instruction_id {
    uint16_t type;                /* One of OFPIT_*. */
    uint16_t len;                 /* Length is 4 or experimenter defined. */
    uint8_t exp_data[0];          /* Optional experimenter id + data. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_id) == 4);
```

The OFPTFPT_NEXT_TABLES, OFPTFPT_NEXT_TABLES_MISS and OFPTFPT_TABLE_SYNC_FROM properties use the following structure and fields:

 © 2015; The Open Networking Foundation

```
/* Next Tables and Table Synchronise From properties */
struct ofp_table_feature_prop_tables {
    uint16_t         type;    /* One of OFPTFPT_NEXT_TABLES,
                                 OFPTFPT_NEXT_TABLES_MISS,
                                 OFPTFPT_TABLE_SYNC_FROM. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the table_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint8_t          table_ids[0];        /* List of table ids. */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_tables) == 4);
```

For OFPTFPT_NEXT_TABLES and OFPTFPT_NEXT_TABLES_MISS, the table_ids field is the array of tables that can be directly reached from the present table using the OFPIT_GOTO_TABLE instruction (see 5.1).

For OFPTFPT_TABLE_SYNC_FROM, the table_ids field is the array of tables the present table is synchronising content from (see 6.6). When a flow entry is added, modified or removed in one of the flow tables listed in the array, a corresponding flow entry is automatically added, modified or removed in the present flow table.

The OFPTFPT_WRITE_ACTIONS, OFPTFPT_WRITE_ACTIONS_MISS, OFPTFPT_APPLY_ACTIONS and OFPTFPT_APPLY_ACTIONS_MISS properties use the following structure and fields:

```
/* Actions property */
struct ofp_table_feature_prop_actions {
    uint16_t         type;    /* One of OFPTFPT_WRITE_ACTIONS,
                                 OFPTFPT_WRITE_ACTIONS_MISS,
                                 OFPTFPT_APPLY_ACTIONS,
                                 OFPTFPT_APPLY_ACTIONS_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the action_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    struct ofp_action_id  action_ids[0];      /* List of actions */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_actions) == 4);
```

The OFPTFPT_WRITE_ACTIONS and OFPTFPT_WRITE_ACTIONS_MISS properties describe actions supported by the table using the OFPIT_WRITE_ACTIONS instruction, whereas the OFPTFPT_APPLY_ACTIONS and OFPTFPT_APPLY_ACTIONS_MISS properties describe actions supported by the table using the OFPIT_APPLY_ACTIONS instruction.

The action_ids field represents all the actions supported by the feature (see 5.8). The elements of that list are variable in size to enable expressing experimenter actions, non-experimenter actions are 4 bytes, and they use the following structure:

```
/* Action ID */
struct ofp_action_id {
    uint16_t type;               /* One of OFPAT_*. */
    uint16_t len;                /* Length is 4 or experimenter defined. */
```

 © 2015; The Open Networking Foundation

```
    uint8_t exp_data[0];          /* Optional experimenter id + data. */
};
OFP_ASSERT(sizeof(struct ofp_action_id) == 4);
```

The OFPTFPT_MATCH, OFPTFPT_WILDCARDS, OFPTFPT_WRITE_SETFIELD, OFPTFPT_WRITE_SETFIELD_MISS, OFPTFPT_APPLY_SETFIELD, OFPTFPT_APPLY_SETFIELD_MISS, OFPTFPT_WRITE_COPYFIELD, OFPTFPT_WRITE_COPYFIELD_MISS, OFPTFPT_APPLY_COPYFIELD and OFPTFPT_APPLY_COPYFIELD_MISS properties use the following structure and fields:

```
/* Match, Wildcard or Set-Field property */
struct ofp_table_feature_prop_oxm {
    uint16_t         type;    /* One of OFPTFPT_MATCH,
                                 OFPTFPT_WILDCARDS,
                                 OFPTFPT_WRITE_SETFIELD,
                                 OFPTFPT_WRITE_SETFIELD_MISS,
                                 OFPTFPT_APPLY_SETFIELD,
                                 OFPTFPT_APPLY_SETFIELD_MISS,
                                 OFPTFPT_WRITE_COPYFIELD,
                                 OFPTFPT_WRITE_COPYFIELD_MISS,
                                 OFPTFPT_APPLY_COPYFIELD,
                                 OFPTFPT_APPLY_COPYFIELD_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the oxm_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         oxm_ids[0];   /* Array of OXM headers */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_oxm) == 4);
```

The oxm_ids field is the list of OXM types for the feature (see 7.2.3.2). The elements of that list are 32-bit OXM headers for non-experimenter OXM fields or 64-bit OXM headers for experimenter OXM fields, those OXM fields don't include any payload. The oxm_length field in OXM headers must be the length value defined for the OXM field, i.e. the payload length if the OXM field had a payload. For experimenter OXM fields with variable payload size, the oxm_length field must be the maximum length of the payload. In some cases, only the packet register with the highest type field needs to be included in the list (see 7.2.3.10).

The OFPTFPT_MATCH property indicates the fields for which that particular table supports matching on (see 7.2.3.7). For example, if the table can match the ingress port, an OXM header with the type OXM_OF_IN_PORT should be included in the list. If the HASMASK bit is set on the OXM header then the switch must support masking for the given type. Fields that are not listed in this property can not be used in the table as a match field, unless they are used as a prerequisite for another field. Fields that can only be used in the table as prerequisite and can't match other values must not be included in this property (see 7.2.3.6). For example, if a table matches IPv4 addresses for IP packets and can't match arbitrary Ethertypes, the table must accept the prerequisite of Ethertype equal to IPv4 in the match, however the Ethertype field must not be listed in this property. On the other hand, if the table matches IPv4 addresses for IP packets and can match arbitrary Ethertypes, the Ethertype field must be listed in this property.

 © 2015; The Open Networking Foundation

The OFPTFPT_WILDCARDS property indicates the fields for which that particular table supports wildcarding (omitting) in the match when their prerequisite can be met. This property must be a strict subset of the OFPTFPT_MATCH property. For example, a direct look-up hash table would have that list empty, while a TCAM or sequentially searched table would have it set to the same value as the OFPTFPT_MATCH property. If a field can be omitted only when one of its prerequisites is invalid (a prerequisite field has a value different from its required value), and must be always present when its prerequisites can be met, it must not be included in the OFPTFPT_WILDCARDS property. For example, if TCP ports can be omitted when the IP protocol is TCP, they would be included in this property. On the other hand if the TCP ports need to be present every time the IP protocol is TCP and are omitted only when the IP protocol is different from TCP (such as UDP), they would not be included in this property.

The OFPTFPT_WRITE_SETFIELD and OFPTFPT_WRITE_SETFIELD_MISS properties describe *Set-Field* action types supported by the table using the OFPIT_WRITE_ACTIONS instruction, whereas the OFPTFPT_APPLY_SETFIELD and OFPTFPT_APPLY_SETFIELD_MISS properties describe *Set-Field* action types supported by the table using the OFPIT_APPLY_ACTIONS instruction (see 7.2.6.7). If the HASMASK bit is set in the OXM header, then the switch must support partial rewrite for the given type (see 7.2.6.7).

The OFPTFPT_WRITE_COPYFIELD and OFPTFPT_WRITE_COPYFIELD_MISS properties describe *Copy-Field* action types supported by the table using the OFPIT_WRITE_ACTIONS instruction, whereas the OFPTFPT_APPLY_COPYFIELD and OFPTFPT_APPLY_COPYFIELD_MISS properties describe *Copy-Field* action types supported by the table using the OFPIT_APPLY_ACTIONS instruction (see 7.2.6.8).

The OFPTFPT_APPLY_ACTIONS, OFPTFPT_APPLY_ACTIONS_MISS, OFPTFPT_APPLY_SETFIELD, and OFPTFPT_APPLY_SETFIELD_MISS properties contain actions and fields the table is capable of applying. For each of these lists, if an element is present it means the table is at least capable of applying the element in isolation one time. There is currently no way to indicate which elements can be applied together, in which order, and how many times an element can be applied in a single flow entry.

The OFPTFPT_PACKET_TYPES property uses the following structure and fields:

```
/* Packet types property */
struct ofp_table_feature_prop_oxm_values {
    uint16_t         type;    /* OFPTFPT_PACKET_TYPES. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the oxm values, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         oxm_values[0];   /* Array of OXM values */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_oxm_values) == 4);
```

The oxm_values is the list of OXM values for the feature (see 7.2.3.2). The elements of that list are complete OXM fields including OXM header and payload.

The OFPTFPT_PACKET_TYPES property contains the set of *Packet Types Match Field* values supported by the table (see 7.2.3.11), defining the packet types supported by the table. For each packet type supported by the table, it either includes one OXM_OF_PACKET_TYPE field (64 bit long) or one experimenter OXM field. This property must be included if the table matches packet types other than Ethernet packets.

 © 2015; The Open Networking Foundation

This version of the specification supports only a single Packet Type per switch, having multiple Packet Types in this property is outside the scope of the specification.

The OFPTFPT_EXPERIMENTER and OFPTFPT_EXPERIMENTER_MISS properties use the following structure and fields:

```
/* Experimenter table feature property */
struct ofp_table_feature_prop_experimenter {
    uint16_t         type;    /* One of OFPTFPT_EXPERIMENTER,
                                 OFPTFPT_EXPERIMENTER_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_experimenter) == 12);
```

The experimenter field is the Experimenter ID and the exp_type field is the experimenter type, they take the same form as in the typical experimenter structure (see 7.2.8). The experimenter_data field is experimenter defined, the whole property needs to be padded to 64 bits.

7.3.5.19 Flow Monitoring

The OFPMP_FLOW_MONITOR multipart type allows a controller to manage flow monitors that keep track of changes to the flow tables. In a multi-controller deployment, this enables a controller to be aware of changes made to the flow tables by other controllers. A controller can create a number of flow monitors, each flow monitor matching a subset of flow entries in some flow tables. Flow monitors will generate events for any change to the flow tables matching one of the monitors.

Flow monitoring is an optional feature and may not be supported by all OpenFlow switches. The number of flow monitors that can be created on a switch may also be limited.

7.3.5.19.1 Flow Monitoring Request

Flow monitor configuration is done with a OFPMP_FLOW_MONITOR multipart request. This request contains an array of ofp_flow_monitor_request structures, and may span multiple multipart messages. The switch may have limited buffer and may accept only a limited number of monitors per request (see 7.3.5).

The body of the OFPMP_FLOW_MONITOR request consists of an array of the following structure:

```
/* Body for ofp_multipart_request of type OFPMP_FLOW_MONITOR.
 *
 * The OFPMP_FLOW_MONITOR request's body consists of an array of zero or more
 * instances of this structure.  The request arranges to monitor the flows
```

 © 2015; The Open Networking Foundation

```
 * that match the specified criteria, which are interpreted in the same way as
 * for OFPMP_FLOW.
 *
 * 'id' identifies a particular monitor for the purpose of allowing it to be
 * canceled later with OFPFMC_DELETE.  'id' must be unique among
 * existing monitors that have not already been canceled.
 */
struct ofp_flow_monitor_request {
    uint32_t monitor_id;        /* Controller-assigned ID for this monitor. */
    uint32_t out_port;          /* Required output port, if not OFPP_ANY. */
    uint32_t out_group;         /* Required group number, if not OFPG_ANY. */
    uint16_t flags;             /* OFPFMF_*. */
    uint8_t table_id;           /* One table's ID or OFPTT_ALL (all tables). */
    uint8_t command;            /* One of OFPFMC_*. */
    struct ofp_match match;     /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_monitor_request) == 24);
```

The `monitor_id` field uniquely identifies a monitor for a specific controller connection within a switch. The same `monitor_id` can be reused for different monitors by two different controller connections on the same switch

The `command` field defines what operation must be done on that monitor and must be one of the following:

```
/* Flow monitor commands */
enum ofp_flow_monitor_command {
    OFPFMC_ADD    = 0,       /* New flow monitor. */
    OFPFMC_MODIFY = 1,       /* Modify existing flow monitor. */
    OFPFMC_DELETE = 2,       /* Delete/cancel existing flow monitor. */
};
```

The command `OFPFMC_ADD` creates a new flow monitor and begins to monitor flow entries based on the request parameters. The command `OFPFMC_ADD` must return an error if the `monitor_id` specified is already in use (see 7.5.4.17). The command `OFPFMC_DELETE` destroys the flow monitor specified by `monitor_id` and stops the associated flow monitoring and flow monitor events. The command `OFPFMC_MODIFY` is equivalent to deleting and creating a flow monitor.

The `table_id` field specifies the flow table to monitor. If this field is set to `OFPTT_ALL`, all tables are monitored.

The `match` field contains a description of the flow entries that should be monitored and may contain wildcarded and masked fields. This field's matching behavior is described in Section 6.4.

The `out_port` and `out_group` fields optionally reduce the scope of the flow monitor by output port. If either `out_port` or `out_group` contains a value other than `OFPP_ANY` or `OFPG_ANY` respectively, it introduces an additional constraint when matching flow entries to monitors. This constraint is that the flow entry must contain an output action directed at that port or that group before or after the change triggering the flow monitor event.

The `flags` field is a bitmap that only applies to flow monitor creation and may include a combination of the following flags :

 © 2015; The Open Networking Foundation

```
/* 'flags' bits in struct of_flow_monitor_request. */
enum ofp_flow_monitor_flags {
    /* When to send updates. */
    OFPFMF_INITIAL = 1 << 0,     /* Initially matching flows. */
    OFPFMF_ADD = 1 << 1,         /* New matching flows as they are added. */
    OFPFMF_REMOVED = 1 << 2,     /* Old matching flows as they are removed. */
    OFPFMF_MODIFY = 1 << 3,      /* Matching flows as they are changed. */

    /* What to include in updates. */
    OFPFMF_INSTRUCTIONS = 1 << 4,/* If set, instructions are included. */
    OFPFMF_NO_ABBREV = 1 << 5,   /* If set, include own changes in full. */
    OFPFMF_ONLY_OWN = 1 << 6,    /* If set, don't include other controllers. */
};
```

When the OFPFMF_INITIAL flag is set, the switch must include, in the reply to the flow monitor request, all flow entries matching the new monitor at the time of the request.

When the OFPFMF_ADD flag is set, the flow monitor must match all additions of flow entries, for example flow mod with OFPFC_ADD command. When the OFPFMF_REMOVED flag is set, the flow monitor must match all removal of flow entries, for example flow mod with OFPFC_DELETE or OFPFC_DELETE_STICT commands, flow expiry due to timeout, evictions, or group or meter deletions. When the OFPFMF_MODIFY flag is set, the flow monitor must match all modifications of flow entries, for example flow mod requests with OFPFC_MODIFY or OFPFC_MODIFY_STICT commands.

Flow updates of type OFPFMF_REMOVED and flow removed messages (see 7.4.2) are independent notification mechanisms. Flow updates do not take into account the value of the OFPFF_SEND_FLOW_REM flag on the flow entry. A single flow entry may be reported by both notification mechanisms.

When the OFPFMF_INSTRUCTIONS flag is set, the switch must include the flow entry instructions in the flow monitor events. If this flag is unset, the switch must omit the flow entry instructions in the flow monitor events.

When the OFPFMF_NO_ABBREV flag is set, the switch must not use abbreviated flow updates in events. If this flag is unset, the switch must use abbreviated flow updates in events related to flow table changes done by the controller that created the flow monitor.

When the OFPFMF_ONLY_OWN flag is set, the switch must generate updates only for flow table changes done by the controller that created the flow monitor. If this flag is unset, the switch must generate updates for flow table changes done by all controllers.

7.3.5.19.2 Flow Monitoring Reply

When the switch receives an OFPMP_FLOW_MONITOR multipart request, it replies to it using an OFPMP_FLOW_MONITOR multipart reply, the transaction id (xid) of this reply must be the same as the request. All flow entries that match one flow monitor of the request with the flag OFPFMF_INITIAL set must be included in the reply as a flow update (with event type OFPFME_INITIAL). A unified multipart reply is generated for all the monitors included in the request.

The OFPMP_FLOW_MONITOR multipart reply is also used as an asynchronous message. At any time, whenever a change to a flow table matches some outstanding flow monitor, the switch must send a notification to the controller using an OFPMP_FLOW_MONITOR multipart reply, the transaction id (xid) of this reply must be 0. All flow entries that match one outstanding flow monitor must be included in the

 © 2015; The Open Networking Foundation

reply. For example, if a flow_mod request modifies multiple flow entries that match some flow monitors, the notification must include one flow update for each flow entry matching a flow monitor.

If a modified flow entry matches multiple flow monitor, it must be included only once in the notification. The flow update must include the aggregate of all information specified by the matched flow monitors. For example, if any of the matching flow monitors specify instructions (OFPFMF_INSTRUCTIONS), the flow update must include instructions, and if any of the matching flow monitors specify full changes for the controller's own changes (OFPFMF_NO_ABBREV), the controller's own changes will be included in full.

A flow_mod message that does not change the flow table will not trigger any notification, even an abbreviated one. For example, a "modify" or "delete" flow_mod message that does not match any flow entries will not trigger a notification. An "add" or "modify" flow_mod message may specifies all the same parameters that a flow entry already has, whether this condition triggers a notification is unspecified and subject to change in future versions of this specification.

An OFPMP_FLOW_MONITOR multipart reply can not cross a barrier handshake (see 7.3.7). The switch must always send the OFPMP_FLOW_MONITOR multipart reply for a given flow table change before the reply to an OFPT_BARRIER_REQUEST request that follows the request responsible for the flow table change.

The body of any OFPMP_FLOW_MONITOR multipart reply consists of an array of flow updates. All flow updates contains the update type and length :

```
/* OFPMP_FLOW_MONITOR reply header.
 *
 * The body of an OFPMP_FLOW_MONITOR reply is an array of variable-length
 * structures, each of which begins with this header.  The 'length' member may
 * be used to traverse the array, and the 'event' member may be used to
 * determine the particular structure.
 *
 * Every instance is a multiple of 8 bytes long. */
struct ofp_flow_update_header {
    uint16_t length;            /* Length of this entry. */
    uint16_t event;             /* One of OFPFME_*. */
    /* ...other data depending on 'event'... */
};
OFP_ASSERT(sizeof(struct ofp_flow_update_header) == 4);
```

The event field is the flow update type. The list of flow update types that are currently defined is :

```
/* 'event' values in struct ofp_flow_update_header. */
enum ofp_flow_update_event {
    /* struct ofp_flow_update_full. */
    OFPFME_INITIAL = 0,          /* Flow present when flow monitor created. */
    OFPFME_ADDED = 1,            /* Flow was added. */
    OFPFME_REMOVED = 2,          /* Flow was removed. */
    OFPFME_MODIFIED = 3,         /* Flow instructions were changed. */

    /* struct ofp_flow_update_abbrev. */
    OFPFME_ABBREV = 4,           /* Abbreviated reply. */

    /* struct ofp_flow_update_header. */
    OFPFME_PAUSED = 5,          /* Monitoring paused (out of buffer space). */
```

 © 2015; The Open Networking Foundation

```
    OFPFME_RESUMED = 6,         /* Monitoring resumed. */
};
```

The OFPFME_ADDED, OFPFME_REMOVED and OFPFME_MODIFIED types denote flow entries matching a flow monitor that have been respectively added, removed or modified in the flow tables and matching an active flow monitor. The OFPFME_INITIAL denotes flow updates part of the initial reply for flow monitors with the flag OFPFMF_INITIAL set. They use the following structure and fields:

```
/* OFPMP_FLOW_MONITOR reply for OFPFME_INITIAL, OFPFME_ADDED, OFPFME_REMOVED,
 * and OFPFME_MODIFIED. */
struct ofp_flow_update_full {
    uint16_t length;            /* Length is 32 + match + instructions. */
    uint16_t event;             /* One of OFPFME_*. */
    uint8_t table_id;           /* ID of flow's table. */
    uint8_t reason;             /* OFPRR_* for OFPFME_REMOVED, else zero. */
    uint16_t idle_timeout;      /* Number of seconds idle before expiration. */
    uint16_t hard_timeout;      /* Number of seconds before expiration. */
    uint16_t priority;          /* Priority of the entry. */
    uint8_t zeros[4];           /* Reserved, currently zeroed. */
    uint64_t cookie;            /* Opaque controller-issued identifier. */
    struct ofp_match match;     /* Fields to match. Variable size. */
    /*     Instruction set.
     *     If OFPFMF_INSTRUCTIONS was not specified, or 'event' is
     *     OFPFME_REMOVED, no instructions are included.
     */
    //struct ofp_instruction instructions[0];
};
OFP_ASSERT(sizeof(struct ofp_flow_update_full) == 32);
```

The fields consist of those provided in the flow_mod that created the flow entry (see 7.3.4.2). For updates of type OFPFME_REMOVED, the reason field is the removal reason (see 7.4.2), for other types this field must be zero.

The OFPFME_ABBREV type denotes an abreviated flow update. When the controller does not specify OFPFMF_NO_ABBREV in a monitor request, any flow tables changes due to the controller's own requests (on the same OpenFlow channel) will be abbreviated, when possible. It uses the following structure and fields:

```
/* OFPMP_FLOW_MONITOR reply for OFPFME_ABBREV.
 *
 * When the controller does not specify OFPFMF_NO_ABBREV in a monitor request,
 * any flow tables changes due to the controller's own requests (on the same
 * OpenFlow channel) will be abbreviated, when possible, to this form, which
 * simply specifies the 'xid' of the OpenFlow request (e.g. an OFPT_FLOW_MOD)
 * that caused the change.
 * Some changes cannot be abbreviated and will be sent in full.
 */
struct ofp_flow_update_abbrev {
    uint16_t length;            /* Length is 8. */
    uint16_t event;             /* OFPFME_ABBREV. */
    uint32_t xid;               /* Controller-specified xid from flow_mod. */
};
OFP_ASSERT(sizeof(struct ofp_flow_update_abbrev) == 8);
```

 © 2015; The Open Networking Foundation

The `xid` field is the transaction id of the controller request this update relates to.

The `OFPFME_PAUSED` and `OFPFME_RESUMED` types denote monitoring pause and resume due to buffer management constraints. They use the following structure and fields:

```
/* OFPMP_FLOW_MONITOR reply for OFPFME_PAUSED and OFPFME_RESUMED.
 */
struct ofp_flow_update_paused {
    uint16_t length;            /* Length is 8. */
    uint16_t event;             /* One of OFPFME_*. */
    uint8_t zeros[4];           /* Reserved, currently zeroed. */
};
OFP_ASSERT(sizeof(struct ofp_flow_update_paused) == 8);
```

7.3.5.19.3 Flow Monitoring Pause

OpenFlow messages for flow monitor notifications can overflow the buffer space available to the switch, either temporarily (e.g. due to network conditions slowing OpenFlow traffic) or more permanently (e.g. the sustained rate of flow table change exceeds the network bandwidth between switch and controller). The flow monitoring pause mechanism disables flow updates when the number of queued flow updates reaches a limit, and enables the switch and controller to recover gracefully. The pause mechanism guarantees that the maximum buffer space requirement for flow updates is bounded by the limit plus the maximum number of supported flow entries.

When the switch's notification buffer space reaches a switch-defined limiting threshold, the switch must react as follows:

1. The switch must send a `OFPFME_PAUSED` flow update to the controller, following all the already queued flow updates. After it receives this message, the controller knows that its view of the flow table, as represented by flow monitor notifications, is incomplete.
2. As long as the notification buffer is not empty:
   - `OFPFMF_ADD` and `OFPFMF_MODIFY` flow updates must not be sent.
   - `OFPFMF_REMOVED` flow updates must still be sent, but only for flow entries that existed before the switch sent the `OFPFME_PAUSED` flow update.
   - `OFPFMF_ABBREV` flow updates must follow the same rules, in particular flow updates related to adding or modifying flow entries must not be sent.
3. When the notification buffer empties, the switch must send `OFPFME_ADDED` flow updates for flow entries added since the buffer reached its limit and still present in a flow table, and it must send `OFPFME_MODIFIED` flow updates for flow entries that existed before the limit was reached, changed after the limit was reached and still present in a flow table.
4. The switch must send an `OFPFME_RESUMED` flow update to the controller. After it receives this message, the controller knows that its view of the flow table, as represented by flow monitor notifications, is again complete.

 © 2015; The Open Networking Foundation

7.3.5.20 Bundle Features Multipart

The bundle features request, using the OFPMP_BUNDLE_FEATURES message type, allows a controller to query a switch about its bundle capabilities, including whether it supports atomic bundles, ordered bundles, and scheduled bundles.

7.3.5.20.1 Bundle Features Request Message Format

The body of the bundle features request message is defined by struct ofp_bundle_features_request, as follows:

```
/* Body of OFPMP_BUNDLE_FEATURES request. */
struct ofp_bundle_features_request {
    uint32_t feature_request_flags;  /* Bitmap of "ofp_bundle_feature_flags". */
    uint8_t pad[4];

    /* Bundle features property list - 0 or more. */
    struct ofp_bundle_features_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_bundle_features_request) == 8);
```

The body consists of a flags field, followed by zero or more property TLV fields. The bundle features properties are specified in Section 7.3.5.20.3. The flags field, feature_request_flags, is defined as follows:

```
enum ofp_bundle_feature_flags {
    OFPBF_TIMESTAMP = 1 << 0,       /* Request includes a timestamp. */
    OFPBF_TIME_SET_SCHED = 1 << 1,  /* Request includes the sched_max_future and
                                     * sched_max_past parameters. */
};
```

- OFPBF_TIMESTAMP indicates that the current request includes a timestamp, i.e., the current time according to the controller's clock. The timestamp is incorporated in the ofp_bundle_features_prop_time property.
- OFPBF_TIME_SET_SCHED indicates that the request includes the scheduling tolerance parameters, sched_max_future and sched_max_past, and that the switch should update its scheduling tolerance according to the received values.

If at least one of the flags OFPBF_TIMESTAMP or OFPBF_TIME_SET_SCHED is set, the bundle features request includes a ofp_bundle_features_prop_time property.

7.3.5.20.2 Bundle Features Reply Message Format

A switch that receives a bundle features request and processes it successfully sends a bundle features reply to the controller. The bundle features reply is an OFPT_MULTIPART_REPLY message with a type OFPMP_BUNDLE_FEATURES. The body of the bundle features reply message is struct ofp_bundle_features, as follows:

 © 2015; The Open Networking Foundation

```
/* Body of reply to OFPMP_BUNDLE_FEATURES request. */
struct ofp_bundle_features {
    uint16_t capabilities; /* Bitmap of "ofp_bundle_flags". */
    uint8_t pad[6];

    /* Bundle features property list - 0 or more. */
    struct ofp_bundle_features_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_bundle_features) == 8);
```

The `capabilities` field indicates the bundle-related features that are supported by the switch. The format of the `capabilities` field is defined by `struct ofp_bundle_flags`, as follows:

```
/* Bundle configuration flags. */
enum ofp_bundle_flags {
    OFPBF_ATOMIC  = 1 << 0,  /* Execute atomically. */
    OFPBF_ORDERED = 1 << 1,  /* Execute in specified order. */
    OFPBF_TIME    = 1 << 2,  /* Execute in specified time. */
};
```

- `OFPBF_ATOMIC` and `OFPBF_ORDERED` indicate that the switch supports atomic bundles, and ordered bundles, respectively (see Section 6.9.4).
- `OFPBF_TIME` indicates that the switch supports scheduled bundles, as described in Section 6.9.6. When this flag is set, the bundle features reply message includes the `ofp_bundle_features_prop_time` property.

The bundle features reply message may include zero or more properties. The format of the bundle features properties is defined below.

If a switch receives a bundle features request and is unable to process it, it replies with an error message of type `OFPET_BAD_REQUEST` (see 7.5.4.2). The code `OFPBRC_MULTIPART_BAD_SCHED` indicates that the request had the `OFPBF_TIME_SET_SCHED` flag enabled, and the switch failed to update the scheduling tolerance values.

7.3.5.20.3 Bundle Features Properties

Bundle features properties are optional TLV fields with a common header format, as follows:

```
struct ofp_bundle_features_prop_header {
    uint16_t type;          /* One of OFPTMPBF_*. */
    uint16_t length;        /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_features_prop_header) == 4);
```

The currently defined types are as follows:

```
/* Bundle features property types. */
enum ofp_bundle_features_prop_type {
    OFPTMPBF_TIME_CAPABILITY = 0x1, /* Time feature property. */
    OFPTMPBF_EXPERIMENTER = 0xFFFF, /* Experimenter property. */
};
```

 © 2015; The Open Networking Foundation

7.3.5.20.4 The Bundle Features Time Property

A bundle feature request in which at least one of the flags OFPBF_TIMESTAMP or OFPBF_TIME_SET_SCHED is set, incorporates the ofp_bundle_features_prop_time property. A bundle feature reply that has the OFPBF_TIME flag set incorporates the ofp_bundle_features_prop_time property.

The ofp_bundle_features_prop_time property is defined as follows:

```
/* Bundle time features. */
struct ofp_bundle_features_prop_time {
    uint16_t type;                    /* OFPTMPBF_TIME_CAPABILITY. */
    uint16_t length;                  /* Length in bytes of this property. */
    uint8_t pad[4];
    struct ofp_time sched_accuracy;   /* The scheduling accuracy, i.e., how
                                       * accurately the switch can perform a
                                       * scheduled commit. This field is used
                                       * only in bundle features replies, and
                                       * is ignored in bundle features
                                       * requests. */
    struct ofp_time sched_max_future; /* The maximal difference between the
                                       * scheduling time and the current time.*/
    struct ofp_time sched_max_past;   /* If the scheduling time occurs in the
                                       * past, defines the maximal difference
                                       * between the current time and the
                                       * scheduling time.*/
    struct ofp_time timestamp;        /* Indicates the time during the
                                       * transmission of this message.*/
};
OFP_ASSERT(sizeof(struct ofp_bundle_features_prop_time) == 72);
```

Time is represented using struct ofp_time, as defined in Section 7.3.9.5.

The ofp_bundle_features_prop_time property in a bundle features request includes:

- sched_accuracy: this field is relevant only to bundle features replies, and the switch must ignore this field in a bundle features request.
- sched_max_future and sched_max_past: a switch that receives a bundle features request with OFPBF_TIME_SET_SCHED set should attempt to change its scheduling tolerance values according to the sched_max_future and sched_max_past values from the time property. If the switch does not successfully update its scheduling tolerance values, it replies with an error message of type OFPET_BAD_REQUEST, and with the OFPBRC_MULTIPART_BAD_SCHED code.
- timestamp, indicating the controller's time during the transmission of this message. A switch that receives a bundle features request with OFPBF_TIMESTAMP set may use the received timestamp to roughly estimate the offset between its clock and the controller's clock.

The ofp_bundle_features_prop_time property in a bundle features reply includes:

- sched_accuracy, indicating the estimated scheduling accuracy of the switch. For example, if the value of sched_accuracy is 1000000 nanoseconds (1 ms), it means that when the switch receives a bundle commit scheduled to time T_s, the commit will in practice be invoked at $T_s \pm 1\ ms$. The factors that affect the scheduling accuracy are discussed in Section 6.9.6.3.

 © 2015; The Open Networking Foundation

- `sched_max_future` and `sched_max_past`, containing the scheduling tolerance values of the switch. If the corresponding bundle features request has the `OFPBF_TIME_SET_SCHED` flag enabled, these two fields are identical to the ones sent be the controller in the request.
- `timestamp`, indicating the switch's time during the transmission of this feature reply. Every bundle feature reply that includes the time property also includes a timestamp. The timestamp may be used by the controller to get a rough estimate of whether the switch's clock is synchronized to the controller's.

7.3.5.21 Experimenter Multipart

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

```
/* Body for ofp_multipart_request/reply of type OFPMP_EXPERIMENTER. */
struct ofp_experimenter_multipart_header {
    uint32_t experimenter;    /* Experimenter ID. */
    uint32_t exp_type;        /* Experimenter defined. */
    /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_multipart_header) == 8);
```

The `experimenter` field is the Experimenter ID and the `exp_type` field is the experimenter type, they take the same form as in the typical experimenter structure (see 7.2.8).

The rest of the request and reply bodies are experimenter-defined and don't need to be padded.

7.3.6 Packet-Out Message

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

```
/* Send packet (controller -> datapath). */
struct ofp_packet_out {
    struct ofp_header header;
    uint32_t buffer_id;           /* ID assigned by datapath (OFP_NO_BUFFER
                                     if none). */
    uint16_t actions_len;         /* Size of action array in bytes. */
    uint8_t pad[2];               /* Align to 64 bits. */
    struct ofp_match match;       /* Packet pipeline fields. Variable size. */
    /* The variable size and padded match is followed by the list of actions. */
    /* struct ofp_action_header actions[0]; *//* Action list - 0 or more. */
    /* The variable size action list is optionally followed by packet data.
     * This data is only present and meaningful if buffer_id == -1. */
    /* uint8_t data[0]; */        /* Packet data.  The length is inferred
                                     from the length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_packet_out) == 24);
```

 © 2015; The Open Networking Foundation

The buffer_id is the same given in the ofp_packet_in message. If the buffer_id is OFP_NO_BUFFER, then the packet data is included in the data array, and the packet encapsulated in the message is processed by the actions of the message. OFP_NO_BUFFER is 0xffffffff. If buffer_id is valid, the corresponding packet is removed from the buffer and processed by the actions of the message.

The match field is a set of OXM TLVs containing the pipeline fields associated with the packet. The TLVs in the match field define the value of the pipeline fields that must be associated with the packet for OpenFlow processing. The OXM TLVs must include the *Packet Type Match Field* to identify the packet type included in the data field, unless the packet is Ethernet (see 7.2.3.11). The match field can only include pipeline fields, and must not include any header fields (see 7.2.3.7). If the match field contains any header fields, the switch must refuse the ofp_packet_out message and send a *Pipeline Fields Only* error message (see 7.5.4.2).

If the packet was received at the controller via an ofp_packet_in message, the match field should be set using the pipeline field values given in the ofp_packet_in message (see 7.4.1).

Pipeline fields not included in the match field must be set to zero by the switch, except for OXM_OF_IN_PORT that is mandatory and for OXM_OF_IN_PHY_PORT that must be set to the value of OXM_OF_IN_PORT.

The OXM_OF_IN_PORT TLV included in the match field specifies the ingress port that must be associated with the packet for OpenFlow processing. This TLV is mandatory in the match field, and it must be set to either a valid standard switch port (see 4.2) or OFPP_CONTROLLER. For example, the value of this TLV is used when processing the packet using groups, OFPP_TABLE, OFPP_IN_PORT and OFPP_ALL. If OXM_OF_IN_PORT TLV is not included in the match field, the switch must refuse the ofp_packet_out message and send a *Bad Port* error message (see 7.5.4.2).

The action field is a list of actions defining how the packet should be processed by the switch. It may include packet modification, group processing and an output port. The list of actions of an OFPT_PACKET_OUT message can also specify the OFPP_TABLE reserved port as an output action to process the packet through the OpenFlow pipeline, starting at the first flow table (see 4.5). If OFPP_TABLE is specified, the match field is used as the set of pipeline fields in the flow table lookup.

If any action in the list of actions is not valid or supported, the switch must generate an error message with type as OFPET_BAD_ACTION (see 7.5.4.3).

In some cases, packets sent to OFPP_TABLE may be forwarded back to the controller as the result of a flow entry action or table miss. Detecting and taking action for such controller-to-switch loops is outside the scope of this specification. In general, OpenFlow messages are not guaranteed to be processed in order, therefore if a OFPT_PACKET_OUT message using OFPP_TABLE depends on a flow that was recently sent to the switch (with a OFPT_FLOW_MOD message), a OFPT_BARRIER_REQUEST message may be required prior to the OFPT_PACKET_OUT message to make sure the flow entry was committed to the flow table prior to execution of OFPP_TABLE.

The data field is either empty, or when buffer_id is OFP_NO_BUFFER the data field contains the packet to be processed. The packet type is determined by the *Packet Type Match Field* included in the match field, if no Packet Type Match Field is present the packet must be Ethernet (Ethernet frame, including an Ethernet header and Ethernet payload, but no Ethernet FCS - see 7.2.3.11). The only processing done by the switch on the packet is processing explicitly specified by the OpenFlow actions in the action field, is particular the switch must not transparently set the Ethernet source address or IP checksum.

 © 2015; The Open Networking Foundation

7.3.7 Barrier Message

When the controller wants to control message processing and ordering by the switch, it may use an OFPT_BARRIER_REQUEST message. This message has no body and is only composed of an OpenFlow header.

The first function of barrier message is to ensure message ordering, so that message dependencies are not violated (see 6.2). The second function of barrier message is for the controller to receive notifications for completed operations. The third function of barrier is to ensure that the state sent by the controller is committed to the datapath.

When the switch receives a OFPT_BARRIER_REQUEST message on a connection, the switch must finish processing all previously-received messages on that connection, including sending corresponding reply or error messages, before executing the barrier request or any messages on the connection received after the barrier request. If all previously-received messages have already been processed when the switch receives a barrier request, the switch can execute the barrier request immediately. When the processing of all message prior to the barrier is complete, the switch executes the barrier request. All state changes corresponding to messages prior to the barrier request must be committed to the datapath, so that any packet received by the switch after sending the barrier response must be processed by the datapath updated with the state changes of all messages prior to the barrier command (this requirement does not apply to packets in flight or queued through the pipeline when the barrier request is executed). When the execution of the barrier request is done, the switch must send an OFPT_BARRIER_REPLY message with the xid of that barrier request. The barrier reply must be sent after the reply or error messages corresponding to messages prior to the barrier request. After sending the barrier reply, the switch can process normally all messages after the barrier request (see 6.2).

7.3.8 Role Request Message

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

```
/* Role request and reply message. */
struct ofp_role_request {
    struct ofp_header header;   /* Type OFPT_ROLE_REQUEST/OFPT_ROLE_REPLY. */
    uint32_t role;              /* One of OFPCR_ROLE_*. */
    uint16_t short_id;          /* ID number for the controller. */
    uint8_t pad[2];             /* Align to 64 bits. */
    uint64_t generation_id;     /* Master Election Generation Id */
};
OFP_ASSERT(sizeof(struct ofp_role_request) == 24);
```

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

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

 © 2015; The Open Networking Foundation

If the role value in the message is OFPCR_ROLE_MASTER or OFPCR_ROLE_SLAVE, the switch must validate generation_id to check for stale messages (see 6.3.7). If the validation fails, the switch must discard the role request and return a *Stale* error message (see 7.5.4.12).

If the role value is OFPCR_ROLE_MASTER, all other controllers whose role was OFPCR_ROLE_MASTER are changed to OFPCR_ROLE_SLAVE (see 6.3.7). If the role value is OFPCR_ROLE_NOCHANGE, the current role of the controller is not changed ; this enables a controller to query its current role without changing it.

Upon receipt of a OFPT_ROLE_REQUEST message, if there is no error, the switch must return a OFPT_ROLE_REPLY message. The structure of this message is exactly the same as the OFPT_ROLE_REQUEST message, and the field role is the current role of the controller. The field generation_id is set to the current generation_id (the generation_id associated with the last successful role request with role OFPCR_ROLE_MASTER or OFPCR_ROLE_SLAVE), if the current generation_id was never set by a controller, the field generation_id in the reply must be set to the maximum field value (the unsigned equivalent of -1).

If the switch must change the role of another controller from OFPCR_ROLE_MASTER to OFPCR_ROLE_SLAVE, it must send that controller a OFPT_ROLE_STATUS message (see 7.4.4).

The OFPT_ROLE_REQUEST message also sets the short_id for the controller. This ID, if defined, uniquely identifies a particular controller. The ID is initially OFPCID_UNDEFINED, and controllers may set the short_id to a value they choose, or set it to OFPCID_UNDEFINED if they do not wish to use it. Only the OFPCID_UNDEFINED ID may be assigned to more than one controller, and setting the short_id to a value already in use will cause the entire OFPT_ROLE_REQUEST message to be rejected (see 7.5.4.12). OFPCID_UNDEFINED is 0.

7.3.9 Bundle messages

7.3.9.1 Bundle control messages

The controller can create, destroy and commit bundles with the OFPT_BUNDLE_CONTROL request. The OFPT_BUNDLE_CONTROL message uses the following structure:

```
/* Message structure for OFPT_BUNDLE_CONTROL. */
struct ofp_bundle_ctrl_msg {
    struct ofp_header     header;
    uint32_t              bundle_id;      /* Identify the bundle. */
    uint16_t              type;           /* OFPBCT_*. */
    uint16_t              flags;          /* Bitmap of OFPBF_* flags. */

    /* Bundle Property list. */
    struct ofp_bundle_prop_header properties[0]; /* Zero or more properties. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_ctrl_msg) == 16);
```

The bundle_id field is the bundle identifier, a 32 bit number chosen by the controller. The bundle identifier should be unique over the connection during the bundle lifetime.

The type field is the control message type. The control types that are currently defined are:

 © 2015; The Open Networking Foundation

```
/* Bundle control message types */
enum ofp_bundle_ctrl_type {
    OFPBCT_OPEN_REQUEST    = 0,
    OFPBCT_OPEN_REPLY      = 1,
    OFPBCT_CLOSE_REQUEST   = 2,
    OFPBCT_CLOSE_REPLY     = 3,
    OFPBCT_COMMIT_REQUEST  = 4,
    OFPBCT_COMMIT_REPLY    = 5,
    OFPBCT_DISCARD_REQUEST = 6,
    OFPBCT_DISCARD_REPLY   = 7,
};
```

The `flags` field is a bitmask of bundle flags (see 7.3.9.3).

The `properties` field is a list of bundle properties (see 7.3.9.4).

7.3.9.2 Bundle Add message

The controller can add requests to a bundle using the `OFPT_BUNDLE_ADD_MESSAGE` message. The `OFPT_BUNDLE_ADD_MESSAGE` message uses the following structure:

```
/* Message structure for OFPT_BUNDLE_ADD_MESSAGE.
 * Adding a message in a bundle is done with. */
struct ofp_bundle_add_msg {
    struct ofp_header     header;
    uint32_t              bundle_id;      /* Identify the bundle. */
    uint16_t              pad;            /* Align to 64 bits. */
    uint16_t              flags;          /* Bitmap of OFPBF_* flags. */

    struct ofp_header     message;        /* Message added to the bundle. */

    /* If there is one property or more, 'message' is followed by:
     *   - Exactly (message.length + 7)/8*8 - (message.length) (between 0 and 7)
     *     bytes of all-zero bytes */

    /* Bundle Property list. */
    /*struct ofp_bundle_prop_header properties[0];*/ /* Zero or more
                                                        properties. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_add_msg) == 24);
```

The `bundle_id` field is the bundle identifier, a 32 bit number chosen by the controller. The bundle identifier should be a bundle that has been previously opened and not yet closed.

The `flags` field is a bitmask of bundle flags (see 7.3.9.3).

The `message` is a OpenFlow message to be added to the bundle, it can be any OpenFlow message that the switch can support in a bundle. The field `xid` in the message must be identical to the field `xid` of the `OFPT_BUNDLE_ADD_MESSAGE` message.

The `properties` field is a list of bundle properties (see 7.3.9.4).

 © 2015; The Open Networking Foundation

7.3.9.3 Bundle flags

Bundle flags modify the behaviour of a bundle.

The bundle flags that are currently defined are:

```
/* Bundle configuration flags. */
enum ofp_bundle_flags {
    OFPBF_ATOMIC  = 1 << 0,  /* Execute atomically. */
    OFPBF_ORDERED = 1 << 1,  /* Execute in specified order. */
    OFPBF_TIME    = 1 << 2,  /* Execute in specified time. */
};
```

- `OFPBF_ATOMIC` is set to request fully atomic application of the bundle.
- `OFPBF_ORDERED` is set to request that the messages of the bundle are applied strictly in order.
- `OFPBF_TIME` is used in a bundle commit message to indicate that the commit message includes the bundle's execution time, using `ofp_bundle_prop_time`. For further details about scheduled bundles, see Section 6.9.6.

The bundle flags `OFPBF_ATOMIC` and `OFPBF_ORDERED` must be specified on every bundle message part of the bundle, and they need to be consistent. The flag `OFPBF_TIME` can be set only in bundle commit messages, and should be ignored by the receiver in other bundle messages.

7.3.9.4 Bundle properties

The list of bundle property types that are currently defined are:

```
/* Bundle property types. */
enum ofp_bundle_prop_type {
    OFPBPT_TIME                   = 1,      /* Time property. */
    OFPBPT_EXPERIMENTER           = 0xFFFF, /* Experimenter property. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all Bundle Properties */
struct ofp_bundle_prop_header {
    uint16_t         type;    /* One of OFPBPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_prop_header) == 4);
```

The `ofp_bundle_prop_time` property is included in bundle messages that have the `OFPBF_TIME` flag set. This property is used in scheduled bundles, as described in Section 6.9.6. This property can only be included in bundle commit messages. The `OFPBPT_TIME` property uses the following structure and fields:

 © 2015; The Open Networking Foundation

```
/* Bundle time property. */
    struct ofp_bundle_prop_time {
    uint16_t type;       /* OFPBPT_TIME */
    uint16_t length;     /* Length in bytes of this property (24). */
    uint8_t  pad[4];
    struct ofp_time scheduled_time;
};
OFP_ASSERT(sizeof(struct ofp_bundle_prop_time)==24);
```

The `scheduled_time` field specifies the time at which the switch is expected to perform the scheduled bundle. The format of this field is `ofp_time`, defined in Section 7.3.9.5.

The `OFPBPT_EXPERIMENTER` property uses the following structure and fields:

```
/* Experimenter bundle property */
struct ofp_bundle_prop_experimenter {
    uint16_t         type;    /* OFPBPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_bundle_prop_experimenter) == 12);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in struct `ofp_experimenter`.

7.3.9.5 Time Format

Bundle commit request messages may include the `scheduled_time`, which specifies *when* the bundle should be committed. The `scheduled_time` field uses `struct ofp_time`. The time format, `ofp_time`, is also used in bundle features messages (see 7.3.5.20.4).

The time format, `ofp_time`, is based on the format defined in the IEEE 1588-2008 standard. It consists of two subfields; a `seconds` field, representing the integer portion of time in seconds[1], and a `nanoseconds` field, representing the fractional portion of time in nanoseconds, i.e., $0 \leq nanoseconds \leq (10^9 - 1)$.

```
/* Time Format. */
struct ofp_time {
    uint64_t seconds;
    uint32_t nanoseconds;
    uint8_t  pad[4];
};
OFP_ASSERT(sizeof(struct ofp_time)==16);
```

[1]The seconds field in IEEE 1588-2008 is 48 bits long. The seconds field defined here is a 64-bit field, but it has the same semantics as the seconds field in the IEEE 1588-2008 time format.

 © 2015; The Open Networking Foundation

As defined in the IEEE 1588-2008 standard, time is measured according to the International Atomic Time (TAI) timescale. The origin of the timescale, also known as the epoch, is defined as 1 January 1970 00:00:00 TAI; at every instant, time is represented by the number of seconds that have elapsed since the epoch.

7.3.9.6 Creating and opening a bundle

To create a bundle, the controller sends a `OFPT_BUNDLE_CONTROL` message with type `OFPBCT_OPEN_REQUEST`. The switch must create a new bundle with id `bundle_id` attached to the current connection, with the options specified in the flags and properties. If the operation is successful, `OFPT_BUNDLE_CONTROL` message with type `OFPBCT_OPEN_REPLY` must be returned by the switch. If an error arises, an error message is returned.

The arguments of the Bundle Open Request must be validated and errors returned if they are not supported (see 7.5.4.18).

If the `bundle_id` already refers to an existing bundle attached to the same connection, the switch must refuse to open the new bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_BAD_ID` code. The existing bundle identified by `bundle_id` must be discarded.

If the switch can not open this bundle because its having too many opened bundles on the switch or attached to the current connection, the switch must refuse to open the new bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_OUT_OF_BUNDLES` code. If the switch can not open the bundle because the connection is using an unreliable transport, the switch must refuse to open the new bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_OUT_OF_BUNDLES` code.

If the `flags` field requests some feature that can not be implemented by the switch, the switch must refuse to open the new bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_BAD_FLAGS` code.

7.3.9.7 Adding messages to a bundle

The switch adds a message to a bundle using the `OFPT_BUNDLE_ADD_MESSAGE`. After the bundle is opened, the controller can send a sequence of `OFPT_BUNDLE_ADD_MESSAGE` messages to populate the bundle. Each `OFPT_BUNDLE_ADD_MESSAGE` includes an OpenFlow message, this OpenFlow message is validated, and if successful it is stored in the bundle specified by `bundle_id` on the current connection. If a message validation error or a bundle error condition arises, an error message is returned (see 7.5.4.18).

Message validation includes at minimum syntax checking and that all features requested in the message are supported, and it may optionally include checking resource availability (see 6.9.3). If a message fails validation, an error message must be returned. The error message must use the `xid` of the offending message, the error data field corresponding to that message and the error code corresponding to the validation failure.

If the `bundle_id` refers to a bundle that does not exist on the current connection, the corresponding bundle is created using arguments from the `OFPT_BUNDLE_ADD_MESSAGE` message. If an error arises from

 © 2015; The Open Networking Foundation

creating the bundle, the relevant error message is returned (see 7.3.9.6). No OFPT_BUNDLE_CONTROL message with type OFPBCT_OPEN_REPLY is returned by the switch in this case.

If the bundle_id refers to a bundle that is already closed, the switch must refuse to add the message to the bundle, discard the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BUNDLE_CLOSED code.

If the flags field is different from the flags that were specified when the bundle was opened, the switch must refuse to add the message to the bundle, discard the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_FLAGS code.

If the message in the request is normally supported on the switch but is not supported in a bundle, the switch must refuse to add the message to the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_MSG_UNSUP code. This is the case for hello, echo, and bundle messages, messages that are not requests (for example replies, status and errors messages), or if the implementation does not support including a specific modification message in a bundle.

If the message in the request is incompatible with another message already stored in the bundle, the switch must refuse to add the message to the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_MSG_CONFLICT code.

If the bundle is full and can not fit the message in the request, the switch must refuse to add the message to the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_MSG_TOO_MANY code.

If the message in the request does not have a valid length field, the switch must refuse to add the message to the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_MSG_BAD_LEN code.

Messages added in a bundle should have a unique xid to help matching errors to messages, except for multipart messages part of the same sequence, and the xid of the bundle add message must be the same as the embedded message. A switch may optionally verify that the two xid fields of a message are consistent or that two non-multipart messages of the bundle don't have the same xid, and if this is not the case refuse to add the new message to the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_MSG_BAD_XID code.

7.3.9.8 Closing a bundle

To finish recording a bundle, the controller may send a OFPT_BUNDLE_CONTROL message with type OFPBCT_CLOSE_REQUEST. The switch must close the bundle specified by bundle_id on the current connection. After closing the bundle, it can no longer be modified and no messages can be added to it, it can only be committed or discarded. Closing a bundle is optional. If the operation is successful, OFPT_BUNDLE_CONTROL message with type OFPBCT_CLOSE_REPLY must be returned by the switch. If an error arises, an error message is returned (see 7.5.4.18).

If the bundle_id refers to a bundle that does not exist, the switch must reject the request and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_ID code.

 © 2015; The Open Networking Foundation

If the bundle_id refers to a bundle that is already closed, the switch must refuse to close to the bundle, discard the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BUNDLE_CLOSED code.

If the flags field is different from the flags that were specified when the bundle was opened, the switch must refuse to close to the bundle, discard the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_FLAGS code.

The switch may optionally do additional validation of the messages part of the bundle as the result of the close request, such as validating resource availability. For each message that fails this additional validation, an error message must be generated that refers to the offending message. After sending those individual error messages, the switch must discard the bundle and send an additional ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_MSG_FAILED code.

7.3.9.9 Committing Bundles

To finish and apply the bundle, the controller sends a OFPT_BUNDLE_CONTROL message with type OFPBCT_COMMIT_REQUEST. The switch must commit the bundle specified by bundle_id on the current connection, it must apply all messages stored in the bundle as a single operation or return an error. The commit operation and the way the message part of the bundle are applied depend on the bundle flags (see 7.3.9.3). If the bundle was not closed prior to this request, it is automatically closed (see 7.3.9.8).

The commit is successful only if all messages that are part of the bundle can be applied without error. If the bundle does not contain any message, commit is always successful. If the commit is successful, the switch must apply all messages of the bundle as a single operation, and a OFPT_BUNDLE_CONTROL message with type OFPBCT_COMMIT_REPLY must be returned by the switch. The OFPT_BUNDLE_CONTROL message with type OFPBCT_COMMIT_REPLY must be sent to the controller after the processing of all messages part of the bundle are guaranteed to no longer fail or produce an error.

If one or more message part of the bundle can not be applied without error, for example due to resource availability, the commit fails and all messages that are part of the bundle must be discarded without being applied. When the commit fails, the switch must generate an error message corresponding to the message that could not be applied. The error message must use the xid of the offending message, the error data field corresponding to that message and the error code corresponding to the failure. If multiple messages are generating errors, the switch may return only the first error found or generate multiple error messages for the same bundle. After sending those individual error messages, the switch must send an additional ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_MSG_FAILED code.

After a commit operation, the bundle is discarded, whether the commit was successful or not. After receiving a successful reply or error message for this operation, the controller can reuse the bundle_id.

Modification requests may require replies to be returned to the controller or events to be generated. Because any message of the bundle may fail, replies and events can only be generated once all modifications in the bundle have been applied. For replies, each of these replies contains the transaction ID of the corresponding request.

 © 2015; The Open Networking Foundation

If the bundle_id refers to a bundle that does not exist, the switch must reject the request and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_ID code.

If the flags field is different from the flags that were specified when the bundle was opened, the switch must refuse to commit the bundle, discard the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_FLAGS code.

7.3.9.10 Discarding Bundles

To finish and discard the bundle, the controller sends a OFPT_BUNDLE_CONTROL message with type OFPBCT_DISCARD_REQUEST. The switch must discard the bundle specified by bundle_id on the current connection, and all messages that are part of the bundle are discarded. If the operation is successful, a OFPT_BUNDLE_CONTROL message with type OFPBCT_DISCARD_REPLY must be returned by the switch. If an error arises, an error message is returned. After receiving either a successful reply or an error message, the controller can reuse the bundle_id.

All implementations must be able to process a discard request on an existing bundle on the current connection without triggering errors. If the bundle_id refers to a bundle that does not exist, the switch must reject the request and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_ID code.

7.3.9.11 Other bundle error conditions

If an OFPT_BUNDLE_CONTROL message contains an invalid type, the switch must reject the request and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_TYPE code.

If an OFPT_BUNDLE_CONTROL or OFPT_BUNDLE_ADD_MESSAGE message specifies flags different for those already specified on an existing bundle, the switch must reject the request and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_FLAGS code. An exception to the latter is the OFPBF_TIME flag, which may have a different value in a bundle commit request message than in previous bundle messages, as described in 7.3.9.3.

If the switch does not receive any OFPT_BUNDLE_CONTROL or OFPT_BUNDLE_ADD_MESSAGE message for an opened bundle_id for a switch defined time greater than 1s, it may send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_TIMEOUT code. If the switch does not receive any new message in a bundle for a switch defined time greater than 1s, it may send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_TIMEOUT code.

If an OpenFlow message can not be processed because a bundle is locking a resource this message is using, the switch must reject that message and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BUNDLE_IN_PROGRESS code. If a switch can not process other messages between opening a bundle and either committing it or discarding it, the switch must reject that message and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BUNDLE_IN_PROGRESS code.

If a switch receives a scheduled commit and is not able to execute it, it sends an error message with type OFPET_BUNDLE_FAILED. The following error codes are specific to scheduled bundles:

- OFPBFC_SCHED_NOT_SUPPORTED - this code is used when the switch does not support scheduled bundle execution and receives a commit message with the OFPBF_TIME flag set.

 © 2015; The Open Networking Foundation

- OFPBFC_SCHED_FUTURE - used when the switch receives a scheduled commit message and the scheduling time exceeds sched_max_future (see Section 6.9.6.4).
- OFPBFC_SCHED_PAST - used when the switch receives a scheduled commit message and the scheduling time exceeds sched_max_past (see Section 6.9.6.4).

7.3.10 Set Asynchronous Configuration Message

The switch manages a per-controller asynchronous configuration, which defines the asynchronous messages that it wants to receive (other than error messages) on a given OpenFlow channel. The controller is able to set and query its asynchronous configuration with the OFPT_SET_ASYNC and OFPT_GET_ASYNC_REQUEST messages, respectively. The switch responds to a OFPT_GET_ASYNC_REQUEST message with an OFPT_GET_ASYNC_REPLY message; it does not reply to a request to set the configuration.

The OFPT_SET_ASYNC message sets whether a controller should receive a given asynchronous message that is generated by the switch. Other OpenFlow features control whether a given message is generated; for example, the OFPFF_SEND_FLOW_REM flag in each flow entry controls whether the switch generates a OFPT_FLOW_REMOVED message when a flow entry is removed. The asynchronous configuration acts as an additional per-controller filter; for example it defines if a specific controller receives those OFPT_FLOW_REMOVED messages.

The configuration set with the OFPT_SET_ASYNC message is specific to a particular OpenFlow channel. It does not affect any other OpenFlow channels, whether currently established or to be established in the future. The configuration set with OFPT_SET_ASYNC does not filter or otherwise affect error messages.

A switch configuration, for example using the OpenFlow Configuration Protocol, may set the initial configuration of asynchronous messages when an OpenFlow connection is established. In the absence of such switch configuration, the initial configuration shall be:

- In the "master" or "equal" role, enable all OFPT_PACKET_IN messages, except those with reason OFPR_INVALID_TTL, enable all OFPT_PORT_STATUS, OFPT_FLOW_REMOVED, OFPT_CONTROLLER_STATUS messages, and disable all OFPT_ROLE_STATUS and OFPT_REQUESTFORWARD messages.
- In the "slave" role, enable all OFPT_PORT_STATUS and OPFT_CONTROLLER_STATUS messages and disable all OFPT_PACKET_IN, OFPT_FLOW_REMOVED, OFPT_ROLE_STATUS and OFPT_REQUESTFORWARD messages.

There is no body for OFPT_GET_ASYNC_REQUEST beyond the OpenFlow header. The OFPT_SET_ASYNC and OFPT_GET_ASYNC_REPLY messages have the following format:

```
/* Asynchronous message configuration. */
struct ofp_async_config {
    struct ofp_header header;     /* OFPT_GET_ASYNC_REPLY or OFPT_SET_ASYNC. */

    /* Async config Property list - 0 or more */
    struct ofp_async_config_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_async_config) == 8);
```

 © 2015; The Open Networking Foundation

The properties field is a list of asynchronous configuration properties, describing if various sets of asynchronous messages are enabled or not. In an OFPT_SET_ASYNC, only the properties that shall be changed need to be included, properties that are omitted from the message are unchanged. In an OFPT_GET_ASYNC_REPLY message, all properties must be included.

The asynchronous configuration types currently defined are:

```
/* Async Config property types.
 * Low order bit cleared indicates a property for the slave role.
 * Low order bit set indicates a property for the master/equal role.
 */
enum ofp_async_config_prop_type {
    OFPACPT_PACKET_IN_SLAVE      = 0,  /* Packet-in mask for slave. */
    OFPACPT_PACKET_IN_MASTER     = 1,  /* Packet-in mask for master. */
    OFPACPT_PORT_STATUS_SLAVE    = 2,  /* Port-status mask for slave. */
    OFPACPT_PORT_STATUS_MASTER   = 3,  /* Port-status mask for master. */
    OFPACPT_FLOW_REMOVED_SLAVE   = 4,  /* Flow removed mask for slave. */
    OFPACPT_FLOW_REMOVED_MASTER  = 5,  /* Flow removed mask for master. */
    OFPACPT_ROLE_STATUS_SLAVE    = 6,  /* Role status mask for slave. */
    OFPACPT_ROLE_STATUS_MASTER   = 7,  /* Role status mask for master. */
    OFPACPT_TABLE_STATUS_SLAVE   = 8,  /* Table status mask for slave. */
    OFPACPT_TABLE_STATUS_MASTER  = 9,  /* Table status mask for master. */
    OFPACPT_REQUESTFORWARD_SLAVE  = 10, /* RequestForward mask for slave. */
    OFPACPT_REQUESTFORWARD_MASTER = 11, /* RequestForward mask for master. */
    OFPACPT_FLOW_STATS_SLAVE     = 12, /* Flow stats mask for slave. */
    OFPACPT_FLOW_STATS_MASTER    = 13, /* Flow stats mask for master. */
    OFPACPT_CONT_STATUS_SLAVE    = 14, /* Controller status mask for slave. */
    OFPACPT_CONT_STATUS_MASTER   = 15, /* Controller status mask for master. */
    OFPACPT_EXPERIMENTER_SLAVE   = 0xFFFE, /* Experimenter for slave. */
    OFPACPT_EXPERIMENTER_MASTER  = 0xFFFF, /* Experimenter for master. */
};
```

Each property type controls whether the controller receives asynchronous messages with a specific enum ofp_type. The properties with the _MASTER suffix specify messages of interest when the controller has a OFPCR_ROLE_EQUAL or OFPCR_ROLE_MASTER role (see 6.3.7). The properties with the _SLAVE suffix specifies messages of interest when the controller has a OFPCR_ROLE_SLAVE role.

A property definition contains the property type, length, and any associated data:

```
/* Common header for all async config Properties */
struct ofp_async_config_prop_header {
    uint16_t         type;    /* One of OFPACPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_async_config_prop_header) == 4);
```

The OFPACPT_PACKET_IN_SLAVE, OFPACPT_PACKET_IN_MASTER, OFPACPT_PORT_STATUS_SLAVE, OFPACPT_PORT_STATUS_MASTER, OFPACPT_FLOW_REMOVED_SLAVE, OFPACPT_FLOW_REMOVED_MASTER, OFPACPT_ROLE_STATUS_SLAVE and OFPACPT_ROLE_STATUS_MASTER, OFPACPT_TABLE_STATUS_SLAVE, OFPACPT_TABLE_STATUS_MASTER, OFPACPT_REQUESTFORWARD_SLAVE, OFPACPT_REQUESTFORWARD_MASTER, OFPACPT_FLOW_STATS_SLAVE, OFPACPT_FLOW_STATS_MASTER, OPFT_CONT_STATUS_SLAVE and OPFT_CONT_STATUS_MASTER properties use the following structure and fields:

 © 2015; The Open Networking Foundation

```
/* Various reason based properties */
struct ofp_async_config_prop_reasons {
    uint16_t         type;    /* One of OFPACPT_PACKET_IN_*,
                                 OFPACPT_PORT_STATUS_*,
                                 OFPACPT_FLOW_REMOVED_*,
                                 OFPACPT_ROLE_STATUS_*,
                                 OFPACPT_TABLE_STATUS_*,
                                 OFPACPT_REQUESTFORWARD_*,
                                 OFPACPT_FLOW_STATS_*,
                                 OFPACPT_CONT_STATUS_*. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         mask;    /* Bitmasks of reason values. */
};
OFP_ASSERT(sizeof(struct ofp_async_config_prop_reasons) == 8);
```

The `mask` field is a bit-mask in which a 0-bit disables receiving a message sent with the `reason` code corresponding to the bit index and a 1-bit enables receiving it. For example, if the property type is `OFPACPT_PORT_STATUS_SLAVE`, the bit with value $2^2 = 4$ in `mask` determines whether the controller will receive `OFPT_PORT_STATUS` messages with reason `OFPPR_MODIFY` (value 2) when the controller has role `OFPCR_ROLE_SLAVE`.

The `OFPACPT_EXPERIMENTER_SLAVE` and `OFPACPT_EXPERIMENTER_MASTER` properties uses the following structure and fields:

```
/* Experimenter async config  property */
struct ofp_async_config_prop_experimenter {
    uint16_t         type;    /* One of OFPTFPT_EXPERIMENTER_SLAVE,
                                 OFPTFPT_EXPERIMENTER_MASTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_async_config_prop_experimenter) == 12);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

7.4 Asynchronous Messages

7.4.1 Packet-In Message

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

 © 2015; The Open Networking Foundation

```
/* Packet received on port (datapath -> controller). */
struct ofp_packet_in {
    struct ofp_header header;
    uint32_t buffer_id;     /* ID assigned by datapath. */
    uint16_t total_len;     /* Full length of frame. */
    uint8_t reason;         /* Reason packet is being sent (one of OFPR_*) */
    uint8_t table_id;       /* ID of the table that was looked up */
    uint64_t cookie;        /* Cookie of the flow entry that was looked up. */
    struct ofp_match match; /* Packet metadata. Variable size. */
    /* The variable size and padded match is always followed by:
     *   - Exactly 2 all-zero padding bytes, then
     *   - An Ethernet frame whose length is inferred from header.length.
     * The padding bytes preceding the Ethernet frame ensure that the IP
     * header (if any) following the Ethernet header is 32-bit aligned.
     */
    //uint8_t pad[2];       /* Align to 64 bit + 16 bit */
    //uint8_t data[0];      /* Ethernet frame */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 32);
```

The `buffer_id` is an opaque value used by the datapath to identify a buffered packet. If the packet associated with the packet-in message is buffered, the `buffer_id` must be an identifier unique on the current connection referring to that packet on the switch. If the packet is not buffered - either because of no available buffers, or because of being explicitly requested via `OFPCML_NO_BUFFER` - the `buffer_id` must be `OFP_NO_BUFFER`.

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

The `total_len` is the full length of the packet that triggered the packet-in message. The actual length of the `data` field in the message may be less than `total_len` in case the packet had been truncated due to buffering (see 6.1.2). The field `total_len` must always correspond to the packet prior to buffering and truncation. The field `total_len` counts the number of bytes in the packet as defined for the *Packet Type* (see 7.2.3.11).

The `reason` field indicates which context triggered the packet-in message and can be any of these values:

```
/* Why is this packet being sent to the controller? */
enum ofp_packet_in_reason {
    OFPR_TABLE_MISS   = 0,   /* No matching flow (table-miss flow entry). */
    OFPR_APPLY_ACTION = 1,   /* Output to controller in apply-actions. */
    OFPR_INVALID_TTL  = 2,   /* Packet has invalid TTL */
    OFPR_ACTION_SET   = 3,   /* Output to controller in action set. */
    OFPR_GROUP        = 4,   /* Output to controller in group bucket. */
    OFPR_PACKET_OUT   = 5,   /* Output to controller in packet-out. */
};
```

`OFPR_INVALID_TTL` indicates that a packet with an invalid IP TTL or MPLS TTL was rejected by the OpenFlow pipeline and passed to the controller. Checking for invalid TTL does not need to

 © 2015; The Open Networking Foundation

be done for every packet, but it must be done at a minimum every time a OFPAT_DEC_MPLS_TTL or OFPAT_DEC_NW_TTL action is applied to a packet.

The cookie field contains the cookie of the flow entry that caused the packet to be sent to the controller. This field must be set to -1 (0xffffffffffffffff) if a cookie cannot be associated with a particular flow. For example, if the packet-in was generated in a group bucket or from the action set.

The match field is a set of OXM TLVs containing the pipeline fields associated with a packet. The pipeline fields values cannot be determined from the packet data, and include for example the input port and the metadata value (see 7.2.3.9). The OXM TLVs must include the *Packet Type Match Field* to identify the packet type included in the data field, unless the packet is Ethernet (see 7.2.3.11). The set of OXM TLVs must include all pipeline fields associated with that packet, supported by the switch and which value is not all-bits-zero. The field OXM_OF_ACTSET_OUTPUT should be ommited from this set. If OXM_OF_IN_PHY_PORT has the same value as OXM_OF_IN_PORT, it should be omitted from this set. The set of OXM TLVs may optionally include pipeline fields whose value is all-bits-zero. The set of OXM TLVs may also optionally include packet header fields. Most switches should not include those optional fields, to minimise the size of the packet-in, and therefore the controller should not depend on their presence and should extract header fields from the data field. The set of OXM TLVs must reflect the packet's headers and context when the event that triggers the packet-in message occurred, they should include all modifications made in the course of previous processing.

The port referenced by the OXM_OF_IN_PORT TLV is the packet ingress port used for matching flow entries and must be a valid standard OpenFlow port (see 7.2.3.9). The port referenced by the OXM_OF_IN_PHY_PORT TLV is the underlying physical port (see 7.2.3.9).

The pad field is additional padding, in addition to the potential padding of the match field, to make sure the IP header of the Ethernet frame is properly aligned. This padding is included even if the data field is empty.

The data field contains part of the packet that triggered the packet-in message, the packet is either included in full in the data field or it is truncated if it is buffered. The packet type is determined by the *Packet Type Match Field* included in the match field, is no Packet Type Match Field is present the packet must be Ethernet (Ethernet frame, including an Ethernet header and Ethernet payload, but no Ethernet FCS - see 7.2.3.11). The packet header reflects any changes applied to the packet in previous processing, but does not include pending changes from the action set.

When a packet is buffered, some number of bytes from the packet beginning will be included in the data field of the message. If the packet is sent because of a "send to controller" action, then this number of bytes must be equal to the max_len value from the ofp_action_output that triggered the message. If the packet is sent for other reasons, such as an invalid TTL, then this number of bytes must be equal to the miss_send_len value configured using the OFPT_SET_CONFIG message. The default miss_send_len is 128 bytes. If the packet is not buffered - either because of no available buffers, or because of being explicitly requested via OFPCML_NO_BUFFER - the entire packet must be included in the data field.

7.4.2 Flow Removed Message

If the controller has requested to be notified when flow entries time out or are deleted from tables (see 6.5), the datapath does this with the OFPT_FLOW_REMOVED message:

 © 2015; The Open Networking Foundation

```
/* Flow removed (datapath -> controller). */
struct ofp_flow_removed {
    struct ofp_header header;
    uint8_t table_id;         /* ID of the table */
    uint8_t reason;           /* One of OFPRR_*. */
    uint16_t priority;        /* Priority level of flow entry. */
    uint16_t idle_timeout;    /* Idle timeout from original flow mod. */
    uint16_t hard_timeout;    /* Hard timeout from original flow mod. */
    uint64_t cookie;          /* Opaque controller-issued identifier. */
    struct ofp_match match;   /* Description of fields. Variable size. */
    //struct ofp_stats stats; /* Statistics list. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_removed) == 32);
```

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

The `reason` field is one of the following:

```
/* Why was this flow removed? */
enum ofp_flow_removed_reason {
    OFPRR_IDLE_TIMEOUT = 0,     /* Flow idle time exceeded idle_timeout. */
    OFPRR_HARD_TIMEOUT = 1,     /* Time exceeded hard_timeout. */
    OFPRR_DELETE       = 2,     /* Evicted by a DELETE flow mod. */
    OFPRR_GROUP_DELETE = 3,     /* Group was removed. */
    OFPRR_METER_DELETE = 4,     /* Meter was removed. */
    OFPRR_EVICTION     = 5,     /* Switch eviction to free resources. */
};
```

The reason values `OFPRR_IDLE_TIMEOUT` and `OFPRR_HARD_TIMEOUT` are used by the expiry process (see 6.5). The reason value `OFPRR_DELETE` is used when the flow entry is removed by a flow-mod request (see 6.4). The reason value `OFPRR_GROUP_DELETE` is used when the flow entry is removed by a group-mod request (see 6.7). The reason value `OFPRR_METER_DELETE` is used when the flow entry is removed by a meter-mod request (see 6.8). The reason value `OFPRR_EVICTION` is used when the flow entry is removed by the eviction process (see 6.5).

The `stats` field contains a list of OXS fields (see 7.2.4.2). The field `OXS_OF_DURATION` is mandatory and tracks how long the flow entry has been installed in the switch. The fields `OXS_OF_PACKET_COUNT` and `OXS_OF_BYTE_COUNT` are optional and indicate the number of packets and bytes that were associated with this flow entry, respectively.

The `idle_timeout` and `hard_timeout` fields are directly copied from the flow mod that created this entry. With the above two fields and `OXS_OF_DURATION`, one can find both the amount of time the flow entry was active, as well as the amount of time the flow entry received traffic.

7.4.3 Port Status Message

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

 © 2015; The Open Networking Foundation

```
/* A physical port has changed in the datapath */
struct ofp_port_status {
    struct ofp_header header;
    uint8_t reason;          /* One of OFPPR_*. */
    uint8_t pad[7];          /* Align to 64-bits. */
    struct ofp_port desc;
};
OFP_ASSERT(sizeof(struct ofp_port_status) == 56);
```

The `reason` field can be one of the following values:

```
/* What changed about the physical port */
enum ofp_port_reason {
    OFPPR_ADD     = 0,         /* The port was added. */
    OFPPR_DELETE  = 1,         /* The port was removed. */
    OFPPR_MODIFY  = 2,         /* Some attribute of the port has changed. */
};
```

The reason value `OFPPR_ADD` denotes a port that did not exist in the datapath and has been added. The reason value `OFPPR_DELETE` denotes a port that has been removed from the datapath and no longer exists. The reason value `OFPPR_MODIFY` denotes a port which state or config has changed (see 7.2.1).

The `desc` field is a port description (see 7.2.1). In the port description, all properties don't need to be included, only those properties that have changed must be included, whereas unchanged properties may optionally be included.

7.4.4 Controller Role Status Message

When a controller has its role changed by the switch, and not directly changed by that controller using a `OFPT_ROLE_REQUEST` message, the corresponding controller must be informed with a `OFPT_ROLE_STATUS` message. The switch may generate `OFPT_ROLE_STATUS` messages at other times, for example when experimenter data change, this is outside the scope of this specification.

The `OFPT_ROLE_STATUS` message uses the following structure and fields:

```
/* Role status event message. */
struct ofp_role_status {
    struct ofp_header header;   /* Type OFPT_ROLE_STATUS. */
    uint32_t role;              /* One of OFPCR_ROLE_*. */
    uint8_t reason;             /* One of OFPCRR_*. */
    uint8_t pad[3];             /* Align to 64 bits. */
    uint64_t generation_id;     /* Master Election Generation Id */

    /* Role Property list */
    struct ofp_role_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_role_status) == 24);
```

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

 © 2015; The Open Networking Foundation

```
/* What changed about the controller role */
enum ofp_controller_role_reason {
    OFPCRR_MASTER_REQUEST = 0,  /* Another controller asked to be master. */
    OFPCRR_CONFIG         = 1,  /* Configuration changed on the switch. */
    OFPCRR_EXPERIMENTER   = 2,  /* Experimenter data changed. */
};
```

The `role` is the new role of the controller (see 7.3.8). The `generation_id` is the generation ID that was included in the role request message that triggered the role change.

The `properties` field is a list of role properties, describing dynamic parameters of table configuration.

The list of role property types that are currently defined are:

```
/* Role property types.
 */
enum ofp_role_prop_type {
    OFPRPT_EXPERIMENTER           = 0xFFFF, /* Experimenter property. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all Role Properties */
struct ofp_role_prop_header {
    uint16_t         type;    /* One of OFPRPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_role_prop_header) == 4);
```

The `OFPTMPT_EXPERIMENTER` property uses the following structure and fields:

```
/* Experimenter role property */
struct ofp_role_prop_experimenter {
    uint16_t         type;    /* One of OFPRPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_role_prop_experimenter) == 12);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in the typical experimenter structure (see 7.2.8).

 © 2015; The Open Networking Foundation

7.4.5 Table Status Message

When the table state changes, the controller needs to be informed with the OFPT_TABLE_STATUS message:

```
/* A table config has changed in the datapath */
struct ofp_table_status {
    struct ofp_header header;
    uint8_t reason;         /* One of OFPTR_*. */
    uint8_t pad[7];         /* Pad to 64 bits */
    struct ofp_table_desc table;   /* New table config. */
};
OFP_ASSERT(sizeof(struct ofp_table_status) == 24);
```

The reason can be one of the following values:

```
/* What changed about the table */
enum ofp_table_reason {
    OFPTR_VACANCY_DOWN  = 3,        /* Vacancy down threshold event. */
    OFPTR_VACANCY_UP    = 4,        /* Vacancy up threshold event. */
};
```

The reason values OFPTR_VACANCY_DOWN and OFPTR_VACANCY_UP identify a vacancy message. The vacancy events are generated when the remaining space in the flow table changes and crosses one of the vacancy threshold (see 7.3.4.1), whatever causes the change in remaining space. This change in remaining space can be the result of OpenFlow operations on the table (flow insertion, expiry or removal), or the result of switch internal processing changing the amount of remaining space in the table.

7.4.6 Request Forward Message

When a controller modifies the state of groups and meters, the request that successfully modified this state may be forwarded to other controllers. Other controllers are informed with the OFPT_REQUESTFORWARD message:

```
/* Group/Meter request forwarding. */
struct ofp_requestforward_header {
    struct ofp_header header;   /* Type OFPT_REQUESTFORWARD. */
    struct ofp_header request;  /* Request being forwarded. */
};
OFP_ASSERT(sizeof(struct ofp_requestforward_header) == 16);
```

Request forwarding is enabled on a per controller channel basis using the Set Asynchronous Configuration Message (see 7.3.10). The following forwarding reasons may be enabled in the requestforward_mask field:

```
/* Request forward reason */
enum ofp_requestforward_reason {
    OFPRFR_GROUP_MOD = 0,       /* Forward group mod requests. */
    OFPRFR_METER_MOD = 1,       /* Forward meter mod requests. */
};
```

 © 2015; The Open Networking Foundation

7.4.7 Controller Status Message

When the status of a controller changes for any of the reasons enumerated below, the switch must send a controller status message to all connected controllers.

```
/* Why is the controller status being reported? */
enum ofp_controller_status_reason {
    OFPCSR_REQUEST            = 0,  /* Controller requested status. */
    OFPCSR_CHANNEL_STATUS     = 1,  /* Oper status of channel changed. */
    OFPCSR_ROLE               = 2,  /* Controller role changed. */
    OFPCSR_CONTROLLER_ADDED   = 3,  /* New controller added. */
    OFPCSR_CONTROLLER_REMOVED = 4,  /* Controller removed from config. */
    OFPCSR_SHORT_ID           = 5,  /* Controller ID changed. */
    OFPCSR_EXPERIMENTER       = 6,  /* Experimenter data changed. */
};
```

Control channel updates are enabled on a per controller channel basis using the Set Asychronous Configuration Messsage (see 7.3.10). The reason code `OFPCSR_REQUEST` is used in multipart messages (see 7.3.5.15), and asynchronous messages should not be generated with this reason code. The switch should ignore the asynchronous configuration for this reason code.

The controller status message consists of the OpenFlow header, and a controller status structure representing the current status of the changed controller with reason code. If a single event causes the status of multiple controllers to change (for example, the failure of a network link over which multiple control channels run), the switch must send a different message for each affected controller.

```
struct ofp_controller_status_header {
    struct ofp_header header;               /* Type OFPT_CONTROLLER_STATUS. */
    struct ofp_controller_status status;    /* Controller status. */
};
```

If a controller is removed from the switch's configuration, the switch must send a controller status to all remaining, connected controllers with `reason` set to `OFPCSR_CONTROLLER_REMOVED`. The controller should ignore the values for `role` and `channel_status` for removed controllers.

The controller status structure is described in 7.2.7.

7.5 Symmetric Messages

7.5.1 Hello

The `OFPT_HELLO` message consists of an OpenFlow header plus a set of variable size hello elements.

```
/* OFPT_HELLO.  This message includes zero or more hello elements having
 * variable size. Unknown elements types must be ignored/skipped, to allow
 * for future extensions. */
struct ofp_hello {
    struct ofp_header header;
```

 © 2015; The Open Networking Foundation

```
    /* Hello element list */
    struct ofp_hello_elem_header elements[0]; /* List of elements - 0 or more */
};
OFP_ASSERT(sizeof(struct ofp_hello) == 8);
```

The `version` field part of the `header` field (see 7.1.1) must be set to the highest OpenFlow switch protocol version supported by the sender (see 6.3.3).

The `elements` field is a set of hello elements, containing optional data to inform the initial handshake of the connection. Implementations must ignore (skip) all elements of a Hello message that they do not support. The list of hello elements types that are currently defined are:

```
/* Hello elements types.
 */
enum ofp_hello_elem_type {
    OFPHET_VERSIONBITMAP          = 1,  /* Bitmap of version supported. */
};
```

An element definition contains the element type, length, and any associated data:

```
/* Common header for all Hello Elements */
struct ofp_hello_elem_header {
    uint16_t         type;    /* One of OFPHET_*. */
    uint16_t         length;  /* Length in bytes of the element,
                                 including this header, excluding padding. */
};
OFP_ASSERT(sizeof(struct ofp_hello_elem_header) == 4);
```

The `OFPHET_VERSIONBITMAP` element uses the following structure and fields:

```
/* Version bitmap Hello Element */
struct ofp_hello_elem_versionbitmap {
    uint16_t         type;    /* OFPHET_VERSIONBITMAP. */
    uint16_t         length;  /* Length in bytes of this element,
                                 including this header, excluding padding. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the bitmaps, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         bitmaps[0];   /* List of bitmaps - supported versions */
};
OFP_ASSERT(sizeof(struct ofp_hello_elem_versionbitmap) == 4);
```

The `bitmaps` field indicates the set of versions of the OpenFlow switch protocol a device supports, and may be used during version negotiation (see 6.3.3). The bits of the set of bitmaps are indexed by the *ofp_version* number of the protocol ; if the bit identified by the number of left bitshift equal to a *ofp_version* number is set, this OpenFlow version is supported. The number of bitmaps included in the field depends on the highest version number supported : *ofp_versions* 0 to 31 are encoded in the first bitmap, *ofp_versions* 32 to 63 are encoded in the second bitmap and so on. For example, if a switch supports only version 1.0 (*ofp_version*=0x01) and version 1.3 (*ofp_version*=0x04), the first bitmap would be set to 0x00000012.

 © 2015; The Open Networking Foundation

7.5.2 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.

7.5.3 Echo Reply

An Echo Reply message consists of an OpenFlow header plus the unmodified data field of an echo request message.

In an OpenFlow switch 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.

7.5.4 Error Message

Error messages are used by the switch or the controller to notify problems to the other side of the connection. They are mostly used by the switch to indicate a failure of a request initiated by the controller.

Error conditions in OpenFlow messages, such as an invalid field value or an unsupported feature, in most cases must return an error message, and in other cases should be ignored (see 7.1.3). If there are multiple error codes that could apply to a specific error condition, if the error code for that error condition is not specified in this document, then the most specific and appropriate error code must be used. If a message contains multiple errors, then one single error message for one of those errors must be returned. Different switches will evaluate message fields in a different order, therefore a controller should not expect a specific error code when multiple errors are present.

The `OFPT_ERROR_MSG` message uses the following structure :

```
/* OFPT_ERROR: Error message. */
struct ofp_error_msg {
    struct ofp_header header;

    uint16_t type;
    uint16_t code;
    uint8_t data[0];          /* Variable-length data.  Interpreted based
                                 on the type and code.  No padding. */
};
OFP_ASSERT(sizeof(struct ofp_error_msg) == 12);
```

The `type` field indicates the high-level type of error. The `code` field is interpreted based on the type, and indicates the exact error within the high level type of error. The list of valid `type` values and `code` values for each type are given below.

 © 2015; The Open Networking Foundation

The field `data` is variable in length and interpreted based on the `type` and `code`. Unless specified otherwise, the `data` field must contain at least 64 bytes of the failed message that caused the error message to be generated, if the failed message is shorter than 64 bytes it must contain the full message without any padding. If the request was a multipart request that can span multiple multipart messages (i.e. defined as an array of structure - see 7.3.5), and if the error specifies that the `data` field contains part of the failed request, then the `data` field must either contain part of the multipart message where the error was found, or be empty.

If the error message is in response to a specific message from the controller, then the `xid` field of the header in the error message must match that of the offending message. If the error is caused by the processing of a request, the error message is sent in place of the reply and the normal reply to the request is not sent.

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

Currently defined error types are:

```
/* Values for 'type' in ofp_error_message.  These values are immutable: they
 * will not change in future versions of the protocol (although new values may
 * be added). */
enum ofp_error_type {
    OFPET_HELLO_FAILED         = 0,  /* Hello protocol failed. */
    OFPET_BAD_REQUEST          = 1,  /* Request was not understood. */
    OFPET_BAD_ACTION           = 2,  /* Error in action description. */
    OFPET_BAD_INSTRUCTION      = 3,  /* Error in instruction list. */
    OFPET_BAD_MATCH            = 4,  /* Error in match. */
    OFPET_FLOW_MOD_FAILED      = 5,  /* Problem modifying flow entry. */
    OFPET_GROUP_MOD_FAILED     = 6,  /* Problem modifying group entry. */
    OFPET_PORT_MOD_FAILED      = 7,  /* Port mod request failed. */
    OFPET_TABLE_MOD_FAILED     = 8,  /* Table mod request failed. */
    OFPET_QUEUE_OP_FAILED      = 9,  /* Queue operation failed. */
    OFPET_SWITCH_CONFIG_FAILED = 10, /* Switch config request failed. */
    OFPET_ROLE_REQUEST_FAILED  = 11, /* Controller Role request failed. */
    OFPET_METER_MOD_FAILED     = 12, /* Error in meter. */
    OFPET_TABLE_FEATURES_FAILED = 13, /* Setting table features failed. */
    OFPET_BAD_PROPERTY         = 14, /* Some property is invalid. */
    OFPET_ASYNC_CONFIG_FAILED  = 15, /* Asynchronous config request failed. */
    OFPET_FLOW_MONITOR_FAILED  = 16, /* Setting flow monitor failed. */
    OFPET_BUNDLE_FAILED        = 17, /* Bundle operation failed. */
    OFPET_EXPERIMENTER = 0xffff      /* Experimenter error messages. */
};
```

7.5.4.1 Hello Failed error codes

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 = 0,    /* No compatible version. */
    OFPHFC_EPERM        = 1,    /* Permissions error. */
};
```

 © 2015; The Open Networking Foundation

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

If the OpenFlow protocol version negotiated via the OFPT_HELLO message is not supported by the recipient, then the recipient must reply with an OFPT_ERROR message with a type field of OFPET_HELLO_FAILED, a code field of OFPHFC_INCOMPATIBLE, and optionally an ASCII string explaining the situation in data.

7.5.4.2 Bad Request error codes

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

```
/* ofp_error_msg 'code' values for OFPET_BAD_REQUEST.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_request_code {
    OFPBRC_BAD_VERSION      = 0,  /* ofp_header.version not supported. */
    OFPBRC_BAD_TYPE         = 1,  /* ofp_header.type not supported. */
    OFPBRC_BAD_MULTIPART    = 2,  /* ofp_multipart_request.type not
                                     supported. */
    OFPBRC_BAD_EXPERIMENTER = 3,  /* Experimenter id not supported
                                   * (in ofp_experimenter_header or
                                   * ofp_multipart_request or
                                   * ofp_multipart_reply). */
    OFPBRC_BAD_EXP_TYPE     = 4,  /* Experimenter type not supported. */
    OFPBRC_EPERM            = 5,  /* Permissions error. */
    OFPBRC_BAD_LEN          = 6,  /* Wrong request length for type. */
    OFPBRC_BUFFER_EMPTY     = 7,  /* Specified buffer has already been used. */
    OFPBRC_BUFFER_UNKNOWN   = 8,  /* Specified buffer does not exist. */
    OFPBRC_BAD_TABLE_ID     = 9,  /* Specified table-id invalid or does not
                                   * exist. */
    OFPBRC_IS_SLAVE         = 10, /* Denied because controller is slave. */
    OFPBRC_BAD_PORT         = 11, /* Invalid port or missing port. */
    OFPBRC_BAD_PACKET       = 12, /* Invalid packet in packet-out. */
    OFPBRC_MULTIPART_BUFFER_OVERFLOW    = 13, /* ofp_multipart_request
                                     overflowed the assigned buffer. */
    OFPBRC_MULTIPART_REQUEST_TIMEOUT = 14,  /* Timeout during multipart
                                               request. */
    OFPBRC_MULTIPART_REPLY_TIMEOUT = 15, /* Timeout during multipart reply. */
    OFPBRC_MULTIPART_BAD_SCHED = 16, /* Switch received a OFPMP_BUNDLE_FEATURES
                                        request and failed to update the
                                        scheduling tolerance. */
    OFPBRC_PIPELINE_FIELDS_ONLY = 17, /* Match fields must include only
                                         pipeline fields. */
    OFPBRC_UNKNOWN          = 18, /* Unspecified error. */
};
```

Error message using the codes OFPBRC_MULTIPART_BUFFER_OVERFLOW, OFPBRC_MULTIPART_REQUEST_TIMEOUT and OFPBRC_MULTIPART_REPLY_TIMEOUT must include an empty data field, and the xid field must match that of the offending multipart message sequence.

If a controller with role set to *Slave* sends to the switch a controller-to-switch commands that send packets or modify the state of the switch, the switch must reply with an OFPT_ERROR message with a type field of OFPET_BAD_REQUEST, a code field of OFPBRC_IS_SLAVE.

 © 2015; The Open Networking Foundation

If the `match` field in a *Packet-Out* message contains any header fields, the switch must refuse the request and send an `ofp_error_msg` with `OFPET_BAD_REQUEST` type and `OFPBRC_PIPELINE_FIELDS_ONLY` code.

If an input port is not included in the `match` field of a *Packet-Out* message, the switch must refuse the request and send an `ofp_error_msg` with `OFPET_BAD_REQUEST` type and `OFPBRC_BAD_PORT` code.

7.5.4.3 Bad Action error codes

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

```
/* ofp_error_msg 'code' values for OFPET_BAD_ACTION.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_action_code {
    OFPBAC_BAD_TYPE           = 0,  /* Unknown or unsupported action type. */
    OFPBAC_BAD_LEN            = 1,  /* Length problem in actions. */
    OFPBAC_BAD_EXPERIMENTER   = 2,  /* Unknown experimenter id specified. */
    OFPBAC_BAD_EXP_TYPE       = 3,  /* Unknown action for experimenter id. */
    OFPBAC_BAD_OUT_PORT       = 4,  /* Problem validating output port. */
    OFPBAC_BAD_ARGUMENT       = 5,  /* Bad action argument. */
    OFPBAC_EPERM              = 6,  /* Permissions error. */
    OFPBAC_TOO_MANY           = 7,  /* Can't handle this many actions. */
    OFPBAC_BAD_QUEUE          = 8,  /* Problem validating output queue. */
    OFPBAC_BAD_OUT_GROUP      = 9,  /* Invalid group id in group action. */
    OFPBAC_MATCH_INCONSISTENT = 10, /* Action can't apply for this match,
                                       or Set-Field missing prerequisite. */
    OFPBAC_UNSUPPORTED_ORDER  = 11, /* Action order is unsupported for the
                                     action list in an Apply-Actions instruction */
    OFPBAC_BAD_TAG            = 12, /* Actions uses an unsupported
                                       tag/encap. */
    OFPBAC_BAD_SET_TYPE       = 13, /* Unsupported type in SET_FIELD action. */
    OFPBAC_BAD_SET_LEN        = 14, /* Length problem in SET_FIELD action. */
    OFPBAC_BAD_SET_ARGUMENT   = 15, /* Bad argument in SET_FIELD action. */
    OFPBAC_BAD_SET_MASK       = 16, /* Bad mask in SET_FIELD action. */
    OFPBAC_BAD_METER          = 17, /* Invalid meter id in meter action. */
};
```

If a message specifies an action that is invalid or not supported in the specified context (table, group, packet-out), the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_TYPE` code.

If a set of actions included in a *Write-Actions* instruction contains more than one instance of an action type or set-field action type, the switch may optionally return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_TOO_MANY` code.

If a list of actions included in an *Apply-Actions* instruction contains more actions than the switch supports, then the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_TOO_MANY` code.

If a list of actions contains a sequence of actions that the switch can not support in the specified order, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_UNSUPPORTED_ORDER` code.

 © 2015; The Open Networking Foundation

If any *action* references a port that will never be valid on a switch, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_OUT_PORT` code. If the referenced port may be valid in the future, e.g. when a linecard is added to a chassis switch, or a port is dynamically added to a software switch, the switch must either silently drop packets sent to the referenced port, or immediately return an `OFPBAC_BAD_OUT_PORT` error and refuse the flow mod.

If an *action* in a message references a group that is not currently defined on the switch, or is a reserved group, such as `OFPG_ALL`, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_OUT_GROUP` code.

If an *action* in a message references a meter that is not currently defined on the switch, or is a reserved meter, such as `OFPM_ALL`, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_METER` code.

If an *action* in a message has a value that is invalid, for example a Push action with an invalid Ethertype, and this situation is not covered by another error codes (such as *bad output port*, *bad group id* or *bad set argument*), the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_ARGUMENT` code.

If an *action* in a message performs an operation which is inconsistent with the match and prior actions of the same message, for example, a *pop VLAN* action with a match specifying no VLAN, or a *set TTL* action with a match wildcarding the Ethertype, the switch may optionally reject the flow mod and immediately return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_MATCH_INCONSISTENT` code.

If a *set-field* action in a message has a value that is invalid, for example a set-field action for OXM `VLAN_VID` with value greater than 4095, or a DSCP value using more than 6 bits, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_SET_ARGUMENT` code.

If a *set-field* action in a flow mod message does not have its prerequisites included in the match or prior actions in the same message, for example, a set IPv4 address action with a match wildcarding the Ethertype, the switch must reject the flow mod and immediately return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_MATCH_INCONSISTENT` code. The effect of any inconsistent actions on matched packets is undefined. Controllers are strongly encouraged to avoid generating combinations of table entries that may yield inconsistent actions.

If a *set-field* action or *Copy-Field* action specifies a field or a class that is not supported in the specified table, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_SET_TYPE` code.

If a *Set-Field* action includes an OXM mask (`oxm_hasmask` field set), and the switch does not support masking for this OXM type, the switch must reject the flow mod and immediately return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBMC_BAD_SET_MASK` code.

If the OXM field in a *Set-Field* action includes a mask, if for a 0-bit in `oxm_mask` the corresponding bit in `oxm_value` is a 1-bit, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_SET_MASK` code.

If a *Copy-Field* action specifies invalid offsets, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_SET_ARGUMENT` code.

 © 2015; The Open Networking Foundation

If a *Copy-Field* action specifies overlapping source and destination and the switch does not support such operation, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_SET_TYPE` code.

7.5.4.4 Bad Instruction error codes

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

```
/* ofp_error_msg 'code' values for OFPET_BAD_INSTRUCTION.  'data' contains at
 * least the first 64 bytes of the failed request. */
enum ofp_bad_instruction_code {
    OFPBIC_UNKNOWN_INST     = 0, /* Unknown instruction. */
    OFPBIC_UNSUP_INST       = 1, /* Switch or table does not support the
                                    instruction. */
    OFPBIC_BAD_TABLE_ID     = 2, /* Invalid Table-ID specified. */
    OFPBIC_UNSUP_METADATA   = 3, /* Metadata value unsupported by datapath. */
    OFPBIC_UNSUP_METADATA_MASK = 4, /* Metadata mask value unsupported by
                                       datapath. */
    OFPBIC_BAD_EXPERIMENTER = 5, /* Unknown experimenter id specified. */
    OFPBIC_BAD_EXP_TYPE     = 6, /* Unknown instruction for experimenter id. */
    OFPBIC_BAD_LEN          = 7, /* Length problem in instructions. */
    OFPBIC_EPERM            = 8, /* Permissions error. */
    OFPBIC_DUP_INST         = 9, /* Duplicate instruction. */
};
```

If the instructions included in a message are unknown, the switch must return an `ofp_error_msg` with `OFPET_BAD_INSTRUCTION` type and `OFPBIC_UNKNOWN_INST` code.

If the instructions included in a message are unsupported the switch must return an `ofp_error_msg` with `OFPET_BAD_INSTRUCTION` type and `OFPBIC_UNSUP_INST` code.

If the instructions included in a set of instructions in a message contain two or more instructions with the same type, the switch must return an `ofp_error_msg` with `OFPET_BAD_INSTRUCTION` type and `OFPBIC_DUP_INST` code.

If the instructions included in a message contain a *Goto-Table* and the next-table-id refers to an invalid table the switch must return an `ofp_error_msg` with `OFPET_BAD_INSTRUCTION` type and `OFPBIC_BAD_TABLE_ID` code.

If the instructions requested contain a *Write-Metadata* and the metadata value or metadata mask value is unsupported then the switch must return an `ofp_error_msg` with `OFPET_BAD_INSTRUCTION` type and `OFPBIC_UNSUP_METADATA` or `OFPBIC_UNSUP_METADATA_MASK` code.

If a *Clear-Actions* instruction contains some actions, the switch must return an `ofp_error_msg` with `OFPET_BAD_INSTRUCTION` type and `OFPBIC_BAD_LEN` code.

 © 2015; The Open Networking Foundation

7.5.4.5 Bad Match error codes

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

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

If the match in a message specifies a field or a class that is not supported in the specified table, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_FIELD code.

If the match in a message specifies a field more than once, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_DUP_FIELD code.

If the match in a message specifies a field but fails to specify its associated prerequisites (see 7.2.3.6), for example specifies an IPv4 address without matching the EtherType to 0x800, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_PREREQ code.

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

If the match in a message specifies values that cannot be matched, for example, a VLAN ID greater than 4095 and not one of the reserved values, or a DSCP value using more than 6 bits, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_VALUE code.

If an OXM field in the match includes a mask, if for a 0-bit in oxm_mask the corresponding bit in oxm_value is a 1-bit, the switch must return an ofp_error_msg with OFPET_BAD_MATCH type and OFPBMC_BAD_WILDCARDS code.

 © 2015; The Open Networking Foundation

If a two OXM fields in the match have the same type, the switch should return an `ofp_error_msg` with `OFPET_BAD_MATCH` type and `OFPBMC_DUP_FIELD` code.

7.5.4.6 Flow-mod Failed error codes

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_UNKNOWN      = 0,   /* Unspecified error. */
    OFPFMFC_TABLE_FULL   = 1,   /* Flow not added because table was full. */
    OFPFMFC_BAD_TABLE_ID = 2,   /* Table does not exist */
    OFPFMFC_OVERLAP      = 3,   /* Attempted to add overlapping flow with
                                   CHECK_OVERLAP flag set. */
    OFPFMFC_EPERM        = 4,   /* Permissions error. */
    OFPFMFC_BAD_TIMEOUT  = 5,   /* Flow not added because of unsupported
                                   idle/hard timeout. */
    OFPFMFC_BAD_COMMAND  = 6,   /* Unsupported or unknown command. */
    OFPFMFC_BAD_FLAGS    = 7,   /* Unsupported or unknown flags. */
    OFPFMFC_CANT_SYNC    = 8,   /* Problem in table synchronisation. */
    OFPFMFC_BAD_PRIORITY = 9,   /* Unsupported priority value. */
    OFPFMFC_IS_SYNC      = 10,  /* Synchronised flow entry is read only. */
};
```

If a switch cannot find any space in the requested table in which to add the flow entry in a *Flow-Mod* message, the switch must send an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_TABLE_FULL` code.

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

If a *Flow-Mod* message with an *add* request has the `OFPFF_CHECK_OVERLAP` flag set and the request has an overlap conflict (see 6.4), the switch must reply with an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_OVERLAP` code.

If a flow entry is added or modified in a source flow table, and if another flow table is synchronised on that source flow table and the switch can't synchronise a flow mod request, the switch must reject the flow mod and immediately return an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_CANT_SYNC` code.

If a flow entry is added in a source flow table with the `OFPFF_CHECK_OVERLAP` flag set, and if another flow table is synchronised on that source flow table and the corresponding flow entry would overlap a flow entry already residing in the synchronised flow table, the switch must refuse the full flow-mod request and respond with an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_CANT_SYNC` code.

If the controller attempts to add, modify or delete a flow entry in a synchronised flow table and the switch can't support such changes, the switch must reject the flow mod and immediately return an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMFC_IS_SYNC` code.

 © 2015; The Open Networking Foundation

If any error which can not be mapped to an existing error code occurs during the processing of the flow mod message, the switch may return an `ofp_error_msg` with `OFPET_FLOW_MOD_FAILED` type and `OFPFMC_UNKNOWN` code.

7.5.4.7 Group-mod Failed error codes

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

```
/* ofp_error_msg 'code' values for OFPET_GROUP_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_group_mod_failed_code {
    OFPGMFC_GROUP_EXISTS         = 0,  /* Group not added because a group ADD
                                          attempted to replace an
                                          already-present group. */
    OFPGMFC_INVALID_GROUP        = 1,  /* Group not added because Group
                                          specified is invalid. */
    OFPGMFC_WEIGHT_UNSUPPORTED   = 2,  /* Switch does not support unequal load
                                          sharing with select groups. */
    OFPGMFC_OUT_OF_GROUPS        = 3,  /* The group table is full. */
    OFPGMFC_OUT_OF_BUCKETS       = 4,  /* The maximum number of action buckets
                                          for a group has been exceeded. */
    OFPGMFC_CHAINING_UNSUPPORTED = 5,  /* Switch does not support groups that
                                          forward to groups. */
    OFPGMFC_WATCH_UNSUPPORTED    = 6,  /* This group cannot watch the watch_port
                                          or watch_group specified. */
    OFPGMFC_LOOP                 = 7,  /* Group entry would cause a loop. */
    OFPGMFC_UNKNOWN_GROUP        = 8,  /* Group not modified because a group
                                          MODIFY attempted to modify a
                                          non-existent group. */
    OFPGMFC_CHAINED_GROUP        = 9,  /* Group not deleted because another
                                          group is forwarding to it. */
    OFPGMFC_BAD_TYPE             = 10, /* Unsupported or unknown group type. */
    OFPGMFC_BAD_COMMAND          = 11, /* Unsupported or unknown command. */
    OFPGMFC_BAD_BUCKET           = 12, /* Error in bucket. */
    OFPGMFC_BAD_WATCH            = 13, /* Error in watch port/group. */
    OFPGMFC_EPERM                = 14, /* Permissions error. */
    OFPGMFC_UNKNOWN_BUCKET       = 15, /* Invalid bucket identifier used in
                                          INSERT BUCKET or REMOVE BUCKET
                                          command. */
    OFPGMFC_BUCKET_EXISTS        = 16, /* Can't insert bucket because a bucket
                                          already exist with that bucket-id. */
};
```

If a Group-Mod request with command **add** specifies a group identifier that already resides in the group table, then the switch must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_GROUP_EXISTS` code.

If a Group-Mod request with command **modify** or **insert bucket** specifies a group identifier that does not already exist, then the switch must refuse the group mod and send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_UNKNOWN_GROUP` code.

 © 2015; The Open Networking Foundation

If a specified group type is invalid or not supported by the switch, then the switch must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_BAD_TYPE` code.

If the group-mod request specifies more than one bucket for a group of type *Indirect*, the switch must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_INVALID_GROUP` code.

If a specified group is invalid (ie: specifies a non-zero `weight` for a group which type is not select) then the switch must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_INVALID_GROUP` code.

If a switch does not support unequal load sharing with select groups (buckets with weight different than 1), it must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_WEIGHT_UNSUPPORTED` code.

If a switch cannot add the incoming group entry due to lack of space, the switch must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_OUT_OF_GROUPS` code.

If a switch cannot add the incoming group entry due to restrictions (hardware or otherwise) limiting the number of group buckets, it must refuse to add the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_OUT_OF_BUCKETS` code.

If a switch cannot add the incoming group because it does not support the proposed liveliness configuration, the switch must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_WATCH_UNSUPPORTED` code. This includes specifying `watch_port` or `watch_group` for a group that does not support liveness, or specifying a port that does not support liveness in `watch_port`, or specifying a group that does not support liveness in `watch_group`.

If a Group-Mod request with command **delete** includes some buckets in the *Group-Mod* message, then the switch must return an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_INVALID_GROUP` code.

For Group-Mod requests with command **insert bucket** or **remove bucket**, if the `command_bucket_id` field value does not already exist in the specified group or is not one of the reserved values for that command, then the switch must refuse the group mod and send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_UNKNOWN_BUCKET` code.

For Group-Mod requests with command **remove bucket**, if the current action bucket list is empty, the switch must refuse the group mod and send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_UNKNOWN_BUCKET` code.

For Group-Mod requests with command **insert bucket**, if a group request would make the group have a duplicate Bucket ID, the switch must refuse the group mod and send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_BUCKET_EXISTS` code.

If a switch does not support group chaining or groups referencing groups, it must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_CHAINING_UNSUPPORTED` code.

If an action in a bucket references a group that is not currently defined on the switch, or is a reserved group, such as `OFPG_ALL`, the switch must return an `ofp_error_msg` with `OFPET_BAD_ACTION` type and `OFPBAC_BAD_OUT_GROUP` code.

 © 2015; The Open Networking Foundation

If a switch supports group chaining and checking for loops, and if the group-mod would create a forwarding loop, the switch must reject the group mod and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_LOOP` code.

If a switch supports group chaining and checking that chained groups are not removed, and if the group-mod would delete a group referenced by another group, the switch must refuse to delete the group entry and must send an `ofp_error_msg` with `OFPET_GROUP_MOD_FAILED` type and `OFPGMFC_CHAINED_GROUP` code.

7.5.4.8 Port-mod Failed error codes

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      = 0,   /* Specified port number does not exist. */
    OFPPMFC_BAD_HW_ADDR   = 1,   /* Specified hardware address does not
                                  * match the port number. */
    OFPPMFC_BAD_CONFIG    = 2,   /* Specified config is invalid. */
    OFPPMFC_BAD_ADVERTISE = 3,   /* Specified advertise is invalid. */
    OFPPMFC_EPERM         = 4,   /* Permissions error. */
};
```

7.5.4.9 Table-mod Failed error codes

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

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

7.5.4.10 Queue-op Failed error codes

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   = 0,     /* Invalid port (or port does not exist). */
    OFPQOFC_BAD_QUEUE  = 1,     /* Queue does not exist. */
    OFPQOFC_EPERM      = 2,     /* Permissions error. */
};
```

 © 2015; The Open Networking Foundation

7.5.4.11 Switch Config Failed error codes

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

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

7.5.4.12 Role Request Failed error codes

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

```
/* ofp_error_msg 'code' values for OFPET_ROLE_REQUEST_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_role_request_failed_code {
    OFPRRFC_STALE      = 0,      /* Stale Message: old generation_id. */
    OFPRRFC_UNSUP      = 1,      /* Controller role change unsupported. */
    OFPRRFC_BAD_ROLE   = 2,      /* Invalid role. */
    OFPRRFC_ID_UNSUP   = 3,      /* Switch doesn't support changing ID. */
    OFPRRFC_ID_IN_USE  = 4,      /* Requested ID is in use. */
};
```

If the switch receives a OFPT_ROLE_REQUEST with role equal to OFPCR_ROLE_MASTER or OFPCR_ROLE_SLAVE and with a generation_id smaller than a previously seen generation id, the switch must reply with an error message with type OFPET_ROLE_REQUEST_FAILED and code OFPRRFC_STALE.

7.5.4.13 Meter-mod Failed error codes

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

```
/* ofp_error_msg 'code' values for OFPET_METER_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_meter_mod_failed_code {
    OFPMMFC_UNKNOWN       = 0,  /* Unspecified error. */
    OFPMMFC_METER_EXISTS  = 1,  /* Meter not added because a Meter ADD
                                 * attempted to replace an existing Meter. */
    OFPMMFC_INVALID_METER = 2,  /* Meter not added because Meter specified
                                 * is invalid,
                                 * or invalid meter in meter action. */
    OFPMMFC_UNKNOWN_METER = 3,  /* Meter not modified because a Meter MODIFY
                                 * attempted to modify a non-existent Meter,
                                 * or bad meter in meter action. */
    OFPMMFC_BAD_COMMAND   = 4,  /* Unsupported or unknown command. */
    OFPMMFC_BAD_FLAGS     = 5,  /* Flag configuration unsupported. */
    OFPMMFC_BAD_RATE      = 6,  /* Rate unsupported. */
    OFPMMFC_BAD_BURST     = 7,  /* Burst size unsupported. */
```

 © 2015; The Open Networking Foundation

```
    OFPMMFC_BAD_BAND      = 8,  /* Band unsupported. */
    OFPMMFC_BAD_BAND_VALUE = 9, /* Band value unsupported. */
    OFPMMFC_OUT_OF_METERS = 10, /* No more meters available. */
    OFPMMFC_OUT_OF_BANDS  = 11, /* The maximum number of properties
                                 * for a meter has been exceeded. */
};
```

If a Meter-Mod request with command **add** specifies a meter identifier that already exists, then the switch must refuse to add the meter and must send an `ofp_error_msg` with `OFPET_METER_MOD_FAILED` type and `OFPMMFC_METER_EXISTS` code.

If a Meter-Mod request with command **modify** specifies a meter identifier that does not already exist, then the switch must refuse the meter mod and send an `ofp_error_msg` with `OFPET_METER_MOD_FAILED` type and `OFPMMFC_UNKNOWN_METER` code.

If a switch cannot add or modify the incoming meter entry due to lack of space, the switch must send an `ofp_error_msg` with `OFPET_METER_MOD_FAILED` type and `OFPMMFC_OUT_OF_METERS` code.

If a switch cannot add or modify the incoming meter entry due to restrictions (hardware or otherwise) limiting the number of bands, it must refuse to add the meter entry and must send an `ofp_error_msg` with `OFPET_METER_MOD_FAILED` type and `OFPMMFC_OUT_OF_BANDS` code.

7.5.4.14 Table-features Failed error codes

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

```
/* ofp_error_msg 'code' values for OFPET_TABLE_FEATURES_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_table_features_failed_code {
    OFPTFFC_BAD_TABLE    = 0,      /* Specified table does not exist. */
    OFPTFFC_BAD_METADATA = 1,      /* Invalid metadata mask. */
    OFPTFFC_EPERM        = 5,      /* Permissions error. */
    OFPTFFC_BAD_CAPA     = 6,      /* Invalid capability field. */
    OFPTFFC_BAD_MAX_ENT  = 7,      /* Invalid max_entries field. */
    OFPTFFC_BAD_FEATURES = 8,      /* Invalid features field. */
    OFPTFFC_BAD_COMMAND  = 9,      /* Invalid command. */
    OFPTFFC_TOO_MANY     = 10,     /* Can't handle this many flow tables. */
};
```

If the switch is unable to set the requested configuration from the table feature request in its entirety, an error of type `OFPET_TABLE_FEATURES_FAILED` is returned with the appropriate error code.

If a switch does not support the requested table features request command, an error of type `OFPET_TABLE_FEATURES_FAILED` and code `OFPTFFC_BAD_COMMAND` is returned.

For commands other than `OFPTFC_REPLACE`, if the number of flow tables is greater than the switch supports, an error of type `OFPET_TABLE_FEATURES_FAILED` and code `OFPTFFC_TOO_MANY` is returned.

 © 2015; The Open Networking Foundation

7.5.4.15 Bad Property error codes

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

```
/* ofp_error_msg 'code' values for OFPET_BAD_PROPERTY. 'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_property_code {
    OFPBPC_BAD_TYPE           = 0,  /* Unknown or unsupported property type. */
    OFPBPC_BAD_LEN            = 1,  /* Length problem in property. */
    OFPBPC_BAD_VALUE          = 2,  /* Unsupported property value. */
    OFPBPC_TOO_MANY           = 3,  /* Can't handle this many properties. */
    OFPBPC_DUP_TYPE           = 4,  /* A property type was duplicated. */
    OFPBPC_BAD_EXPERIMENTER   = 5,  /* Unknown experimenter id specified. */
    OFPBPC_BAD_EXP_TYPE       = 6,  /* Unknown exp_type for experimenter id. */
    OFPBPC_BAD_EXP_VALUE      = 7,  /* Unknown value for experimenter id. */
    OFPBPC_EPERM              = 8,  /* Permissions error. */
};
```

7.5.4.16 Async Config Failed error codes

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

```
/* ofp_error_msg 'code' values for OFPET_ASYNC_CONFIG_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_async_config_failed_code {
    OFPACFC_INVALID      = 0,      /* One mask is invalid. */
    OFPACFC_UNSUPPORTED  = 1,      /* Requested configuration not supported. */
    OFPACFC_EPERM        = 2,      /* Permissions error. */
};
```

7.5.4.17 Flow-monitor Failed error codes

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

```
/* ofp_error_msg 'code' values for OFPET_FLOW_MONITOR_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_flow_monitor_failed_code {
    OFPMOFC_UNKNOWN       = 0,  /* Unspecified error. */
    OFPMOFC_MONITOR_EXISTS = 1, /* Monitor not added because a Monitor ADD
                                 * attempted to replace an existing Monitor. */
    OFPMOFC_INVALID_MONITOR = 2, /* Monitor not added because Monitor specified
                                 * is invalid. */
    OFPMOFC_UNKNOWN_MONITOR = 3, /* Monitor not modified because a Monitor
                                   MODIFY attempted to modify a non-existent
                                   Monitor. */
    OFPMOFC_BAD_COMMAND   = 4,  /* Unsupported or unknown command. */
    OFPMOFC_BAD_FLAGS     = 5,  /* Flag configuration unsupported. */
    OFPMOFC_BAD_TABLE_ID  = 6,  /* Specified table does not exist. */
    OFPMOFC_BAD_OUT       = 7,  /* Error in output port/group. */
};
```

 © 2015; The Open Networking Foundation

7.5.4.18 Bundle Failed error codes

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

```
/* ofp_error_msg 'code' values for OFPET_BUNDLE_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_bundle_failed_code {
  OFPBFC_UNKNOWN        = 0,  /* Unspecified error. */
  OFPBFC_EPERM          = 1,  /* Permissions error. */
  OFPBFC_BAD_ID         = 2,  /* Bundle ID doesn't exist. */
  OFPBFC_BUNDLE_EXIST   = 3,  /* Bundle ID already exist. */
  OFPBFC_BUNDLE_CLOSED  = 4,  /* Bundle ID is closed. */
  OFPBFC_OUT_OF_BUNDLES = 5,  /* Too many bundles IDs. */
  OFPBFC_BAD_TYPE       = 6,  /* Unsupported or unknown message control type. */
  OFPBFC_BAD_FLAGS      = 7,  /* Unsupported, unknown, or inconsistent flags. */
  OFPBFC_MSG_BAD_LEN    = 8,  /* Length problem in included message. */
  OFPBFC_MSG_BAD_XID    = 9,  /* Inconsistent or duplicate XID. */
  OFPBFC_MSG_UNSUP      = 10, /* Unsupported message in this bundle. */
  OFPBFC_MSG_CONFLICT   = 11, /* Unsupported message combination in this
                                 bundle. */
  OFPBFC_MSG_TOO_MANY   = 12, /* Can't handle this many messages in bundle. */
  OFPBFC_MSG_FAILED     = 13, /* One message in bundle failed. */
  OFPBFC_TIMEOUT        = 14, /* Bundle is taking too long. */
  OFPBFC_BUNDLE_IN_PROGRESS = 15,  /* Bundle is locking the resource. */
  OFPBFC_SCHED_NOT_SUPPORTED = 16, /* Scheduled commit was received and
                                      scheduling is not supported. */
  OFPBFC_SCHED_FUTURE = 17,   /* Scheduled commit time exceeds upper bound. */
  OFPBFC_SCHED_PAST = 18,     /* Scheduled commit time exceeds lower bound. */
};
```

If the bundle_id in a *Bundle Open Request* refers to an existing bundle attached to the same connection, the switch must refuse to open the new bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_ID code. The existing bundle identified by bundle_id must be discarded.

If the switch can not open the bundle specified in a *Bundle Open Request* because it is having too many opened bundles on the switch or attached to the current connection, the switch must refuse to open the new bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_OUT_OF_BUNDLES code.

If the switch can not open the bundle specified in a *Bundle Open Request* because the connection is using an unreliable transport, the switch must refuse to open the new bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_OUT_OF_BUNDLES code.

If the flags field specified in a *Bundle Request* specifies some feature that can not be implemented by the switch, the switch must refuse to open the new bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BAD_FLAGS code.

If the bundle_id in a *Bundle Add Request* refers to a bundle that is already closed, the switch must refuse to add the message to the bundle, discard the bundle and send an ofp_error_msg with OFPET_BUNDLE_FAILED type and OFPBFC_BUNDLE_CLOSED code.

If the flags field in a *Bundle Add Request*, *Bundle Close Request* or *Bundle Commit Request* is different from the flags that were specified when the bundle was opened, the switch must refuse to add the message

 © 2015; The Open Networking Foundation

to the bundle, discard the bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_BAD_FLAGS` code.

If the message in a *Bundle Add Request* is normally supported on the switch but is not supported in a bundle, the switch must refuse to add the message to the bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_MSG_UNSUP` code. This is the case for hello, echo, and bundle messages, messages that are not requests (for example replies, status and errors messages), or if the implementation does not support including a specific modification message in a bundle.

If the message in a *Bundle Add Request* is incompatible with another message already stored in the bundle, the switch must refuse to add the message to the bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_MSG_CONFLICT` code.

If the bundle is full and can not fit the message in the *Bundle Add Request*, the switch must refuse to add the message to the bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_MSG_TOO_MANY` code.

If the message in the *Bundle Add Request* does not have a valid `length` field, the switch must refuse to add the message to the bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_MSG_BAD_LEN` code.

If the `bundle_id` in a *Bundle Close Request* or *Bundle Commit Request* refers to a bundle that does not exist, the switch must reject the request and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_BAD_ID` code.

If the `bundle_id` in a *Bundle Close Request* refers to a bundle that is already closed, the switch must refuse to close to the bundle, discard the bundle and send an `ofp_error_msg` with `OFPET_BUNDLE_FAILED` type and `OFPBFC_BUNDLE_CLOSED` code.

7.5.4.19 Experimenter Error Message

For the `OFPET_EXPERIMENTER` error `type`, the error message is defined by the following structure and fields:

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

    uint16_t type;            /* OFPET_EXPERIMENTER. */
    uint16_t exp_code;        /* Experimenter defined. */
    uint32_t experimenter;    /* Experimenter ID. */
    uint8_t data[0];          /* Variable-length data.  Interpreted based
                                 on the type and experimenter.  No padding. */
};
OFP_ASSERT(sizeof(struct ofp_error_experimenter_msg) == 16);
```

The `experimenter` field is the Experimenter ID , which takes the same form as in the typical experimenter structure (see 7.2.8). The `exp_code` field is the experimenter code, a 16 bit field managed by the experimenter and it can have any value and function the experimenter wishes. The `data` field is experimenter defined and does not need to be padded.

 © 2015; The Open Networking Foundation

7.5.5 Experimenter Message

The Experimenter message is defined as follows:

```
/* Experimenter extension message. */
struct ofp_experimenter_msg {
    struct ofp_header header;   /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter;      /* Experimenter ID:
                                 * - MSB 0: low-order bytes are IEEE OUI.
                                 * - MSB != 0: defined by ONF. */
    uint32_t exp_type;          /* Experimenter defined. */
    /* Experimenter-defined arbitrary additional data. */
    uint8_t  experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_experimenter_msg) == 16);
```

The `experimenter` field identifies the experimenter, the `exp_type` field is the experimenter type, and `experimenter_data` is the exprimenter arbitrary data. They take the same form as in the typical experimenter structure (see 7.2.8). The Experimenter message does not need to be padded.

As opposed to other Experimenter extensions, Experimenter messages are not associated with a specific OpenFlow object, and therefore can be used to create entirely new APIs and manage entirely new objects.

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

Appendix A Header file openflow.h

The file `openflow.h` contains all the structures, defines, and enumerations used by the OpenFlow protocol. The version of `openflow.h` defined for the present version of the specification 1.5.1 is both included below and attached to this document here (not available in all PDF reader).


```
/* Copyright (c) 2008 The Board of Trustees of The Leland Stanford
 * Junior University
 * Copyright (c) 2011, 2013, 2014 Open Networking Foundation
 *
 * We are making the OpenFlow specification and associated documentation
 * (Software) available for public use and benefit with the expectation
 * that others will use, modify and enhance the Software and contribute
 * those enhancements back to the community. However, since we would
 * like to make the Software available for broadest use, with as few
 * restrictions as possible permission is hereby granted, free of
 * charge, to any person obtaining a copy of this Software to deal in
 * the Software under the copyrights without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Software, and to
 * permit persons to whom the Software is furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
```


 © 2015; The Open Networking Foundation

```
 * SOFTWARE.
 *
 * The name and trademarks of copyright holder(s) may NOT be used in
 * advertising or publicity pertaining to the Software or any
 * derivatives without specific, written prior permission.
 */

/* OpenFlow: protocol between controller and datapath. */

#ifndef OPENFLOW_OPENFLOW_H
#define OPENFLOW_OPENFLOW_H 1

#ifdef __KERNEL__
#include <linux/types.h>
#else
#include <stdint.h>
#endif

#ifdef SWIG
#define OFP_ASSERT(EXPR)        /* SWIG can't handle OFP_ASSERT. */
#elif !defined(__cplusplus)
/* Build-time assertion for use in a declaration context. */
#define OFP_ASSERT(EXPR)                                                \
        extern int (*build_assert(void))[ sizeof(struct {               \
                    unsigned int build_assert_failed : (EXPR) ? 1 : -1; })]
#else /* __cplusplus */
#define OFP_ASSERT(_EXPR) typedef int build_assert_failed[(_EXPR) ? 1 : -1]
#endif /* __cplusplus */

#ifndef SWIG
#define OFP_PACKED __attribute__((packed))
#else
#define OFP_PACKED              /* SWIG doesn't understand __attribute. */
#endif

/* Version number:
 * OpenFlow versions released: 0x01 = 1.0 ; 0x02 = 1.1 ; 0x03 = 1.2
 *     0x04 = 1.3.X ; 0x05 = 1.4.X ; 0x06 = 1.5.X
 */
/* The most significant bit in the version field is reserved and must
 * be set to zero.
 */
#define OFP_VERSION   0x06

#define OFP_MAX_TABLE_NAME_LEN 32
#define OFP_MAX_PORT_NAME_LEN  16

/* Official IANA registered port for OpenFlow. */
#define OFP_TCP_PORT  6653
#define OFP_SSL_PORT  6653

#define OFP_ETH_ALEN 6          /* Bytes in an Ethernet address. */

/* Port numbering. Ports are numbered starting from 1. */
enum ofp_port_no {
    /* Maximum number of physical and logical switch ports. */
    OFPP_MAX        = 0xffffff00,

    /* Reserved OpenFlow Port (fake output "ports"). */
    OFPP_UNSET      = 0xfffffff7,  /* Output port not set in action-set.
                                      used only in OXM_OF_ACTSET_OUTPUT. */
    OFPP_IN_PORT    = 0xfffffff8,  /* Send the packet out the input port.  This
                                      reserved port must be explicitly used
                                      in order to send back out of the input
                                      port. */
    OFPP_TABLE      = 0xfffffff9,  /* Submit the packet to the first flow table
                                      NB: This destination port can only be
                                      used in packet-out messages. */
    OFPP_NORMAL     = 0xfffffffa,  /* Forward using non-OpenFlow pipeline. */
    OFPP_FLOOD      = 0xfffffffb,  /* Flood using non-OpenFlow pipeline. */
    OFPP_ALL        = 0xfffffffc,  /* All standard ports except input port. */
    OFPP_CONTROLLER = 0xfffffffd,  /* Send to controller. */
    OFPP_LOCAL      = 0xfffffffe,  /* Local openflow "port". */
    OFPP_ANY        = 0xffffffff   /* Special value used in some requests when
                                      no port is specified (i.e. wildcarded). */
};

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

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

 © 2015; The Open Networking Foundation

```
    OFPT_SET_CONFIG         = 9,  /* Controller/switch message */

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

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

    /* Multipart messages. */
    OFPT_MULTIPART_REQUEST      = 18, /* Controller/switch message */
    OFPT_MULTIPART_REPLY        = 19, /* Controller/switch message */

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

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

    /* Asynchronous message configuration. */
    OFPT_GET_ASYNC_REQUEST  = 26, /* Controller/switch message */
    OFPT_GET_ASYNC_REPLY    = 27, /* Controller/switch message */
    OFPT_SET_ASYNC          = 28, /* Controller/switch message */

    /* Meters and rate limiters configuration messages. */
    OFPT_METER_MOD          = 29, /* Controller/switch message */

    /* Controller role change event messages. */
    OFPT_ROLE_STATUS        = 30, /* Async message */

    /* Asynchronous messages. */
    OFPT_TABLE_STATUS       = 31, /* Async message */

    /* Request forwarding by the switch. */
    OFPT_REQUESTFORWARD     = 32, /* Async message */

    /* Bundle operations (multiple messages as a single operation). */
    OFPT_BUNDLE_CONTROL     = 33, /* Controller/switch message */
    OFPT_BUNDLE_ADD_MESSAGE = 34, /* Controller/switch message */

    /* Controller Status async message. */
    OFPT_CONTROLLER_STATUS  = 35, /* Async message */

};

/* 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);

/* Hello elements types.
 */
enum ofp_hello_elem_type {
    OFPHET_VERSIONBITMAP          = 1,  /* Bitmap of version supported. */
};

/* Common header for all Hello Elements */
struct ofp_hello_elem_header {
    uint16_t         type;    /* One of OFPHET_*. */
    uint16_t         length;  /* Length in bytes of the element,
                                 including this header, excluding padding. */
};
OFP_ASSERT(sizeof(struct ofp_hello_elem_header) == 4);

/* Version bitmap Hello Element */
struct ofp_hello_elem_versionbitmap {
    uint16_t         type;    /* OFPHET_VERSIONBITMAP. */
    uint16_t         length;  /* Length in bytes of this element,
                                 including this header, excluding padding. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the bitmaps, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         bitmaps[0];   /* List of bitmaps - supported versions */
};
OFP_ASSERT(sizeof(struct ofp_hello_elem_versionbitmap) == 4);
```

 © 2015; The Open Networking Foundation

```
/* OFPT_HELLO.  This message includes zero or more hello elements having
 * variable size. Unknown elements types must be ignored/skipped, to allow
 * for future extensions. */
struct ofp_hello {
    struct ofp_header header;

    /* Hello element list */
    struct ofp_hello_elem_header elements[0]; /* List of elements - 0 or more */
};
OFP_ASSERT(sizeof(struct ofp_hello) == 8);

#define OFP_DEFAULT_MISS_SEND_LEN   128

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

/* Switch configuration. */
struct ofp_switch_config {
    struct ofp_header header;
    uint16_t flags;             /* Bitmap of OFPC_* flags. */
    uint16_t miss_send_len;     /* Max bytes of packet that datapath
                                   should send to the controller. See
                                   ofp_controller_max_len for valid values.
                                   */
};
OFP_ASSERT(sizeof(struct ofp_switch_config) == 12);

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

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

/* Flags to configure the table. */
enum ofp_table_config {
    OFPTC_DEPRECATED_MASK    = 3,       /* Deprecated bits */
    OFPTC_EVICTION           = 1 << 2,  /* Authorise table to evict flows. */
    OFPTC_VACANCY_EVENTS     = 1 << 3,  /* Enable vacancy events. */
};

/* Table Mod property types.
 */
enum ofp_table_mod_prop_type {
    OFPTMPT_EVICTION               = 0x2,    /* Eviction property. */
    OFPTMPT_VACANCY                = 0x3,    /* Vacancy property. */
    OFPTMPT_EXPERIMENTER           = 0xFFFF, /* Experimenter property. */
};

/* Common header for all Table Mod Properties */
struct ofp_table_mod_prop_header {
    uint16_t         type;    /* One of OFPTMPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_table_mod_prop_header) == 4);

/* Eviction flags. */
enum ofp_table_mod_prop_eviction_flag {
    OFPTMPEF_OTHER           = 1 << 0,     /* Using other factors. */
    OFPTMPEF_IMPORTANCE      = 1 << 1,     /* Using flow entry importance. */
    OFPTMPEF_LIFETIME        = 1 << 2,     /* Using flow entry lifetime. */
};

/* Eviction table mod Property. Mostly used in OFPMP_TABLE_DESC replies. */
struct ofp_table_mod_prop_eviction {
    uint16_t         type;    /* OFPTMPT_EVICTION. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         flags;   /* Bitmap of OFPTMPEF_* flags */
};
OFP_ASSERT(sizeof(struct ofp_table_mod_prop_eviction) == 8);

/* Vacancy table mod property */
struct ofp_table_mod_prop_vacancy {
    uint16_t         type;   /* OFPTMPT_VACANCY. */
    uint16_t         length; /* Length in bytes of this property. */
    uint8_t vacancy_down;    /* Vacancy threshold when space decreases (%). */
    uint8_t vacancy_up;      /* Vacancy threshold when space increases (%). */
    uint8_t vacancy;         /* Current vacancy (%) - only in ofp_table_desc. */
    uint8_t pad[1];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_table_mod_prop_vacancy) == 8);
```

 © 2015; The Open Networking Foundation

```
/* Experimenter table mod property */
struct ofp_table_mod_prop_experimenter {
    uint16_t         type;    /* OFPTMPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_table_mod_prop_experimenter) == 12);

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

    /* Table Mod Property list */
    struct ofp_table_mod_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_table_mod) == 16);

/* Capabilities supported by the datapath. */
enum ofp_capabilities {
    OFPC_FLOW_STATS     = 1 << 0,  /* Flow statistics. */
    OFPC_TABLE_STATS    = 1 << 1,  /* Table statistics. */
    OFPC_PORT_STATS     = 1 << 2,  /* Port statistics. */
    OFPC_GROUP_STATS    = 1 << 3,  /* Group statistics. */
    OFPC_IP_REASM       = 1 << 5,  /* Can reassemble IP fragments. */
    OFPC_QUEUE_STATS    = 1 << 6,  /* Queue statistics. */
    OFPC_PORT_BLOCKED   = 1 << 8,  /* Switch will block looping ports. */
    OFPC_BUNDLES        = 1 << 9,  /* Switch supports bundles. */
    OFPC_FLOW_MONITORING = 1 << 10,  /* Switch supports flow monitoring. */
};

/* Flags to indicate behavior of the physical port.  These flags are
 * used in ofp_port to describe the current configuration.  They are
 * used in the ofp_port_mod message to configure the port's behavior.
 */
enum ofp_port_config {
    OFPPC_PORT_DOWN    = 1 << 0,  /* Port is administratively down. */

    OFPPC_NO_RECV      = 1 << 2,  /* Drop all packets received by port. */
    OFPPC_NO_FWD       = 1 << 5,  /* Drop packets forwarded to port. */
    OFPPC_NO_PACKET_IN = 1 << 6   /* Do not send packet-in msgs for port. */
};

/* Current state of the physical port.  These are not configurable from
 * the controller.
 */
enum ofp_port_state {
    OFPPS_LINK_DOWN    = 1 << 0,  /* No physical link present. */
    OFPPS_BLOCKED      = 1 << 1,  /* Port is blocked */
    OFPPS_LIVE         = 1 << 2,  /* Live for Fast Failover Group. */
};

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

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

/* Port description property types.
 */
enum ofp_port_desc_prop_type {
    OFPPDPT_ETHERNET          = 0,      /* Ethernet property. */
    OFPPDPT_OPTICAL           = 1,      /* Optical property. */
    OFPPDPT_PIPELINE_INPUT    = 2,      /* Ingress pipeline fields. */
    OFPPDPT_PIPELINE_OUTPUT   = 3,      /* Egress pipeline fields. */
```

 © 2015; The Open Networking Foundation

```
    OFPPDPT_RECIRCULATE       = 4,      /* Recirculation property. */
    OFPPDPT_EXPERIMENTER      = 0xFFFF, /* Experimenter property. */
};

/* Common header for all port description properties. */
struct ofp_port_desc_prop_header {
    uint16_t         type;    /* One of OFPPDPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_header) == 4);

/* Ethernet port description property. */
struct ofp_port_desc_prop_ethernet {
    uint16_t         type;    /* OFPPDPT_ETHERNET. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint8_t          pad[4];  /* Align to 64 bits. */
    /* Bitmaps of OFPPF_* that describe features.  All bits zeroed if
     * unsupported or unavailable. */
    uint32_t curr;          /* Current features. */
    uint32_t advertised;    /* Features being advertised by the port. */
    uint32_t supported;     /* Features supported by the port. */
    uint32_t peer;          /* Features advertised by peer. */

    uint32_t curr_speed;    /* Current port bitrate in kbps. */
    uint32_t max_speed;     /* Max port bitrate in kbps */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_ethernet) == 32);

/* Features of optical ports available in switch. */
enum ofp_optical_port_features {
    OFPOPF_RX_TUNE   = 1 << 0,  /* Receiver is tunable */
    OFPOPF_TX_TUNE   = 1 << 1,  /* Transmit is tunable */
    OFPOPF_TX_PWR    = 1 << 2,  /* Power is configurable */
    OFPOPF_USE_FREQ  = 1 << 3,  /* Use Frequency, not wavelength */
};

/* Optical port description property. */
struct ofp_port_desc_prop_optical {
    uint16_t         type;    /* OFPPDPT_3OPTICAL. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint8_t          pad[4];  /* Align to 64 bits. */

    uint32_t supported;     /* Features supported by the port. */
    uint32_t tx_min_freq_lmda;   /* Minimum TX Frequency/Wavelength */
    uint32_t tx_max_freq_lmda;   /* Maximum TX Frequency/Wavelength */
    uint32_t tx_grid_freq_lmda;  /* TX Grid Spacing Frequency/Wavelength */
    uint32_t rx_min_freq_lmda;   /* Minimum RX Frequency/Wavelength */
    uint32_t rx_max_freq_lmda;   /* Maximum RX Frequency/Wavelength */
    uint32_t rx_grid_freq_lmda;  /* RX Grid Spacing Frequency/Wavelength */
    uint16_t tx_pwr_min;         /* Minimum TX power */
    uint16_t tx_pwr_max;         /* Maximum TX power */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_optical) == 40);

/* Ingress or egress pipeline fields. */
struct ofp_port_desc_prop_oxm {
    uint16_t         type;    /* One of OFPPDPT_PIPELINE_INPUT or
                                 OFPPDPT_PIPELINE_OUTPUT. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the oxm_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         oxm_ids[0];   /* Array of OXM headers */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_oxm) == 4);

/* Recirculate port description property. */
struct ofp_port_desc_prop_recirculate {
    uint16_t     type;          /* OFPPDPT_RECIRCULATE. */
    uint16_t     length;        /* Length in bytes of the property,
                                   including this header, excluding padding. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the port numbers, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t     port_nos[0];    /* List of recirculated input port numbers.
                                    0 or more. The number of port numbers
                                    is inferred from the length field in
                                    the header. */
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_recirculate) == 4);

/* Experimenter port description property. */
struct ofp_port_desc_prop_experimenter {
    uint16_t         type;    /* OFPPDPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
```

 © 2015; The Open Networking Foundation

```
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_port_desc_prop_experimenter) == 12);

/* Description of a port */
struct ofp_port {
    uint32_t port_no;
    uint16_t length;
    uint8_t pad[2];
    uint8_t hw_addr[OFP_ETH_ALEN];
    uint8_t pad2[2];                 /* Align to 64 bits. */
    char name[OFP_MAX_PORT_NAME_LEN]; /* Null-terminated */

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

    /* Port description property list - 0 or more properties */
    struct ofp_port_desc_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_port) == 40);

/* 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 auxiliary_id;   /* Identify auxiliary connections */
    uint8_t pad[2];         /* Align to 64-bits. */

    /* Features. */
    uint32_t capabilities;  /* Bitmap of support "ofp_capabilities". */
    uint32_t reserved;
};
OFP_ASSERT(sizeof(struct ofp_switch_features) == 32);

/* What changed about the physical port */
enum ofp_port_reason {
    OFPPR_ADD    = 0,         /* The port was added. */
    OFPPR_DELETE  = 1,        /* The port was removed. */
    OFPPR_MODIFY  = 2,        /* Some attribute of the port has changed. */
};

/* A physical port has changed in the datapath */
struct ofp_port_status {
    struct ofp_header header;
    uint8_t reason;          /* One of OFPPR_*. */
    uint8_t pad[7];          /* Align to 64-bits. */
    struct ofp_port desc;
};
OFP_ASSERT(sizeof(struct ofp_port_status) == 56);

/* Port mod property types.
 */
enum ofp_port_mod_prop_type {
    OFPPMPT_ETHERNET          = 0,      /* Ethernet property. */
    OFPPMPT_OPTICAL           = 1,      /* Optical property. */
    OFPPMPT_EXPERIMENTER      = 0xFFFF, /* Experimenter property. */
};

/* Common header for all port mod properties. */
struct ofp_port_mod_prop_header {
    uint16_t         type;    /* One of OFPPMPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_port_mod_prop_header) == 4);

/* Ethernet port mod property. */
struct ofp_port_mod_prop_ethernet {
    uint16_t      type;       /* OFPPMPT_ETHERNET. */
    uint16_t      length;     /* Length in bytes of this property. */
    uint32_t      advertise;  /* Bitmap of OFPPF_*.  Zero all bits to prevent
                                 any action taking place. */
};
OFP_ASSERT(sizeof(struct ofp_port_mod_prop_ethernet) == 8);

struct ofp_port_mod_prop_optical {
    uint16_t      type;       /* OFPPMPT_OPTICAL. */
    uint16_t      length;     /* Length in bytes of this property. */
    uint32_t      configure;  /* Bitmap of OFPOPF_*. */
    uint32_t      freq_lmda;  /* The "center" frequency */
```

 © 2015; The Open Networking Foundation

```
    int32_t       fl_offset;  /* signed frequency offset */
    uint32_t      grid_span;  /* The size of the grid for this port */
    uint32_t      tx_pwr;     /* tx power setting */
};
OFP_ASSERT(sizeof(struct ofp_port_mod_prop_optical) == 24);

/* Experimenter port mod property. */
struct ofp_port_mod_prop_experimenter {
    uint16_t         type;    /* OFPPMPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_port_mod_prop_experimenter) == 12);

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

    /* Port mod property list - 0 or more properties */
    struct ofp_port_mod_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_port_mod) == 32);

/* ## ---------------------- ## */
/* ## OpenFlow Header Types. ## */
/* ## ---------------------- ## */

/* Header type structure. */
struct ofp_header_type {
    uint16_t    namespace;    /* One of OFPHTN_*. */
    uint16_t    ns_type;      /* Type within namespace. */
};
OFP_ASSERT(sizeof(struct ofp_header_type) == 4);

enum ofp_header_type_namespaces {
    OFPHTN_ONF              = 0,  /* ONF namespace. */
    OFPHTN_ETHERTYPE        = 1,  /* ns_type is an Ethertype. */
    OFPHTN_IP_PROTO         = 2,  /* ns_type is a IP protocol number. */
    OFPHTN_UDP_TCP_PORT     = 3,  /* ns_type is a TCP or UDP port. */
    OFPHTN_IPV4_OPTION      = 4,  /* ns_type is an IPv4 option number. */
};

enum ofp_header_type_onf {
    OFPHTO_ETHERNET         = 0,  /* Ethernet (DIX or IEEE 802.3) - default. */
    OFPHTO_NO_HEADER        = 1,  /* No header, ex. circuit switch. */
    OFPHTO_OXM_EXPERIMENTER = 0xFFFF, /* Use Experimenter OXM. */
};

/* ## -------------------------- ## */
/* ## OpenFlow Extensible Match. ## */
/* ## -------------------------- ## */

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

/* Fields to match against flows */
struct ofp_match {
    uint16_t type;             /* One of OFPMT_* */
    uint16_t length;           /* Length of ofp_match (excluding padding) */
    /* Followed by:
     *   - Exactly (length - 4) (possibly 0) bytes containing OXM TLVs, then
     *   - Exactly ((length + 7)/8*8 - length) (between 0 and 7) bytes of
     *     all-zero bytes
```

 © 2015; The Open Networking Foundation

```
     * In summary, ofp_match is padded as needed, to make its overall size
     * a multiple of 8, to preserve alignment in structures using it.
     */
    uint8_t oxm_fields[0];     /* 0 or more OXM match fields */
    uint8_t pad[4];            /* Zero bytes - see above for sizing */
};
OFP_ASSERT(sizeof(struct ofp_match) == 8);

/* Components of a OXM TLV header.
 * Those macros are not valid for the experimenter class, macros for the
 * experimenter class will depend on the experimenter header used. */
#define OXM_HEADER__(CLASS, FIELD, HASMASK, LENGTH) \
    (((CLASS) << 16) | ((FIELD) << 9) | ((HASMASK) << 8) | (LENGTH))
#define OXM_HEADER(CLASS, FIELD, LENGTH) \
    OXM_HEADER__(CLASS, FIELD, 0, LENGTH)
#define OXM_HEADER_W(CLASS, FIELD, LENGTH) \
    OXM_HEADER__(CLASS, FIELD, 1, (LENGTH) * 2)
#define OXM_CLASS(HEADER) ((HEADER) >> 16)
#define OXM_FIELD(HEADER) (((HEADER) >> 9) & 0x7f)
#define OXM_TYPE(HEADER) (((HEADER) >> 9) & 0x7fffff)
#define OXM_HASMASK(HEADER) (((HEADER) >> 8) & 1)
#define OXM_LENGTH(HEADER) ((HEADER) & 0xff)

#define OXM_MAKE_WILD_HEADER(HEADER) \
        OXM_HEADER_W(OXM_CLASS(HEADER), OXM_FIELD(HEADER), OXM_LENGTH(HEADER))

/* OXM Class IDs.
 * The high order bit differentiate reserved classes from member classes.
 * Classes 0x0000 to 0x7FFF are member classes, allocated by ONF.
 * Classes 0x8000 to 0xFFFE are reserved classes, reserved for standardisation.
 */
enum ofp_oxm_class {
    OFPXMC_NXM_0          = 0x0000,    /* Backward compatibility with NXM */
    OFPXMC_NXM_1          = 0x0001,    /* Backward compatibility with NXM */
    OFPXMC_OPENFLOW_BASIC = 0x8000,    /* Basic class for OpenFlow */
    OFPXMC_PACKET_REGS    = 0x8001,    /* Packet registers (pipeline fields). */
    OFPXMC_EXPERIMENTER   = 0xFFFF,    /* Experimenter class */
};

/* OXM Flow match field types for OpenFlow basic class. */
enum oxm_ofb_match_fields {
    OFPXMT_OFB_IN_PORT        = 0,  /* Switch input port. */
    OFPXMT_OFB_IN_PHY_PORT    = 1,  /* Switch physical input port. */
    OFPXMT_OFB_METADATA       = 2,  /* Metadata passed between tables. */
    OFPXMT_OFB_ETH_DST        = 3,  /* Ethernet destination address. */
    OFPXMT_OFB_ETH_SRC        = 4,  /* Ethernet source address. */
    OFPXMT_OFB_ETH_TYPE       = 5,  /* Ethernet frame type. */
    OFPXMT_OFB_VLAN_VID       = 6,  /* VLAN id. */
    OFPXMT_OFB_VLAN_PCP       = 7,  /* VLAN priority. */
    OFPXMT_OFB_IP_DSCP        = 8,  /* IP DSCP (6 bits in ToS field). */
    OFPXMT_OFB_IP_ECN         = 9,  /* IP ECN (2 bits in ToS field). */
    OFPXMT_OFB_IP_PROTO       = 10, /* IP protocol. */
    OFPXMT_OFB_IPV4_SRC       = 11, /* IPv4 source address. */
    OFPXMT_OFB_IPV4_DST       = 12, /* IPv4 destination address. */
    OFPXMT_OFB_TCP_SRC        = 13, /* TCP source port. */
    OFPXMT_OFB_TCP_DST        = 14, /* TCP destination port. */
    OFPXMT_OFB_UDP_SRC        = 15, /* UDP source port. */
    OFPXMT_OFB_UDP_DST        = 16, /* UDP destination port. */
    OFPXMT_OFB_SCTP_SRC       = 17, /* SCTP source port. */
    OFPXMT_OFB_SCTP_DST       = 18, /* SCTP destination port. */
    OFPXMT_OFB_ICMPV4_TYPE    = 19, /* ICMP type. */
    OFPXMT_OFB_ICMPV4_CODE    = 20, /* ICMP code. */
    OFPXMT_OFB_ARP_OP         = 21, /* ARP opcode. */
    OFPXMT_OFB_ARP_SPA        = 22, /* ARP source IPv4 address. */
    OFPXMT_OFB_ARP_TPA        = 23, /* ARP target IPv4 address. */
    OFPXMT_OFB_ARP_SHA        = 24, /* ARP source hardware address. */
    OFPXMT_OFB_ARP_THA        = 25, /* ARP target hardware address. */
    OFPXMT_OFB_IPV6_SRC       = 26, /* IPv6 source address. */
    OFPXMT_OFB_IPV6_DST       = 27, /* IPv6 destination address. */
    OFPXMT_OFB_IPV6_FLABEL    = 28, /* IPv6 Flow Label */
    OFPXMT_OFB_ICMPV6_TYPE    = 29, /* ICMPv6 type. */
    OFPXMT_OFB_ICMPV6_CODE    = 30, /* ICMPv6 code. */
    OFPXMT_OFB_IPV6_ND_TARGET = 31, /* Target address for ND. */
    OFPXMT_OFB_IPV6_ND_SLL    = 32, /* Source link-layer for ND. */
    OFPXMT_OFB_IPV6_ND_TLL    = 33, /* Target link-layer for ND. */
    OFPXMT_OFB_MPLS_LABEL     = 34, /* MPLS label. */
    OFPXMT_OFB_MPLS_TC        = 35, /* MPLS TC. */
    OFPXMT_OFP_MPLS_BOS       = 36, /* MPLS BoS bit. */
    OFPXMT_OFB_PBB_ISID       = 37, /* PBB I-SID. */
    OFPXMT_OFB_TUNNEL_ID      = 38, /* Logical Port Metadata. */
    OFPXMT_OFB_IPV6_EXTHDR    = 39, /* IPv6 Extension Header pseudo-field */
    OFPXMT_OFB_PBB_UCA        = 41, /* PBB UCA header field. */
    OFPXMT_OFB_TCP_FLAGS      = 42, /* TCP flags. */
    OFPXMT_OFB_ACTSET_OUTPUT  = 43, /* Output port from action set metadata. */
    OFPXMT_OFB_PACKET_TYPE    = 44, /* Packet type value. */
};

#define OFPXMT_OFB_ALL    ((UINT64_C(1) << 45) - 1)

/* OpenFlow port on which the packet was received.
```

 © 2015; The Open Networking Foundation

```
 * May be a physical port, a logical port, or the reserved port OFPP_LOCAL
 *
 * Prereqs: None.
 *
 * Format: 32-bit integer in network byte order.
 *
 * Masking: Not maskable. */
#define OXM_OF_IN_PORT    OXM_HEADER  (0x8000, OFPXMT_OFB_IN_PORT, 4)

/* Physical port on which the packet was received.
 *
 * Consider a packet received on a tunnel interface defined over a link
 * aggregation group (LAG) with two physical port members.  If the tunnel
 * interface is the logical port bound to OpenFlow.  In this case,
 * OFPXMT_OF_IN_PORT is the tunnel's port number and OFPXMT_OF_IN_PHY_PORT is
 * the physical port number of the LAG on which the tunnel is configured.
 *
 * When a packet is received directly on a physical port and not processed by a
 * logical port, OFPXMT_OF_IN_PORT and OFPXMT_OF_IN_PHY_PORT have the same
 * value.
 *
 * This field is usually not available in a regular match and only available
 * in ofp_packet_in messages when it's different from OXM_OF_IN_PORT.
 *
 * Prereqs: OXM_OF_IN_PORT must be present.
 *
 * Format: 32-bit integer in network byte order.
 *
 * Masking: Not maskable. */
#define OXM_OF_IN_PHY_PORT OXM_HEADER  (0x8000, OFPXMT_OFB_IN_PHY_PORT, 4)

/* Table metadata.
 *
 * Prereqs: None.
 *
 * Format: 64-bit integer in network byte order.
 *
 * Masking: Arbitrary masks.
 */
#define OXM_OF_METADATA   OXM_HEADER  (0x8000, OFPXMT_OFB_METADATA, 8)
#define OXM_OF_METADATA_W OXM_HEADER_W(0x8000, OFPXMT_OFB_METADATA, 8)

/* Source or destination address in Ethernet header.
 *
 * Prereqs: None.
 *
 * Format: 48-bit Ethernet MAC address.
 *
 * Masking: Arbitrary masks. */
#define OXM_OF_ETH_DST    OXM_HEADER  (0x8000, OFPXMT_OFB_ETH_DST, 6)
#define OXM_OF_ETH_DST_W  OXM_HEADER_W(0x8000, OFPXMT_OFB_ETH_DST, 6)
#define OXM_OF_ETH_SRC    OXM_HEADER  (0x8000, OFPXMT_OFB_ETH_SRC, 6)
#define OXM_OF_ETH_SRC_W  OXM_HEADER_W(0x8000, OFPXMT_OFB_ETH_SRC, 6)

/* Packet's Ethernet type.
 *
 * Prereqs: None.
 *
 * Format: 16-bit integer in network byte order.
 *
 * Masking: Not maskable. */
#define OXM_OF_ETH_TYPE   OXM_HEADER  (0x8000, OFPXMT_OFB_ETH_TYPE, 2)

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

/* 802.1Q VID.
 *
 * For a packet with an 802.1Q header, this is the VLAN-ID (VID) from the
 * outermost tag, with the CFI bit forced to 1. For a packet with no 802.1Q
 * header, this has value OFPVID_NONE.
 *
 * Prereqs: None.
 *
 * Format: 16-bit integer in network byte order with bit 13 indicating
 * presence of VLAN header and 3 most-significant bits forced to 0.
 * Only the lower 13 bits have meaning.
 *
 * Masking: Arbitrary masks.
 *
 * This field can be used in various ways:
 *
 *   - If it is not constrained at all, the nx_match matches packets without
```

 © 2015; The Open Networking Foundation

```
 *     an 802.1Q header or with an 802.1Q header that has any VID value.
 *
 *   - Testing for an exact match with 0x0 matches only packets without
 *     an 802.1Q header.
 *
 *   - Testing for an exact match with a VID value with CFI=1 matches packets
 *     that have an 802.1Q header with a specified VID.
 *
 *   - Testing for an exact match with a nonzero VID value with CFI=0 does
 *     not make sense.  The switch may reject this combination.
 *
 *   - Testing with nxm_value=0, nxm_mask=0x0fff matches packets with no 802.1Q
 *     header or with an 802.1Q header with a VID of 0.
 *
 *   - Testing with nxm_value=0x1000, nxm_mask=0x1000 matches packets with
 *     an 802.1Q header that has any VID value.
 */
#define OXM_OF_VLAN_VID   OXM_HEADER  (0x8000, OFPXMT_OFB_VLAN_VID, 2)
#define OXM_OF_VLAN_VID_W OXM_HEADER_W(0x8000, OFPXMT_OFB_VLAN_VID, 2)

/* 802.1Q PCP.
 *
 * For a packet with an 802.1Q header, this is the VLAN-PCP from the
 * outermost tag.  For a packet with no 802.1Q header, this has value
 * 0.
 *
 * Prereqs: OXM_OF_VLAN_VID must be different from OFPVID_NONE.
 *
 * Format: 8-bit integer with 5 most-significant bits forced to 0.
 * Only the lower 3 bits have meaning.
 *
 * Masking: Not maskable.
 */
#define OXM_OF_VLAN_PCP   OXM_HEADER  (0x8000, OFPXMT_OFB_VLAN_PCP, 1)

/* The Diff Serv Code Point (DSCP) bits of the IP header.
 * Part of the IPv4 ToS field or the IPv6 Traffic Class field.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must be either 0x0800 or 0x86dd,
 *   or PACKET_TYPE must be either (1,0x800) or (1,0x86dd).
 *
 * Format: 8-bit integer with 2 most-significant bits forced to 0.
 * Only the lower 6 bits have meaning.
 *
 * Masking: Not maskable. */
#define OXM_OF_IP_DSCP     OXM_HEADER  (0x8000, OFPXMT_OFB_IP_DSCP, 1)

/* The ECN bits of the IP header.
 * Part of the IPv4 ToS field or the IPv6 Traffic Class field.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must be either 0x0800 or 0x86dd,
 *   or PACKET_TYPE must be either (1,0x800) or (1,0x86dd).
 *
 * Format: 8-bit integer with 6 most-significant bits forced to 0.
 * Only the lower 2 bits have meaning.
 *
 * Masking: Not maskable. */
#define OXM_OF_IP_ECN     OXM_HEADER  (0x8000, OFPXMT_OFB_IP_ECN, 1)

/* The "protocol" byte in the IP header.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must be either 0x0800 or 0x86dd,
 *   or PACKET_TYPE must be either (1,0x800) or (1,0x86dd).
 *
 * Format: 8-bit integer.
 *
 * Masking: Not maskable. */
#define OXM_OF_IP_PROTO   OXM_HEADER  (0x8000, OFPXMT_OFB_IP_PROTO, 1)

/* The source or destination address in the IP header.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x0800 exactly,
 *   or PACKET_TYPE must match (1,0x800) exactly.
 *
 * Format: 32-bit integer in network byte order.
 *
 * Masking: Arbitrary masks.
 */
#define OXM_OF_IPV4_SRC     OXM_HEADER  (0x8000, OFPXMT_OFB_IPV4_SRC, 4)
#define OXM_OF_IPV4_SRC_W   OXM_HEADER_W(0x8000, OFPXMT_OFB_IPV4_SRC, 4)
#define OXM_OF_IPV4_DST     OXM_HEADER  (0x8000, OFPXMT_OFB_IPV4_DST, 4)
#define OXM_OF_IPV4_DST_W   OXM_HEADER_W(0x8000, OFPXMT_OFB_IPV4_DST, 4)

/* The source or destination port in the TCP header.
 *
 * Prereqs:
```

 © 2015; The Open Networking Foundation

```
 *   either OXM_OF_ETH_TYPE must be either 0x0800 or 0x86dd,
 *   or PACKET_TYPE must be either (1,0x800) or (1,0x86dd),
 *   OXM_OF_IP_PROTO must match 6 exactly.
 *
 * Format: 16-bit integer in network byte order.
 *
 * Masking: Not maskable. */
#define OXM_OF_TCP_SRC    OXM_HEADER  (0x8000, OFPXMT_OFB_TCP_SRC, 2)
#define OXM_OF_TCP_DST    OXM_HEADER  (0x8000, OFPXMT_OFB_TCP_DST, 2)

/* The flags in the TCP header.
*
* Prereqs:
*   OXM_OF_ETH_TYPE must be either 0x0800 or 0x86dd.
*   OXM_OF_IP_PROTO must match 6 exactly.
*
* Format: 16-bit integer with 4 most-significant bits forced to 0.
*
* Masking: Bits 0-11 fully maskable. */
#define OXM_OF_TCP_FLAGS   OXM_HEADER  (0x8000, OFPXMT_OFB_TCP_FLAGS, 2)
#define OXM_OF_TCP_FLAGS_W OXM_HEADER_W(0x8000, OFPXMT_OFB_TCP_FLAGS, 2)

/* The source or destination port in the UDP header.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match either 0x0800 or 0x86dd,
 *   or PACKET_TYPE must be either (1,0x800) or (1,0x86dd),
 *   OXM_OF_IP_PROTO must match 17 exactly.
 *
 * Format: 16-bit integer in network byte order.
 *
 * Masking: Not maskable. */
#define OXM_OF_UDP_SRC    OXM_HEADER  (0x8000, OFPXMT_OFB_UDP_SRC, 2)
#define OXM_OF_UDP_DST    OXM_HEADER  (0x8000, OFPXMT_OFB_UDP_DST, 2)

/* The source or destination port in the SCTP header.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match either 0x0800 or 0x86dd,
 *   or PACKET_TYPE must be either (1,0x800) or (1,0x86dd),
 *   OXM_OF_IP_PROTO must match 132 exactly.
 *
 * Format: 16-bit integer in network byte order.
 *
 * Masking: Not maskable. */
#define OXM_OF_SCTP_SRC   OXM_HEADER  (0x8000, OFPXMT_OFB_SCTP_SRC, 2)
#define OXM_OF_SCTP_DST   OXM_HEADER  (0x8000, OFPXMT_OFB_SCTP_DST, 2)

/* The type or code in the ICMP header.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x0800 exactly,
 *   or PACKET_TYPE must match (1,0x800) exactly.
 *   OXM_OF_IP_PROTO must match 1 exactly.
 *
 * Format: 8-bit integer.
 *
 * Masking: Not maskable. */
#define OXM_OF_ICMPV4_TYPE  OXM_HEADER  (0x8000, OFPXMT_OFB_ICMPV4_TYPE, 1)
#define OXM_OF_ICMPV4_CODE  OXM_HEADER  (0x8000, OFPXMT_OFB_ICMPV4_CODE, 1)

/* ARP opcode.
 *
 * For an Ethernet+IP ARP packet, the opcode in the ARP header.  Always 0
 * otherwise.
 *
 * Prereqs: OXM_OF_ETH_TYPE must match 0x0806 exactly.
 *
 * Format: 16-bit integer in network byte order.
 *
 * Masking: Not maskable. */
#define OXM_OF_ARP_OP     OXM_HEADER  (0x8000, OFPXMT_OFB_ARP_OP, 2)

/* For an Ethernet+IP ARP packet, the source or target protocol address
 * in the ARP header.  Always 0 otherwise.
 *
 * Prereqs: OXM_OF_ETH_TYPE must match 0x0806 exactly.
 *
 * Format: 32-bit integer in network byte order.
 *
 * Masking: Arbitrary masks.
 */
#define OXM_OF_ARP_SPA    OXM_HEADER  (0x8000, OFPXMT_OFB_ARP_SPA, 4)
#define OXM_OF_ARP_SPA_W  OXM_HEADER_W(0x8000, OFPXMT_OFB_ARP_SPA, 4)
#define OXM_OF_ARP_TPA    OXM_HEADER  (0x8000, OFPXMT_OFB_ARP_TPA, 4)
#define OXM_OF_ARP_TPA_W  OXM_HEADER_W(0x8000, OFPXMT_OFB_ARP_TPA, 4)

/* For an Ethernet+IP ARP packet, the source or target hardware address
 * in the ARP header.  Always 0 otherwise.
 *
```

 © 2015; The Open Networking Foundation

```
 * Prereqs: OXM_OF_ETH_TYPE must match 0x0806 exactly.
 *
 * Format: 48-bit Ethernet MAC address.
 *
 * Masking: Not maskable. */
#define OXM_OF_ARP_SHA    OXM_HEADER  (0x8000, OFPXMT_OFB_ARP_SHA, 6)
#define OXM_OF_ARP_THA    OXM_HEADER  (0x8000, OFPXMT_OFB_ARP_THA, 6)

/* The source or destination address in the IPv6 header.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x86dd exactly,
 *   or PACKET_TYPE must match (1,0x86dd) exactly.
 *
 * Format: 128-bit IPv6 address.
 *
 * Masking: Arbitrary masks.
 */
#define OXM_OF_IPV6_SRC    OXM_HEADER  (0x8000, OFPXMT_OFB_IPV6_SRC, 16)
#define OXM_OF_IPV6_SRC_W  OXM_HEADER_W(0x8000, OFPXMT_OFB_IPV6_SRC, 16)
#define OXM_OF_IPV6_DST    OXM_HEADER  (0x8000, OFPXMT_OFB_IPV6_DST, 16)
#define OXM_OF_IPV6_DST_W  OXM_HEADER_W(0x8000, OFPXMT_OFB_IPV6_DST, 16)

/* The IPv6 Flow Label
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x86dd exactly,
 *   or PACKET_TYPE must match (1,0x86dd) exactly.
 *
 * Format: 32-bit integer with 12 most-significant bits forced to 0.
 * Only the lower 20 bits have meaning.
 *
 * Masking: Arbitrary masks.
 */
#define OXM_OF_IPV6_FLABEL   OXM_HEADER  (0x8000, OFPXMT_OFB_IPV6_FLABEL, 4)
#define OXM_OF_IPV6_FLABEL_W OXM_HEADER_W(0x8000, OFPXMT_OFB_IPV6_FLABEL, 4)

/* The type or code in the ICMPv6 header.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x86dd exactly,
 *   or PACKET_TYPE must match (1,0x86dd) exactly,
 *   OXM_OF_IP_PROTO must match 58 exactly.
 *
 * Format: 8-bit integer.
 *
 * Masking: Not maskable. */
#define OXM_OF_ICMPV6_TYPE OXM_HEADER  (0x8000, OFPXMT_OFB_ICMPV6_TYPE, 1)
#define OXM_OF_ICMPV6_CODE OXM_HEADER  (0x8000, OFPXMT_OFB_ICMPV6_CODE, 1)

/* The target address in an IPv6 Neighbor Discovery message.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x86dd exactly,
 *   or PACKET_TYPE must match (1,0x86dd) exactly,
 *   OXM_OF_IP_PROTO must match 58 exactly.
 *   OXM_OF_ICMPV6_TYPE must be either 135 or 136.
 *
 * Format: 128-bit IPv6 address.
 *
 * Masking: Not maskable. */
#define OXM_OF_IPV6_ND_TARGET OXM_HEADER (0x8000, OFPXMT_OFB_IPV6_ND_TARGET, 16)

/* The source link-layer address option in an IPv6 Neighbor Discovery
 * message.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x86dd exactly,
 *   or PACKET_TYPE must match (1,0x86dd) exactly,
 *   OXM_OF_IP_PROTO must match 58 exactly.
 *   OXM_OF_ICMPV6_TYPE must be exactly 135.
 *
 * Format: 48-bit Ethernet MAC address.
 *
 * Masking: Not maskable. */
#define OXM_OF_IPV6_ND_SLL  OXM_HEADER  (0x8000, OFPXMT_OFB_IPV6_ND_SLL, 6)

/* The target link-layer address option in an IPv6 Neighbor Discovery
 * message.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x86dd exactly,
 *   or PACKET_TYPE must match (1,0x86dd) exactly,
 *   OXM_OF_IP_PROTO must match 58 exactly.
 *   OXM_OF_ICMPV6_TYPE must be exactly 136.
 *
 * Format: 48-bit Ethernet MAC address.
 *
 * Masking: Not maskable. */
#define OXM_OF_IPV6_ND_TLL  OXM_HEADER  (0x8000, OFPXMT_OFB_IPV6_ND_TLL, 6)
```

 © 2015; The Open Networking Foundation

```
/* The LABEL in the first MPLS shim header.
 *
 * Prereqs:
 *   OXM_OF_ETH_TYPE must match 0x8847 or 0x8848 exactly.
 *
 * Format: 32-bit integer in network byte order with 12 most-significant
 * bits forced to 0. Only the lower 20 bits have meaning.
 *
 * Masking: Not maskable. */
#define OXM_OF_MPLS_LABEL  OXM_HEADER  (0x8000, OFPXMT_OFB_MPLS_LABEL, 4)

/* The TC in the first MPLS shim header.
 *
 * Prereqs:
 *   OXM_OF_ETH_TYPE must match 0x8847 or 0x8848 exactly.
 *
 * Format: 8-bit integer with 5 most-significant bits forced to 0.
 * Only the lower 3 bits have meaning.
 *
 * Masking: Not maskable. */
#define OXM_OF_MPLS_TC     OXM_HEADER  (0x8000, OFPXMT_OFB_MPLS_TC, 1)

/* The BoS bit in the first MPLS shim header.
 *
 * Prereqs:
 *   OXM_OF_ETH_TYPE must match 0x8847 or 0x8848 exactly.
 *
 * Format: 8-bit integer with 7 most-significant bits forced to 0.
 * Only the lowest bit have a meaning.
 *
 * Masking: Not maskable. */
#define OXM_OF_MPLS_BOS     OXM_HEADER  (0x8000, OFPXMT_OFB_MPLS_BOS, 1)

/* IEEE 802.1ah I-SID.
 *
 * For a packet with a PBB header, this is the I-SID from the
 * outermost service tag.
 *
 * Prereqs:
 *   OXM_OF_ETH_TYPE must match 0x88E7 exactly.
 *
 * Format: 24-bit integer in network byte order.
 *
 * Masking: Arbitrary masks. */
#define OXM_OF_PBB_ISID   OXM_HEADER  (0x8000, OFPXMT_OFB_PBB_ISID, 3)
#define OXM_OF_PBB_ISID_W OXM_HEADER_W(0x8000, OFPXMT_OFB_PBB_ISID, 3)

/* Logical Port Metadata.
 *
 * Metadata associated with a logical port.
 * If the logical port performs encapsulation and decapsulation, this
 * is the demultiplexing field from the encapsulation header.
 * For example, for a packet received via GRE tunnel including a (32-bit) key,
 * the key is stored in the low 32-bits and the high bits are zeroed.
 * For a MPLS logical port, the low 20 bits represent the MPLS Label.
 * For a VxLAN logical port, the low 24 bits represent the VNI.
 * If the packet is not received through a logical port, the value is 0.
 *
 * Prereqs: None.
 *
 * Format: 64-bit integer in network byte order.
 *
 * Masking: Arbitrary masks. */
#define OXM_OF_TUNNEL_ID    OXM_HEADER  (0x8000, OFPXMT_OFB_TUNNEL_ID, 8)
#define OXM_OF_TUNNEL_ID_W  OXM_HEADER_W(0x8000, OFPXMT_OFB_TUNNEL_ID, 8)

/* The IPv6 Extension Header pseudo-field.
 *
 * Prereqs:
 *   either OXM_OF_ETH_TYPE must match 0x86dd exactly,
 *   or PACKET_TYPE must match (1,0x86dd) exactly.
 *
 * Format: 16-bit integer with 7 most-significant bits forced to 0.
 * Only the lower 9 bits have meaning.
 *
 * Masking: Maskable. */
#define OXM_OF_IPV6_EXTHDR   OXM_HEADER  (0x8000, OFPXMT_OFB_IPV6_EXTHDR, 2)
#define OXM_OF_IPV6_EXTHDR_W OXM_HEADER_W(0x8000, OFPXMT_OFB_IPV6_EXTHDR, 2)

/* Bit definitions for IPv6 Extension Header pseudo-field. */
enum ofp_ipv6exthdr_flags {
    OFPIEH_NONEXT = 1 << 0,     /* "No next header" encountered. */
    OFPIEH_ESP    = 1 << 1,     /* Encrypted Sec Payload header present. */
    OFPIEH_AUTH   = 1 << 2,     /* Authentication header present. */
    OFPIEH_DEST   = 1 << 3,     /* 1 or 2 dest headers present. */
    OFPIEH_FRAG   = 1 << 4,     /* Fragment header present. */
    OFPIEH_ROUTER = 1 << 5,     /* Router header present. */
    OFPIEH_HOP    = 1 << 6,     /* Hop-by-hop header present. */
    OFPIEH_UNREP  = 1 << 7,     /* Unexpected repeats encountered. */
```

 © 2015; The Open Networking Foundation

```
    OFPIEH_UNSEQ  = 1 << 8,     /* Unexpected sequencing encountered. */
};


/* IEEE 802.1ah UCA.
 *
 * For a packet with a PBB header, this is the UCA (Use Customer Address)
 * from the outermost service tag.
 *
 * Prereqs:
 *   OXM_OF_ETH_TYPE must match 0x88E7 exactly.
 *
 * Format: 8-bit integer with 7 most-significant bits forced to 0.
 * Only the lower 1 bit has meaning.
 *
 * Masking: Not maskable. */
#define OXM_OF_PBB_UCA    OXM_HEADER  (0x8000, OFPXMT_OFB_PBB_UCA, 1)

/* Output port from action set Metadata.
 *
 * Metadata representing the forwarding decision in the action set.
 * If the action set contains an output action, this field equals
 * the output port.
 * Else, the field equals its initial value, OFPP_UNSET.
 *
 * Prereqs: None.
 *
 * Format: 32-bit integer in network byte order.
 *
 * Masking: Not maskable. */
#define OXM_OF_ACTSET_OUTPUT OXM_HEADER  (0x8000, OFPXMT_OFB_ACTSET_OUTPUT, 4)

/* Packet type.
 *
 * Packet type to identify packets. This is the canonical header type of
 * the outermost header.
 * If not specified, default to Ethernet header type.
 *
 * Prereqs: None.
 *
 * Format: 16-bit integer for namespace followed by 16 bit interget for ns_type.
 *
 * Masking: Not maskable. */
#define OXM_OF_PACKET_TYPE  OXM_HEADER  (0x8000, OFPXMT_OFB_PACKET_TYPE, 4)

/* Header for OXM experimenter match fields.
 * The experimenter class should not use OXM_HEADER() macros for defining
 * fields due to this extra header. */
struct ofp_oxm_experimenter_header {
    uint32_t oxm_header;        /* oxm_class = OFPXMC_EXPERIMENTER */
    uint32_t experimenter;      /* Experimenter ID. */
};
OFP_ASSERT(sizeof(struct ofp_oxm_experimenter_header) == 8);

/* Packet registers (scratch space).
 *
 * Storage registers associated with a packet.
 * Set to zero when packet enters the pipeline.
 * May be used to stored values through the pipeline.
 * Usually not usable as a match field.
 * Switch implement a variable number of registers, may be 0.
 *
 * Prereqs: None.
 *
 * Format: 64-bit integer in network byte order.
 *
 * Masking: Arbitrary masks.
 */
#define OXM_OF_PKT_REG(N)    (OXM_HEADER  (0x8001, N, 8))
#define OXM_OF_PKT_REG_W(N)  (OXM_HEADER_W(0x8001, N, 8))

/* ## ------------------------- ## */
/* ## OpenFlow Extensible Stats. ## */
/* ## ------------------------- ## */

/* Flow statistics structure - list of statistic fields. */
struct ofp_stats {
    uint16_t reserved;         /* Reserved for future use, currently zeroed. */
    uint16_t length;           /* Length of ofp_stats (excluding padding) */
    /* Followed by:
     *   - Exactly (length - 4) (possibly 0) bytes containing OXS TLVs, then
     *   - Exactly ((length + 7)/8*8 - length) (between 0 and 7) bytes of
     *     all-zero bytes
     * In summary, ofp_stats is padded as needed, to make its overall size
     * a multiple of 8, to preserve alignement in structures using it.
     */
    uint8_t oxs_fields[0];     /* 0 or more OXS stat fields */
    uint8_t pad[4];            /* Zero bytes - see above for sizing */
};
OFP_ASSERT(sizeof(struct ofp_stats) == 8);
```

 © 2015; The Open Networking Foundation

```
/* Components of a OXS TLV header. */
#define OXS_HEADER__(CLASS, FIELD, RESERVED, LENGTH) \
    (((CLASS) << 16) | ((FIELD) << 9) | ((RESERVED) << 8) | (LENGTH))
#define OXS_HEADER(CLASS, FIELD, LENGTH) \
    OXS_HEADER__(CLASS, FIELD, 0, LENGTH)
#define OXS_CLASS(HEADER) ((HEADER) >> 16)
#define OXS_FIELD(HEADER) (((HEADER) >> 9) & 0x7f)
#define OXS_TYPE(HEADER) (((HEADER) >> 9) & 0x7fffff)
#define OXS_RESERVED(HEADER) (((HEADER) >> 8) & 1)
#define OXS_LENGTH(HEADER) ((HEADER) & 0xff)

/* OXS Class IDs.
 * The high order bit differentiate reserved classes from member classes.
 * Classes 0x0000 to 0x7FFF are member classes, allocated by ONF.
 * Classes 0x8000 to 0xFFFE are reserved classes, reserved for standardisation.
 */
enum ofp_oxs_class {
    OFPXSC_OPENFLOW_BASIC = 0x8002,    /* Basic stats class for OpenFlow */
    OFPXSC_EXPERIMENTER   = 0xFFFF,    /* Experimenter class */
};

/* OXS flow stat field types for OpenFlow basic class. */
enum oxs_ofb_stat_fields {
    OFPXST_OFB_DURATION        = 0,  /* Time flow entry has been alive. */
    OFPXST_OFB_IDLE_TIME       = 1,  /* Time flow entry has been idle. */
    OFPXST_OFB_FLOW_COUNT      = 3,  /* Number of aggregated flow entries. */
    OFPXST_OFB_PACKET_COUNT    = 4,  /* Number of packets in flow entry. */
    OFPXST_OFB_BYTE_COUNT      = 5,  /* Number of bytes in flow entry. */
};

#define OFPXST_OFB_ALL    ((UINT64_C(1) << 6) - 1)

/* OpenFlow flow duration.
 * Time flow entry has been alive in seconds and nanoseconds.
 *
 * Format: A pair of 32-bit integer in network byte order.
 *    First 32-bit number is duration in seconds.
 *    Second 32-bit number is nanoseconds beyond duration in seconds.
 * Second number must be set to zero if not supported.
 */
#define OXS_OF_DURATION      OXS_HEADER  (0x8002, OFPXST_OFB_DURATION, 8)

/* OpenFlow flow idle time.
 * Time flow entry has been idle (no packets matched) in seconds
 * and nanoseconds.
 *
 * Format: A pair of 32-bit integer in network byte order.
 *    First 32-bit number is idle time in seconds.
 *    Second 32-bit number is nanoseconds beyond idle time in seconds.
 * Second number must be set to zero if not supported.
 */
#define OXS_OF_IDLE_TIME      OXS_HEADER  (0x8002, OFPXST_OFB_IDLE_TIME, 8)

/* OpenFlow flow entry count.
 * Number of aggregated flow entries.
 * Required in OFPMP_AGGREGATE replies, undefined in other context.
 *
 * Format: 32-bit integer in network byte order.
 */
#define OXS_OF_FLOW_COUNT    OXS_HEADER  (0x8002, OFPXST_OFB_FLOW_COUNT, 4)

/* OpenFlow flow entry packet count.
 * Number of packets matched by a flow entry.
 *
 * Format: 64-bit integer in network byte order.
 */
#define OXS_OF_PACKET_COUNT  OXS_HEADER  (0x8002, OFPXST_OFB_PACKET_COUNT, 8)

/* OpenFlow flow entry packet count.
 * Number of bytes matched by a flow entry.
 *
 * Format: 64-bit integer in network byte order.
 */
#define OXS_OF_BYTE_COUNT    OXS_HEADER  (0x8002, OFPXST_OFB_BYTE_COUNT, 8)

/* Header for OXS experimenter stat fields. */
struct ofp_oxs_experimenter_header {
    uint32_t oxs_header;        /* oxs_class = OFPXSC_EXPERIMENTER */
    uint32_t experimenter;      /* Experimenter ID. */
};
OFP_ASSERT(sizeof(struct ofp_oxs_experimenter_header) == 8);

/* ## ----------------- ## */
/* ## OpenFlow Actions. ## */
/* ## ----------------- ## */

enum ofp_action_type {
    OFPAT_OUTPUT       = 0,  /* Output to switch port. */
    OFPAT_COPY_TTL_OUT = 11, /* Copy TTL "outwards" -- from next-to-outermost
```


© 2015; The Open Networking Foundation

```
                              to outermost */
    OFPAT_COPY_TTL_IN  = 12, /* Copy TTL "inwards" -- from outermost to
                                next-to-outermost */
    OFPAT_SET_MPLS_TTL = 15, /* MPLS TTL */
    OFPAT_DEC_MPLS_TTL = 16, /* Decrement MPLS TTL */

    OFPAT_PUSH_VLAN    = 17, /* Push a new VLAN tag */
    OFPAT_POP_VLAN     = 18, /* Pop the outer VLAN tag */
    OFPAT_PUSH_MPLS    = 19, /* Push a new MPLS tag */
    OFPAT_POP_MPLS     = 20, /* Pop the outer MPLS tag */
    OFPAT_SET_QUEUE    = 21, /* Set queue id when outputting to a port */
    OFPAT_GROUP        = 22, /* Apply group. */
    OFPAT_SET_NW_TTL   = 23, /* IP TTL. */
    OFPAT_DEC_NW_TTL   = 24, /* Decrement IP TTL. */
    OFPAT_SET_FIELD    = 25, /* Set a header field using OXM TLV format. */
    OFPAT_PUSH_PBB     = 26, /* Push a new PBB service tag (I-TAG) */
    OFPAT_POP_PBB      = 27, /* Pop the outer PBB service tag (I-TAG) */
    OFPAT_COPY_FIELD   = 28, /* Copy value between header and register. */
    OFPAT_METER        = 29, /* Apply meter (rate limiter) */
    OFPAT_EXPERIMENTER = 0xffff
};

/* 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 this struct in bytes. */
};
OFP_ASSERT(sizeof(struct ofp_action_header) == 4);

enum ofp_controller_max_len {
        OFPCML_MAX       = 0xffe5, /* maximum max_len value which can be used
                                      to request a specific byte length. */
        OFPCML_NO_BUFFER = 0xffff  /* indicates that no buffering should be
                                      applied and the whole packet is to be
                                      sent to the controller. */
};

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

/* Action structure for OFPAT_COPY_TTL_OUT, OFPAT_COPY_TTL_IN,
 * OFPAT_DEC_MPLS_TTL, OFPAT_DEC_NW_TTL, OFPAT_POP_VLAN and OFPAT_POP_PBB. */
struct ofp_action_generic {
    uint16_t type;                  /* One of OFPAT_*. */
    uint16_t len;                   /* Length is 8. */
    uint8_t pad[4];                 /* Pad to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_action_generic) == 8);

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

/* Action structure for OFPAT_PUSH_VLAN/MPLS/PBB. */
struct ofp_action_push {
    uint16_t type;                  /* OFPAT_PUSH_VLAN/MPLS/PBB. */
    uint16_t len;                   /* Length is 8. */
    uint16_t ethertype;             /* Ethertype */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_push) == 8);

/* Action structure for OFPAT_POP_MPLS. */
struct ofp_action_pop_mpls {
    uint16_t type;                  /* OFPAT_POP_MPLS. */
    uint16_t len;                   /* Length is 8. */
    uint16_t ethertype;             /* Ethertype */
    uint8_t pad[2];
};
OFP_ASSERT(sizeof(struct ofp_action_pop_mpls) == 8);
```

 © 2015; The Open Networking Foundation

```
/* Action structure for OFPAT_SET_QUEUE. */
struct ofp_action_set_queue {
    uint16_t type;            /* OFPAT_SET_QUEUE. */
    uint16_t len;             /* Len is 8. */
    uint32_t queue_id;        /* Queue id for the packets. */
};
OFP_ASSERT(sizeof(struct ofp_action_set_queue) == 8);

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

/* Action structure for OFPAT_SET_NW_TTL. */
struct ofp_action_nw_ttl {
    uint16_t type;                  /* OFPAT_SET_NW_TTL. */
    uint16_t len;                   /* Length is 8. */
    uint8_t nw_ttl;                 /* IP TTL */
    uint8_t pad[3];
};
OFP_ASSERT(sizeof(struct ofp_action_nw_ttl) == 8);

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

/* Action structure for OFPAT_COPY_FIELD. */
struct ofp_action_copy_field {
    uint16_t type;                  /* OFPAT_COPY_FIELD. */
    uint16_t len;                   /* Length is padded to 64 bits. */
    uint16_t n_bits;                /* Number of bits to copy. */
    uint16_t src_offset;            /* Starting bit offset in source. */
    uint16_t dst_offset;            /* Starting bit offset in destination. */
    uint8_t pad[2];                 /* Align to 32 bits. */
    /* Followed by:
     *   - Exactly 8, 12 or 16 bytes containing the oxm_ids, then
     *   - Enough all-zero bytes (either 0 or 4) to make the action a whole
     *     multiple of 8 bytes in length */
    uint32_t         oxm_ids[0];   /* Source and destination OXM headers */
};
OFP_ASSERT(sizeof(struct ofp_action_copy_field) == 12);

/* Action structure for OFPAT_METER */
struct ofp_action_meter {
    uint16_t type;                /* OFPAT_METER */
    uint16_t len;                 /* Length is 8. */
    uint32_t meter_id;            /* Meter instance. */
};
OFP_ASSERT(sizeof(struct ofp_action_meter) == 8);

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

/* ## ---------------------- ## */
/* ## OpenFlow Instructions. ## */
/* ## ---------------------- ## */

enum ofp_instruction_type {
    OFPIT_GOTO_TABLE = 1,       /* Setup the next table in the lookup
                                   pipeline */
    OFPIT_WRITE_METADATA = 2,   /* Setup the metadata field for use later in
                                   pipeline */
    OFPIT_WRITE_ACTIONS = 3,    /* Write the action(s) onto the datapath action
                                   set */
    OFPIT_APPLY_ACTIONS = 4,    /* Applies the action(s) immediately */
    OFPIT_CLEAR_ACTIONS = 5,    /* Clears all actions from the datapath
                                   action set */
    OFPIT_DEPRECATED = 6,       /* Deprecated (was apply meter) */
    OFPIT_STAT_TRIGGER = 7,     /* Statistics triggers */

    OFPIT_EXPERIMENTER = 0xFFFF  /* Experimenter instruction */
};
```

 © 2015; The Open Networking Foundation

```
/* Instruction header that is common to all instructions.  The length includes
 * the header and any padding used to make the instruction 64-bit aligned.
 * NB: The length of an instruction *must* always be a multiple of eight. */
struct ofp_instruction_header {
    uint16_t type;                /* One of OFPIT_*. */
    uint16_t len;                 /* Length of this struct in bytes. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_header) == 4);

/* Instruction structure for OFPIT_GOTO_TABLE */
struct ofp_instruction_goto_table {
    uint16_t type;                /* OFPIT_GOTO_TABLE */
    uint16_t len;                 /* Length is 8. */
    uint8_t table_id;             /* Set next table in the lookup pipeline */
    uint8_t pad[3];               /* Pad to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_goto_table) == 8);

/* Instruction structure for OFPIT_WRITE_METADATA */
struct ofp_instruction_write_metadata {
    uint16_t type;                /* OFPIT_WRITE_METADATA */
    uint16_t len;                 /* Length is 24. */
    uint8_t pad[4];               /* Align to 64-bits */
    uint64_t metadata;            /* Metadata value to write */
    uint64_t metadata_mask;       /* Metadata write bitmask */
};
OFP_ASSERT(sizeof(struct ofp_instruction_write_metadata) == 24);

/* Instruction structure for OFPIT_WRITE/APPLY/CLEAR_ACTIONS */
struct ofp_instruction_actions {
    uint16_t type;              /* One of OFPIT_*_ACTIONS */
    uint16_t len;               /* Length is padded to 64 bits. */
    uint8_t pad[4];             /* Align to 64-bits */
    struct ofp_action_header actions[0];  /* 0 or more actions associated with
                                             OFPIT_WRITE_ACTIONS and
                                             OFPIT_APPLY_ACTIONS */
};
OFP_ASSERT(sizeof(struct ofp_instruction_actions) == 8);

enum ofp_stat_trigger_flags {
    OFPSTF_PERIODIC    = 1 << 0,  /* Trigger for all multiples of thresholds. */
    OFPSTF_ONLY_FIRST  = 1 << 1,  /* Trigger on only first reach threshold. */
};

/* Instruction structure for OFPIT_STAT_TRIGGER */
struct ofp_instruction_stat_trigger {
    uint16_t type;                /* OFPIT_STAT_TRIGGER */
    uint16_t len;                 /* Length is padded to 64 bits. */
    uint32_t flags;               /* Bitmap of OFPSTF_* flags. */
    struct ofp_stats thresholds;  /* Threshold list. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_stat_trigger) == 16);

/* Instruction structure for experimental instructions */
struct ofp_instruction_experimenter_header {
    uint16_t type;              /* OFPIT_EXPERIMENTER. */
    uint16_t len;               /* Length is padded to 64 bits. */
    uint32_t experimenter;      /* Experimenter ID. */
    /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_experimenter_header) == 8);

/* ## -------------------------- ## */
/* ## OpenFlow Flow Modification. ## */
/* ## -------------------------- ## */

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

/* Value used in "idle_timeout" and "hard_timeout" to indicate that the entry
 * is permanent. */
#define OFP_FLOW_PERMANENT 0

/* By default, choose a priority in the middle. */
#define OFP_DEFAULT_PRIORITY 0x8000

enum ofp_flow_mod_flags {
    OFPFF_SEND_FLOW_REM = 1 << 0,  /* Send flow removed message when flow
                                    * expires or is deleted. */
    OFPFF_CHECK_OVERLAP = 1 << 1,  /* Check for overlapping entries first. */
    OFPFF_RESET_COUNTS  = 1 << 2,  /* Reset flow packet and byte counts. */
    OFPFF_NO_PKT_COUNTS = 1 << 3,  /* Don't keep track of packet count. */
```

 © 2015; The Open Networking Foundation

```
    OFPFF_NO_BYT_COUNTS = 1 << 4,  /* Don't keep track of byte count. */
};

/* Flow setup and teardown (controller -> datapath). */
struct ofp_flow_mod {
    struct ofp_header header;
    uint64_t cookie;              /* Opaque controller-issued identifier. */
    uint64_t cookie_mask;         /* Mask used to restrict the cookie bits
                                     that must match when the command is
                                     OFPFC_MODIFY* or OFPFC_DELETE*. A value
                                     of 0 indicates no restriction. */
    uint8_t table_id;             /* ID of the table to put the flow in.
                                     For OFPFC_DELETE_* commands, OFPTT_ALL
                                     can also be used to delete matching
                                     flows from all tables. */
    uint8_t command;              /* One of OFPFC_*. */
    uint16_t idle_timeout;        /* Idle time before discarding (seconds). */
    uint16_t hard_timeout;        /* Max time before discarding (seconds). */
    uint16_t priority;            /* Priority level of flow entry. */
    uint32_t buffer_id;           /* Buffered packet to apply to, or
                                     OFP_NO_BUFFER.
                                     Not meaningful for OFPFC_DELETE*. */
    uint32_t out_port;            /* For OFPFC_DELETE* commands, require
                                     matching entries to include this as an
                                     output port.  A value of OFPP_ANY
                                     indicates no restriction. */
    uint32_t out_group;           /* For OFPFC_DELETE* commands, require
                                     matching entries to include this as an
                                     output group.  A value of OFPG_ANY
                                     indicates no restriction. */
    uint16_t flags;               /* Bitmap of OFPFF_* flags. */
    uint16_t importance;          /* Eviction precedence (optional). */
    struct ofp_match match;       /* Fields to match. Variable size. */
    /* The variable size and padded match is always followed by instructions. */
    //struct ofp_instruction_header instructions[0];
                         /* Instruction set - 0 or more. The length
                                     of the instruction set is inferred from
                                     the length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_flow_mod) == 56);

/* Group numbering. Groups can use any number up to OFPG_MAX. */
enum ofp_group {
    /* Last usable group number. */
    OFPG_MAX        = 0xffffff00,

    /* Fake groups. */
    OFPG_ALL        = 0xfffffffc,  /* Represents all groups for group delete
                                      commands. */
    OFPG_ANY        = 0xffffffff   /* Special wildcard: no group specified. */
};

/* Group commands */
enum ofp_group_mod_command {
    OFPGC_ADD    = 0,       /* New group. */
    OFPGC_MODIFY = 1,       /* Modify all matching groups. */
    OFPGC_DELETE = 2,       /* Delete all matching groups. */
    OFPGC_INSERT_BUCKET = 3,/* Insert action buckets to the already available
                               list of action buckets in a matching group */
    /* OFPGC_??? = 4, */    /* Reserved for future use. */
    OFPGC_REMOVE_BUCKET = 5,/* Remove all action buckets or any specific action
                               bucket from matching group */
};

/* Group bucket property types.  */
enum ofp_group_bucket_prop_type {
    OFPGBPT_WEIGHT                  = 0,  /* Select groups only. */
    OFPGBPT_WATCH_PORT              = 1,  /* Fast failover groups only. */
    OFPGBPT_WATCH_GROUP             = 2,  /* Fast failover groups only. */
    OFPGBPT_EXPERIMENTER      = 0xFFFF,  /* Experimenter defined. */
};

/* Common header for all group bucket properties. */
struct ofp_group_bucket_prop_header {
    uint16_t         type;    /* One of OFPGBPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_group_bucket_prop_header) == 4);

/* Group bucket weight property, for select groups only. */
struct ofp_group_bucket_prop_weight {
    uint16_t         type;    /* OFPGBPT_WEIGHT. */
    uint16_t         length;  /* 8. */
    uint16_t         weight;  /* Relative weight of bucket. */
    uint8_t          pad[2];  /* Pad to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_group_bucket_prop_weight) == 8);

/* Group bucket watch port or watch group property, for fast failover groups
 * only. */
```

 © 2015; The Open Networking Foundation

```
struct ofp_group_bucket_prop_watch {
    uint16_t         type;    /* OFPGBPT_WATCH_PORT or OFPGBPT_WATCH_GROUP. */
    uint16_t         length;  /* 8. */
    uint32_t         watch;   /* The port or the group.  */
};
OFP_ASSERT(sizeof(struct ofp_group_bucket_prop_watch) == 8);

/* Experimenter group bucket property */
struct ofp_group_bucket_prop_experimenter {
    uint16_t         type;         /* OFPGBPT_EXPERIMENTER. */
    uint16_t         length;       /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_group_bucket_prop_experimenter) == 12);

/* Bucket for use in groups. */
struct ofp_bucket {
    uint16_t len;                    /* Length of the bucket in bytes, including
                                       this header and any padding to make it
                                       64-bit aligned. */
    uint16_t action_array_len;      /* Length of all actions in bytes. */
    uint32_t bucket_id;             /* Bucket Id used to identify bucket*/
    /* Followed by:
     *   - Exactly 'action_array_len' bytes containing an array of
     *     struct ofp_action_*.
     *   - Zero or more bytes of group bucket properties to fill out the
     *     overall length in header.length. */
    struct ofp_action_header actions[0]; /* The length of the action array is
                                            action_array_len bytes. */
    //struct ofp_group_bucket_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_bucket) == 8);

/* Bucket Id can be any value between 0 and OFPG_BUCKET_MAX */
enum ofp_group_bucket {
    OFPG_BUCKET_MAX   = 0xffffff00, /* Last usable bucket ID. */
    OFPG_BUCKET_FIRST = 0xfffffffd, /* First bucket ID in the list of action
                                       buckets of a group. This is applicable
                                       for OFPGC_INSERT_BUCKET and
                                       OFPGC_REMOVE_BUCKET commands. */
    OFPG_BUCKET_LAST  = 0xfffffffe, /* Last bucket ID in the list of action
                                       buckets of a group. This is applicable
                                       for OFPGC_INSERT_BUCKET and
                                       OFPGC_REMOVE_BUCKET commands. */
    OFPG_BUCKET_ALL   = 0xffffffff  /* All action buckets in a group,
                                       This is applicable for
                                       only OFPGC_REMOVE_BUCKET command. */
};

/* Group property types.  */
enum ofp_group_prop_type {
    OFPGPT_EXPERIMENTER      = 0xFFFF,  /* Experimenter defined. */
};

/* Common header for all group properties. */
struct ofp_group_prop_header {
    uint16_t         type;    /* One of OFPGPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_group_prop_header) == 4);

/* Experimenter group property */
struct ofp_group_prop_experimenter {
    uint16_t         type;         /* OFPGPT_EXPERIMENTER. */
    uint16_t         length;       /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_group_prop_experimenter) == 12);

/* Group setup and teardown (controller -> datapath). */
struct ofp_group_mod {
    struct ofp_header header;
    uint16_t command;             /* One of OFPGC_*. */
    uint8_t type;                 /* One of OFPGT_*. */
```

 © 2015; The Open Networking Foundation

```
    uint8_t pad;                /* Pad to 64 bits. */
    uint32_t group_id;          /* Group identifier. */
    uint16_t bucket_array_len;  /* Length of action buckets data. */
    uint8_t pad2[2];            /* Pad to 64 bits. */
    uint32_t command_bucket_id; /* Bucket Id used as part of
                                   OFPGC_INSERT_BUCKET and OFPGC_REMOVE_BUCKET
                                   commands execution.*/
    /* Followed by:
     *   - Exactly 'bucket_array_len' bytes containing an array of
     *     struct ofp_bucket.
     *   - Zero or more bytes of group properties to fill out the overall
     *     length in header.length. */
    struct ofp_bucket buckets[0]; /* The length of the bucket array is
                                   bucket_array_len bytes. */
    //struct ofp_group_prop_header properties[0];
};
OFP_ASSERT (sizeof(struct ofp_group_mod) == 24);

/* Group types.  Values in the range [128, 255] are reserved for experimental
 * use. */
enum ofp_group_type {
    OFPGT_ALL      = 0, /* All (multicast/broadcast) group.  */
    OFPGT_SELECT   = 1, /* Select group. */
    OFPGT_INDIRECT = 2, /* Indirect group. */
    OFPGT_FF       = 3, /* Fast failover group. */
};

/* Special buffer-id to indicate 'no buffer' */
#define OFP_NO_BUFFER 0xffffffff

/* Send packet (controller -> datapath). */
struct ofp_packet_out {
    struct ofp_header header;
    uint32_t buffer_id;           /* ID assigned by datapath (OFP_NO_BUFFER
                                     if none). */
    uint16_t actions_len;         /* Size of action array in bytes. */
    uint8_t pad[2];               /* Align to 64 bits. */
    struct ofp_match match;       /* Packet pipeline fields. Variable size. */
    /* The variable size and padded match is followed by the list of actions. */
    /* struct ofp_action_header actions[0]; *//* Action list - 0 or more. */
    /* The variable size action list is optionally followed by packet data.
     * This data is only present and meaningful if buffer_id == -1. */
    /* uint8_t data[0]; */        /* Packet data.  The length is inferred
                                     from the length field in the header. */
};
OFP_ASSERT(sizeof(struct ofp_packet_out) == 24);

/* Why is this packet being sent to the controller? */
enum ofp_packet_in_reason {
    OFPR_TABLE_MISS   = 0,   /* No matching flow (table-miss flow entry). */
    OFPR_APPLY_ACTION = 1,   /* Output to controller in apply-actions. */
    OFPR_INVALID_TTL  = 2,   /* Packet has invalid TTL */
    OFPR_ACTION_SET   = 3,   /* Output to controller in action set. */
    OFPR_GROUP        = 4,   /* Output to controller in group bucket. */
    OFPR_PACKET_OUT   = 5,   /* Output to controller in packet-out. */
};

/* Packet received on port (datapath -> controller). */
struct ofp_packet_in {
    struct ofp_header header;
    uint32_t buffer_id;     /* ID assigned by datapath. */
    uint16_t total_len;     /* Full length of frame. */
    uint8_t reason;         /* Reason packet is being sent (one of OFPR_*) */
    uint8_t table_id;       /* ID of the table that was looked up */
    uint64_t cookie;        /* Cookie of the flow entry that was looked up. */
    struct ofp_match match; /* Packet metadata. Variable size. */
    /* The variable size and padded match is always followed by:
     *   - Exactly 2 all-zero padding bytes, then
     *   - An Ethernet frame whose length is inferred from header.length.
     * The padding bytes preceding the Ethernet frame ensure that the IP
     * header (if any) following the Ethernet header is 32-bit aligned.
     */
    //uint8_t pad[2];       /* Align to 64 bit + 16 bit */
    //uint8_t data[0];      /* Ethernet frame */
};
OFP_ASSERT(sizeof(struct ofp_packet_in) == 32);

/* Why was this flow removed? */
enum ofp_flow_removed_reason {
    OFPRR_IDLE_TIMEOUT = 0,     /* Flow idle time exceeded idle_timeout. */
    OFPRR_HARD_TIMEOUT = 1,     /* Time exceeded hard_timeout. */
    OFPRR_DELETE       = 2,     /* Evicted by a DELETE flow mod. */
    OFPRR_GROUP_DELETE = 3,     /* Group was removed. */
    OFPRR_METER_DELETE = 4,     /* Meter was removed. */
    OFPRR_EVICTION     = 5,     /* Switch eviction to free resources. */
};

/* Flow removed (datapath -> controller). */
struct ofp_flow_removed {
    struct ofp_header header;
```


© 2015; The Open Networking Foundation

```
    uint8_t table_id;         /* ID of the table */
    uint8_t reason;           /* One of OFPRR_*. */
    uint16_t priority;        /* Priority level of flow entry. */
    uint16_t idle_timeout;    /* Idle timeout from original flow mod. */
    uint16_t hard_timeout;    /* Hard timeout from original flow mod. */
    uint64_t cookie;          /* Opaque controller-issued identifier. */
    struct ofp_match match;   /* Description of fields. Variable size. */
    //struct ofp_stats stats; /* Statistics list. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_removed) == 32);

/* Meter numbering. Flow meters can use any number up to OFPM_MAX. */
enum ofp_meter {
    /* Last usable meter. */
    OFPM_MAX        = 0xffff0000,

    /* Virtual meters. */
    OFPM_SLOWPATH   = 0xfffffffd,  /* Meter for slow datapath. */
    OFPM_CONTROLLER = 0xfffffffe,  /* Meter for controller connection. */
    OFPM_ALL        = 0xffffffff,  /* Represents all meters for stat requests
                                      commands. */
};

/* Meter band types */
enum ofp_meter_band_type {
    OFPMBT_DROP           = 1,      /* Drop packet. */
    OFPMBT_DSCP_REMARK    = 2,      /* Remark DSCP in the IP header. */
    OFPMBT_EXPERIMENTER   = 0xFFFF  /* Experimenter meter band. */
};

/* Common header for all meter bands */
struct ofp_meter_band_header {
    uint16_t        type;    /* One of OFPMBT_*. */
    uint16_t        len;     /* Length in bytes of this band. */
    uint32_t        rate;    /* Rate for this band. */
    uint32_t        burst_size; /* Size of bursts. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_header) == 12);

/* OFPMBT_DROP band - drop packets */
struct ofp_meter_band_drop {
    uint16_t        type;    /* OFPMBT_DROP. */
    uint16_t        len;     /* Length is 16. */
    uint32_t        rate;    /* Rate for dropping packets. */
    uint32_t        burst_size; /* Size of bursts. */
    uint8_t         pad[4];
};
OFP_ASSERT(sizeof(struct ofp_meter_band_drop) == 16);

/* OFPMBT_DSCP_REMARK band - Remark DSCP in the IP header */
struct ofp_meter_band_dscp_remark {
    uint16_t        type;    /* OFPMBT_DSCP_REMARK. */
    uint16_t        len;     /* Length is 16. */
    uint32_t        rate;    /* Rate for remarking packets. */
    uint32_t        burst_size; /* Size of bursts. */
    uint8_t         prec_level; /* Number of drop precedence level to add. */
    uint8_t         pad[3];
};
OFP_ASSERT(sizeof(struct ofp_meter_band_dscp_remark) == 16);

/* OFPMBT_EXPERIMENTER band - Experimenter type.
 * The rest of the band is experimenter-defined. */
struct ofp_meter_band_experimenter {
    uint16_t        type;    /* OFPMBT_EXPERIMENTER. */
    uint16_t        len;     /* Length in bytes of this band. */
    uint32_t        rate;    /* Rate for this band. */
    uint32_t        burst_size;   /* Size of bursts. */
    uint32_t        experimenter; /* Experimenter ID. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_experimenter) == 16);

/* Meter commands */
enum ofp_meter_mod_command {
    OFPMC_ADD    = 0,       /* New meter. */
    OFPMC_MODIFY = 1,       /* Modify specified meter. */
    OFPMC_DELETE = 2,       /* Delete specified meter. */
};

/* Meter configuration flags */
enum ofp_meter_flags {
    OFPMF_KBPS        = 1 << 0,  /* Rate value in kb/s (kilo-bit per second). */
    OFPMF_PKTPS       = 1 << 1,  /* Rate value in packet/sec. */
    OFPMF_BURST       = 1 << 2,  /* Do burst size. */
    OFPMF_STATS       = 1 << 3,  /* Collect statistics. */
};

/* Meter configuration. OFPT_METER_MOD. */
struct ofp_meter_mod {
    struct ofp_header header;
    uint16_t            command;        /* One of OFPMC_*. */
```

 © 2015; The Open Networking Foundation

```
    uint16_t            flags;          /* Bitmap of OFPMF_* flags. */
    uint32_t            meter_id;       /* Meter instance. */
    struct ofp_meter_band_header bands[0]; /* The band list length is
                                          inferred from the length field
                                          in the header. */
};
OFP_ASSERT(sizeof(struct ofp_meter_mod) == 16);

/* ## ---------------- ## */
/* ## OpenFlow Errors. ## */
/* ## ---------------- ## */

/* Values for 'type' in ofp_error_message.  These values are immutable: they
 * will not change in future versions of the protocol (although new values may
 * be added). */
enum ofp_error_type {
    OFPET_HELLO_FAILED         = 0,  /* Hello protocol failed. */
    OFPET_BAD_REQUEST          = 1,  /* Request was not understood. */
    OFPET_BAD_ACTION           = 2,  /* Error in action description. */
    OFPET_BAD_INSTRUCTION      = 3,  /* Error in instruction list. */
    OFPET_BAD_MATCH            = 4,  /* Error in match. */
    OFPET_FLOW_MOD_FAILED      = 5,  /* Problem modifying flow entry. */
    OFPET_GROUP_MOD_FAILED     = 6,  /* Problem modifying group entry. */
    OFPET_PORT_MOD_FAILED      = 7,  /* Port mod request failed. */
    OFPET_TABLE_MOD_FAILED     = 8,  /* Table mod request failed. */
    OFPET_QUEUE_OP_FAILED      = 9,  /* Queue operation failed. */
    OFPET_SWITCH_CONFIG_FAILED = 10, /* Switch config request failed. */
    OFPET_ROLE_REQUEST_FAILED  = 11, /* Controller Role request failed. */
    OFPET_METER_MOD_FAILED     = 12, /* Error in meter. */
    OFPET_TABLE_FEATURES_FAILED = 13, /* Setting table features failed. */
    OFPET_BAD_PROPERTY         = 14, /* Some property is invalid. */
    OFPET_ASYNC_CONFIG_FAILED  = 15, /* Asynchronous config request failed. */
    OFPET_FLOW_MONITOR_FAILED  = 16, /* Setting flow monitor failed. */
    OFPET_BUNDLE_FAILED        = 17, /* Bundle operation failed. */
    OFPET_EXPERIMENTER = 0xffff      /* Experimenter error messages. */
};

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

/* ofp_error_msg 'code' values for OFPET_BAD_REQUEST.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_request_code {
    OFPBRC_BAD_VERSION      = 0,  /* ofp_header.version not supported. */
    OFPBRC_BAD_TYPE         = 1,  /* ofp_header.type not supported. */
    OFPBRC_BAD_MULTIPART    = 2,  /* ofp_multipart_request.type not
                                     supported. */
    OFPBRC_BAD_EXPERIMENTER = 3,  /* Experimenter id not supported
                                   * (in ofp_experimenter_header or
                                   * ofp_multipart_request or
                                   * ofp_multipart_reply). */
    OFPBRC_BAD_EXP_TYPE     = 4,  /* Experimenter type not supported. */
    OFPBRC_EPERM            = 5,  /* Permissions error. */
    OFPBRC_BAD_LEN          = 6,  /* Wrong request length for type. */
    OFPBRC_BUFFER_EMPTY     = 7,  /* Specified buffer has already been used. */
    OFPBRC_BUFFER_UNKNOWN   = 8,  /* Specified buffer does not exist. */
    OFPBRC_BAD_TABLE_ID     = 9,  /* Specified table-id invalid or does not
                                   * exist. */
    OFPBRC_IS_SLAVE         = 10, /* Denied because controller is slave. */
    OFPBRC_BAD_PORT         = 11, /* Invalid port or missing port. */
    OFPBRC_BAD_PACKET       = 12, /* Invalid packet in packet-out. */
    OFPBRC_MULTIPART_BUFFER_OVERFLOW    = 13, /* ofp_multipart_request
                                     overflowed the assigned buffer. */
    OFPBRC_MULTIPART_REQUEST_TIMEOUT = 14,  /* Timeout during multipart
                                               request. */
    OFPBRC_MULTIPART_REPLY_TIMEOUT = 15, /* Timeout during multipart reply. */
    OFPBRC_MULTIPART_BAD_SCHED = 16, /* Switch received a OFPMP_BUNDLE_FEATURES
                                        request and failed to update the
                                        scheduling tolerance. */
    OFPBRC_PIPELINE_FIELDS_ONLY = 17, /* Match fields must include only
                                         pipeline fields. */
    OFPBRC_UNKNOWN          = 18, /* Unspecified error. */
};

/* ofp_error_msg 'code' values for OFPET_BAD_ACTION.  'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_action_code {
    OFPBAC_BAD_TYPE           = 0,  /* Unknown or unsupported action type. */
    OFPBAC_BAD_LEN            = 1,  /* Length problem in actions. */
    OFPBAC_BAD_EXPERIMENTER   = 2,  /* Unknown experimenter id specified. */
    OFPBAC_BAD_EXP_TYPE       = 3,  /* Unknown action for experimenter id. */
    OFPBAC_BAD_OUT_PORT       = 4,  /* Problem validating output port. */
    OFPBAC_BAD_ARGUMENT       = 5,  /* Bad action argument. */
    OFPBAC_EPERM              = 6,  /* Permissions error. */
    OFPBAC_TOO_MANY           = 7,  /* Can't handle this many actions. */
    OFPBAC_BAD_QUEUE          = 8,  /* Problem validating output queue. */
```

 © 2015; The Open Networking Foundation

```
    OFPBAC_BAD_OUT_GROUP      = 9,  /* Invalid group id in group action. */
    OFPBAC_MATCH_INCONSISTENT = 10, /* Action can't apply for this match,
                                       or Set-Field missing prerequisite. */
    OFPBAC_UNSUPPORTED_ORDER  = 11, /* Action order is unsupported for the
                                       action list in an Apply-Actions instruction */
    OFPBAC_BAD_TAG            = 12, /* Actions uses an unsupported
                                       tag/encap. */
    OFPBAC_BAD_SET_TYPE       = 13, /* Unsupported type in SET_FIELD action. */
    OFPBAC_BAD_SET_LEN        = 14, /* Length problem in SET_FIELD action. */
    OFPBAC_BAD_SET_ARGUMENT   = 15, /* Bad argument in SET_FIELD action. */
    OFPBAC_BAD_SET_MASK       = 16, /* Bad mask in SET_FIELD action. */
    OFPBAC_BAD_METER          = 17, /* Invalid meter id in meter action. */
};

/* ofp_error_msg 'code' values for OFPET_BAD_INSTRUCTION.  'data' contains at
 * least the first 64 bytes of the failed request. */
enum ofp_bad_instruction_code {
    OFPBIC_UNKNOWN_INST     = 0, /* Unknown instruction. */
    OFPBIC_UNSUP_INST       = 1, /* Switch or table does not support the
                                    instruction. */
    OFPBIC_BAD_TABLE_ID     = 2, /* Invalid Table-ID specified. */
    OFPBIC_UNSUP_METADATA   = 3, /* Metadata value unsupported by datapath. */
    OFPBIC_UNSUP_METADATA_MASK = 4, /* Metadata mask value unsupported by
                                       datapath. */
    OFPBIC_BAD_EXPERIMENTER = 5, /* Unknown experimenter id specified. */
    OFPBIC_BAD_EXP_TYPE     = 6, /* Unknown instruction for experimenter id. */
    OFPBIC_BAD_LEN          = 7, /* Length problem in instructions. */
    OFPBIC_EPERM            = 8, /* Permissions error. */
    OFPBIC_DUP_INST         = 9, /* Duplicate instruction. */
};

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

/* ofp_error_msg 'code' values for OFPET_FLOW_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_flow_mod_failed_code {
    OFPFMFC_UNKNOWN      = 0,   /* Unspecified error. */
    OFPFMFC_TABLE_FULL   = 1,   /* Flow not added because table was full. */
    OFPFMFC_BAD_TABLE_ID = 2,   /* Table does not exist */
    OFPFMFC_OVERLAP      = 3,   /* Attempted to add overlapping flow with
                                   CHECK_OVERLAP flag set. */
    OFPFMFC_EPERM        = 4,   /* Permissions error. */
    OFPFMFC_BAD_TIMEOUT  = 5,   /* Flow not added because of unsupported
                                   idle/hard timeout. */
    OFPFMFC_BAD_COMMAND  = 6,   /* Unsupported or unknown command. */
    OFPFMFC_BAD_FLAGS    = 7,   /* Unsupported or unknown flags. */
    OFPFMFC_CANT_SYNC    = 8,   /* Problem in table synchronisation. */
    OFPFMFC_BAD_PRIORITY = 9,   /* Unsupported priority value. */
    OFPFMFC_IS_SYNC      = 10,  /* Synchronised flow entry is read only. */
};

/* ofp_error_msg 'code' values for OFPET_GROUP_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_group_mod_failed_code {
    OFPGMFC_GROUP_EXISTS         = 0,  /* Group not added because a group ADD
                                          attempted to replace an
                                          already-present group. */
    OFPGMFC_INVALID_GROUP        = 1,  /* Group not added because Group
                                          specified is invalid. */
    OFPGMFC_WEIGHT_UNSUPPORTED   = 2,  /* Switch does not support unequal load
                                          sharing with select groups. */
    OFPGMFC_OUT_OF_GROUPS        = 3,  /* The group table is full. */
    OFPGMFC_OUT_OF_BUCKETS       = 4,  /* The maximum number of action buckets
                                          for a group has been exceeded. */
    OFPGMFC_CHAINING_UNSUPPORTED = 5,  /* Switch does not support groups that
                                          forward to groups. */
    OFPGMFC_WATCH_UNSUPPORTED    = 6,  /* This group cannot watch the watch_port
                                          or watch_group specified. */
    OFPGMFC_LOOP                 = 7,  /* Group entry would cause a loop. */
```

 © 2015; The Open Networking Foundation

```
    OFPGMFC_UNKNOWN_GROUP       = 8,  /* Group not modified because a group
                                         MODIFY attempted to modify a
                                         non-existent group. */
    OFPGMFC_CHAINED_GROUP       = 9,  /* Group not deleted because another
                                         group is forwarding to it. */
    OFPGMFC_BAD_TYPE            = 10, /* Unsupported or unknown group type. */
    OFPGMFC_BAD_COMMAND         = 11, /* Unsupported or unknown command. */
    OFPGMFC_BAD_BUCKET          = 12, /* Error in bucket. */
    OFPGMFC_BAD_WATCH           = 13, /* Error in watch port/group. */
    OFPGMFC_EPERM               = 14, /* Permissions error. */
    OFPGMFC_UNKNOWN_BUCKET      = 15, /* Invalid bucket identifier used in
                                         INSERT BUCKET or REMOVE BUCKET
                                         command. */
    OFPGMFC_BUCKET_EXISTS       = 16, /* Can't insert bucket because a bucket
                                         already exist with that bucket-id. */
};

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

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

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

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

/* ofp_error_msg 'code' values for OFPET_ROLE_REQUEST_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_role_request_failed_code {
    OFPRRFC_STALE      = 0,      /* Stale Message: old generation_id. */
    OFPRRFC_UNSUP      = 1,      /* Controller role change unsupported. */
    OFPRRFC_BAD_ROLE   = 2,      /* Invalid role. */
    OFPRRFC_ID_UNSUP   = 3,      /* Switch doesn't support changing ID. */
    OFPRRFC_ID_IN_USE  = 4,      /* Requested ID is in use. */
};

/* ofp_error_msg 'code' values for OFPET_METER_MOD_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_meter_mod_failed_code {
    OFPMMFC_UNKNOWN       = 0,  /* Unspecified error. */
    OFPMMFC_METER_EXISTS  = 1,  /* Meter not added because a Meter ADD
                                 * attempted to replace an existing Meter. */
    OFPMMFC_INVALID_METER = 2,  /* Meter not added because Meter specified
                                 * is invalid,
                                 * or invalid meter in meter action. */
    OFPMMFC_UNKNOWN_METER = 3,  /* Meter not modified because a Meter MODIFY
                                 * attempted to modify a non-existent Meter,
                                 * or bad meter in meter action. */
    OFPMMFC_BAD_COMMAND   = 4,  /* Unsupported or unknown command. */
    OFPMMFC_BAD_FLAGS     = 5,  /* Flag configuration unsupported. */
    OFPMMFC_BAD_RATE      = 6,  /* Rate unsupported. */
    OFPMMFC_BAD_BURST     = 7,  /* Burst size unsupported. */
    OFPMMFC_BAD_BAND      = 8,  /* Band unsupported. */
    OFPMMFC_BAD_BAND_VALUE = 9, /* Band value unsupported. */
    OFPMMFC_OUT_OF_METERS = 10, /* No more meters available. */
    OFPMMFC_OUT_OF_BANDS  = 11, /* The maximum number of properties
                                 * for a meter has been exceeded. */
};

/* ofp_error_msg 'code' values for OFPET_TABLE_FEATURES_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_table_features_failed_code {
    OFPTFFC_BAD_TABLE    = 0,      /* Specified table does not exist. */
    OFPTFFC_BAD_METADATA = 1,      /* Invalid metadata mask. */
```


© 2015; The Open Networking Foundation

```
    OFPTFFC_EPERM        = 5,      /* Permissions error. */
    OFPTFFC_BAD_CAPA     = 6,      /* Invalid capability field. */
    OFPTFFC_BAD_MAX_ENT  = 7,      /* Invalid max_entries field. */
    OFPTFFC_BAD_FEATURES = 8,      /* Invalid features field. */
    OFPTFFC_BAD_COMMAND  = 9,      /* Invalid command. */
    OFPTFFC_TOO_MANY     = 10,     /* Can't handle this many flow tables. */
};

/* ofp_error_msg 'code' values for OFPET_BAD_PROPERTY. 'data' contains at least
 * the first 64 bytes of the failed request. */
enum ofp_bad_property_code {
    OFPBPC_BAD_TYPE           = 0,  /* Unknown or unsupported property type. */
    OFPBPC_BAD_LEN            = 1,  /* Length problem in property. */
    OFPBPC_BAD_VALUE          = 2,  /* Unsupported property value. */
    OFPBPC_TOO_MANY           = 3,  /* Can't handle this many properties. */
    OFPBPC_DUP_TYPE           = 4,  /* A property type was duplicated. */
    OFPBPC_BAD_EXPERIMENTER   = 5,  /* Unknown experimenter id specified. */
    OFPBPC_BAD_EXP_TYPE       = 6,  /* Unknown exp_type for experimenter id. */
    OFPBPC_BAD_EXP_VALUE      = 7,  /* Unknown value for experimenter id. */
    OFPBPC_EPERM              = 8,  /* Permissions error. */
};

/* ofp_error_msg 'code' values for OFPET_ASYNC_CONFIG_FAILED. 'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_async_config_failed_code {
    OFPACFC_INVALID      = 0,      /* One mask is invalid. */
    OFPACFC_UNSUPPORTED  = 1,      /* Requested configuration not supported. */
    OFPACFC_EPERM        = 2,      /* Permissions error. */
};

/* ofp_error_msg 'code' values for OFPET_FLOW_MONITOR_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_flow_monitor_failed_code {
    OFPMOFC_UNKNOWN       = 0,  /* Unspecified error. */
    OFPMOFC_MONITOR_EXISTS = 1, /* Monitor not added because a Monitor ADD
                                 * attempted to replace an existing Monitor. */
    OFPMOFC_INVALID_MONITOR = 2, /* Monitor not added because Monitor specified
                                 * is invalid. */
    OFPMOFC_UNKNOWN_MONITOR = 3, /* Monitor not modified because a Monitor
                                   MODIFY attempted to modify a non-existent
                                   Monitor. */
    OFPMOFC_BAD_COMMAND   = 4,  /* Unsupported or unknown command. */
    OFPMOFC_BAD_FLAGS     = 5,  /* Flag configuration unsupported. */
    OFPMOFC_BAD_TABLE_ID  = 6,  /* Specified table does not exist. */
    OFPMOFC_BAD_OUT       = 7,  /* Error in output port/group. */
};

/* ofp_error_msg 'code' values for OFPET_BUNDLE_FAILED.  'data' contains
 * at least the first 64 bytes of the failed request. */
enum ofp_bundle_failed_code {
  OFPBFC_UNKNOWN        = 0,  /* Unspecified error. */
  OFPBFC_EPERM          = 1,  /* Permissions error. */
  OFPBFC_BAD_ID         = 2,  /* Bundle ID doesn't exist. */
  OFPBFC_BUNDLE_EXIST   = 3,  /* Bundle ID already exist. */
  OFPBFC_BUNDLE_CLOSED  = 4,  /* Bundle ID is closed. */
  OFPBFC_OUT_OF_BUNDLES = 5,  /* Too many bundles IDs. */
  OFPBFC_BAD_TYPE       = 6,  /* Unsupported or unknown message control type. */
  OFPBFC_BAD_FLAGS      = 7,  /* Unsupported, unknown, or inconsistent flags. */
  OFPBFC_MSG_BAD_LEN    = 8,  /* Length problem in included message. */
  OFPBFC_MSG_BAD_XID    = 9,  /* Inconsistent or duplicate XID. */
  OFPBFC_MSG_UNSUP      = 10, /* Unsupported message in this bundle. */
  OFPBFC_MSG_CONFLICT   = 11, /* Unsupported message combination in this
                                 bundle. */
  OFPBFC_MSG_TOO_MANY   = 12, /* Can't handle this many messages in bundle. */
  OFPBFC_MSG_FAILED     = 13, /* One message in bundle failed. */
  OFPBFC_TIMEOUT        = 14, /* Bundle is taking too long. */
  OFPBFC_BUNDLE_IN_PROGRESS = 15,  /* Bundle is locking the resource. */
  OFPBFC_SCHED_NOT_SUPPORTED = 16, /* Scheduled commit was received and
                                      scheduling is not supported. */
  OFPBFC_SCHED_FUTURE = 17,   /* Scheduled commit time exceeds upper bound. */
  OFPBFC_SCHED_PAST = 18,     /* Scheduled commit time exceeds lower bound. */
};

/* OFPT_ERROR: Error message. */
struct ofp_error_msg {
    struct ofp_header header;

    uint16_t type;
    uint16_t code;
    uint8_t data[0];          /* Variable-length data.  Interpreted based
                                 on the type and code.  No padding. */
};
OFP_ASSERT(sizeof(struct ofp_error_msg) == 12);

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

    uint16_t type;            /* OFPET_EXPERIMENTER. */
    uint16_t exp_code;        /* Experimenter defined. */
```

 © 2015; The Open Networking Foundation

```
    uint32_t experimenter;    /* Experimenter ID. */
    uint8_t data[0];          /* Variable-length data.  Interpreted based
                                 on the type and experimenter.  No padding. */
};
OFP_ASSERT(sizeof(struct ofp_error_experimenter_msg) == 16);

/* ## -------------------- ## */
/* ## OpenFlow Multiparts. ## */
/* ## -------------------- ## */

enum ofp_multipart_type {
    /* Description of this OpenFlow switch.
     * The request body is empty.
     * The reply body is struct ofp_desc. */
    OFPMP_DESC = 0,

    /* Individual flow descriptions and statistics.
     * The request body is struct ofp_flow_stats_request.
     * The reply body is an array of struct ofp_flow_desc. */
    OFPMP_FLOW_DESC = 1,

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

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

    /* Port statistics.
     * The request body is struct ofp_port_multipart_request.
     * The reply body is an array of struct ofp_port_stats. */
    OFPMP_PORT_STATS = 4,

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

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

    /* Group description.
     * The request body is struct ofp_group_multipart_request.
     * The reply body is an array of struct ofp_group_desc. */
    OFPMP_GROUP_DESC = 7,

    /* Group features.
     * The request body is empty.
     * The reply body is struct ofp_group_features. */
    OFPMP_GROUP_FEATURES = 8,

    /* Meter statistics.
     * The request body is struct ofp_meter_multipart_request.
     * The reply body is an array of struct ofp_meter_stats. */
    OFPMP_METER_STATS = 9,

    /* Meter configuration.
     * The request body is struct ofp_meter_multipart_request.
     * The reply body is an array of struct ofp_meter_desc. */
    OFPMP_METER_DESC = 10,

    /* Meter features.
     * The request body is empty.
     * The reply body is struct ofp_meter_features. */
    OFPMP_METER_FEATURES = 11,

    /* Table features.
     * The request body is either empty or contains an array of
     * struct ofp_table_features containing the controller's
     * desired view of the switch. If the switch is unable to
     * set the specified view an error is returned.
     * The reply body is an array of struct ofp_table_features. */
    OFPMP_TABLE_FEATURES = 12,

    /* Port description.
     * The request body is struct ofp_port_multipart_request.
     * The reply body is an array of struct ofp_port. */
    OFPMP_PORT_DESC = 13,

    /* Table description.
     * The request body is empty.
     * The reply body is an array of struct ofp_table_desc. */
    OFPMP_TABLE_DESC = 14,

    /* Queue description.
```

 © 2015; The Open Networking Foundation

```
     * The request body is struct ofp_queue_multipart_request.
     * The reply body is an array of struct ofp_queue_desc. */
    OFPMP_QUEUE_DESC = 15,

    /* Flow monitors. Reply may be an asynchronous message.
     * The request body is an array of struct ofp_flow_monitor_request.
     * The reply body is an array of struct ofp_flow_update_header. */
    OFPMP_FLOW_MONITOR = 16,

    /* Individual flow statistics (without description).
     * The request body is struct ofp_flow_stats_request.
     * The reply body is an array of struct ofp_flow_stats. */
    OFPMP_FLOW_STATS = 17,

    /* Controller status.
     * The request body is empty.
     * The reply body is an array of struct ofp_controller_status. */
    OFPMP_CONTROLLER_STATUS = 18,

    /* Bundle features.
    * The request body is ofp_bundle_features_request.
    * The reply body is struct ofp_bundle_features. */
    OFPMP_BUNDLE_FEATURES = 19,

    /* Experimenter extension.
     * The request and reply bodies begin with
     * struct ofp_experimenter_multipart_header.
     * The request and reply bodies are otherwise experimenter-defined. */
    OFPMP_EXPERIMENTER = 0xffff
};

/* Backward compatibility with 1.3.1 - avoid breaking the API. */
#define ofp_multipart_types ofp_multipart_type

enum ofp_multipart_request_flags {
    OFPMPF_REQ_MORE  = 1 << 0  /* More requests to follow. */
};

struct ofp_multipart_request {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPMP_* constants. */
    uint16_t flags;             /* OFPMPF_REQ_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the request. 0 or more bytes. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_request) == 16);

enum ofp_multipart_reply_flags {
    OFPMPF_REPLY_MORE  = 1 << 0  /* More replies to follow. */
};

struct ofp_multipart_reply {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPMP_* constants. */
    uint16_t flags;             /* OFPMPF_REPLY_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the reply. 0 or more bytes. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_reply) == 16);

#define DESC_STR_LEN   256
#define SERIAL_NUM_LEN 32
/* Body of reply to OFPMP_DESC request.  Each entry is a NULL-terminated
 * ASCII string. */
struct ofp_desc {
    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) == 1056);

/* Body for ofp_multipart_request of type OFPMP_FLOW_DESC & OFPMP_FLOW_STATS. */
struct ofp_flow_stats_request {
    uint8_t table_id;         /* ID of table to read (from ofp_table_desc),
                                 OFPTT_ALL for all tables. */
    uint8_t pad[3];           /* Align to 32 bits. */
    uint32_t out_port;        /* Require matching entries to include this
                                 as an output port.  A value of OFPP_ANY
                                 indicates no restriction. */
    uint32_t out_group;       /* Require matching entries to include this
                                 as an output group.  A value of OFPG_ANY
                                 indicates no restriction. */
    uint8_t pad2[4];          /* Align to 64 bits. */
    uint64_t cookie;          /* Require matching entries to contain this
                                 cookie value */
    uint64_t cookie_mask;     /* Mask used to restrict the cookie bits that
                                 must match. A value of 0 indicates
```

 © 2015; The Open Networking Foundation

```
                                no restriction. */
    struct ofp_match match;   /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats_request) == 40);

/* Body of reply to OFPMP_FLOW_DESC request. */
struct ofp_flow_desc {
    uint16_t length;          /* Length of this entry. */
    uint8_t pad2[2];          /* Align to 64-bits. */
    uint8_t table_id;         /* ID of table flow came from. */
    uint8_t pad;
    uint16_t priority;        /* Priority of the entry. */
    uint16_t idle_timeout;    /* Number of seconds idle before expiration. */
    uint16_t hard_timeout;    /* Number of seconds before expiration. */
    uint16_t flags;           /* Bitmap of OFPFF_* flags. */
    uint16_t importance;      /* Eviction precedence. */
    uint64_t cookie;          /* Opaque controller-issued identifier. */
    struct ofp_match match;   /* Description of fields. Variable size. */
    //struct ofp_stats stats; /* Statistics list. Variable size. */
    //struct ofp_instruction_header instructions[0];
                              /* Instruction set - 0 or more. */
};
OFP_ASSERT(sizeof(struct ofp_flow_desc) == 32);

/* Reason for generating flow stats. */
enum ofp_flow_stats_reason {
    OFPFSR_STATS_REQUEST = 0,   /* Reply to a OFPMP_FLOW_STATS request. */
    OFPFSR_STAT_TRIGGER  = 1,   /* Status generated by OFPIT_STAT_TRIGGER. */
};

/* Body of reply to OFPMP_FLOW_STATS request
 * and body for OFPIT_STAT_TRIGGER generated status. */
struct ofp_flow_stats {
    uint16_t length;          /* Length of this entry. */
    uint8_t pad2[2];          /* Align to 64-bits. */
    uint8_t table_id;         /* ID of table flow came from. */
    uint8_t reason;           /* One of OFPFSR_*. */
    uint16_t priority;        /* Priority of the entry. */
    struct ofp_match match;   /* Description of fields. Variable size. */
    //struct ofp_stats stats; /* Statistics list. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_stats) == 16);

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

/* Body of reply to OFPMP_AGGREGATE_STATS request. */
struct ofp_aggregate_stats_reply {
    struct ofp_stats stats;   /* Aggregated statistics list. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_aggregate_stats_reply) == 8);

/* Table Feature property types.
 * Low order bit cleared indicates a property for a regular Flow Entry.
 * Low order bit set indicates a property for the Table-Miss Flow Entry.
 */
enum ofp_table_feature_prop_type {
    OFPTFPT_INSTRUCTIONS           = 0,  /* Instructions property. */
    OFPTFPT_INSTRUCTIONS_MISS      = 1,  /* Instructions for table-miss. */
    OFPTFPT_NEXT_TABLES            = 2,  /* Next Table property. */
    OFPTFPT_NEXT_TABLES_MISS       = 3,  /* Next Table for table-miss. */
    OFPTFPT_WRITE_ACTIONS          = 4,  /* Write Actions property. */
    OFPTFPT_WRITE_ACTIONS_MISS     = 5,  /* Write Actions for table-miss. */
    OFPTFPT_APPLY_ACTIONS          = 6,  /* Apply Actions property. */
    OFPTFPT_APPLY_ACTIONS_MISS     = 7,  /* Apply Actions for table-miss. */
    OFPTFPT_MATCH                  = 8,  /* Match property. */
    OFPTFPT_WILDCARDS              = 10, /* Wildcards property. */
    OFPTFPT_WRITE_SETFIELD         = 12, /* Write Set-Field property. */
    OFPTFPT_WRITE_SETFIELD_MISS    = 13, /* Write Set-Field for table-miss. */
    OFPTFPT_APPLY_SETFIELD         = 14, /* Apply Set-Field property. */
    OFPTFPT_APPLY_SETFIELD_MISS    = 15, /* Apply Set-Field for table-miss. */
    OFPTFPT_TABLE_SYNC_FROM        = 16, /* Table synchronisation property. */
```

 © 2015; The Open Networking Foundation

```
    OFPTFPT_WRITE_COPYFIELD        = 18, /* Write Copy-Field property. */
    OFPTFPT_WRITE_COPYFIELD_MISS   = 19, /* Write Copy-Field for table-miss. */
    OFPTFPT_APPLY_COPYFIELD        = 20, /* Apply Copy-Field property. */
    OFPTFPT_APPLY_COPYFIELD_MISS   = 21, /* Apply Copy-Field for table-miss. */
    OFPTFPT_PACKET_TYPES           = 22, /* Packet types property. */
    OFPTFPT_EXPERIMENTER           = 0xFFFE, /* Experimenter property. */
    OFPTFPT_EXPERIMENTER_MISS      = 0xFFFF, /* Experimenter for table-miss. */
};

/* Common header for all Table Feature Properties */
struct ofp_table_feature_prop_header {
    uint16_t         type;    /* One of OFPTFPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_header) == 4);

/* Instruction ID */
struct ofp_instruction_id {
    uint16_t type;                /* One of OFPIT_*. */
    uint16_t len;                 /* Length is 4 or experimenter defined. */
    uint8_t exp_data[0];          /* Optional experimenter id + data. */
};
OFP_ASSERT(sizeof(struct ofp_instruction_id) == 4);

/* Instructions property */
struct ofp_table_feature_prop_instructions {
    uint16_t         type;    /* One of OFPTFPT_INSTRUCTIONS,
                                 OFPTFPT_INSTRUCTIONS_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the instruction ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    struct ofp_instruction_id  instruction_ids[0];   /* List of instructions */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_instructions) == 4);

/* Next Tables and Table Synchronise From properties */
struct ofp_table_feature_prop_tables {
    uint16_t         type;    /* One of OFPTFPT_NEXT_TABLES,
                                 OFPTFPT_NEXT_TABLES_MISS,
                                 OFPTFPT_TABLE_SYNC_FROM. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the table_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint8_t          table_ids[0];        /* List of table ids. */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_tables) == 4);

/* Action ID */
struct ofp_action_id {
    uint16_t type;                /* One of OFPAT_*. */
    uint16_t len;                 /* Length is 4 or experimenter defined. */
    uint8_t exp_data[0];          /* Optional experimenter id + data. */
};
OFP_ASSERT(sizeof(struct ofp_action_id) == 4);

/* Actions property */
struct ofp_table_feature_prop_actions {
    uint16_t         type;    /* One of OFPTFPT_WRITE_ACTIONS,
                                 OFPTFPT_WRITE_ACTIONS_MISS,
                                 OFPTFPT_APPLY_ACTIONS,
                                 OFPTFPT_APPLY_ACTIONS_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the action_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    struct ofp_action_id  action_ids[0];      /* List of actions */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_actions) == 4);

/* Match, Wildcard or Set-Field property */
struct ofp_table_feature_prop_oxm {
    uint16_t         type;    /* One of OFPTFPT_MATCH,
                                 OFPTFPT_WILDCARDS,
                                 OFPTFPT_WRITE_SETFIELD,
                                 OFPTFPT_WRITE_SETFIELD_MISS,
                                 OFPTFPT_APPLY_SETFIELD,
                                 OFPTFPT_APPLY_SETFIELD_MISS,
                                 OFPTFPT_WRITE_COPYFIELD,
                                 OFPTFPT_WRITE_COPYFIELD_MISS,
                                 OFPTFPT_APPLY_COPYFIELD,
                                 OFPTFPT_APPLY_COPYFIELD_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the oxm_ids, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
```

 © 2015; The Open Networking Foundation

```
     *     bytes of all-zero bytes */
    uint32_t         oxm_ids[0];   /* Array of OXM headers */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_oxm) == 4);

/* Packet types property */
struct ofp_table_feature_prop_oxm_values {
    uint16_t         type;    /* OFPTFPT_PACKET_TYPES. */
    uint16_t         length;  /* Length in bytes of this property. */
    /* Followed by:
     *   - Exactly (length - 4) bytes containing the oxm values, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         oxm_values[0];   /* Array of OXM values */
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_oxm_values) == 4);

/* Experimenter table feature property */
struct ofp_table_feature_prop_experimenter {
    uint16_t         type;    /* One of OFPTFPT_EXPERIMENTER,
                                 OFPTFPT_EXPERIMENTER_MISS. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_table_feature_prop_experimenter) == 12);

/* Flags of features supported by the table. */
enum ofp_table_feature_flag {
    OFPTFF_INGRESS_TABLE     = 1 << 0, /* Can be configured as ingress table. */
    OFPTFF_EGRESS_TABLE      = 1 << 1, /* Can be configured as egress table. */
    OFPTFF_FIRST_EGRESS      = 1 << 4, /* Is the first egress table. */
};

/* Table Features request commands */
enum ofp_table_features_command {
    OFPTFC_REPLACE = 0,       /* Replace full pipeline. */
    OFPTFC_MODIFY  = 1,       /* Modify flow tables capabilities. */
    OFPTFC_ENABLE  = 2,       /* Enable flow tables in the pipeline. */
    OFPTFC_DISABLE = 3,       /* Disable flow tables in pipeline. */
};

/* Body for ofp_multipart_request of type OFPMP_TABLE_FEATURES./
 * Body of reply to OFPMP_TABLE_FEATURES request. */
struct ofp_table_features {
    uint16_t length;         /* Length is padded to 64 bits. */
    uint8_t table_id;        /* Identifier of table.  Lower numbered tables
                                are consulted first. */
    uint8_t command;         /* One of OFPTFC_*. */
    uint32_t features;       /* Bitmap of OFPTFF_* values. */
    char name[OFP_MAX_TABLE_NAME_LEN];
    uint64_t metadata_match; /* Bits of metadata table can match. */
    uint64_t metadata_write; /* Bits of metadata table can write. */
    uint32_t capabilities;   /* Bitmap of OFPTC_* values. */
    uint32_t max_entries;    /* Max number of entries supported. */

    /* Table Feature Property list */
    struct ofp_table_feature_prop_header properties[0]; /* List of properties */
};
OFP_ASSERT(sizeof(struct ofp_table_features) == 64);

/* Body of reply to OFPMP_TABLE_STATS 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. */
    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) == 24);

/* Body of reply to OFPMP_TABLE_DESC request. */
struct ofp_table_desc {
    uint16_t length;         /* Length is padded to 64 bits. */
    uint8_t table_id;        /* Identifier of table.  Lower numbered tables
                                are consulted first. */
    uint8_t pad[1];          /* Align to 32-bits. */
    uint32_t config;         /* Bitmap of OFPTC_* values. */

    /* Table Mod Property list - 0 or more. */
    struct ofp_table_mod_prop_header properties[0];
};
```

 © 2015; The Open Networking Foundation

```
OFP_ASSERT(sizeof(struct ofp_table_desc) == 8);

/* Body for ofp_multipart_request of types OFPMP_PORT_STATS and
 * OFPMP_PORT_DESC. */
struct ofp_port_multipart_request {
    uint32_t port_no;        /* OFPMP_PORT message must request statistics
                              * either for a single port (specified in
                              * port_no) or for all ports (if port_no ==
                              * OFPP_ANY). */
    uint8_t pad[4];
};
OFP_ASSERT(sizeof(struct ofp_port_multipart_request) == 8);

/* Port stats property types.
 */
enum ofp_port_stats_prop_type {
    OFPPSPT_ETHERNET          = 0,      /* Ethernet property. */
    OFPPSPT_OPTICAL           = 1,      /* Optical property. */
    OFPPSPT_EXPERIMENTER      = 0xFFFF, /* Experimenter property. */
};

/* Common header for all port stats properties. */
struct ofp_port_stats_prop_header {
    uint16_t         type;    /* One of OFPPSPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats_prop_header) == 4);

/* Ethernet port stats property. */
struct ofp_port_stats_prop_ethernet {
    uint16_t         type;    /* OFPPSPT_ETHERNET. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint8_t          pad[4];  /* Align to 64 bits. */

    uint64_t rx_frame_err;   /* Number of frame alignment errors. */
    uint64_t rx_over_err;    /* Number of packets with RX overrun. */
    uint64_t rx_crc_err;     /* Number of CRC errors. */
    uint64_t collisions;     /* Number of collisions. */
};
OFP_ASSERT(sizeof(struct ofp_port_stats_prop_ethernet) == 40);

/* Optical port stats property. */
struct ofp_port_stats_prop_optical {
    uint16_t         type;    /* OFPPSPT_OPTICAL. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint8_t          pad[4];  /* Align to 64 bits. */

    uint32_t flags;          /* Features enabled by the port. */
    uint32_t tx_freq_lmda;   /* Current TX Frequency/Wavelength */
    uint32_t tx_offset;      /* TX Offset */
    uint32_t tx_grid_span;   /* TX Grid Spacing */
    uint32_t rx_freq_lmda;   /* Current RX Frequency/Wavelength */
    uint32_t rx_offset;      /* RX Offset */
    uint32_t rx_grid_span;   /* RX Grid Spacing */
    uint16_t tx_pwr;         /* Current TX power */
    uint16_t rx_pwr;         /* Current RX power */
    uint16_t bias_current;   /* TX Bias Current */
    uint16_t temperature;    /* TX Laser Temperature */
};
OFP_ASSERT(sizeof(struct ofp_port_stats_prop_optical) == 44);

/* Flags is one of OFPOSF_ below */
enum ofp_port_stats_optical_flags  {
    OFPOSF_RX_TUNE   = 1 << 0,  /* Receiver tune info valid */
    OFPOSF_TX_TUNE   = 1 << 1,  /* Transmit tune info valid */
    OFPOSF_TX_PWR    = 1 << 2,  /* TX Power is valid */
    OFPOSF_RX_PWR    = 1 << 4,  /* RX power is valid */
    OFPOSF_TX_BIAS   = 1 << 5,  /* Transmit bias is valid */
    OFPOSF_TX_TEMP   = 1 << 6,  /* TX Temp is valid */
};

/* Experimenter port stats property. */
struct ofp_port_stats_prop_experimenter {
    uint16_t         type;    /* OFPPSPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_port_stats_prop_experimenter) == 12);

/* Body of reply to OFPMP_PORT_STATS request. If a counter is unsupported,
 * set the field to all ones. */
struct ofp_port_stats {
```

 © 2015; The Open Networking Foundation

```
    uint16_t length;         /* Length of this entry. */
    uint8_t pad[2];          /* Align to 64 bits. */
    uint32_t port_no;
    uint32_t duration_sec;   /* Time port has been alive in seconds. */
    uint32_t duration_nsec;  /* Time port has been alive in nanoseconds beyond
                                duration_sec. */
    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 in properties. */
    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.) */

    /* Port description property list - 0 or more properties */
    struct ofp_port_stats_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_port_stats) == 80);

/* Body of OFPMP_GROUP_STATS and OFPMP_GROUP_DESC requests. */
struct ofp_group_multipart_request {
    uint32_t group_id;       /* All groups if OFPG_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_group_multipart_request) == 8);

/* Used in group stats replies. */
struct ofp_bucket_counter {
    uint64_t packet_count;   /* Number of packets processed by bucket. */
    uint64_t byte_count;     /* Number of bytes processed by bucket. */
};
OFP_ASSERT(sizeof(struct ofp_bucket_counter) == 16);

/* Body of reply to OFPMP_GROUP_STATS request. */
struct ofp_group_stats {
    uint16_t length;         /* Length of this entry. */
    uint8_t pad[2];          /* Align to 64 bits. */
    uint32_t group_id;       /* Group identifier. */
    uint32_t ref_count;      /* Number of flows or groups that directly forward
                                to this group. */
    uint8_t pad2[4];         /* Align to 64 bits. */
    uint64_t packet_count;   /* Number of packets processed by group. */
    uint64_t byte_count;     /* Number of bytes processed by group. */
    uint32_t duration_sec;   /* Time group has been alive in seconds. */
    uint32_t duration_nsec;  /* Time group has been alive in nanoseconds beyond
                                duration_sec. */
    struct ofp_bucket_counter bucket_stats[0]; /* One counter set per bucket. */
};
OFP_ASSERT(sizeof(struct ofp_group_stats) == 40);

/* Body of reply to OFPMP_GROUP_DESC request. */
struct ofp_group_desc {
    uint16_t length;              /* Length of this entry. */
    uint8_t type;                 /* One of OFPGT_*. */
    uint8_t pad;                  /* Pad to 64 bits. */
    uint32_t group_id;            /* Group identifier. */
    uint16_t bucket_array_len;    /* Length of action buckets data. */
    uint8_t pad2[6];              /* Pad to 64 bits. */
    /* Followed by:
     *   - Exactly 'bucket_array_len' bytes containing an array of
     *     struct ofp_bucket.
     *   - Zero or more bytes of group properties to fill out the overall
     *     length in the length field. */
    struct ofp_bucket buckets[0];   /* List of buckets - 0 or more. */
    //struct ofp_group_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_group_desc) == 16);

/* Backward compatibility with 1.3.1 - avoid breaking the API. */
#define ofp_group_desc_stats ofp_group_desc

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

/* Body of reply to OFPMP_GROUP_FEATURES request. Group features. */
struct ofp_group_features {
    uint32_t  types;           /* Bitmap of (1 << OFPGT_*) values supported. */
```

 © 2015; The Open Networking Foundation

```
    uint32_t  capabilities;    /* Bitmap of OFPGFC_* capability supported. */
    uint32_t  max_groups[4];   /* Maximum number of groups for each type. */
    uint32_t  actions[4];      /* Bitmaps of (1 << OFPAT_*) values supported. */
};
OFP_ASSERT(sizeof(struct ofp_group_features) == 40);

/* Body of OFPMP_METER_STATS and OFPMP_METER_DESC requests. */
struct ofp_meter_multipart_request {
    uint32_t meter_id;       /* Meter instance, or OFPM_ALL. */
    uint8_t pad[4];          /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_meter_multipart_request) == 8);

/* Statistics for each meter band */
struct ofp_meter_band_stats {
    uint64_t        packet_band_count;   /* Number of packets in band. */
    uint64_t        byte_band_count;     /* Number of bytes in band. */
};
OFP_ASSERT(sizeof(struct ofp_meter_band_stats) == 16);

/* Body of reply to OFPMP_METER_STATS request. Meter statistics. */
struct ofp_meter_stats {
    uint32_t        meter_id;         /* Meter instance. */
    uint16_t        len;              /* Length in bytes of this stats. */
    uint8_t         pad[6];
    uint32_t        ref_count;        /* Number of flows or groups that
                                         directly reference this meter. */
    uint64_t        packet_in_count;  /* Number of packets in input. */
    uint64_t        byte_in_count;    /* Number of bytes in input. */
    uint32_t   duration_sec;  /* Time meter has been alive in seconds. */
    uint32_t   duration_nsec; /* Time meter has been alive in nanoseconds beyond
                                 duration_sec. */
    struct ofp_meter_band_stats band_stats[0]; /* The band_stats length is
                                         inferred from the length field. */
};
OFP_ASSERT(sizeof(struct ofp_meter_stats) == 40);

/* Body of reply to OFPMP_METER_DESC request. Meter configuration. */
struct ofp_meter_desc {
    uint16_t        length;           /* Length of this entry. */
    uint16_t        flags;            /* All OFPMF_* that apply. */
    uint32_t        meter_id;         /* Meter instance. */
    struct ofp_meter_band_header bands[0]; /* The bands length is
                                         inferred from the length field. */
};
OFP_ASSERT(sizeof(struct ofp_meter_desc) == 8);

/* Meter feature flags */
enum ofp_meter_feature_flags {
    OFPMFF_ACTION_SET   = 1 << 0, /* Support meter action in action set. */
    OFPMFF_ANY_POSITION = 1 << 1, /* Support any position in action list. */
    OFPMFF_MULTI_LIST   = 1 << 2, /* Support multiple actions in action list. */
};

/* Body of reply to OFPMP_METER_FEATURES request. Meter features. */
struct ofp_meter_features {
    uint32_t    max_meter;    /* Maximum number of meters. */
    uint32_t    band_types;   /* Bitmaps of (1 << OFPMBT_*) values supported. */
    uint32_t    capabilities; /* Bitmaps of "ofp_meter_flags". */
    uint8_t     max_bands;    /* Maximum bands per meters */
    uint8_t     max_color;    /* Maximum color value */
    uint8_t     pad[2];
    uint32_t    features;     /* Bitmaps of "ofp_meter_feature_flags". */
    uint8_t     pad2[4];
};
OFP_ASSERT(sizeof(struct ofp_meter_features) == 24);

/* All ones is used to indicate all queues in a port (for stats retrieval). */
#define OFPQ_ALL      0xffffffff

/* Min rate > 1000 means not configured. */
#define OFPQ_MIN_RATE_UNCFG      0xffff

/* Max rate > 1000 means not configured. */
#define OFPQ_MAX_RATE_UNCFG      0xffff

enum ofp_queue_desc_prop_type {
    OFPQDPT_MIN_RATE      = 1,      /* Minimum datarate guaranteed. */
    OFPQDPT_MAX_RATE      = 2,      /* Maximum datarate. */
    OFPQDPT_EXPERIMENTER  = 0xffff  /* Experimenter defined property. */
};

/* Common header for all queue properties */
struct ofp_queue_desc_prop_header {
    uint16_t         type;    /* One of OFPQDPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_queue_desc_prop_header) == 4);

/* Min-Rate queue property description. */
```

 © 2015; The Open Networking Foundation

```
struct ofp_queue_desc_prop_min_rate {
    uint16_t         type;    /* OFPQDPT_MIN_RATE. */
    uint16_t         length;  /* Length is 8. */
    uint16_t rate;        /* In 1/10 of a percent; >1000 -> disabled. */
    uint8_t pad[2];       /* 64-bit alignment */
};
OFP_ASSERT(sizeof(struct ofp_queue_desc_prop_min_rate) == 8);

/* Max-Rate queue property description. */
struct ofp_queue_desc_prop_max_rate {
    uint16_t         type;    /* OFPQDPT_MAX_RATE. */
    uint16_t         length;  /* Length is 8. */
    uint16_t rate;        /* In 1/10 of a percent; >1000 -> disabled. */
    uint8_t pad[2];       /* 64-bit alignment */
};
OFP_ASSERT(sizeof(struct ofp_queue_desc_prop_max_rate) == 8);

/* Experimenter queue property description. */
struct ofp_queue_desc_prop_experimenter {
    uint16_t         type;    /* OFPQDPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_queue_desc_prop_experimenter) == 12);

/* Body for ofp_multipart_request of types OFPMP_QUEUE_DESC and
 * OFPMP_QUEUE_STATS. */
struct ofp_queue_multipart_request {
    uint32_t port_no;        /* All ports if OFPP_ANY. */
    uint32_t queue_id;       /* All queues if OFPQ_ALL. */
};
OFP_ASSERT(sizeof(struct ofp_queue_multipart_request) == 8);

/* Body of reply to OFPMP_QUEUE_DESC request. */
struct ofp_queue_desc {
    uint32_t port_no;      /* Port this queue is attached to. */
    uint32_t queue_id;     /* id for the specific queue. */
    uint16_t len;          /* Length in bytes of this queue desc. */
    uint8_t pad[6];        /* 64-bit alignment. */

    struct ofp_queue_desc_prop_header properties[0]; /* List of properties. */
};
OFP_ASSERT(sizeof(struct ofp_queue_desc) == 16);

enum ofp_queue_stats_prop_type {
    OFPQSPT_EXPERIMENTER  = 0xffff  /* Experimenter defined property. */
};

/* Common header for all queue properties */
struct ofp_queue_stats_prop_header {
    uint16_t         type;    /* One of OFPQSPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats_prop_header) == 4);

/* Experimenter queue property description. */
struct ofp_queue_stats_prop_experimenter {
    uint16_t         type;    /* OFPQSPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_queue_stats_prop_experimenter) == 12);

/* Body of reply to OFPMP_QUEUE_STATS request. */
struct ofp_queue_stats {
    uint16_t length;         /* Length of this entry. */
    uint8_t pad[6];          /* Align to 64 bits. */
    uint32_t port_no;        /* Port the queue is attached to. */
    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. */
    uint32_t duration_sec;   /* Time queue has been alive in seconds. */
    uint32_t duration_nsec;  /* Time queue has been alive in nanoseconds beyond
```

 © 2015; The Open Networking Foundation

```
                                duration_sec. */

    struct ofp_queue_stats_prop_header properties[0]; /* List of properties. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats) == 48);

/* Body for ofp_multipart_request of type OFPMP_FLOW_MONITOR.
 *
 * The OFPMP_FLOW_MONITOR request's body consists of an array of zero or more
 * instances of this structure.  The request arranges to monitor the flows
 * that match the specified criteria, which are interpreted in the same way as
 * for OFPMP_FLOW.
 *
 * 'id' identifies a particular monitor for the purpose of allowing it to be
 * canceled later with OFPFMC_DELETE.  'id' must be unique among
 * existing monitors that have not already been canceled.
 */
struct ofp_flow_monitor_request {
    uint32_t monitor_id;        /* Controller-assigned ID for this monitor. */
    uint32_t out_port;          /* Required output port, if not OFPP_ANY. */
    uint32_t out_group;         /* Required group number, if not OFPG_ANY. */
    uint16_t flags;             /* OFPFMF_*. */
    uint8_t table_id;           /* One table's ID or OFPTT_ALL (all tables). */
    uint8_t command;            /* One of OFPFMC_*. */
    struct ofp_match match;     /* Fields to match. Variable size. */
};
OFP_ASSERT(sizeof(struct ofp_flow_monitor_request) == 24);

/* Flow monitor commands */
enum ofp_flow_monitor_command {
    OFPFMC_ADD    = 0,       /* New flow monitor. */
    OFPFMC_MODIFY = 1,       /* Modify existing flow monitor. */
    OFPFMC_DELETE = 2,       /* Delete/cancel existing flow monitor. */
};

/* 'flags' bits in struct of_flow_monitor_request. */
enum ofp_flow_monitor_flags {
    /* When to send updates. */
    OFPFMF_INITIAL = 1 << 0,     /* Initially matching flows. */
    OFPFMF_ADD = 1 << 1,         /* New matching flows as they are added. */
    OFPFMF_REMOVED = 1 << 2,     /* Old matching flows as they are removed. */
    OFPFMF_MODIFY = 1 << 3,      /* Matching flows as they are changed. */

    /* What to include in updates. */
    OFPFMF_INSTRUCTIONS = 1 << 4,/* If set, instructions are included. */
    OFPFMF_NO_ABBREV = 1 << 5,   /* If set, include own changes in full. */
    OFPFMF_ONLY_OWN = 1 << 6,    /* If set, don't include other controllers. */
};

/* OFPMP_FLOW_MONITOR reply header.
 *
 * The body of an OFPMP_FLOW_MONITOR reply is an array of variable-length
 * structures, each of which begins with this header.  The 'length' member may
 * be used to traverse the array, and the 'event' member may be used to
 * determine the particular structure.
 *
 * Every instance is a multiple of 8 bytes long. */
struct ofp_flow_update_header {
    uint16_t length;            /* Length of this entry. */
    uint16_t event;             /* One of OFPFME_*. */
    /* ...other data depending on 'event'... */
};
OFP_ASSERT(sizeof(struct ofp_flow_update_header) == 4);

/* 'event' values in struct ofp_flow_update_header. */
enum ofp_flow_update_event {
    /* struct ofp_flow_update_full. */
    OFPFME_INITIAL = 0,          /* Flow present when flow monitor created. */
    OFPFME_ADDED = 1,            /* Flow was added. */
    OFPFME_REMOVED = 2,          /* Flow was removed. */
    OFPFME_MODIFIED = 3,         /* Flow instructions were changed. */

    /* struct ofp_flow_update_abbrev. */
    OFPFME_ABBREV = 4,           /* Abbreviated reply. */

    /* struct ofp_flow_update_header. */
    OFPFME_PAUSED = 5,          /* Monitoring paused (out of buffer space). */
    OFPFME_RESUMED = 6,         /* Monitoring resumed. */
};

/* OFPMP_FLOW_MONITOR reply for OFPFME_INITIAL, OFPFME_ADDED, OFPFME_REMOVED,
 * and OFPFME_MODIFIED. */
struct ofp_flow_update_full {
    uint16_t length;            /* Length is 32 + match + instructions. */
    uint16_t event;             /* One of OFPFME_*. */
    uint8_t table_id;           /* ID of flow's table. */
    uint8_t reason;             /* OFPRR_* for OFPFME_REMOVED, else zero. */
    uint16_t idle_timeout;      /* Number of seconds idle before expiration. */
    uint16_t hard_timeout;      /* Number of seconds before expiration. */
    uint16_t priority;          /* Priority of the entry. */
```

 © 2015; The Open Networking Foundation

```
    uint8_t zeros[4];           /* Reserved, currently zeroed. */
    uint64_t cookie;            /* Opaque controller-issued identifier. */
    struct ofp_match match;     /* Fields to match. Variable size. */
    /*     Instruction set.
     *     If OFPFMF_INSTRUCTIONS was not specified, or 'event' is
     *     OFPFME_REMOVED, no instructions are included.
     */
    //struct ofp_instruction instructions[0];
};
OFP_ASSERT(sizeof(struct ofp_flow_update_full) == 32);

/* OFPMP_FLOW_MONITOR reply for OFPFME_ABBREV.
 *
 * When the controller does not specify OFPFMF_NO_ABBREV in a monitor request,
 * any flow tables changes due to the controller's own requests (on the same
 * OpenFlow channel) will be abbreviated, when possible, to this form, which
 * simply specifies the 'xid' of the OpenFlow request (e.g. an OFPT_FLOW_MOD)
 * that caused the change.
 * Some changes cannot be abbreviated and will be sent in full.
 */
struct ofp_flow_update_abbrev {
    uint16_t length;            /* Length is 8. */
    uint16_t event;             /* OFPFME_ABBREV. */
    uint32_t xid;               /* Controller-specified xid from flow_mod. */
};
OFP_ASSERT(sizeof(struct ofp_flow_update_abbrev) == 8);

/* OFPMP_FLOW_MONITOR reply for OFPFME_PAUSED and OFPFME_RESUMED.
 */
struct ofp_flow_update_paused {
    uint16_t length;            /* Length is 8. */
    uint16_t event;             /* One of OFPFME_*. */
    uint8_t zeros[4];           /* Reserved, currently zeroed. */
};
OFP_ASSERT(sizeof(struct ofp_flow_update_paused) == 8);

/* Controller status property types.
 */
enum ofp_controller_status_prop_type {
    OFPCSPT_URI                   = 0,      /* Connection URI property. */
    OFPCSPT_EXPERIMENTER          = 0xFFFF, /* Experimenter property. */
};

/* Common header for all Controller Status Properties */
struct ofp_controller_status_prop_header {
    uint16_t         type;    /* One of OFPCSPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_controller_status_prop_header) == 4);

/* Connection URI controller status property */
struct ofp_controller_status_prop_uri {
    uint16_t         type;    /* OFPCSPT_URI. */
    uint16_t         length;  /* Length in bytes of this property. */

    /* Followed by:
     *   - Exactly (length - 4) bytes containing Connection URI, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    char             uri[0];
};
OFP_ASSERT(sizeof(struct ofp_controller_status_prop_uri) == 4);

/* Experimenter controller status property */
struct ofp_controller_status_prop_experimenter {
    uint16_t         type;    /* OFPCSPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_controller_status_prop_experimenter) == 12);

/* Why is the controller status being reported? */
enum ofp_controller_status_reason {
    OFPCSR_REQUEST            = 0,  /* Controller requested status. */
    OFPCSR_CHANNEL_STATUS     = 1,  /* Oper status of channel changed. */
    OFPCSR_ROLE               = 2,  /* Controller role changed. */
    OFPCSR_CONTROLLER_ADDED   = 3,  /* New controller added. */
    OFPCSR_CONTROLLER_REMOVED = 4,  /* Controller removed from config. */
    OFPCSR_SHORT_ID           = 5,  /* Controller ID changed. */
    OFPCSR_EXPERIMENTER       = 6,  /* Experimenter data changed. */
};
```

 © 2015; The Open Networking Foundation

```
/* Body of OFPMP_CONTROLLER_STATUS reply message and body of async
 * OFPT_CONTROLLER_STATUS message */
struct ofp_controller_status {
   uint16_t length;           /* Length of this entry. */
   uint16_t short_id;         /* ID number which identifies the controller. */
   uint32_t role;             /* Controller's role. One of OFPCR_ROLE_*. */
   uint8_t reason;            /* One of OFPCSR_* reason codes. */
   uint8_t channel_status;    /* Status of control channel.
                               * One of OFPCT_STATUS_*. */
   uint8_t pad[6];            /* Align to 64-bits. */

    /* Controller Status Property list.  The Connection URI property is
       required; other properties are optional. */
    struct ofp_controller_status_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_controller_status) == 16);

struct ofp_controller_status_header {
    struct ofp_header header;               /* Type OFPT_CONTROLLER_STATUS. */
    struct ofp_controller_status status;    /* Controller status. */
};

/* Control channel status. */
enum ofp_control_channel_status {
    OFPCT_STATUS_UP     = 0,    /* Control channel is operational. */
    OFPCT_STATUS_DOWN   = 1,    /* Control channel is not operational. */
};

struct ofp_bundle_features_prop_header {
    uint16_t type;          /* One of OFPTMPBF_*. */
    uint16_t length;        /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_features_prop_header) == 4);

/* Body of OFPMP_BUNDLE_FEATURES request. */
struct ofp_bundle_features_request {
    uint32_t feature_request_flags;  /* Bitmap of "ofp_bundle_feature_flags". */
    uint8_t pad[4];

    /* Bundle features property list - 0 or more. */
    struct ofp_bundle_features_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_bundle_features_request) == 8);

/* Bundle features property types. */
enum ofp_bundle_features_prop_type {
    OFPTMPBF_TIME_CAPABILITY = 0x1, /* Time feature property. */
    OFPTMPBF_EXPERIMENTER = 0xFFFF, /* Experimenter property. */
};

/* Time Format. */
struct ofp_time {
    uint64_t seconds;
    uint32_t nanoseconds;
    uint8_t  pad[4];
};
OFP_ASSERT(sizeof(struct ofp_time)==16);

/* Bundle time features. */
struct ofp_bundle_features_prop_time {
    uint16_t type;                    /* OFPTMPBF_TIME_CAPABILITY. */
    uint16_t length;                  /* Length in bytes of this property. */
    uint8_t pad[4];
    struct ofp_time sched_accuracy;   /* The scheduling accuracy, i.e., how
                                       * accurately the switch can perform a
                                       * scheduled commit. This field is used
                                       * only in bundle features replies, and
                                       * is ignored in bundle features
                                       * requests. */
    struct ofp_time sched_max_future; /* The maximal difference between the
                                       * scheduling time and the current time.*/
    struct ofp_time sched_max_past;   /* If the scheduling time occurs in the
                                       * past, defines the maximal difference
                                       * between the current time and the
                                       * scheduling time.*/
    struct ofp_time timestamp;        /* Indicates the time during the
                                       * transmission of this message.*/
};
OFP_ASSERT(sizeof(struct ofp_bundle_features_prop_time) == 72);

enum ofp_bundle_feature_flags {
    OFPBF_TIMESTAMP = 1 << 0,       /* Request includes a timestamp. */
    OFPBF_TIME_SET_SCHED = 1 << 1,  /* Request includes the sched_max_future and
                                     * sched_max_past parameters. */
};

/* Body of reply to OFPMP_BUNDLE_FEATURES request. */
struct ofp_bundle_features {
    uint16_t capabilities; /* Bitmap of "ofp_bundle_flags". */
    uint8_t pad[6];
```

 © 2015; The Open Networking Foundation

```
    /* Bundle features property list - 0 or more. */
    struct ofp_bundle_features_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_bundle_features) == 8);

/* Body for ofp_multipart_request/reply of type OFPMP_EXPERIMENTER. */
struct ofp_experimenter_multipart_header {
    uint32_t experimenter;    /* Experimenter ID. */
    uint32_t exp_type;        /* Experimenter defined. */
    /* Experimenter-defined arbitrary additional data. */
};
OFP_ASSERT(sizeof(struct ofp_experimenter_multipart_header) == 8);

/* Typical Experimenter structure. */
struct ofp_experimenter_structure {
    uint32_t experimenter;      /* Experimenter ID:
                                 * - MSB 0: low-order bytes are IEEE OUI.
                                 * - MSB != 0: defined by ONF. */
    uint32_t exp_type;          /* Experimenter defined. */
    uint8_t  experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_experimenter_structure) == 8);

/* Experimenter extension message. */
struct ofp_experimenter_msg {
    struct ofp_header header;   /* Type OFPT_EXPERIMENTER. */
    uint32_t experimenter;      /* Experimenter ID:
                                 * - MSB 0: low-order bytes are IEEE OUI.
                                 * - MSB != 0: defined by ONF. */
    uint32_t exp_type;          /* Experimenter defined. */
    /* Experimenter-defined arbitrary additional data. */
    uint8_t  experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_experimenter_msg) == 16);

/* Configures the "role" of the sending controller.  The default role is:
 *
 *    - Equal (OFPCR_ROLE_EQUAL), which allows the controller access to all
 *      OpenFlow features. All controllers have equal responsibility.
 *
 * The other possible roles are a related pair:
 *
 *    - Master (OFPCR_ROLE_MASTER) is equivalent to Equal, except that there
 *      may be at most one Master controller at a time: when a controller
 *      configures itself as Master, any existing Master is demoted to the
 *      Slave role.
 *
 *    - Slave (OFPCR_ROLE_SLAVE) allows the controller read-only access to
 *      OpenFlow features.  In particular attempts to modify the flow table
 *      will be rejected with an OFPBRC_EPERM error.
 *
 *      Slave controllers do not receive OFPT_PACKET_IN or OFPT_FLOW_REMOVED
 *      messages, but they do receive OFPT_PORT_STATUS messages.
 *
 * Also configures the ID for the controller.  The default ID is
 * OFPCID_UNDEFINED, and controllers may set the short_id to change their ID.
 * Only the OFPCID_UNDEFINED ID may be assigned to more than one controller,
 * and setting the short_id to a value in use will be rejected.
 */

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

/* Role request and reply message. */
struct ofp_role_request {
    struct ofp_header header;   /* Type OFPT_ROLE_REQUEST/OFPT_ROLE_REPLY. */
    uint32_t role;              /* One of OFPCR_ROLE_*. */
    uint16_t short_id;          /* ID number for the controller. */
    uint8_t pad[2];             /* Align to 64 bits. */
    uint64_t generation_id;     /* Master Election Generation Id */
};
OFP_ASSERT(sizeof(struct ofp_role_request) == 24);

#define OFPCID_UNDEFINED 0

/* Role property types.
 */
enum ofp_role_prop_type {
    OFPRPT_EXPERIMENTER           = 0xFFFF, /* Experimenter property. */
};

/* Common header for all Role Properties */
struct ofp_role_prop_header {
    uint16_t         type;    /* One of OFPRPT_*. */
```

 © 2015; The Open Networking Foundation

```
    uint16_t        length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_role_prop_header) == 4);

/* Experimenter role property */
struct ofp_role_prop_experimenter {
    uint16_t        type;    /* One of OFPRPT_EXPERIMENTER. */
    uint16_t        length;  /* Length in bytes of this property. */
    uint32_t        experimenter;  /* Experimenter ID which takes the same
                                      form as in struct
                                      ofp_experimenter_header. */
    uint32_t        exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t        experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_role_prop_experimenter) == 12);

/* What changed about the controller role */
enum ofp_controller_role_reason {
    OFPCRR_MASTER_REQUEST = 0,  /* Another controller asked to be master. */
    OFPCRR_CONFIG         = 1,  /* Configuration changed on the switch. */
    OFPCRR_EXPERIMENTER   = 2,  /* Experimenter data changed. */
};

/* Role status event message. */
struct ofp_role_status {
    struct ofp_header header;   /* Type OFPT_ROLE_STATUS. */
    uint32_t role;              /* One of OFPCR_ROLE_*. */
    uint8_t reason;             /* One of OFPCRR_*. */
    uint8_t pad[3];             /* Align to 64 bits. */
    uint64_t generation_id;     /* Master Election Generation Id */

    /* Role Property list */
    struct ofp_role_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_role_status) == 24);

/* Async Config property types.
 * Low order bit cleared indicates a property for the slave role.
 * Low order bit set indicates a property for the master/equal role.
 */
enum ofp_async_config_prop_type {
    OFPACPT_PACKET_IN_SLAVE      = 0,  /* Packet-in mask for slave. */
    OFPACPT_PACKET_IN_MASTER     = 1,  /* Packet-in mask for master. */
    OFPACPT_PORT_STATUS_SLAVE    = 2,  /* Port-status mask for slave. */
    OFPACPT_PORT_STATUS_MASTER   = 3,  /* Port-status mask for master. */
    OFPACPT_FLOW_REMOVED_SLAVE   = 4,  /* Flow removed mask for slave. */
    OFPACPT_FLOW_REMOVED_MASTER  = 5,  /* Flow removed mask for master. */
    OFPACPT_ROLE_STATUS_SLAVE    = 6,  /* Role status mask for slave. */
    OFPACPT_ROLE_STATUS_MASTER   = 7,  /* Role status mask for master. */
    OFPACPT_TABLE_STATUS_SLAVE   = 8,  /* Table status mask for slave. */
    OFPACPT_TABLE_STATUS_MASTER  = 9,  /* Table status mask for master. */
    OFPACPT_REQUESTFORWARD_SLAVE  = 10, /* RequestForward mask for slave. */
    OFPACPT_REQUESTFORWARD_MASTER = 11, /* RequestForward mask for master. */
    OFPACPT_FLOW_STATS_SLAVE     = 12, /* Flow stats mask for slave. */
    OFPACPT_FLOW_STATS_MASTER    = 13, /* Flow stats mask for master. */
    OFPACPT_CONT_STATUS_SLAVE    = 14, /* Controller status mask for slave. */
    OFPACPT_CONT_STATUS_MASTER   = 15, /* Controller status mask for master. */
    OFPACPT_EXPERIMENTER_SLAVE   = 0xFFFE, /* Experimenter for slave. */
    OFPACPT_EXPERIMENTER_MASTER  = 0xFFFF, /* Experimenter for master. */
};

/* Common header for all async config Properties */
struct ofp_async_config_prop_header {
    uint16_t        type;    /* One of OFPACPT_*. */
    uint16_t        length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_async_config_prop_header) == 4);

/* Various reason based properties */
struct ofp_async_config_prop_reasons {
    uint16_t        type;    /* One of OFPACPT_PACKET_IN_*,
                                OFPACPT_PORT_STATUS_*,
                                OFPACPT_FLOW_REMOVED_*,
                                OFPACPT_ROLE_STATUS_*,
                                OFPACPT_TABLE_STATUS_*,
                                OFPACPT_REQUESTFORWARD_*,
                                OFPACPT_FLOW_STATS_*,
                                OFPACPT_CONT_STATUS_*. */
    uint16_t        length;  /* Length in bytes of this property. */
    uint32_t        mask;    /* Bitmasks of reason values. */
};
OFP_ASSERT(sizeof(struct ofp_async_config_prop_reasons) == 8);

/* Experimenter async config  property */
struct ofp_async_config_prop_experimenter {
    uint16_t        type;    /* One of OFPTFPT_EXPERIMENTER_SLAVE,
```

 © 2015; The Open Networking Foundation

```
                                OFPTFPT_EXPERIMENTER_MASTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_async_config_prop_experimenter) == 12);

/* Asynchronous message configuration. */
struct ofp_async_config {
    struct ofp_header header;     /* OFPT_GET_ASYNC_REPLY or OFPT_SET_ASYNC. */

    /* Async config Property list - 0 or more */
    struct ofp_async_config_prop_header properties[0];
};
OFP_ASSERT(sizeof(struct ofp_async_config) == 8);

/* What changed about the table */
enum ofp_table_reason {
    OFPTR_VACANCY_DOWN  = 3,        /* Vacancy down threshold event. */
    OFPTR_VACANCY_UP    = 4,        /* Vacancy up threshold event. */
};

/* A table config has changed in the datapath */
struct ofp_table_status {
    struct ofp_header header;
    uint8_t reason;         /* One of OFPTR_*. */
    uint8_t pad[7];         /* Pad to 64 bits */
    struct ofp_table_desc table;   /* New table config. */
};
OFP_ASSERT(sizeof(struct ofp_table_status) == 24);

/* Request forward reason */
enum ofp_requestforward_reason {
    OFPRFR_GROUP_MOD = 0,       /* Forward group mod requests. */
    OFPRFR_METER_MOD = 1,       /* Forward meter mod requests. */
};

/* Group/Meter request forwarding. */
struct ofp_requestforward_header {
    struct ofp_header header;   /* Type OFPT_REQUESTFORWARD. */
    struct ofp_header request;  /* Request being forwarded. */
};
OFP_ASSERT(sizeof(struct ofp_requestforward_header) == 16);

/* Bundle property types. */
enum ofp_bundle_prop_type {
    OFPBPT_TIME                   = 1,      /* Time property. */
    OFPBPT_EXPERIMENTER           = 0xFFFF, /* Experimenter property. */
};

/* Common header for all Bundle Properties */
struct ofp_bundle_prop_header {
    uint16_t         type;    /* One of OFPBPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_prop_header) == 4);

/* Experimenter bundle property */
struct ofp_bundle_prop_experimenter {
    uint16_t         type;    /* OFPBPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct ofp_bundle_prop_experimenter) == 12);

/* Bundle time property. */
    struct ofp_bundle_prop_time {
    uint16_t type;       /* OFPBPT_TIME */
    uint16_t length;     /* Length in bytes of this property (24). */
    uint8_t  pad[4];
    struct ofp_time scheduled_time;
};
OFP_ASSERT(sizeof(struct ofp_bundle_prop_time)==24);

/* Bundle control message types */
```

 © 2015; The Open Networking Foundation

```
enum ofp_bundle_ctrl_type {
    OFPBCT_OPEN_REQUEST    = 0,
    OFPBCT_OPEN_REPLY      = 1,
    OFPBCT_CLOSE_REQUEST   = 2,
    OFPBCT_CLOSE_REPLY     = 3,
    OFPBCT_COMMIT_REQUEST  = 4,
    OFPBCT_COMMIT_REPLY    = 5,
    OFPBCT_DISCARD_REQUEST = 6,
    OFPBCT_DISCARD_REPLY   = 7,
};

/* Bundle configuration flags. */
enum ofp_bundle_flags {
    OFPBF_ATOMIC  = 1 << 0,  /* Execute atomically. */
    OFPBF_ORDERED = 1 << 1,  /* Execute in specified order. */
    OFPBF_TIME    = 1 << 2,  /* Execute in specified time. */
};

/* Message structure for OFPT_BUNDLE_CONTROL. */
struct ofp_bundle_ctrl_msg {
    struct ofp_header     header;
    uint32_t              bundle_id;      /* Identify the bundle. */
    uint16_t              type;           /* OFPBCT_*. */
    uint16_t              flags;          /* Bitmap of OFPBF_* flags. */

    /* Bundle Property list. */
    struct ofp_bundle_prop_header properties[0]; /* Zero or more properties. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_ctrl_msg) == 16);

/* Message structure for OFPT_BUNDLE_ADD_MESSAGE.
 * Adding a message in a bundle is done with. */
struct ofp_bundle_add_msg {
    struct ofp_header     header;
    uint32_t              bundle_id;      /* Identify the bundle. */
    uint16_t              pad;            /* Align to 64 bits. */
    uint16_t              flags;          /* Bitmap of OFPBF_* flags. */

    struct ofp_header     message;        /* Message added to the bundle. */

    /* If there is one property or more, 'message' is followed by:
     *   - Exactly (message.length + 7)/8*8 - (message.length) (between 0 and 7)
     *     bytes of all-zero bytes */

    /* Bundle Property list. */
    /*struct ofp_bundle_prop_header properties[0];*/ /* Zero or more
                                                        properties. */
};
OFP_ASSERT(sizeof(struct ofp_bundle_add_msg) == 24);

#endif /* openflow/openflow.h */
```

Appendix B Release Notes

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

The text of the release notes is informative and historical, and should not be considered normative. Many items of the release notes refer to features and text that have been removed, replaced or updated in subsequent versions of this specification, and therefore do not necessarily match the actual specification.

B.1 OpenFlow version 0.2.0

Release date : March 28,2008
Protocol version : 1

 © 2015; The Open Networking Foundation

B.2 OpenFlow version 0.2.1

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

B.3 OpenFlow version 0.8.0

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

- Reorganise OpenFlow message types
- Add `OFPP_TABLE` virtual port to send packet-out packet to the tables
- Add global flag `OFPC_SEND_FLOW_EXP` to configure flow expired messages generation
- Add flow priority
- Remove flow Group-ID (experimental QoS support)
- Add Error messages
- Make stat request and stat reply more generic, with a generic header and stat specific body
- Change fragmentation strategy for stats reply, use explicit flag `OFPSF_REPLY_MORE` instead of empty packet
- Add table stats and port stats messages

B.4 OpenFlow version 0.8.1

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

B.5 OpenFlow version 0.8.2

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

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

B.6 OpenFlow version 0.8.9

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

 © 2015; The Open Networking Foundation

B.6.1 IP Netmasks

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

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

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

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

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

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

B.6.2 New Physical Port Stats

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

```
/* Body of reply to OFPST_PORT request. If a counter is unsupported, set
 * the field to all ones. */
struct ofp_port_stats {
    uint16_t port_no;
```

 © 2015; The Open Networking Foundation

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

B.6.3 IN_PORT Virtual Port

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

B.6.4 Port and Link Status and Configuration

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

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

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

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

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

B.6.5 Echo Request/Reply Messages

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

 © 2015; The Open Networking Foundation

B.6.6 Vendor Extensions

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

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

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

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

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

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

B.6.7 Explicit Handling of IP Fragments

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

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

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

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

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

 © 2015; The Open Networking Foundation

B.6.8 802.1D Spanning Tree

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

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

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

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

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

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

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

B.6.9 Modify Actions in Existing Flow Entries

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

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

 © 2015; The Open Networking Foundation

B.6.10 More Flexible Description of Tables

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

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

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

B.6.11 Lookup Count in Tables

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

B.6.12 Modifying Flags in Port-Mod More Explicit

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

The new `ofp_port_mod` message looks like the following:

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

B.6.13 New Packet-Out Message Format

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

 © 2015; The Open Networking Foundation

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

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

B.6.14 Hard Timeout for Flow Entries

A hard timeout value has been added to flow entries. If set, then the entry must be expired in the specified number of seconds regardless of whether or not packets are hitting the entry. A `hard_timeout` field has been added to the `flow_mod` message to support this. The `max_idle` field has been renamed `idle_timeout`. A value of zero means that a timeout has not been set. If both `idle_timeout` and `hard_timeout` are zero, then the flow is permanent and should not be deleted without an explicit deletion.

The new `ofp_flow_mod` format looks like this:

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

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

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

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

The new `ofp_flow_expired` message looks like the following:

 © 2015; The Open Networking Foundation

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

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

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

B.6.15 Reworked initial handshake to support backwards compatibility

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

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

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

B.6.16 Description of Switch Stat

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

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

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

 © 2015; The Open Networking Foundation

B.6.17 Variable Length and Vendor Actions

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

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

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

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

The vendor-defined action header looks like the following:

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

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

 © 2015; The Open Networking Foundation

B.6.18 VLAN Action Changes

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

The OFPAT_SET_VLAN_VID action looks like the following:

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

The OFPAT_SET_VLAN_PCP action looks like the following:

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

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

B.6.19 Max Supported Ports Set to 65280

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

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

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

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

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

 © 2015; The Open Networking Foundation

B.6.21 Behavior Defined When Controller Connection Lost

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

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

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

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

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

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

B.6.22 ICMP Type and Code Fields Now Matchable

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

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

How: Changed spec to allow matching on these fields.

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

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

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

To support this, an `out_port` field has been added to the `ofp_flow_mod`, `ofp_flow_stats_request`, and `ofp_aggregate_stats_request` messages. If an `out_port` contains a value other than `OFPP_NONE`, it introduces a constraint when matching. This constraint is that the rule must contain an output action directed at that port. Other constraints such as `ofp_match` structs and priorities are still used; this is purely an *additional* constraint. Note that to get previous behavior, though, `out_port` must be set

 © 2015; The Open Networking Foundation

to `OFPP_NONE`, since "0" is a valid port id. This only applies to the `delete` and `delete_strict` flow mod commands; the field is ignored by `add`, `modify`, and `modify_strict`.

B.7 OpenFlow version 0.9

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

B.7.1 Failover

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

B.7.2 Emergency Flow Cache

The protocol and reference implementation have been extended to allow insertion and management of emergency flow entries.

Emergency-specific flow entries are inactive until a switch loses connectivity from the controller. If this happens, the switch invalidates all normal flow table entries and copies all emergency flows into the normal flow table.

Upon connecting to a controller again, all entries in the flow cache stay active. The controller then has the option of resetting the flow cache if needed.

B.7.3 Barrier Command

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

B.7.4 Match on VLAN Priority Bits

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

B.7.5 Selective Flow Expirations

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

 © 2015; The Open Networking Foundation

B.7.6 Flow Mod Behavior

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

B.7.7 Flow Expiration Duration

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

B.7.8 Notification for Flow Deletes

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

B.7.9 Rewrite DSCP in IP ToS header

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

B.7.10 Port Enumeration now starts at 1

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

B.7.11 Other changes to the Specification

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

 © 2015; The Open Networking Foundation

B.8 OpenFlow version 1.0

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

B.8.1 Slicing

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

B.8.2 Flow cookies

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

B.8.3 User-specifiable datapath description

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

B.8.4 Match on IP fields in ARP packets

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

B.8.5 Match on IP ToS/DSCP bits

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

B.8.6 Querying port stats for individual ports

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

 © 2015; The Open Networking Foundation

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

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

B.8.8 Other changes to the Specification

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

B.9 OpenFlow version 1.1

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

B.9.1 Multiple Tables

Prior versions of the OpenFlow specification did expose to the controller the abstraction of a single table. The OpenFlow pipeline could internally be mapped to multiple tables, such as having a separate wildcard and exact match table, but those tables would always act logically as a single table.

OpenFlow 1.1 introduces a more flexible pipeline with multiple tables. Exposing multiple tables has many advantages. The first advantage is that a lot of hardware has multiple tables internally (for example L2 table, L3 table, multiple TCAM lookups), and the multiple tables support of OpenFlow may allow to expose this hardware with greater efficiency and flexibility. The second advantage is that many network deployments combine orthogonal processing of packets (for example ACL, QoS and routing), forcing all those processing in a single table creates huge ruleset due to the cross product of individual rules. Multiple tables helps to decouple separate processing functions and to map them efficiently to the various hardware resources.

The new OpenFlow pipeline with multiple tables is quite different from the simple pipeline of prior OpenFlow versions. The new OpenFlow pipeline exposes a set of completely generic tables, supporting the full match and full set of actions. It's difficult to build a pipeline abstraction that accurately

 © 2015; The Open Networking Foundation

represents all possible hardware, therefore OpenFlow 1.1 is based on a generic and flexible pipeline that may be mapped to the hardware. Some limited table capabilities are available to denote what each table is capable of supporting.

Packets are processed through the pipeline, they are matched and processed in the first table, and may be matched and processed in other tables. As it goes through the pipeline, a packet is associated with an *action set*, accumulating action, and a generic *metadata* register. The action set is resolved at the end of the pipeline and applied to the packet. The metadata can be matched and written at each table and allows state to be carried between tables.

OpenFlow introduces a new protocol object called *instruction* to control pipeline processing. Actions which were directly attached to flows in previous versions are now encapsulated in instructions, instructions may apply those actions between tables or accumulate them in the packet action set. Instructions can also change the *metadata*, or direct a packet to another table.

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

B.9.2 Groups

The new group abstraction enables OpenFlow to represent a set of ports as a single entity for forwarding packets. Different types of groups are provided to represent different abstractions such as multicasting or multipathing. Each group is composed of a set group buckets, each group bucket contains the set of actions to be applied before forwarding to the port. Groups buckets can also forward to other groups, enabling groups to be chained together.

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

B.9.3 Tags : MPLS & VLAN

Prior versions of the OpenFlow specification had limited VLAN support, it only supported a single level of VLAN tagging with ambiguous semantic. The new tagging support has explicit actions to add,

 © 2015; The Open Networking Foundation

modify and remove VLAN tags, and can support multiple levels of VLAN tagging. It also adds similar support the MPLS shim headers.

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

B.9.4 Virtual ports

Prior versions of the OpenFlow specification assumed that all the ports of the OpenFlow switch were physical ports. This version of the specification adds support for virtual ports, which can represent complex forwarding abstractions such as LAGs or tunnels.

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

B.9.5 Controller connection failure

Prior versions of the OpenFlow specification introduced the emergency flow cache as a way to deal with the loss of connectivity with the controller. The emergency flow cache feature was removed in this version of the specification, due to the lack of adoption, the complexity to implement it and other issues with the feature semantic.

This version of the specification adds two simpler modes to deal with the loss of connectivity with the controller. In fail secure mode, the switch continues operating in OpenFlow mode, until it reconnects to a controller. In fail standalone mode, the switch reverts to using normal processing (Ethernet switching).

- Remove Emergency Flow Cache from spec
- Connection interruption triggers fail secure or fail standalone mode

B.9.6 Other changes

- Remove 802.1d-specific text from the specification
- Cookie Enhancements Proposal - cookie mask for filtering
- Set_queue action (unbundled from output port action)
- Maskable DL and NW address match fields
- Add TTL decrement, set and copy actions for IPv4 and MPLS
- SCTP header matching and rewriting support
- Set ECN action
- Define message handling : no loss, may reorder if no barrier
- Rename `VENDOR` APIs to `EXPERIMENTER` APIs
- Many other bug fixes, rewording and clarifications

 © 2015; The Open Networking Foundation

B.10 OpenFlow version 1.2

Release date : December 5, 2011
Protocol version : 0x03

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

B.10.1 Extensible match support

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

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

List of features for OpenFlow Extensible Match :

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

B.10.2 Extensible 'set_field' packet rewriting support

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

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

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

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

- Reuse match TLV structure (OXM) to describe metadata in packet-in (EXT-6)

 © 2015; The Open Networking Foundation

- Include the 'metadata' field in packet-in
- Move ingress port and physical port from static field to OXM encoding
- Allow to optionally include packet header fields in TLV structure

B.10.4 Extensible Error messages via experimenter error type

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

B.10.5 IPv6 support added

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

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

B.10.6 Simplified behaviour of flow-mod request

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

- MODIFY and MODIFY_STRICT commands never insert new flows in the table
- New flag OFPFF_RESET_COUNTS to control counter reset
- Remove quirky behaviour for cookie field.

B.10.7 Removed packet parsing specification

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

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

B.10.8 Controller role change mechanism

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

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

 © 2015; The Open Networking Foundation

B.10.9 Other changes

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

B.11 OpenFlow version 1.3

Release date : April 13, 2012
Protocol version : 0x04

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

B.11.1 Refactor capabilities negotiation

Prior versions of the OpenFlow specification included limited expression of the capabilities of an OpenFlow switch. OpenFlow 1.3 includes a more flexible framework to express capabilities (EXT-123).

The main change is the improved description of table capabilities. Those capabilities have been moved out of the table statistics structure in its own request/reply message, and encoded using a flexible TLV format. This enables the additions of next-table capabilities, table-miss flow entry capabilities and experimenter capabilities.

Other changes include renaming the 'stats' framework into the 'multipart' framework to reflect the fact that it is now used for both statistics and capabilities, and the move of port descriptions into its own multipart message to enable support of a greater number of ports.

List of features for Refactor capabilities negotiation :

- Rename 'stats' framework into the 'multipart' framework.
- Enable 'multipart' requests (requests spanning multiple messages).
- Move port list description to its own multipart request/reply.
- Move table capabilities to its own multipart request/reply.
- Create flexible property structure to express table capabilities.
- Enable to express experimenter capabilities.
- Add capabilities for table-miss flow entries.

 © 2015; The Open Networking Foundation

- Add next-table (i.e. goto) capabilities

B.11.2 More flexible table miss support

Prior versions of the OpenFlow specification included table configuration flags to select one of three 3 behaviour for handling table-misses (packet not matching any flows in the table). OpenFlow 1.3 replaces those limited flags with the table-miss flow entry, a special flow entry describing the behaviour on table miss (EXT-108).

The table-miss flow entry uses standard OpenFlow instructions and actions to process table-miss packets, this enables the use of OpenFlow's full flexibility in processing those packets. All previous behaviour expressed by the table-miss config flags can be expressed using the table-miss flow entry. Many new ways of handling a table-miss, such as processing table-miss with normal, can now trivially be described by the OpenFlow protocol.

- Remove table-miss config flags (EXT-108).
- Define table-miss flow entry as the all wildcard, lowest priority flow entry (EXT-108).
- Mandate support of the table-miss flow entry in every table to process table-miss packets (EXT-108).
- Add capabilities to describe the table-miss flow entry (EXT-123).
- Change table-miss default to drop packets (EXT-119).

B.11.3 IPv6 Extension Header handling support

Add the ability to match the presence of common IPv6 extension headers, and some anomalous conditions in IPv6 extension headers (EXT-38). A new OXM pseudo header field `OXM_OF_IPV6_EXTHDR` enables to match the following conditions :

- Hop-by-hop IPv6 extension header is present.
- Router IPv6 extension header is present.
- Fragmentation IPv6 extension header is present.
- Destination options IPv6 extension headers is present.
- Authentication IPv6 extension header is present.
- Encrypted Security Payload IPv6 extension header is present.
- No Next Header IPv6 extension header is present.
- IPv6 extension headers out of preferred order.
- Unexpected IPv6 extension header encountered.

B.11.4 Per flow meters

Add support for per-flow meters (EXT-14). Per-flow meters can be attached to flow entries and can measure and control the rate of packets. One of the main applications of per-flow meters is to rate limit packets sent to the controller.

The per-flow meter feature is based on a new flexible meter framework, which includes the ability to describe complex meters through the use of multiple metering bands, metering statistics and capabilities. Currently, only simple rate-limiter meters are defined over this framework. Support for color-aware

 © 2015; The Open Networking Foundation

meters, which support Diff-Serv style operation and are tightly integrated in the pipeline, was postponed to a later release.

- Flexible meter framework based on per-flow meters and meter bands.
- Meter statistics, including per band statistics.
- Enable to attach meters flexibly to flow entries.
- Simple rate-limiter support (drop packets).

B.11.5 Per connection event filtering

Version 1.2 of the specification introduced the ability for a switch to connect to multiple controllers for fault tolerance and load balancing. Per connection event filtering improves the multi-controller support by enabling each controller to filter events from the switch it does not want (EXT-120).

A new set of OpenFlow messages enables a controller to configure an event filter on its own connection to the switch. Asynchronous messages can be filtered by type and reason. This event filter comes in addition to other existing mechanisms that enable or disable asynchronous messages, for example the generation of flow-removed events can be configured per flow. Each controller can have a separate filter for the slave role and the master/equal role.

- Add asynchronous message filter for each controller connection.
- Controller message to set/get the asynchronous message filter.
- Set default filter value to match OpenFlow 1.2 behaviour.
- Remove `OFPC_INVALID_TTL_TO_CONTROLLER` config flag.

B.11.6 Auxiliary connections

In previous versions of the specification, the channel between the switch and the controller is exclusively made of a single TCP connection, which did not allow the exploitation of the parallelism available in most switch implementations. OpenFlow 1.3 enables a switch to create auxiliary connections to supplement the main connection between the switch and the controller (EXT-114). Auxiliary connections are mostly useful to carry packet-in and packet-out messages.

- Enable switch to create auxiliary connections to the controller.
- Mandate that auxiliary connection can not exist when main connection is not alive.
- Add auxiliary-id to the protocol to disambiguate the type of connection.
- Enable auxiliary connection over UDP and DTLS.

B.11.7 MPLS BoS matching

A new OXM field `OXM_OF_MPLS_BOS` has been added to match the Bottom of Stack bit (BoS) from the MPLS header (EXT-85). The BoS bit indicates if other MPLS shim headers are in the payload of the present MPLS packet, and matching this bit can help to disambiguate cases where the MPLS label is reused across levels of MPLS encapsulation.

 © 2015; The Open Networking Foundation

B.11.8 Provider Backbone Bridging tagging

Add support for tagging packets using Provider Backbone Bridging (PBB) encapsulation (EXT-105). This enables OpenFlow to support various network deployments based on PBB, such as regular PBB and PBB-TE.

- Push and Pop operation to add PBB header as a tag.
- New OXM field to match I-SID for the PBB header.

B.11.9 Rework tag order

In previous versions of the specification, the final order of tags in a packet was statically specified. For example, an MPLS shim header was always inserted after all VLAN tags in the packet. OpenFlow 1.3 removes this restriction, the final order of tags in a packet is dictated by the order of the tagging operations, each tagging operation adds its tag in the outermost position (EXT-121).

- Remove defined order of tags in packet from the specification.
- Tags are now always added in the outermost possible position.
- Action-list can add tags in arbitrary order.
- Tag order is predefined for tagging in the action-set.

B.11.10 Tunnel-ID metadata

The logical port abstraction enables OpenFlow to support a wide variety of encapsulations. The tunnel-id metadata `OXM_OF_TUNNEL_ID` is a new OXM field that exposes metadata associated with the logical port to the OpenFlow pipeline. It is most commonly the demultiplexing field from the encapsulation header (EXT-107).

For example, if the logical port performs GRE encapsulation, the tunnel-id field would map to the GRE key field from the GRE header. After decapsulation, OpenFlow would be able to match the GRE key in the tunnel-id match field. Similarly, by setting the tunnel-id, OpenFlow would be able to set the GRE key in an encapsulated packet.

B.11.11 Cookies in packet-in

A cookie field was added to the packet-in message (EXT-7). This field takes its value from the flow that sends the packet to the controller. If the packet was not sent by a flow, this field is set to 0xffffffffffffffff.

Having the cookie in the packet-in enables the controller to more efficiently classify packet-in, rather than having to match the packet against the full flow table.

B.11.12 Duration for stats

A duration field was added to most statistics, including port statistics, group statistics, queue statistics and meter statistics (EXT-102). The duration field allows to more accurately calculate packet and byte rates from the counters included in those statistics.

 © 2015; The Open Networking Foundation

B.11.13 On demand flow counters

New flow-mod flags have been added to disable packet and byte counters on a per-flow basis. Disabling such counters may improve flow handling performance in the switch.

B.11.14 Other changes

- Fix a bug describing VLAN matching (EXT-145).
- Flow entry description now mention priority (EXT-115).
- Flow entry description now mention timeout and cookies (EXT-147).
- Unavailable counters must now be set to all 1 (EXT-130).
- Correctly refer to flow entry instead of rule (EXT-132).
- Many other bug fixes, rewording and clarifications.

B.12 OpenFlow version 1.3.1

Release date : September 06, 2012
Protocol version : 0x04

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

B.12.1 Improved version negotiation

Prior versions of the OpenFlow specification included a simple scheme for version negotiation, picking the lowest of the highest version supported by each side. Unfortunately this scheme does not work properly in all cases ; if both implementations don't implement all versions up to their highest version, the scheme can fail to negotiate a version they have in common (EXT-157).

The main change is adding a bitmap of version numbers in the Hello messages using during negotiation. By having the full list of version numbers, negotiation can always negotiate the appropriate version if one is available. This version bitmap is encoded in a flexible TLV format to retain future extensibility of the Hello message.

List of features for Improved version negotiation :

- Hello Elements, new flexible TLV format for Hello message
- Optional version bitmap in Hello messages.
- Improve version negotiation using optional version bitmaps.

B.12.2 Other changes

- Mandate that table-miss flow entry support drop and controller (EXT-158).
- Clarify the mapping of encapsulation data in `OXM_OF_TUNNEL_ID` (EXT-161).
- Rules and restrictions for UDP connections (EXT-162).
- Clarify virtual meters (EXT-165).
- Remove reference to switch fragmentation - confusing (EXT-172).
- Fix meter constant names to always be multipart (`OFPST_ => OFPMT_`) (EXT-184).

 © 2015; The Open Networking Foundation

- Add `OFPG_*` definitions to spec (EXT-198).
- Add `ofp_instruction` and `ofp_table_feature_prop_header` in spec text (EXT-200).
- Bad error code in connection setup, must be `OFPHFC_INCOMPATIBLE` (EXT-201).
- Instructions must be a multiple of 8 bytes in length (EXT-203).
- Port status includes a reason, not a status (EXT-204).
- Clarify usage of table config field (EXT-205).
- Clarify that required match fields don't need to be supported in every flow table (EXT-206).
- Clarify that prerequisite does not require full match field support (EXT-206).
- Include in the spec missing definitions from openflow.h (EXT-207).
- Fix invalid error code `OFPQCFC_EPERM -> OFPSCFC_EPERM` (EXT-208).
- Clarify PBB language about B-VLAN (EXT-215)
- Fix inversion between source and destination ethernet addresses (EXT-215)
- Clarify how to reorder group buckets, and associated group bucket clarifications (EXT-217).
- Add disclaimer that release notes may not match specification (EXT-218)
- Figure 1 still says "Secure Channel" (EXT-222).
- OpenFlow version must be calculated (EXT-223).
- Meter band drop precedence should be increased, not reduced (EXT-225)
- Fix ambiguous uses of may/can/should/must (EXT-227)
- Fix typos (EXT-228)
- Many typos (EXT-231)

B.13 OpenFlow version 1.3.2

Release date : April 25, 2013
Protocol version : 0x04

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

B.13.1 Changes

- Mandate in OXM that 0-bits in mask must be 0-bits in value (EXT-238).
- Allow connection initiated from one of the controllers (EXT-252).
- Add clause on frame misordering to spec (EXT-259).
- Set table features doesn't generate flow removed messages (EXT-266).
- Fix description of set table features error response (EXT-267).
- Define use of `generation_id` in role reply messages (EXT-272).
- Switches with only one flow table are not mandated to implement goto (EXT-280).

B.13.2 Clarifications

- Clarify that MPLS Pop action uses Ethertype regardless of BOS bit (EXT-194).
- Controller message priorities using auxiliary connections (EXT-240).
- Clarify padding rules and variable size arrays (EXT-251).
- Better description buffer-id in flow mod (EXT-257).
- Semantic of `OFPPS_LIVE` (EXT-258).
- Improve multipart introduction (EXT-263).

 © 2015; The Open Networking Foundation

- Clarify set table features description (EXT-266).
- Clarify meter flags and burst fields (EXT-270).
- Clarify slave access rights (EXT-271).
- Clarify that a switch can't change a controller role (EXT-276).
- Clarify roles of coexisting master and equal controllers (EXT-277).
- Various typos and rewording (EXT-282, EXT-288, EXT-290)

B.14 OpenFlow version 1.3.3

Release date : September 27, 2013
Protocol version : 0x04

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

B.14.1 Changes

- Update with IANA registered TCP port : 6653 (EXT-133).
- Clarify that IPv6 flow label is not maskable by default (EXT-101).
- Clarify multipart segmentation rules, clarify use of empty multipart messages (EXT-321).
- Specify the normal fragment handling is mandatory, drop/reasm optional (EXT-99).
- Explain that prerequisites are cumulative (EXT-285).
- Specify that buffer-id is unique per connection (EXT-286).
- Clarify which OXM types can be used in set-field actions (EXT-289).
- Define `oxm_len` for OXM IDs in table feature to have the payload length (EXT-330).
- Set-field prerequisite may be met through other actions (EXT-331).
- Clarify error codes for invalid group type and invalid weight (EXT-344).
- Specify group and meter feature bitmaps (EXT-345).

B.14.2 Clarifications

- Explain that `OFP_TABLE_MOD` is deprecated in 1.3.X (EXT-269).
- Minor clarification, replace "Goto" with "Goto-Table", replace "read message" with "multipart message" (EXT-297).
- Mention flags in the description of flow entries (EXT-298).
- Clarify policing of packet-in to controllers (EXT-300).
- Clarify invalid DSCP values, all six bits are valid (EXT-305).
- Add many new definitions to the glossary (EXT-309).
- Improve many existing glossary definitions (EXT-309).
- Detail UDP congestion control for auxiliary channels (EXT-311).
- Better document controller initiated connections (EXT-311).
- Clarify that there is only one request/reply per multipart sequence (EXT-321).
- Clarify connection maintenance messages on auxiliary connections (EXT-323).
- Clarify padding in set-field and hello elements (EXT-326).
- Clarify `padding`, `data` and `total_len` fields in packet-in (EXT-286).
- Clarify that actions in table-feature don't have padding (EXT-287).
- In fail-standalone, the switch owns the flow tables and flow entries (EXT-291).

 © 2015; The Open Networking Foundation

- Clarify queue relation to ports and packets, and that queues are optional (EXT-293).
- Action set may be executed before generating packet-in (EXT-296).
- Add bytes column in table describing OXM types (EXT-313).
- Clarify that `OFPP_MAX` is a usable port number (EXT-315).
- Specify how to pack OpenFlow messages in UDP (EXT-332).
- Flow-mod modify: instructions are replaced, not updated (EXT-294).
- Clarify that `OFPBAC_BAD_TYPE` applies to unsupported actions (EXT-343).
- Explain flow removed reasons in the spec (EXT-261).
- Removing ports does not remove flow entries (EXT-281).
- Clarify that header field must be presents for set-field action (EXT-331).
- Clarify default values for fields on push-tag action (EXT-342).
- Clarify the use of the priority field in flow-mods (EXT-354).
- Replace WhitePaper specific URL with ONF generic URL (EXT-83/EXT-356).
- Clarify that the action-set is not always executed (EXT-359).
- Connection setup may be for an in-band connection (EXT-359).
- Clarify error for group forwarding to invalid group (EXT-359).
- Replace "OpenFlow protocol" into "OpenFlow switch protocol" (EXT-357).
- Replace "wire protocol" with "protocol version"

B.15 OpenFlow version 1.3.4

Release date : March 27, 2014
Protocol version : 0x04

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

B.15.1 Changes

- Make IPv6 flow label maskable (EXT-101).
- Clarify statistics when group/meter are modified (EXT-341).
- Clarify that table feature match list should not include prerequisite only fields (EXT-387).
- Clarify table feature wildcard list should not include fields that are mandatory in some context only (EXT-387).
- Add section about control channel maintenance (EXT-435).
- Push MPLS should add a MPLS header before the IP header and before MPLS tags, not before VLAN which is not valid (EXT-457).

B.15.2 Clarifications

- Specify error for bad meter in meter action (EXT-327).
- Fix invalid prefix on meter multipart constants (EXT-302).
- Add a section about reserved values and reserved bit positions (EXT-360).
- Better describe the protocol basic format (EXT-360).
- Fix comment about experimenter band type (EXT-363).
- Clarify that port description multipart only list standard ports (EXT-364).
- Update flow-mod description with `OFPFF_RESET_COUNTS` (EXT-365).

 © 2015; The Open Networking Foundation

- Clarify `flow_count` for meter stats (EXT-374).
- Experimenter actions/types can't be reported in bitmaps (EXT-376).
- Clarify action bad argument errors (EXT-393).
- Many small clarifications, implementation defined features (EXT-395).
- Clarify that actions in a buckets always apply as an action-set (EXT-408).
- Merging action-set need to be set-field aware (EXT-409).
- Change action-list to list of actions for consistency (EXT-409).
- Introduce properly set of actions in the glossary (EXT-409).
- Clarify DSCP remark meter band (EXT-416).
- Add a section about pipeline consistency (EXT-415).
- Clarify handling of actions inconsistent with the match or packet (EXT-417).
- Clarify in-port and in-phy-port OXM field definitions (EXT-418).
- Clarify that `OFPP_CONTROLLER` is a valid ingress port (EXT-418).
- Clarification on Flow Match Field length for experimenter fields with masks (EXT-420).
- Clarify handling of duplicate action in a write-action instruction or group bucket, allow either to return an error or filter duplicate actions (EXT-421).
- Clarify error code for Clear-Actions instruction with non-empty set of actions (EXT-422).
- Clarify error on unsupported `OXM_CLASS` and `OXM_FIELD` (EXT-423).
- Add section about reserved property/TLV types (EXT-429).
- Clarify meaning of `OFPG_ANY` for watching group in group bucket (EXT-431).
- Separate pipeline field definitions from header field definitions (EXT-432).
- Clarify presence of header fields in Packet-In OXM list (EXT-432).
- Barrier reply must be generated when no pending request (EXT-433).
- Clarify error code for unsupported actions (EXT-434).
- Clarify error codes when setting table features is not supported or enabled (EXT-436).
- Port description must include all standard port, regardless of config or state (EXT-437).
- Improve channel reconnection recommendations (EXT-439).
- Wrong prefix, fix `OFPPFL_NO_PACKET_IN` into `OFPPC_NO_PACKET_IN` (EXT-443).
- Specify properly packet data field in packet-in and packet-out, especially CRCs (EXT-452).
- Specify how tunnel-id interract with logical ports, especially in output (EXT-453).
- More precise description of Tunnel ID pipeline field (EXT-453).
- Various typos, grammar and spelling fixes (EXT-455).

B.16 OpenFlow version 1.3.5

Release date : March 26, 2015
Protocol version : 0x04

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

B.16.1 Changes

- Flow-mod request with command loose modify and loose delete can not be partial (EXT-362).
- Clarify definition of overlap for flow entries, equal is not overlap (EXT-406).
- Add specification for alternate OpenFlow connection transports (EXT-463).
- Add specification for Controller Channel Connection URI from OF 1.5 (EXT-275).

 © 2015; The Open Networking Foundation

- Table numbers are no longer sequential, table are only numbered in potential packet traversal order (EXT-467).
- Clarify which flow-mod commands and flags are mandatory (EXT-478).
- Clarify that physical port is optional, and if used it's an OpenFlow port (EXT-491).
- Specify that data included in error for multipart request is either the current mutlipart message or empty (EXT-506).
- Specify that counters must use the full bit range (EXT-529).

B.16.2 Clarifications

- Controller must be able to read Table-miss flow entry (EXT-464).
- Clarify the meaning of `total_len` in packet-in structure (EXT-468).
- Allow delayed CRC recalculations for set-field actions (EXT-469).
- Allow Experimenter OXM fields in Set-Field actions (EXT-471).
- Make the highest bit of version reserved instead of experimental (EXT-480).
- Fix incorrect prefix OFPMC_ with OFPMF_ in comments (EXT-481).
- Be more explicit about `oxm_length` for Experimenter OXMs (EXT-482).
- In-band control channel is outside the spec, no longer mention default flows (EXT-497).
- Required instructions and actions must be supported in all flow tables (EXT-498).
- Specify errors for multiparts with bad `queue_id`, `group_id` or `meter_id` (EXT-500).
- Clarify that Clear-Actions instruction is required for table-miss flow entries (EXT-509).
- Clarify error code for group delete requests including buckets (EXT-510).
- Clarify that empty buckets or group with no buckets just drop the packet (EXT-511).
- Clarify that a switch can support flow a priority 0 that are not table-miss flow entries (EXT-512).
- Clarify that barrier replies correspond to barrier requests (EXT-513).
- Specify that the most specific error should be used when multiple error codes are possible (EXT-514).
- Specify that if there are multiple errors in a message, the switch return only one error message (EXT-514).
- Clarify that error must have a matching xid for all replies (EXT-514).
- Clarify definition of `OFPC_FRAG_MASK` (EXT-517).
- Improve the definition of connection in glossary (EXT-518).
- Broaden definition of counters in glossary (EXT-518).
- Fix various specification technical errors (EXT-518).
- Fix typos and improper english (EXT-518).
- Fix formatting of tables and enums (EXT-518).
- Improved generic description of matching (EXT-519).
- Improve various descriptions to be more precise (EXT-519).
- Add cross references and fix duplicate cross reference (EXT-519).
- Better specify bitmaps and flags (EXT-520).
- Clarify `oxm_length` (EXT-520).
- Clarify that rate field in queue properties refers to a fraction of the output port current speed (EXT-522).
- Specify that a secure version of TLS is recommended (EXT-525).
- Clarify certificate configuration of the switch (EXT-304).
- Specify that malformed packet refer to those in the datapath (EXT-528).

 © 2015; The Open Networking Foundation

- Specify how to deal with malformed OpenFlow messages (EXT-528).
- Spelling, gramar and other typos (EXT-542).

B.17 OpenFlow version 1.4.0

Release date : August 5, 2013
Protocol version : 0x05

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

B.17.1 More extensible wire protocol

The OpenFlow protocol was initially designed with many static fixed structures and limited extensibility. The introduction of the OpenFlow Extensible Match (OXM) in version 1.2 added much needed extensibility in the OpenFlow classifier. In version 1.4, many other parts of the protocol have been retrofitted with TLV structures for improved extensibility (EXT-262).

This TLV work affected many areas of the protocol. New TLVs have been added in previously fixed structures in the form of properties at the end of the structure. In some areas, the existing TLVs have been changed to use the common property TLV format. TLVs rules have been clarified. This additional extensibility of the protocol will allow a much easier way to add new features to the protocol in the future, and also greatly extend the Experimenter Extension API.

- Port structures: add port description properties, add port mod properties and add port stats properties.
- Table structures: add table mod properties, add table descriptions multipart, add table status asynchronous message.
- Queue structures: migrate queue description to multipart, convert queue description properties to standardised TLVs, add queue stats properties.
- Set-async structures: convert set-async-config to TLVs, add set-async experimenter property.
- Instruction structures: clarify instruction TLVs.
- Actions structures: clarify actions TLVs.
- Experimented structures: clarify experimenter TLVs.
- Properties errors: add a set of unified error codes for all properties.

B.17.2 More descriptive reasons for packet-in

The OpenFlow pipeline saw extensive changes since 1.0, however, the `reason` values in the `ofp_packet_in` messages did not change. As a result, many distinct parts of the pipeline are using the same `reason` value. Version 1.4 introduces more descriptive reasons, so that the controller can properly distinguish which part of the pipeline redirected the packet to the controller (EXT-136).

The main change is that the the "output action" reason `OFPR_ACTION` is effectively split into four reasons, "apply-action", "action-set", "group bucket" and "packet-out", representing the four distinct context where this action is used. The "no match" reason `OFPR_NO_MATCH` is renamed to properly reflect the fact that it is generated by the table miss flow entry.

 © 2015; The Open Networking Foundation

The new set of `reason` values for `ofp_packet_in` message is :

- `OFPR_TABLE_MISS`: No matching flow (table-miss flow entry).
- `OFPR_APPLY_ACTION`: Output to controller in apply-actions.
- `OFPR_INVALID_TTL`: Packet has invalid TTL.
- `OFPR_ACTION_SET`: Output to controller in action set.
- `OFPR_GROUP`: Output to controller in group bucket.
- `OFPR_PACKET_OUT`: Output to controller in packet-out.

B.17.3 Optical port properties

A new set of port properties adds support for Optical ports, they include fields to configure and monitor the transmit and receive frequency of a laser, as well as its power (EXT-154). Those new properties can be used to configure and monitor either Ethernet optical port or optical ports on circuit switches.

- Optical port mod property `ofp_port_mod_prop_optical` to configure optical ports.
- Optical port stats property `ofp_port_stats_prop_optical` to monitor optical ports.
- Optical port description property `ofp_port_desc_prop_optical` to describe optical port capabilities.

B.17.4 Flow-removed reason for meter delete

Add a new `reason` value `OFPRR_METER_DELETE` for the `ofp_flow_removed` message to denote that the flow entry was removed as a result of a meter deletion (EXT-261).

When a meter is deleted on the switch, all the flow entries that use that meter are removed. This is similar to how group operates. The flow entry that is removed may generate a flow-removed message (depending on config). In version 1.3, we did not have a `reason` for this condition, version 1.4 fixes that bug.

B.17.5 Flow monitoring

The OpenFlow protocol defines a multi-controller scheme where multiple controllers can manage a switch. Flow monitoring allows a controller to monitor in real time the changes to any subsets of the flow table done by other controllers (EXT-187).

The flow monitoring framework allows a controller to define a number of monitors, each selecting a subset of the flow tables. Each monitor includes a table id and a match pattern that defines the subset monitored. When any flow entry is added, modified or removed in one of the subsets defined by a flow monitor, an event is sent to the controller to inform it of the change.

- Multipart request `ofp_flow_monitor_request` to set flow monitors on the switch.
- Flow monitor update events sent to controller, with full details using `ofp_flow_update_full` or abbreviated using `ofp_flow_update_abbrev`.
- Monitor flags to define the format of the updates.
- Flow control mechanism to avoid backlog of monitor updates.

 © 2015; The Open Networking Foundation

B.17.6 Role status events

Version 1.2 of the specification added the ability for a controller to set its role in a multi-controller environment. When a controller elected itself to "master" role, the previous master controller is demoted to "slave" role, however that controller was not informed about it. In version 1.4, the Role Status message enables the switch to inform a controller about changes to its role (EXT-191).

- Role status event `OFPT_ROLE_STATUS` to inform controller to change to role.
- Role status properties for experimenter data, `ofp_role_prop_experimenter`.

B.17.7 Eviction

Most flow tables have finite capacity. In previous versions of the specification, when a flow table is full, new flow entries are not inserted in the flow table and an error is returned to the controller. However, reaching that point is pretty problematic, as the controller needs time to operate on the flow table and this may cause a disruption of service. Eviction adds a mechanism enabling the switch to automatically eliminate entries of lower importance to make space for newer entries (EXT-192). This enables smoother degradation of behaviour when the table is full.

- Table-mod flag `OFPTC_EVICTION` to enable or disable eviction on a table.
- Flow-mod `importance` to optionally denote the importance of a flow entry for eviction.
- Table-desc eviction property `ofp_table_mod_prop_eviction` to describe the type of eviction performed by the switch.

B.17.8 Vacancy events

Most flow tables have finite capacity. In previous versions of the specification, when a flow table is full, new flow entries are not inserted in the flow table and an error is returned to the controller. However, reaching that point is pretty problematic, as the controller needs time to operate on the flow table and this may cause a disruption of service. Vacancy events add a mechanism enabling the controller to get an early warning based on a capacity threshold chosen by the controller (EXT-192). This allows the controller to react in advance and avoid getting the table full.

- Table status event `OFPT_TABLE_STATUS` with reasons `OFPTR_VACANCY_DOWN` and `OFPTR_VACANCY_UP` to inform controller of vacancy change.
- Hysteresis mechanism to avoid spurious events using two threshold, `vacancy_down` and `vacancy_up`.
- Table-mod vacancy property to set vacancy thresholds, `ofp_table_mod_prop_vacancy`.

B.17.9 Bundles

Add the bundle mechanism, enabling to apply a group of OpenFlow message as a single operation (EXT-230). This enables the quasi-atomic application of related changes, and to better synchronise changes across a series of switches.

- Bundle control message `OFPT_BUNDLE_CONTROL` to create, destroy and commit bundles.
- Bundle add message `OFPT_BUNDLE_ADD_MESSAGE` to add an OpenFlow message into a bundle.

 © 2015; The Open Networking Foundation

- Bundle error type OFPET_BUNDLE_FAILED to report bundle operation errors.

B.17.10 Synchronised tables

Many switches can perform multiple lookups on the same lookup data. For example, a standard Ethernet learning table performs a learning lookup and a forwarding lookup on the same set of MAC addresses. Another example is RPF checks which reuse the IP forwarding data. The synchronised table feature allow to represent those constructs as a set of two tables whose data is synchronised (EXT-232).

Synchronised table is expressed using a new property in the table feature, OFPTFPT_TABLE_SYNC_FROM. It defines the synchronisation abstraction between the two flow tables, however it does not define and express the flow entry transformation between the flow tables.

B.17.11 Group and Meter change notifications

The OpenFlow protocol defines a multi-controller scheme where multiple controllers can manage a switch. Group and Meter change notifications allow a controller to monitor in real time the changes to the group table or meter table done by other controllers (EXT-235).

The "group-mod" and "meter-mod" requests are simply encapsulated in an OFPT_REQUESTFORWARD asynchronous message sent to other controllers. Those notifications are enabled and disabled via the "set-async-config" message.

B.17.12 Error code for bad priority

Some switches may have restrictions on the priorities that can be used in a table. For example a switch may be enforcing some "longest prefix match" rule in a table, requiring the priority to be related to the mask. A new error code, OFPFMFC_BAD_PRIORITY, enables the switch to properly inform the controller when this happens (EXT-236).

B.17.13 Error code for Set-async-config

The OFPT_GET_ASYNC_REQUEST feature was introduced in version 1.3.0. There was no error messages defined for that feature, however it is possible for this request to fail. A new error type, OFPET_ASYNC_CONFIG_FAILED, with an appropriate set of error codes, enables the switch to properly inform the controller when this happens (EXT-237).

The set of error codes defined for OFPET_ASYNC_CONFIG_FAILED are :

- OFPACFC_INVALID: One mask is invalid.
- OFPACFC_UNSUPPORTED: Requested configuration not supported.
- OFPACFC_EPERM: Permissions error.

 © 2015; The Open Networking Foundation

B.17.14 PBB UCA header field

A new OXM field `OFPXMT_OFB_PBB_UCA` has been added to match the "use customer address" header field from the PBB header (EXT-256).

B.17.15 Error code for duplicate instruction

The OpenFlow specification defines the instructions included in a flow entry as a set, and that an instruction can not be duplicated in that set. A new error code, `OFPBIC_DUP_INST`, enables the switch to properly inform the controller when flow entries contain duplicate instructions (EXT-260).

B.17.16 Error code for multipart timeout

Multipart requests and replies are encoded as a sequence of messages. This version of the specification defines how to deal with unterminated sequences, some minimum timeouts are defined as well as error codes (EXT-264).

- Define minimum timeout (100 ms) and error code (`OFPBRC_MULTIPART_REQUEST_TIMEOUT`) for unterminated multipart request sequences.
- Define minimum timeout (1 s) and error code (`OFPBRC_MULTIPART_REPLY_TIMEOUT`) for unterminated multipart reply sequences.

B.17.17 Change default TCP port to 6653

IANA allocated to ONF the TCP port number 6653 to be used by the OpenFlow switch protocol. All uses of the previous port numbers, 6633 and 976, should be discontinued. OpenFlow switches and OpenFlow controllers must use 6653 by default (when not using a user specified port number).

B.18 OpenFlow version 1.4.1

Release date : March 26, 2015
Protocol version : 0x05

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

B.18.1 Changes

- When adding a flow entry due to synchronisation, overwrite an existing entry instead of merging them (EXT-355).
- Extend the scope of the overlap checking flag to the synchronised flow table (EXT-355).
- Mandate reversible translation for bidirectional synchronised tables (EXT-361).
- Add synchronisation error `OFPFMFC_IS_SYNC` (EXT-378).
- Add capabilities in switch features for bundles and flow monitoring (EXT-404).

 © 2015; The Open Networking Foundation

B.18.2 Clarifications

- Clarify that a table can synchronise on itself (EXT-346).
- Clarify that the vacancy property is optional in table-mods (EXT-346).
- Fix typo, replace `OFPFMF_OWN` with `OFPFMF_NO_ABBREV` in flow monitoring section (EXT-346).
- Fix mistake, Send error if bundled message have the same xid, not the reverse (EXT-346).
- Clarify that bundles can't be applied on different switches exactly at the same time, only approximately (EXT-355).
- Clarify that if the synchronised flow entry does not exist, no flow entry is removed (EXT-355).
- Clarify learning example in synchronisation section, and clarify that it is only illustrative (EXT-355).
- Recommend against complex table synchronisation (EXT-361).
- Clarify that the flow monitor reply message may be an asynchronous message (EXT-371).
- Clarify usage of synchronisation error `OFPFMFC_CANT_SYNC` (EXT-378).
- Clarify that Bundle is optional (EXT-404).
- Clarify that Flow monitoring is optional (EXT-404).
- Add description for duplicate instruction error `OFPBIC_DUP_INST` (EXT-424).
- Various typos, grammar and spelling fixes (EXT-346, EXT-355).
- Specify that importance is unchanged on flow-mod modify (EXT-496).
- Explain consequences of eviction success and failure, specify error (EXT-502).
- Be explicit that less important flow entries don't trigger eviction (EXT-502).
- Clarify that monitor request can have multiple monitors and parts (EXT-505).
- Clarify that table-miss flow entries can be evicted (EXT-538).
- Spelling, gramar and other typos (EXT-543).

B.19 OpenFlow version 1.5.0

Release date : December, 2014
Protocol version : 0x06

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

B.19.1 Egress Tables

In previous versions of the specification, all processing was done in the context of the input port. Version 1.5 introduces Egress Tables, enabling processing to be done in the context of the output port (EXT-306). When a packet is output to a port, it will start processing at the first egress table where flow entries can define processing and redirect the packet to other egress tables. A new OXM field, `OXM_OF_ACTSET_OUTPUT`, enables egress flow entry to match the outgoing port (EXT-233).

- Packet sent to an output port is processed by the first egress table.
- Group processing and reserved port substitution happen before egress tables.
- Define behaviour of egress table and egress flow entries, mostly similar to ingress.
- New match field `OXM_OF_ACTSET_OUTPUT` to match output value from action-set, mandatory for egress tables, optional for ingress tables.
- Forbid adding output or group action in the egress action-set to prevent changing output port.
- Allow egress flow entry to use output action in action list for egress mirroring.

 © 2015; The Open Networking Foundation

- Pipeline fields are carried from ingress to egress tables.
- Table Feature capabilities to set the first egress table.
- Table Feature capabilities to identify table than can be used for ingress and/or egress.
- Introduce table feature request commands for simpler table features updates.

B.19.2 Packet Type aware pipeline

In previous versions of the specification, all packets must be Ethernet packets. Version 1.5 introduces a packet aware pipeline, enabling processing of other types of packets, such as IP packets or PPP packets (EXT-112). A new OXM pipeline field identifies the packet type. The packet type is defined in an extensible manner, it uses multiple namespaces defined by other standards and allows experimenter packet types. The packet type field can be used in matches and acts as a prerequisite for header fields. The packet type field can also be used in packet-in and packet-out messages to identify the payload carried by those messages.

- Header Type definition based on various existing namespaces, define Canonical Header Types.
- New Packet Type pipeline field. This OXM contains the Canonical Header Type of the packet outermost header.
- Packet Type OXM can be used in match to match specific packet types.
- Packet Type OXM used as prerequisite for various header match fields.
- Packet Type OXM used in packet-in to identify the payload.
- Packet Type OXM used in packet-out to identify the payload.
- Table Feature property expose the packet types supported by each flow table.

The specification is currently restricted to a single packet type per switch because packet type conversions are not defined.

B.19.3 Extensible Flow Entry Statistics

Previous versions of the specification use a fixed structure for flow entry statistics. Version 1.5 introduces a flexible encoding, OpenFlow eXtensible Statistics (OXS), to encode any arbitrary flow entry statistics (EXT-334). Existing flow entry statistics are redefined as standard OXS fields.

- OXS TLV format to express flow entry statistics, based on OXM and compatible with OXM for simpler implementation.
- Express existing flow entry statistics as standard OXS fields : flow duration, flow count, packet count, byte count.
- Introduce new standard flow entry statistic : flow idle time.
- Enable arbitrary class based or experimenter based flow entry statistics.
- Use OXS in all messages carrying flow entry statistics : flow removed message, flow statistics multipart, flow aggregate mutipart.
- Introduce lightweight flow statistics multipart, rename existing flow statistics as flow description.

 © 2015; The Open Networking Foundation

B.19.4 Flow Entry Statistics Trigger

Polling flow entry statistics can induce high overhead and utilisation for the switch. A new statistics trigger mechanism enable statistics to be automatically sent to the controller based on various statistics thresholds (EXT-335).

- New instruction `OFPIT_STAT_TRIGGER` to define a set of statistics thresholds using OXS.
- Statistics trigger flags to allow periodic thresholds.
- Efficient batching of statistics threshold to the controller using the lightweight flow statistics multipart.

B.19.5 Copy-Field action to copy between two OXM fields

The existing *Set-Field* action allows to set a header field or pipeline field with a static value. The new *Copy-Field* action allows to copy the value from one header or pipeline field into another header or pipeline field (EXT-320).

- New instruction `OFPAT_COPY_FIELD` to copy value between fields.
- Source and destination fields defined as OXM IDs, support standard OXMs, class based OXMs and experimenter OXMs.
- The Copy-Field instruction can copy bit sub-fields of any length from source to destination at any offset.
- New Table Features property to express support for Copy-Field.

B.19.6 Packet Register pipeline fields

Introduce Packet Registers, a set pipeline fields that can be used as generic scratch space for the *Copy-Field* action (EXT-244). The packet registers are optionally matchable.

- New OXM class for Packet Registers pipeline fields.
- Switch can support as many register as it wants, each register is 64 bits.

B.19.7 TCP flags matching

A new OXM field `OFPXMT_OFB_TCP_FLAGS` has been added to match the flag bits in the TCP header (EXT-109). This field allows to match all flags, such as SYN, ACK and FIN, and may be used to detect the start and the end of TCP connections.

B.19.8 Group command for selective bucket operation

In previous versions of the specification, group operations can only change the entire set of group buckets in a group. Two new group commands enable the insertion or removal of specific group buckets without impacting the other group buckets in the group (EXT-350).

- Group command `OFPGC_INSERT_BUCKET` to insert individual group buckets into a group.
- Group command `OFPGC_REMOVE_BUCKET` to remove specific group buckets from a group.

 © 2015; The Open Networking Foundation

- Add `bucket_id` field to buckets to identify buckets.
- Add group property to extend groups.
- Add group bucket properties to extend group buckets.
- Redefine bucket fields `weight`, `watch_port` and `watch_group` as group bucket properties.

B.19.9 Allow set-field action to set metadata field

Previous versions of the specification had a restriction on the set-field action, the pipeline field `OXM_OF_METADATA` could not be a valid argument for the set-field action. In this specification, the restriction is removed and `OXM_OF_METADATA` can be a valid argument for the set-field action (EXT-46).

B.19.10 Allow wildcard to be used in set-field action

In previous versions of the specification, the *Set-Field* action could only set the entire field referenced. This version of the specification enables the switch to support mask in Set-Field, so that only the specified bits of the field are modified (EXT-314).

B.19.11 Scheduled Bundles

The previous version of the specification introduced *bundle messages*. A bundle is a sequence of OpenFlow modification requests from the controller that is applied as a single OpenFlow operation.

The current specification extends the bundle feature (EXT-340), allowing:

- Scheduled bundles: a bundle commit message may include an *execution time*, specifying *when* the switch is expected to commit the bundle.
- Bundle features request: allows a controller to query a switch about its bundle capabilities, including whether it supports atomic bundles, ordered bundles, and scheduled bundles.

B.19.12 Controller connection status

The Controller Connection Status enables controller to know the status of all the connections from the switch to controllers (EXT-454). This allows a controller to detect control network partitioning or monitor the status of other controllers. Controllers are identified using a standard URI (EXT-275).

- Controller status multipart messages to query status of controller connections.
- Define URI scheme to identify controller connections.
- Enable controller to set a short id to identify themselves.
- Controller status message to inform controller of changes in connection status.

 © 2015; The Open Networking Foundation

B.19.13 Meter action

In previous versions of the specification, metering was done through a Meter instruction. This version replaces that instruction with a Meter action (EXT-379). As the result, multiple meters can be attached to a flow entry, and meters can be used in group buckets.

- Introduce Meter action `OFPAT_METER`.
- Deprecate Meter instruction `OFPIT_METER`.

B.19.14 Enable setting all pipeline fields in packet-out

Previous versions of the specification support setting the *In-Port* pipeline field of the packet in the *Packet-Out* message. This version of the specification supports setting all pipeline field of the packet in the *Packet-Out* message (EXT-427). This enables, for example, setting the tunnel-id pipeline field for the logical port.

- Add match structure in `packet_out` messages to encode pipeline fields as OXMs.
- Move the `in_port` field as an OXM in the match.
- Specify that packet-out set the pipeline fields of the packet based on fields included in the match.

B.19.15 Port properties for pipeline fields

Logical ports may consume and generate pipeline fields. For example, the pipeline field *Tunnel ID* can be used by a logical port to express the metadata associated with an encapsulation. New port properties allow logical ports to express which pipeline fields are consumed and which are produced by the logical port (EXT-388).

- Port description property `OFPPDPT_PIPELINE_INPUT` is the list of OXM fields provided for packet received on that port.
- Port description property `OFPPDPT_PIPELINE_OUTPUT` is the list of pipeline fields that are consumed by the port when packets are sent to it.

B.19.16 Port property for recirculation

Logical ports may recirculate packets back to the OpenFlow pipeline after having processed them. A new property allows a port to associate another port where packets are returned to the OpenFlow pipeline (EXT-399). Such recirculation may preserve pipeline fields.

B.19.17 Clarify and improve barrier

The description of the barrier message and its effect on the switch was unified throughout the specification and clarified (EXT-189). Barriers also are changed to commit state to the datapath.

- Clarify that barrier controls message ordering.
- Clarify that barrier notifies of completed processing.
- Specify that barrier commits state to datapath.

 © 2015; The Open Networking Foundation

- Clarify that ordered bundles can be used as an alternative to barriers.

B.19.18 Always generate port status on port config change

In previous versions of the specification, the port status message was sent to the controllers only in case the port state changed or if the configuration was changed outside OpenFlow. On the other hand, if one controller changed the configuration, no port status message was sent to the controllers. This is problematic in multi-controller setup, as other controllers can't be notified of port configuration changes done by one controller.

In this specification, the port status message is sent to the controllers for any change of the port configuration, including those done by one OpenFlow controller (EXT-338).

B.19.19 Make all Experimenter OXM-IDs 64 bits

Previous versions of the specification were unclear on how to encode Experimenter OXMs. This specification mandates that Experimenter OXMs encode the experimenter type in the `oxm_field` field of the OXM header (EXT-380). This enables all OXM-IDs used to identified OXMs to be uniformly 64 bits.

B.19.20 Unified requests for group, port and queue multiparts

The multipart requests for statistics and descriptions have been changed to use a common structure in the case of group, port and queue (EXT-69).

- Port statistics and port descriptions use a common request format.
- Individual port descriptions can be requested.
- Group statistics and group descriptions use a common request format.
- Individual group descriptions can be requested.
- Queue statistics and queue descriptions use a common request format.

B.19.21 Rename some type for consistency

Some OpenFlow types were renamed for consistency.

- Statistics multiparts now have the `_STATS` suffix (EXT-302).
- Multipart `METER_CONFIG` was renamed to `METER_DESC` (EXT-302).
- Meter statistics field `flow_count` was renamed to `ref_count` (EXT-374).

 © 2015; The Open Networking Foundation

B.19.22 Specification reorganisation

Some text was moved in the specification to improve readability and make more accurate cross references (EXT-507).

- Add paragraph headings for error codes.
- Move specification of error conditions to the error message section.
- Move group and meters at the end of the OpenFlow table section.
- Move subsection about flow removal into section about control channel.
- Move Multipart tables subsections further down in multipart section.
- Add paragraph headings for action structures.

B.20 OpenFlow version 1.5.1

Release date : March 26, 2015
Protocol version : 0x06

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

B.20.1 Changes

- Add new error `OFPBAC_BAD_METER` for bad meter in flow-mod (EXT-530).
- Don't specify how each packet is mapped into each meter band (EXT-474).

B.20.2 Clarifications

- Add figure for Meters (EXT-474).
- Properly specify meter flags and explain `max_bands` capability (EXT-474).
- Use explicit values in enum used to describe meter commands (EXT-539).
- Spelling, gramar and other typos (EXT-544).

Appendix C Credits

Spec contributions, in alphabetical order:

Anders Nygren, Ben Pfaff, Bob Lantz, Brandon Heller, Casey Barker, Curt Beckmann, Dan Cohn, Dan Malek, Dan Talayco, David Erickson, David McDysan, David Ward, Edward Crabbe, Fabian Schneider, Glen Gibb, Guido Appenzeller, Jean Tourrilhes, Johann Tonsing, Justin Pettit, KK Yap, Leon Poutievski, Linda Dunbar, Lorenzo Vicisano, Martin Casado, Masahiko Takahashi, Masayoshi Kobayashi, Michael Orr, Navindra Yadav, Nick McKeown, Nico dHeureuse, Peter Balland, Rajesh Madabushi, Rajiv Ramanathan, Reid Price, Rob Sherwood, Saurav Das, Shashidhar Gandham, Spike Curtis, Sriram Natarajan, Tal Mizrahi, Tatsuya Yabe, Wanfu Ding, Yiannis Yiakoumis, Yoram Moses, Zoltán Lajos Kis.

 © 2015; The Open Networking Foundation