Open Optical Monitoring (OOM)

A set of APIs and a decode layer to simplify accessing the information in a pluggable module

Revision 1.0

Executive Summary

Open Optical Monitoring (OOM) defines an interface for applications to access (read and write) the content of optical module EEPROMs. The interface is uniform across all conforming NOS implementations, and across pluggable modules from different vendors (both optical and electrical), as long as they conform to the SFF committee standards. It provides access to the data as decoded key/value pairs, eliminating the need for the developer to know the location and format of the data in the EEPROM. The interface is consistent across different module types, though different keys are defined as appropriate for each module type.

OOM should greatly expand the use of available optical module data by data center operators, by simplifying and standardizing the interface across NOS and module vendor implementations. We believe this will enrich the OCP networking standard and increase the value of OCP networking implementations.

Contents

Figures

Revision History

Name	Date	Version	Description
Steve Joiner	2015-11-18	0.1	Initial Version for Discussion on 20151118
Steve Joiner	2015-12-09	0.2	Additions reflecting the discussion of 20151118
Steve Joiner	2015-12-15	0.3	Additions from discussions during meeting of 20151209. Decision to simplify and focus on northbound interface from decode layer, decode layer, and southbound. Note: None of this text has been approved by the project.
Steve Joiner	2016-01-13	0.4	Note: None of this text has been approved by the project. Updates to Section on governance Update of the SouthBound Definition Update to checklist for Maintenance. Added Github location in Checklist for Governance and Supporting Documents. NOTE: Northbound Interface is Out of Date for this revision.
Don Bollinger	2016-03-16	0.5	Substantial Revision to bring this doc up to date with the implementation as demonstrated at OCP Summit, March 2016
Steve Joiner	2016-06-07	0.6	Updating status of progress. Some items have moved from planned to completed. Also updated future plans.
Steve Joiner	2016-9-14	1.0	Same as version 0.6 but recognizing vote of the OCP Incubation Committee.

Overview

The OOM project provides an open source method to easily access the information inside of the pluggable modules in OCP switches. This project defines a consistent interface between a user application and software layers that enables the user to request information from the pluggable module without knowing the details of how to access the module, or how the information is stored inside the pluggable module.

The high level architecture, depicted in Figure 1, implements a decode library, a Southbound API that the library uses to get module information from the Network Operating System (NOS), and a Northbound API that application developers use to request and receive the decoded module information.

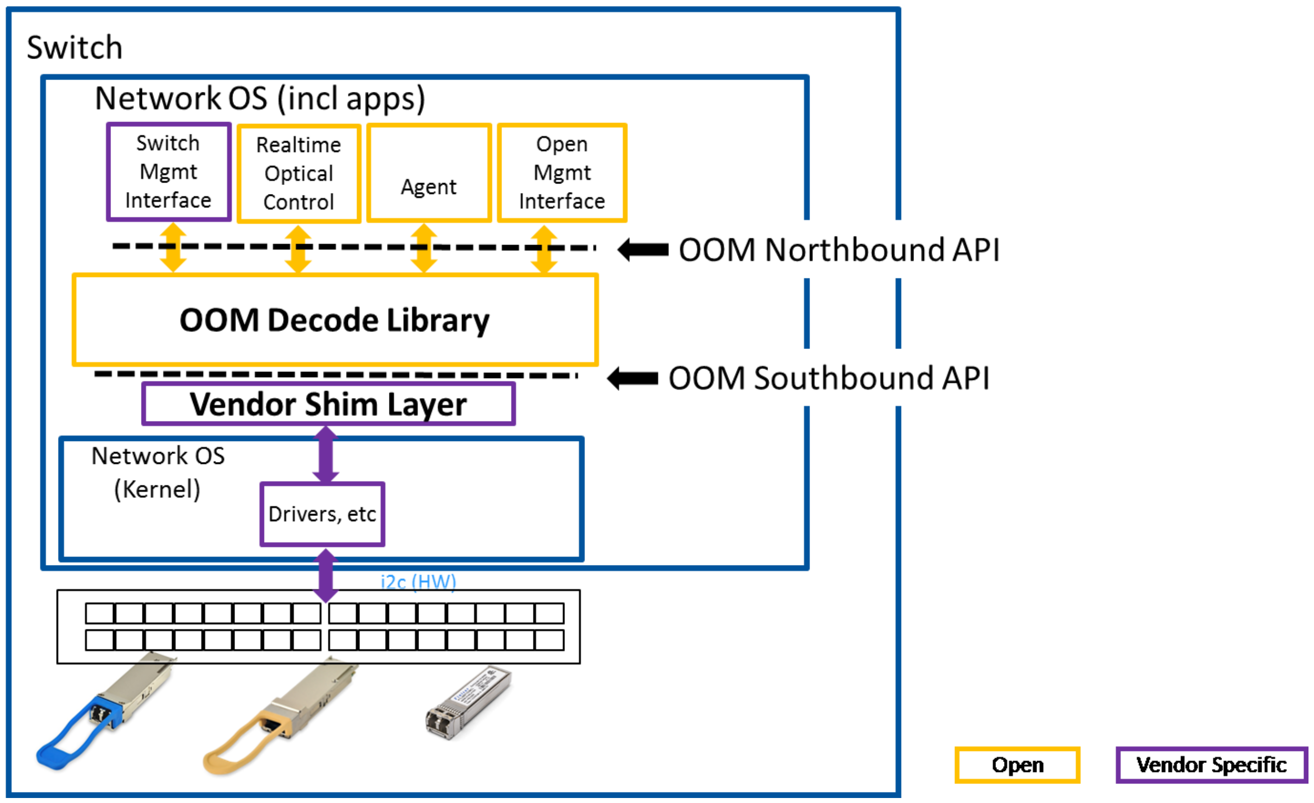

Figure 1: Software Architecture levels

The Southbound API is defined by the include file *oom_south.h*, which specifies a 'C' interface which will be called by the decode library. This API specifies how to get a list of available ports, and routines to read and write specific locations in module EEPROM. This API will be implemented in a 'Vendor Shim', which provides a translation between the native NOS functions and the standardized OOM API. As depicted in Figure 2, each conforming NOS will provide an appropriate shim to support OOM.

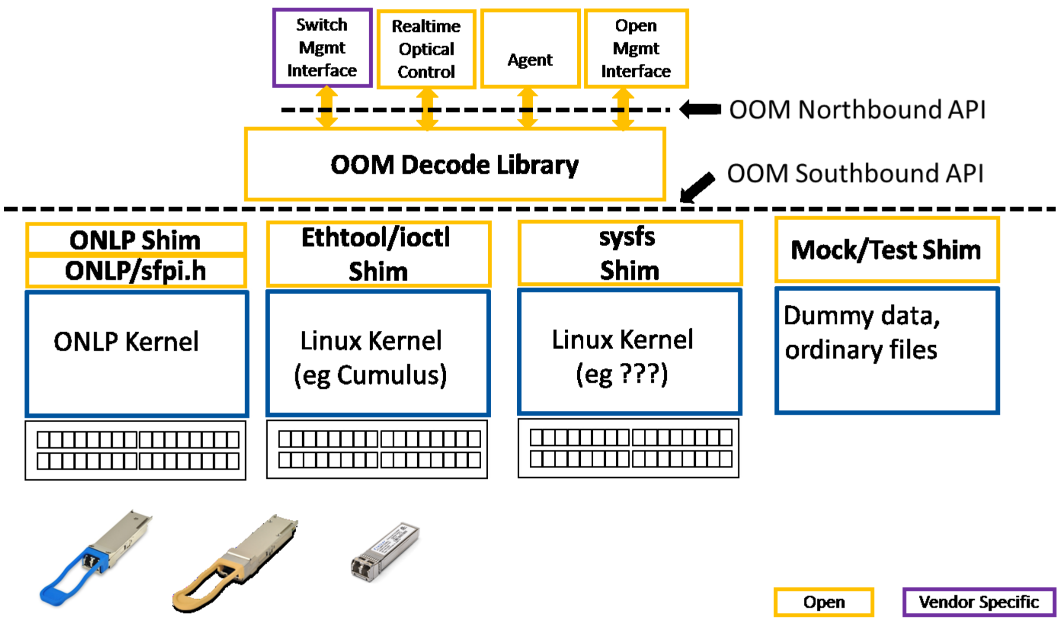

Figure 2: NOS provides a shim to connect the decode layer to the native implementation.

The Northbound API is defined by the python file *oom.py,* which specifies a Python interface which will be called by applications consuming OOM information. This API specifies how to get a list of available ports, how to read and write values in EEPROM using pre-defined keys, how to read pre-defined collections of keys with a single call, and how to read and write raw EEPROM data directly, without any decode layer interpretation.

The decode library itself is implemented in Python. It includes (among other things) *oomlib.py*, which contains most of the architectural framework and logic, and *decode.py*, which contains various routines to decode each unique form of data in the various module types.

OOM provides a collection of python files that encode the actual location and format of the data for each key, for each type of module. s*fp.py* and *qsfp_plus.py* are available in the initial implementation of OOM. More such files will be provided to implement support for additional module types. Application developers should examine these files to determine the available keys for the module types of interest, and the format of the data provided by each key. These files also specify which keys can be accessed as a collection (eg SERIAL_ID and DOM), and which keys can be written back to EEPROM. Typically there are lots of readable keys, and only a handful of writable keys.

OOM supports adding additional keys to existing module types. This can be used by module vendors to access non-standard or proprietary capabilities, or by application developers to extend the list of supported keys beyond the current implementation. An example is in the file *addonsample.py*.

All of the files referenced above are available at https://github.com/ocpnetworking-wip/oom.

License

This software is offered under the MIT License.

Background

Motivation for project:

A Call for Interest posted by Carlos Cardenas on 10/14/2015 as a result of discussions that occurred during the OCP engineering workshop in September 2015 at Fidelity. There was a request for a common diagnostic monitoring utility for NOSs to use since at present, most of them are only able to dump encoded pages.

This call for interest evolved into an "open optical monitoring" group that wrote and tested code that became OOM.

Design

Open Optical Monitoring consists of the following Interfaces and Code modules:

1. Northbound Interface between User Applications and Decode layer
2. Decode Layer
3. South bound Interface between Decode Layer and Linux Kernel / Physical Layer

Northbound Interface Definition (from Decode Layer)

The Northbound API is defined in *oom.py*. The code supersedes this specification.

As a starting point, consider a complete executable python script, printing the part number of each module in each port. Using the Northbound API, it looks like this:

```
from oom import *                # import the Northbound API routines
for port in oom_get_portlist():  # loop through each port
  print port.port_name + ' ' + oom_get_keyvalue(port, 'VENDOR_PN')
```

Table of Python routines in *oom.py*:

Discovery	
API Function	**Description**
oom_get_portlist()	returns a list of **port**s available on the host switch. **Port** is a python class. User accessible class members include: **c_port**: the C port structure returned by the Southbound API (see oom_south.h)

	port_name: The name of the port provided by the Southbound API **port_type**: The type of the port, per the SFF specs. For example: SFP is type 3, QSFP+ is type 13 **mmap**: The dictionary of keys, decoders and locations for everything OOM knows how to access in this port. See *qsfp_plus.py* for the list of QSFP+ keys, for example. **fmap**: The list of keys that form function groups (for oom_get_memory()) **wmap**: The list of writable keys, and the encoder to pack the data to write for each key
Decoded Access to Data Fields	
API Function	**Description**
oom_get_keyvalue(port, key**)**	For the given port, returns the value for the specified key. **Port**: The OOM port as returned by oom_get_portlist() **Key**: Available keys are defined by module type. Refer to the appropriate file for the complete list of keys. Currently *sfp.py* and *qsfp_plus.py* are available. Look for the definition of the MM dictionary to find the function list (MM = Memory Map). Note that the type of the returned value will depend on the key, including strings (eg VENDOR_NAME), integers (eg IDENTIFIER), byte strings (eg VENDOR_OUI), floating point (eg TEMPERATURE), etc. Key names are, to the extent practical, derived directly from the SFF specification for the module. See the key files (eg *sfp.py*, *qsfp.py*) for the SFF Spec reference used.
oom_get_memory(port, function)	Gets a collection of keys from memory (aka EEPROM) at port. Returns a python dictionary of key/value pairs. **Port**: The OOM port as returned by oom_get_portlist() **Function:** Available functions are defined by module type. Refer to the appropriate file for the complete list of functions and the keys returned for those functions. Currently *sfp.py* and *qsfp_plus.py* are available. Look for the definition of the FM dictionary to find the function list (FM = Function Map). SERIAL_ID and DOM should be defined for every port type. The key values returned will match what oom_get_keyvalue() would return individually for each key.

oom_set_keyvalue(port, key, value)	For the given port, ***writes*** the value for the specified key, into module EEPROM. **Port**: The OOM port as returned by oom_get_portlist() **Key**: Available keys are defined by module type. Refer to the appropriate file for the complete list of writable keys. Currently *sfp.py* and *qsfp_plus.py* are available. Look for the definition of the WMAP dictionary to find the list of writable keys (WMAP = Write Map). **Value**: The value to be written into EEPROM. Value should be of the same type and format as returned for that key by oom_get_keyvalue(). The value will be reformatted as appropriate for that key and written into the same location in EEPROM that oom_get_keyvalue() would read that key from. Note for example, that if oom_get_keyvalue() extracts 3 bits from the middle of a byte in EEPROM, and returns those bits in the low 3 bits of a byte, then oom_set_keyvalue() (for that key) will take the low 3 bits of a byte, and the current value of the target byte in EEPROM, merge them appropriately, and write the byte back to EEPROM. If oom_get_keyvalue() for that key returns a string, then oom_set_keyvalue() for that key will expect a string and write it into the appropriate sequential bytes in EEPROM.
R/W Memory (Raw)	
API Function	**Description**
oom_get_memory_sff(*port, address, page, offset, length*)	Gets contents of EEPROM from pluggable module and returns *length* bytes of binary data **Port**: The OOM port as returned by oom_get_portlist() **Address**: A0h, A2h, A8h, as appropriate for the specific module. **Page**: The page (as defined in the SFF spec) where the desired data resides. **Offset**: Byte location for start of read. IMPORTANT NOTE: ALL offsets less than 128 are in the low memory specified by Address, independent of the page specified. From offset 0 to 127, the page is ignored. The first byte on Page <n> will be at offset 128, for all pages. Only one page can be read per call, consecutive pages are not available sequentially in memory (offset + length may never exceed 255). **Length**: Number of bytes to be read
oom_set_memory_sff(*port, address, page, offset, length, data*)	Writes *data* of *length* bytes to the pluggable module at the *address, page, offset* specified See oom_get_memory_sff() for a description of the parameters. Heed the important note regarding pages and offsets.

	Data: Binary data to be written to EEPROM.
R/W "PINs"	
API Function	**Description**
oom_get_function(port, function, Status)	Reads status of function at port and returns a Boolean value **Port**: The OOM port as returned by oom_get_portlist() **Status:** Boolean; True is asserted or False is de-asserted. **Function:** Python key (Note:Availability is Port Type dependent) **Function** **Status** **TxFault** R Asserted Tx has fault; Deasserted Fault not present **TxDisable** RW Asserted is Tx off; Deasserted Tx is on **Mod_ABS** R Asserted module absent; Deasserted Module Present **RS0** RW Asserted RS0 is high; Deasserted RS0 is Low **RS1** RW Asserted RS1 is high; Deasserted RS1 is Low **LOS** R Asserted = Loss of Rx Signal (Rx signal power too low) **Interrupt** R Asserted = Read module status to determine source of interrupt
oom_set_function(port, function, status)	Sets status of function at the port. **Port**: The OOM port as returned by oom_get_portlist() **Status**: asserted or de-asserted. **Function:** Python key (Note:Availability is Port Type dependent) **Function** **Status** **TxDisable** RW Asserted is Tx off; Deasserted Tx is on **RS0** RW RS0 and RS1 should be written together **RS1** RW **Reset** W Asserted HOLDS module in Reset until Deasserted. (causes data flow to be stopped)

Decode layer

The Decode layer is coded in Python. It runs on the switch, receives requests from user applications via the Northbound API, accesses the pluggable modules via the Southbound API, and performs the necessary actions to provide the requested services.

The decode layer is meant to remove the tedium of knowing the specific addresses and definitions of MSA defined fields found in pluggable modules used in OCP related systems.

Currently those modules are limited to SFP+, QSFP+ and QSFP28. In the near future other pluggable platforms are expected to be introduced into the OCP networking domain such as CFP2, CFP4, etc.

These pluggable devices are usually defined by an MSA group. In the case of the SFP, QSFP, and QSFP28, the MSA documents for the Serial ID information and digital monitoring information are controlled by the Small Form Factor (SFF) Committee.

As products evolve, errors are found, and new features are added, the decode layer will be regularly maintained and updated.

Key definitions (location of the data, how to decode them), collections of keys (eg SERIAL_ID, DOM), and writable keys are defined in python files for use by the decode layer. Additional types can be added by properly coding the keys for that type into a new python file, and naming that file appropriately (see 'port_type_e' in *oomlib.py* for the appropriate name for each type).

Additional keys can be added separately via the 'addons' directory. See *addonsample.py* for detailed info on how to add keys, including read keys, collections of keys and writable keys. This capability can be used to create user defined data areas in the scratchpad registers in EEPROM, to describe proprietary data for a specific vendor or module, or to access data fields that are defined in the SFF spec but not included in the OOM key definitions. Addon keys are handled in all respects the same as predefined keys by the decode layer.

SouthBound Interface Definition (from Decode Layer)

The Southbound API is defined in oom_south.h. As of May 2016, the decode layer consumes 3 functions from the Southbound API:

1. oom_get_portlist() returns a list of ports that may contain optical modules. These ports provide the handle to specify which port is the target of subsequent EEPROM read and write calls.
2. oom_get_memory_sff() returns the content of the EEPROM, at the location and for the number of bytes specified.
3. Oom_set_memory_sff() writes the specified buffer to EEPROM at the location and for the number of bytes specified.

In addition to these 3 functions, the data structure for a port is defined, as well as an enumerated type to identify what class of port this is.

There are I/O pin control functions specified but not yet implemented by any shim or consumed by the decode layer. These are oom_get_function() and oom_set_function().

There are two additional read/write functions specified but not yet implemented by any shim or consumed by the decode layer. These are oom_get_memory_cfp() and oom_set_memory_cfp(). They provide the same services as their 'oom_*et_memory_sff' counterparts, but the memory layout of CFP class modules is sufficiently different that a different function call is warranted.

Experience suggests that the currently unused functions are likely to change when first implemented.

Test Plan

The test plan involved several steps. After testing within the labs of pluggable vendors, OCP box providers and OCP NOSs, OOM was demoed live at OCP Summit 2016 in March. In addition it was presented during one of the breakout sections. OOM was then tested with success at the OCP Interoperability event at UnH IOL during the week of May 2, 2016. We expect OOM to be used as part of the OCP Interoperability test process at UNH. The test framework and test code for OOM will be aligned for maximum sharing with that process.

Checklist for Maintenance

Currently the code is maintained in GitHub and the development uses GitHub-based best practices. All code changes are reviewed publicly (using GitHub's online code review tools) and approved by someone with commit rights. The current list of committers/maintainers includes:

- Steve Joiner, Finisar
- Craig Thompson, Finisar
- Don Bollinger, Finisar
- Carlos Cardenas, Cumulus Networks / OCP

It is mandatory that all entities (including the ones listed above) with code approval or commit capability, i.e., are either committers/maintainers into the project be OCP members. We are open to expanding the committers list as other contributors/authors emerge. New contributors/authors cannot become committers/maintainers without first being an OCP member.

In the event that all maintainers are permanently unavailable, a duly appointed representative of the Open Compute Project may take over the project.

Software releases will be made as time and major features are committed. While many open source projects with regular committers have a time-based release model, at least for the near future until the projects popularity increases, we will follow a feature-based release schedule.

Checklist for Governance

This is the list of current governance sites which may change with acceptance into OCP.

Website: N/A

Mailing list: opencompute-networking@lists.opencompute.org

IRC: N/A

Mirror: N/A

GitHub: https://github.com/ocpnetworking-wip/oom

Wiki: http://www.opencompute.org/wiki/Networking/SpecsAndDesigns#Common_Module_Interface

Roadmap

- Working OOM software was demonstrated at the OCP Summit in San Jose in March 2016. The shims also need to implement oom_set_memory_sff(). The decode layer currently writes to EEPROM, both raw and via oom_set_keyvalue(), but has only been demonstrated on a simulator shim.
- Date TBD: "I/O pins" functions (in both the decode layer and the shims)

- As demand arises: CFP support, both raw read/write and keys with decoders.
- Extensions will be needed such as
    - Additional module types
    - MDIO
    - Additional form factors as they become used on OCP switches

Supporting Documents

The OOM code is available in the Github repository. It is reasonably well commented to allow the curious to understand how it works in detail.

Project documents, architecture slides, proposals, meeting minutes, etc are available at http://www.opencompute.org/wiki/Networking/SpecsAndDesigns (look for "Common Module Interface")

SFF documents can be found http://sffcommittee.org/ie/Specifications.html