A L I E N

ABSTRACTION LAYER FOR IMPLEMENTATION OF EXTENSIONS IN PROGRAMMABLE NETWORKS

Collaborative project co-funded by the European Commission within the Seventh Framework Programme

Grant agreement no: **317880**
Project acronym: **ALIEN**
Project full title: **"Abstraction Layer for Implementation of Extensions in programmable Networks"**
Project start date: **01/10/12**
Project duration: **24 months**

Deliverable D3.2
Specification of Hardware Specific Parts

Version 1.0

Due date: 31/07/2013
Submission date: 14/10/2013
Deliverable leader: **Remigiusz Rajewski (PUT)**
Author list: Artur Binczewski, Bartosz Belter, Krzysztof Dombek, Artur Juszczyk, Łukasz Ogrodowczyk, Iwo Olszewski, Damian Parniewicz (PSNC), Janusz Kleban, Marek Michalski, Remigiusz Rajewski, Mariusz Żal (PUT), Hagen Woesner (EICT), Mehdi Rashidi Fard (UNIBRIS), Richard Clegg (UCL), Marc Bruyere (FORCE), Eduardo Jacob, Jon Matias, Maider Huarte (UPV/EHU)

Internal Reviewers: Eduardo Jacob (UPV/EHU), Bartosz Belter (PSNC)

Dissemination Level

☒	**PU:**	Public
☐	**PP:**	Restricted to other programme participants (including the Commission Services)
☐	**RE:**	Restricted to a group specified by the consortium (including the Commission Services)
☐	**CO:**	Confidential, only for members of the consortium (including the Commission Services)

Abstract

This report provides a definition of Hardware Specific Parts (HSPs) of different hardware platforms in the ALIEN project. Report introduces a high-level specification of hardware dependent parts of Hardware Abstraction Layer (HAL) for seven network devices used in the ALIEN project: NetFPGA cards, EZChip NP-3 network processors (in EZappliance platform), Cavium OCTEON network processors (in ACTA system and DELL switch), ADVA optical switches, DOCSIS systems and GEPON systems. The detailed description of hardware specific software components provided in this document has been developed from the official HAL whitepaper released in July 2013. A part of this deliverable is devoted to the functional overview of a reference HAL implementation, based on xDPd and ROFL libraries. This deliverable presents also a software development strategy and plans for the next steps towards the HAL implementation.

Table of Contents

Figure Summary

Table Summary

Executive Summary

This report is focused on the design and specification of hardware specific parts which will be implemented as a part of the Hardware Abstraction Layer (HAL). The innovative network abstraction layer will enable connecting non-OpenFlow capable equipment to OpenFlow networks. The following non-OpenFlow capable platforms (named also "alien" hardware) are considered in the ALIEN project: NetFPGA cards, EZChip NP-3 network processors (EZAppliance), Cavium OCTEON Plus AMC network processor (a module in ATCA systems and DELL switches), ADVA optical switches, DOCSIS and GEPON systems. In this deliverable terms and assumptions for the further implementation of HAL's functional blocks are presented. The hardware abstraction allows to represent features and capabilities of non-OpenFlow capable hardware towards OpenFlow controllers.

Section 1 (***Introduction***) presents the Hardware Abstraction Layer concept based on the HAL whitepaper released by the ALIEN project.

Section 2 (***Functional Overview of the Reference HAL Implementation***) describes a reference HAL implementation, based on the xDPd/ROFL (eXtensible DataPath Daemon/Revised OpenFlow Library) libraries. This section provides also details related to the Hardware Agnostic and Hardware Specific parts, as well as a functional overview of HAL interfaces available for the HSP developers.

Section 3 (***Hardware Specific Parts***) constitutes the main part of the Hardware Specific Parts specification. This section describes the software architecture of the HSP of each platform used in the ALIEN project. The high-level description of architecture components as well as software decomposition and platform interfaces have been drafted. For each platform limitations concerning the supported OpenFlow functionalities have been identified and described. A common structure for all sections has been employed, although some specific hardware can either remove or redefine the contents.

Section 4 (***Software Development Strategy***) describes a guideline and tips to Hardware Specific Parts developers. Specifically, the testing methodology, programming language and code repositories have been introduced.

Section 5 (***Conclusions and Future Steps Towards HAL Implementation***) provides set of common functionalities that will be implemented on each hardware platform. The section has been concluded with directions towards the HAL implementation.

Two appendixes have been attached to the document. The appendixes present the definition of the AFA and the ROFL Pipeline Interfaces and act as a guide for software developers in the ALIEN project.

This report will be used in task T3.3 as an initial document for hardware specific parts implementation and validation.

This deliverable is public and may be followed by people interested in issues related to implementation of OpenFlow functionality at non-OpenFlow capable devices.

1 Introduction

The ALIEN project aims to deliver an innovative network abstraction mechanism targeting the control and management convergence and interoperability of heterogeneous network elements, building strong foundations for Software Defined Networks (SDN). To achieve this goal, building blocks for a novel Hardware Adaptation Layer (HAL) must be designed and implemented. The HAL can facilitate unified integration of “alien” network hardware elements (i.e. any type of network hardware and their capabilities/functionalities that doesn’t support OpenFlow) with an OpenFlow control framework. The “alien” hardware includes: traditional packet switching network elements (e.g., L2 switches), non-packet switching network elements (e.g., optical switch), network monitoring and network processing elements (e.g. NetFPGA, network processors).

This deliverable is focused on the specification of the Hardware Specific Parts (HSPs) of the ALIEN HAL. The HSP is a part of ALIEN HAL implementation which is hardware dependent, therefore it must be designed and implemented consequently for each hardware platform identified by the ALIEN project. In the HAL architecture the northbound interface of the HSP is the Abstract Forwarding API (AFA) interface. This unified interface will be exposed by each HSP implementation and will be used by a hardware agnostic part of the ALIEN HAL, which will be implemented within WP2. The southbound interface of the HSP is a direct interface to the hardware itself.

This document is based on a current version of the specification which has been presented in the ALIEN whitepaper [1] and the preliminary reference implementation. The final version of the HAL specification will be published in Deliverable D2.2.

HSP specifications presented in this document will form a base for the HAL implementation for specific devices that will be performed within WP3.

1.1 Towards the HAL architecture

Following the SDN concept and according to the OpenFlow architecture, the data path and the forwarding engine inside of an OpenFlow device must be controlled and managed by a controller which resides outside of the device and communicates to the device via a control channel. The data path is represented to the controller in an abstracted fashion as a table or a set of tables with so-called flow entries or "flowmods", holding packets information and actions associated to them which could be manipulated (programmed) by the controller software.

ALIEN integrates non-OpenFlow hardware, legacy or new systems into an OpenFlow environment. In order to achieve project goals a Hardware Abstraction Layer (HAL) is introduced. On one hand, the abstraction mechanism allows to hide hardware complexity as well as technology and vendor specific features and, specifically, to present capabilities of the

hardware using the native concept of OpenFlow. On the other hand, all OpenFlow commands are abstracted by the HAL into common requests, generic for any type of hardware and technology. In this environment, hardware specific part, as a part of the HAL, understanding and executing abstracted messages is also required.

The following features of the HAL should be emphasized:

- The HAL resides between OpenFlow Controller and the non-OpenFlow network device i.e. OpenFlow Controller can get control access on data path only through HAL.
- HAL components are reusable. The HAL layer is divided into two parts, i.e. hardware dependent and independent sub-layers. This approach is similar to the Java concept, where we have platform dependent (JVM) and independent (Java Compiler) parts. This approach leads to a smaller code, better troubleshooting and software management.
- The HAL is portable, giving an opportunity to implement OpenFlow on various, currently non-OpenFlow platforms.

The HAL provides certain functionalities for OpenFlow controllers to manage the datapath on a specific device:

- It provides an interface to upper layers (applications) for handling hardware dependent issues.
- It provides information rich enough to support full-featured network operating systems (controllers).
- It exposes interfaces for hardware re-programmability. For example, an ASIC packet-processing fast path could support programmability for deep-packet inspection operations. Another example could be NetFPGAs which users can define their own function on the hardware.
- It provides mechanism for controlling shared abstracted resources such as forwarding-table in virtualized environment.

1.2 The HAL Architecture, basic components and interfaces

The HAL proposed in the ALIEN Whitepaper consists of the two separate layers: the upper Hardware Interface Layer (HIL) and the lower Hardware Presentation Layer (HPL). It can be seen in Figure 1.1.

Bare Metal Management – Although the ultimate goal of creating HAL is to provide an interface for controlling hardware, each hardware platform has different protocol for device management and configuration. The bare metal management component is responsible for hardware initialization and management. In case of configuration changes in hardware platform or introducing new user-defined functionality into the platform, this component will provide tools and interfaces for such events.

HIL – Components in the HIL altogether provide an implementation of common device control and device management protocols i.e. OpenFlow, etc. The HIL components are independent of underlying hardware platform.

OpenFlow end-point – a protocol endpoint responsible for maintaining connection with a controller.

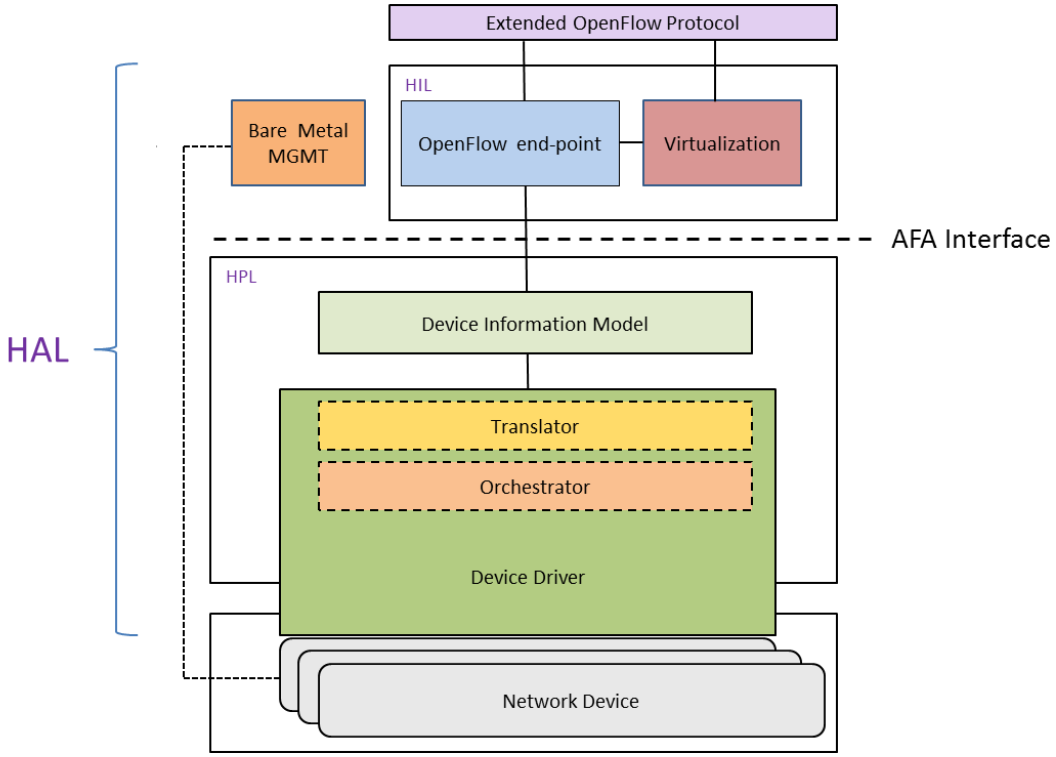

Figure 1.1 The HAL architecture

AFA Interface – Abstract Forwarding API (AFA) Interface between HIL and HPL sub-layers. It lies between hardware agnostic and hardware specific parts of the HAL. AFA is binding the platform specific forwarding module with control and management module, which is keeping the abstracted switching information.

HPL – The Hardware Presentation Layer (HPL), play similar role as sixth layer (presentation layer) of OSI RM. Its major task is to "present" the capabilities of the hardware to hardware independent HAL layer and on the other hand it interacts with lower layers represented by network device. HPL includes few components which have distinct functionality. The northbound of HPL provides an API for upper layer to create extended OpenFlow table. In the following, HPL components and their role and functionality are explained.

Device Information Model (DIM) – Is a model of configured OF switch that represents switch's state which probably is platform independent. It's a pure data model without any data processing capability.

Device Driver – Is a piece of software that performs data processing using hardware device/accelerators. In case of closed devices, it only configures a device to obtain the same behavior like modeled by DIM OF switch (a kind of translation). It is a platform dependent component of the HAL.

Translator – Is part of device driver. it is responsible for translating all entries and actions from OpenFlow switch model (DIM) into platform specific commands and configurations. This module is not mandatory, e.g. for devices that supports OpenFlow data model.

Orchestrator – Is part of driver components and responsible for configuring the entire device network if available, i.e. multiple components, to simulate the behavior of OpenFlow switch modeled by DIM. As an example, GEPON needs configuration for both optical and electrical parts for appropriate functioning.

The remainder of this deliverable is organized as follows. Chapter 2 introduces HAL interfaces. Next chapters describe hardware specific parts for EZappliance (Chapter 3.1), hardware specific parts for ATCA (Chapter 3.2), hardware specific parts for NetFPGA cards (Chapter 3.3), hardware specific parts for L0 switch (Chapter 3.4), hardware specific parts for Dell/Force10 Switch (Chapter 3.5), hardware specific parts for GEPON (Chapter 3.6), and hardware specific parts for DOCSIS (Chapter 3.7). In Chapter 4 is described software development strategy. Chapter 5 includes the common OpenFlow functionalities for a different hardware described in Chapter 3. It includes also conclusions and directions towards HAL implementation.

2 Functional Overview of the HAL Reference Implementation

In the ALIEN project, xDPd/ROFL (eXtensible DataPath Daemon/Revised OpenFlow Library) libraries [6, 7] are used as a reference HAL concept implementation and the framework for creating OpenFlow agents communicating with different types of hardware platforms.

The ROFL library is a set of libraries to build both OpenFlow data paths and controllers. On one hand, the controller ("rofl-common") part of the library includes a full C/C++ OpenFlow v1.0, v1.2 and v1.3.2 endpoint to be embedded both in data path elements as well as in controllers, including hierarchical (like FlowVisor) or recursive controllers. On the other, the ROFL library includes two main libraries for building data path elements, the Abstract Forwarding API ("rofl-afa") and the Pipeline ("rofl-pipeline") sub-libraries. The Abstract Forwarding API is a set of header files which define a hardware agnostic API for heterogeneous forwarding engine management. ROFL-pipeline in its turn, is a complete OpenFlow engine and forwarding data model, for OpenFlow versions v1.0, v1.2 and v1.3.2. The "rofl-afa" library currently uses "rofl-pipeline" as its data model.

The xDPd is a multi-platform OpenFlow data path element, build using ROFL libraries. The architecture of xDPd is made of basically three components (see Figure 2.1):

- the Control and Management Module in charge of the common control part of the logical switches (OpenFlow endpoint) and the management,
- the AFA interface of HAL, which abstracts the details of the underlying platform via the hardware agnostic C interface,
- the forwarding modules (also known as drivers) for the specific platform, which fulfills the AFA interface.

The usage of Pipeline platform interface for a platform driver implementation is suggested for devices which could compile and execute xDPd/ROFL libraries inside the device and Pipeline module could have access to incoming network packets and will be able to control packet processing within the device (e.g.: OCTEON-based network platforms). The benefit of using this approach for creating platform driver is that ROFL library already implements the whole logic related to packet processing (the logic is based on OpenFlow protocol pipeline) and driver developer has to implement only low level system operations like packet IO handling, memory management, etc. The AFA interface could be used on any platform where it is not possible to utilize ROFL pipeline implementation. In this case platform driver developer is in charge of implementing OpenFlow pipeline packet processing with usage platform specific features.

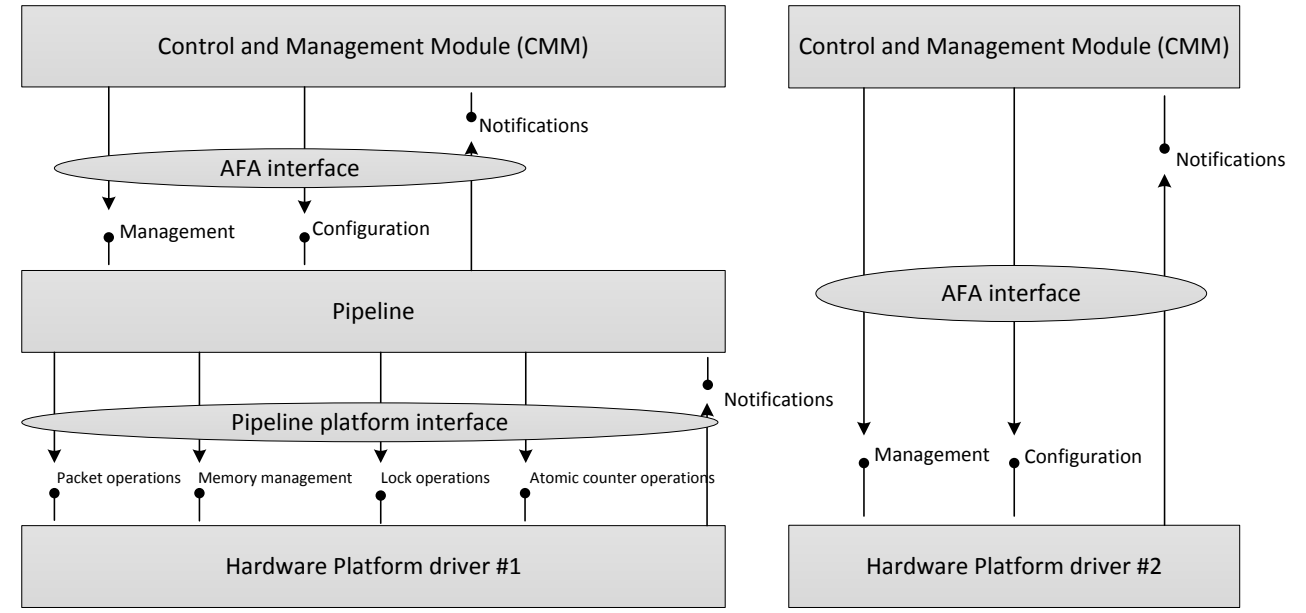

Figure 2.1 Overview of interfaces (provided by ROFL implementation of the HAL concept) to be used by Platform driver developer

2.1 AFA Interface

Abstract Forwarding API is binding the platform specific, forwarding module with control and management module, which is keeping the abstracted switching information. AFA interface provide below features:

- Management actions: allow for creating and destructing logical switches within the forwarding module (a "driver"). It binds the logical switch instance and ports or network interfaces,
- Configuration actions: passing OpenFlow requests to the forwarding module (e.g.: flow tables entry addition, flow statistics query, etc.),
- Notification of events : Passing events generated by the forwarding module to CMM module (e.g.: port status changes, OpenFlow packet-in and other OpenFlow messages and errors).

2.1.1 Datapath Management

The management API of AFA interface [16] is responsible for managing the abstracted platform, fundamentally the logical switches and switch ports, as well as their relation, during life time of the OpenFlow logical switch. At the beginning, the forwarding module for a specific hardware platform has to be initialized and device interfaces have to be discovered. After this, the AFA interface allows the creation of parallel and independent logical switches. Any switch port (device interface) can be attached to only one logical switch. In this way, a hardware platform can be logically partitioned into several OpenFlow-controlled data path elements. Figure 2.2 presents an overview of the logical relations between forwarding module, logical switches and switch ports, and partition of functions between entities.

The management part of the AFA interface is available as a set of common data structures and functions whose details are listed below. These functions should be implemented for each hardware platform which will be managed by the AFA interface.

Most important data structures are described in Appendix A.1.

Functions of the AFA interface for datapath management are described in detail in Appendix A.2.

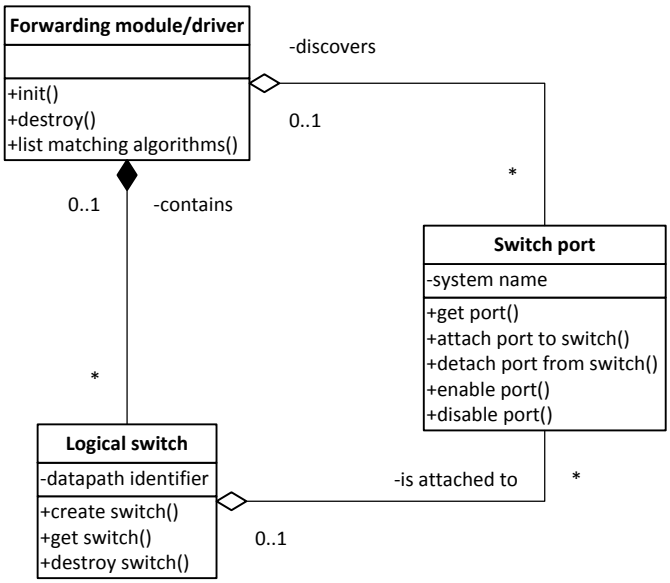

Figure 2.2 Overview of the logical relations between AFA management API entities[1]

2.1.2 Datapath Configuration

The operational part of the AFA interface API [14, 15] represents a collection of functions which are triggered by messages coming from OpenFlow controller:

- configuration messages,
- flow and group table modifications,
- requests for flow and group table statistics.

Figure 2.3 presents an overview of the logical relations between entities controlled by OpenFlow protocol: logical switch, tables, groups and switch ports as well as functions related to each kind of entity.

Functions of operational part of the AFA interface should be implemented for each hardware platform which will be operated by the AFA interface.

Functions of the AFA interface for datapath configuration are described in detail in Appendix A.3.

[1] The AFA forwarding module API is composed of C-style functions, not objects and its methods – Figure presents only simplified overview of available functions by adapting UML notation.

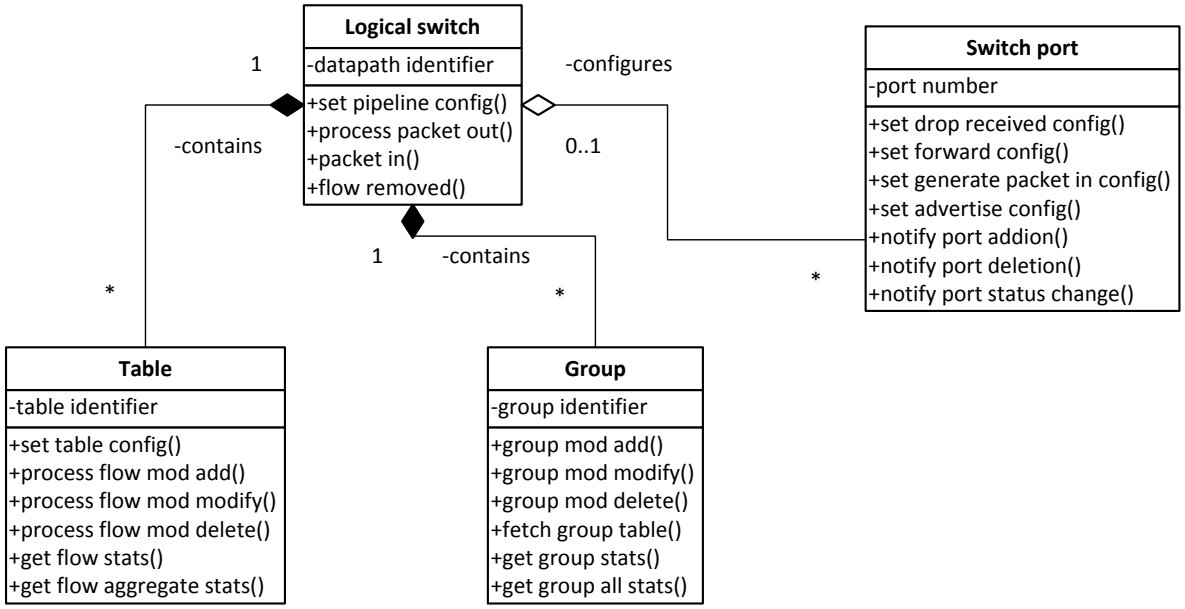

Figure 2.3 Overview of the logical relations between AFA operational API entities[2]

2.1.3 Notifications

Platform specific datapath implementation can generate notifications (e.g.: port change, packet for controller, flow entries expiration) towards CMM module existing on north side of AFA interface. These functions are implemented by CMM module [16] and used within datapath module.

Functions of the AFA interface for notifications are described in detail in Appendix A.4.

2.2 Pipeline Platform Interface

Hardware platforms forwarding module (or hardware driver) can, at discretion, use pipeline platform library API to: a) maintain the state of the forwarding module OpenFlow (regardless of its nature; hardware, software or hybrid) b) use the runtime APIs to process packets via software (e.g. in software switches, certain network processors, or pack out events on ASICs or FPGAs).

The ROFL-pipeline provides an implementation of an OpenFlow forwarding engine or pipeline. It also includes a runtime sub-API which is able to process abstract representations of real data packets across an OpenFlow pipeline. The pipeline platform interface exists in order to synchronize the abstract representation of packet in ROFL-pipeline with the real packet in a network device (mangling the packet, like reading packets fields, modifying packet and apply forwarding decisions).

Additionally, ROFL-pipeline deployment on particular hardware platforms, especially on software ones, require specific memory management and synchronization mechanisms. For this reason ROFL-pipeline, memory, locking and atomic

[2] The AFA forwarding module API is composed of C-style functions, not objects and its methods – Figure presents only simplified overview of available functions by adapting UML notation.

operations platform interfaces contain also additional API calls to abstract these platform specific functions from the hardware-agnostic OpenFlow pipeline implementation.

2.2.1 Packet Operations

Packet operations functions [17] allows the ROFL pipeline to get information about and manipulate real packets processed by the network platform. Thanks to this part of API, the pipeline doesn't have to pass a full packet between device and pipeline. Instead, ROFL pipeline creates abstracted packet representation to be processed through pipeline tables. These functions should be implemented for each hardware platform which will be operated by ROFL pipeline interface. If the forwarding module does not process packet through pipeline, the implementation of these function can be empty.

Functions and data structures of ROFL pipeline for packet operations are described in Appendix B.1.

2.2.2 Memory Allocation, Lock and Atomic Counter Operations

These functions provide interface for memory management [18], synchronization mechanisms [19] and atomic counter operations [20] which is important for programmable network processors. Functions declarations are reasonably compliant with POSIX standard so standard Linux functions can be easily used if available within particular network processor platform. These functions should be implemented for each hardware platform which will be operated by ROFL pipeline interface.

Functions and data structures of ROFL pipeline for memory allocation, lock operations and atomic counter operations are described in Appendix B.2, B.3 and B.4.

2.2.3 Notifications

This interface [21] is implemented by ROFL Pipeline and can be used by platform driver to activate packet processing within OpenFlow pipeline.

Function of ROFL pipeline for notifications is described in Appendix B.5.

3 Hardware Specific Parts

As presented in the introduction the Hardware Specific Part (HSP) is a part of HAL layer below AFA interface. It is a hardware-dependent part of the HAL. It consists of Hardware Presentation Layer (HPL) of the HAL as well as a network device with a corresponding driver. HSPs are sets of driver libraries that provide interfaces for detection of hardware capabilities, hardware configuration and statistic collection.

AFA is a northbound interface common for all HSPs. It is C-language interface provided by WP2 and it will be described in details in a D2.2 deliverable. Brief description of the preliminary version of the AFA interface is provided in chapter 2.2 of this deliverable.

Specifications of HSPs are presented in sub-chapters below for all ALIEN hardware platforms:

- EZappliance,
- ATCA,
- NetFPGA,
- Layer 0 switch,
- Dell switch,
- GEPON,
- DOCSIS.

Architectures of HSPs for all above-mentioned network devices are described and compared to general HAL architecture. Software decomposition with modules and interfaces descriptions are done together with software deployment plans. Finally limitations due to nature of the platforms and their switching constraints are presented. As explained before a common presentation for every platform is proposed, although small variations can appear. It's worth mentioning that when in specific cases some similarities appear between hardware platforms (i.e. ATCA Cavium module and Dell/Force10 SDP Cavium based module or GEPON and DOCSIS platforms) the sections are self-contained and don't point to descriptions in other sections.

3.1 EZappliance

3.1.1 Logical Architecture

The EZappliance is a platform for developing and deploying network applications. It is composed of the programmable NP-3 network processor capable of Ethernet packet manipulation and Host CPU build-in Linux system, where any management or control plane software can be deployed. More details of the EZappliance platform is presented in ALIEN project's deliverable D3.1 [2].

Hardware Specific Part (HSP) for EZappliance platform will be a low-level part of full featured OpenFlow switch, supporting all standard (i.e. OpenFlow-defined) packet matching rules and actions. In order to negate all hardware limitations, it will be a hardware-software composition which is interfaced with hardware agnostic part using the AFA interface. The hardware part implemented on the NP-3 network processor will provide a fast data path for flows supported by the NP-3 processor (most of the flows) and act as an processing accelerator for a software part. Packets which processing goes beyond the capabilities of hardware part will be processed by a slow datapath provided by software modules. Software modules are also responsible for configuration and management of hardware part.

Figure 3.1 presents a high level logical architecture of Hardware Specific Part of HAL for EZappliance. This design extends overall HAL architecture of several platform specific modules.

Translator (EZ Forward) is responsible for transforming OpenFlow switch model from DIM into form understandable by Hardware Datapath. Flow definitions, action sets, etc. are translated into EZappliance configurations and search structure entries. Translator also takes decisions which flows will be configured on the Hardware Datapath.

Device state stored in **Device Information Module (DIM)** is used by EZ Forward module and by Software Datapath.

Hardware Datapath module is implemented using NP-3 network processor in EZappliance device. This results in very high traffic processing rate but introduces some limitations. Processor architecture does not allow to match every flow and execute every set of actions. Each data packet will be processed by this module because all data plane interfaces are part of it. However some packets will be passed to the Software Datapath for further processing.

Software Datapath is a pure software OpenFlow pipeline implementation. This module processes packets which could not be successfully processed within Hardware Datapath. This module is slow in comparison with Hardware Datapath, but implements full OpenFlow 1.2 specification [3]. It is also responsible for generating *packet_in* and processing *packet_out* actions.

Orchestrator, an optional module described in general ALIEN HAL architecture, is not used in EZappliance HSP architecture. It is designed for orchestrating multiple hardware components to behave as a single OF switch but HSP for EZappliance is built on top of an only one hardware component.

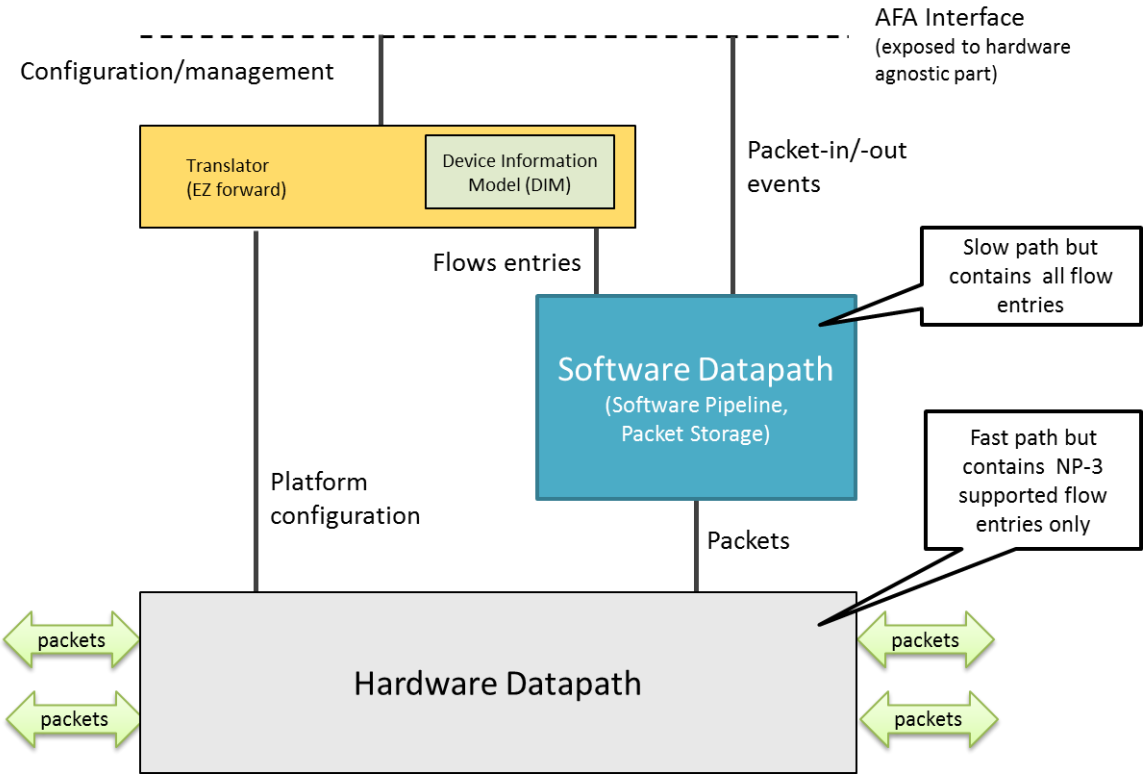

Figure 3.1 High level overview of Hardware Specific part for EZappliance platform

3.1.2 Software Decomposition

The Hardware Specific part for EZappliance platform will be implemented as a set of software components presented in Figure 3.2. Some part of these components will be reused or adapted from xDPd and ROFL software projects (Hardware Agnostic Part, Software Pipeline, Packet Storage). The rest of the Hardware Specific Part components have to be developed from scratch.

Hardware Agnostic Part

Hardware Agnostic Part is a common part of OpenFlow agent for all hardware platforms. This component is provided by WP2 activity in form of Control and Management Module (CMM) within xDPd software project. The xDPd CMM software component is a standalone Linux program which provides the platform for instantiating EZappliance platform specific components and is capable of tasks scheduling both for hardware agnostic and EZ driver workflows. Design of this software component is not covered within EZappliance Hardware Specific Part section.

EZ Forward

EZ Forward component must be developed from scratch and implements the AFA interface for the Hardware Agnostic Part. EZ Forward is responsible for transforming OpenFlow switch model from DIM into form understandable by Hardware Datapath. Flow definitions, action sets, etc. are translated into EZappliance configurations and search structure entries. EZ Forward takes decisions which flows will be configured on Hardware Datapath and translates logical

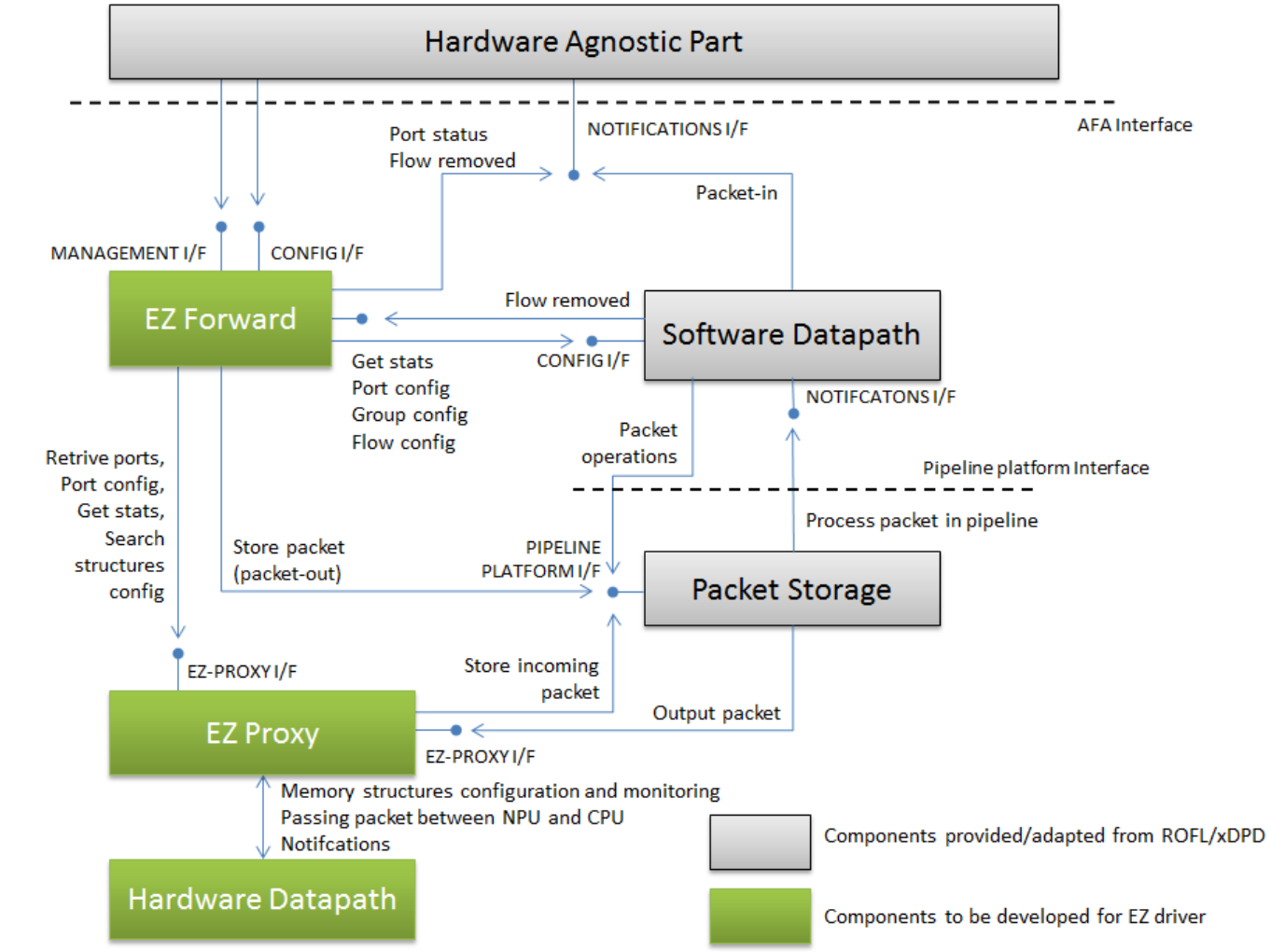

Figure 3.2 Software structure of the driver components for EZappliance platform

port numbers contained by flow entries to platform physical network port identifiers. EZ Forward instantiates and controls Software Datapath and Packet Storage components. External software interfaces within EZ Forward component are presented in Table 1.

Software Datapath

Software Datapath is a pure software OpenFlow pipeline implementation. This module processes packets which could not be successfully processed within Hardware Datapath. This module is slow in comparison with Hardware Datapath, but implements full OpenFlow specification. It is also responsible for generating *packet_in* and processing *packet_out* actions. This component will be reused from ROFL software project and small adaptation development should be made (i.e.: flow removed notification will be sent to EZ Forward instead of CMM module). Design of this software component is not covered within EZappliance Specific Part section.

Table 1 External software interfaces within EZ Forward component

Interface	Usage	Peer module	Description
AFA MANAGEMENT	Provider	CMM	Allows for creation of a logical switch basing on EZappliance device, retrieval of physical ports, ports assignment to a logical switch and setting port configuration.
AFA CONFIG	Provider	CMM	Allows for configuration of packet processing rules (OpenFlow flows) inside EZappliance platform as well as getting statistics related to OpenFlow flows and groups.
FORWARD NOTIFICATION	Provider	Software Datapath	Allows for notification about flow removal within Software Datapath. This interface is triggering a flow removal from Hardware Datapath.
EZ-PROXY	Client	EZ Proxy	Used to retrieve physical data plane ports available in EZappliance, setting port configuration, traffic processing rules configuration in NP-3 processor by accessing, setting values in search memory structures located inside NP-3 and retrieving packet processing statistics.
PLATFORM DATAPATH	Client	Packet Storage	Used to send packet, originated by OpenFlow controller, to Software Datapath. The packet has to be processed through Software Datapath but Software Datapath is requiring to store the packet first.
AFA NOTIFICATION	Client	CMM	Used to notify OpenFlow controller about port status change or flow removal from EZappliance datapath.

Packet Storage

The aim of Packet Storage is to support data processing in Software Datapath. Every packet, which cannot be handled by Hardware Datapath, is buffered in this module while its headers are processed locally or are sent to controller in *packet_in* action. This component will be reused from ROFL software project and some adaptation development should be made (i.e.: *output_packet* action should use EZ-PROXY interface). Design of this software component is not covered within EZappliance Specific Part section.

EZ Proxy

EZ Proxy is a component implemented from scratch by PSNC. It is designed to partially hide Network Processor Unit (NPU) NP-3 complexity. This module exposes platform management and configuration functions easy for use by aforementioned modules. This component can be adapted in case of new requirements. External software interfaces within EZ Proxy component can be seen in Table 2.

EZ Proxy module uses vendor API to interact with Hardware Datapath. Direct access to search structures and statistic counters in NP-3 memory allows controlling microcode behavior. API allows also for bidirectional frame transmission and for initial configuration of network processor and its memories.

Table 2 External software interfaces within EZ Proxy component

Interface	Usage	Peer module	Description
EZ PROXY	Provider	EZ Forward Packet Storage	Allow for NP-3 network processor configuration and network packets receiving from and sending to the network.

More detailed information about EZ Proxy component structure and EZ PROXY interface are provided in Section 3.1.4.

Hardware Datapath

Hardware Datapath module will be implemented within NP-3 network processor of the EZappliance device. It will provide very high traffic processing rate but some NP-3 processor limitations must be complied. Processor architecture does not allow matching every flow and executing every set of actions. Each data packet will be processed by this module because all data plane interfaces are part of it. However some packets will be passed to the Software Datapath for further processing. Other components can interact with Hardware Datapath only via EZ Proxy module.

3.1.3 Software Deployment

Software modules presented in Section 3.1.2 will require to be deployed on various types of execution environments. Hardware specific part for EZAppliance consists of software for NP-3 network processor, EZ Host Power PC Linux and x86 Linux platform (see Figure 3.3).

The deployment of OpenFlow agent for EZAppliance will require the installation of the following software components:

- xDPd on x86 linux,
- EZProxy on Power PC embedded Linux,
- TOP microcode on EZchip NP-3 network processor.

The xDPd package with all needed libraries (https://www.codebasin.net/redmine/projects/xdpd/wiki) will be hosted on any GNU Linux. EZProxy package will be hosted on EZappliance embedded Linux with a set of required libraries (libomniorb, DENX ELDK system libraries). Network processor is a NP-3 processor built-in into EZappliance device. Hardware Datapath microcode will be installed directly on this NPU.

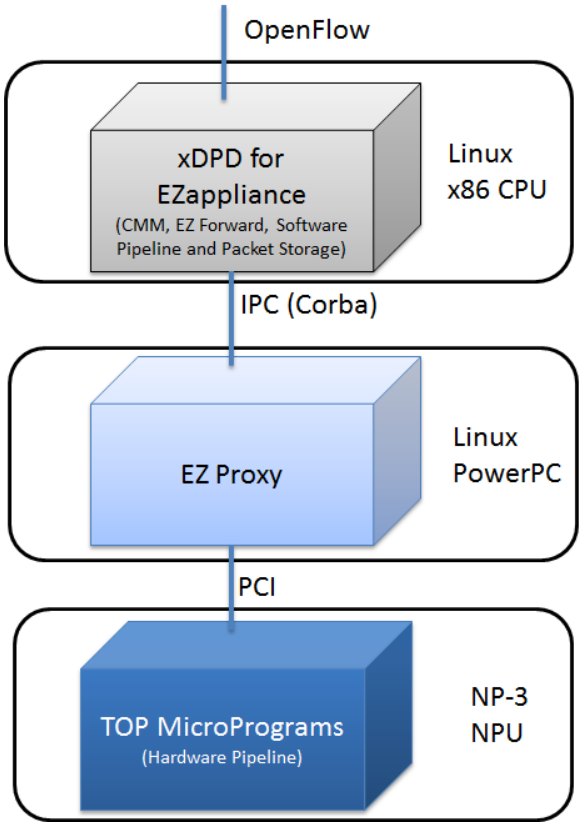

Figure 3.3 Mapping of logical software components to programs (executables) and its environmental requirements

3.1.4 Platform Interfaces Description

This section describes an EZappliance NP-3 interface exposed by EZ Proxy component. The access to real NP-3 network processor interface is protected by non-disclosure agreements and cannot be presented in this document so we provide details of EZ-PROXY interface which mirrors some subset of methods available within real NP-3 network processor interface.

EZ-PROXY interface can be splitted into three functional groups:

- functions for Hardware Configuration
- functions for Hardware Capabilities Detection, Monitoring and Statistics Collection
- functions for frames and packets management

These three groups of functions are available by EZdriver API library (C library provided by EZChip Company [4]) and exposed by EZ Proxy through Corba technology [5] using omniorb library [6]. EZ-PROXY interface provides access to NP-3

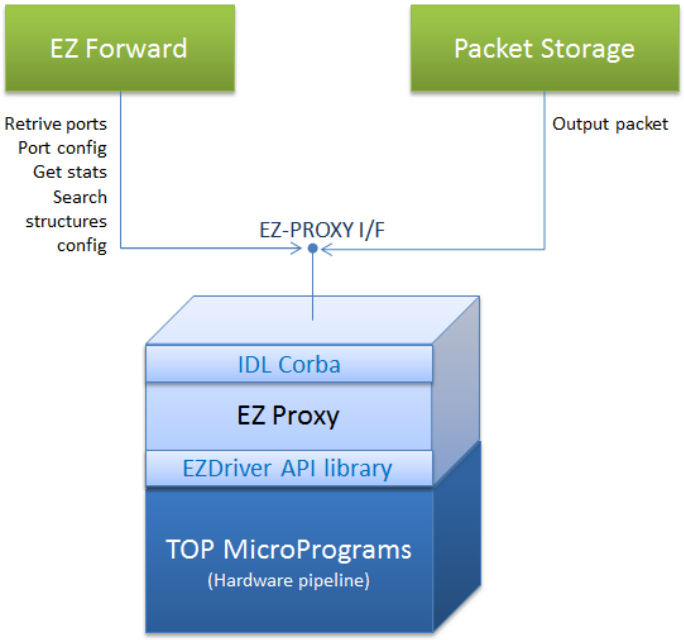

Figure 3.4 EZ-PROXY interface

network processor configuration, getting statistics and parameters monitoring as well as frames and packets management. Software interfaces of EZappliance platform allow access to NP-3 EZchip network processor which act as a hardware accelerator of pipeline processing (Hardware Datapath).

Interface exposed by EZ Proxy component will be used by EZ Forward as well as Packed Storage modules (see Section 3.1.2) through EZ-PROXY interface as depicted on Figure 3.4.

EZ-PROXY interface provides functions for:

- **frames/packets exchange between NP-3 network processor and upper layers**
  Frames (packets) could be exchange between NPU and CPU through PCI bus. Performance of this interface is limited to about 2Gbps. All frames that couldn't be processed into network processor will be sent to upper layer to Software Datapath module. This functionality will be used during *packet_in*/*packet_out* operations.

- **search structures configuration (addition/deletion of entries)**
  Each search structures consist of key with mask (in some types of search structures) and result. Entries are stored within TOP Search processor memory structure. Structures differ in type and key/result length and are declared during memory partitioning of NP. Search structures will be used by NP-3 network processor as a flow tables during hardware pipeline processing. They may be also read by upper layers for hardware monitoring and capabilities detection. As mentioned above search structures types like binary trees or hash tables are defined by vendor. Max length of search key for tree structure is 38 bytes.

- **configuration and monitoring of traffic conditioning mechanisms (e.g. shaper) in Traffic Managers entities**
  Traffic Management entities (TMs) provide queue management mechanisms built in NP-3 network processor. TMs are not programmable, but only configurable entities (they may be configured by process installed in the

Host CPU system by using EZdriver API). Traffic Managers allow for configuration of traffic conditioning mechanisms. Different mechanisms could be configured to meet the Quality of Service guarantees e.g. WRED, DRR, WFQ.

- **Token Bucket configuration for implementation of policer mechanism**
  EZchip provide Token bucket algorithm. It is programmed in Top Resolve and is preconfigured in NPsl's scripts. Token Bucket is configured with associated profiles that define used parameters. Operation over profiles is handled by EZ-PROXY interface.

- **Management of statistic counters used for traffic monitoring.**
  For device management and statistics purpose EZchip provide set of build-in features which will be used by ALIEN developers. EZAppliance give access to monitor physical parameters of device, as port up/down, temperature, SFP port etc. Also more complicated statistics could be performed by statistic counters. The process depends on how it is programmed in EZchip but give flexible approach for platform developers. All mentioned management and statistics mechanism will be reflected in EZ-PROXY interface.

All abovementioned functions will be defined in IDL language and exposed using Corba technology [5].

3.1.5 Supported OpenFlow Match Fields

All match fields defined as required and optional in OpenFlow 1.2 specification [3] will be supported by EZappliance HSP implementation.

3.1.6 Supported OpenFlow Actions

The NP-3 network processor is designed as a general solution for creating any packet processing devices. Thanks to that it is supporting any required actions related to the packet processing. They are collected in Table 3.

Table 3 Supported OpenFlow actions

Action	Description
Forward to physical port (Output)	Processed packet is directly sent out by given data plane interface. In NP-3 processor, packet must always be sent out through QoS queue - *output* and *queue* actions are equivalent within NP-3.
Forward to all physical port (Output ALL)	Processed packet is directly sent out by all interfaces (excluding input port of frame). In NP-3 processor, packet must always be sent out through QoS queue.
Forward along spanning tree ports (Output FLOOD)	Processed packet is directly sent out along the minimum spanning tree interfaces (excluding input port of frame). This action is identical with *output_all* action because we will not support spanning tree protocol. In NP-3 processor, packet must always be sent out through QoS queue.
Forward to controller (Output CONTROLLER)	Processed packet is sent to the controller.
Forward to a table	Start processing a packet in a specified flow table.

(Output TABLE) (Goto-Table)	
Forward to input port (Output IN_PORT)	Processed packet is sent out by the input interface.
Multiple forwards	A single action list can contain multiple forward actions.
Enqueue (Set-Queue)	Forward packet through a queue attached to a port.
Drop	Matched packet should be dropped.
Modify-Field actions (Set-Field)	Modify-Field actions allow adding, modifying and removing most important fields, structures in processed packet headers.
Apply-Actions	Applies actions immediately in current OpenFlow table processing. In NP-3 processor, the TOP Modify is taking part within a single OpenFlow table processing.
Write-Actions	Merge the specified actions into a current action set. The action set can be implemented using TOP messages with are used to pass additional information between TOP processors as well as for the next round of TOP pipeline processing.
Clear-Actions	Clears action set immediately. See above.

3.1.7 Host CPU Limitations

Host CPU is the small board in EZappliance device with embedded Linux like system that enables developing high-level applications. However because of limited CPU power and operating memory resources (800 MHz of CPU and 512 MB of RAM) not all EZappliance Specific Parts can be deployed there. Some software components of Hardware Specific Part for EZappliance platform will be installed on external Linux System. There is also 32-bit 66 MHz PCI between NP-3 network processor and Host CPU board so performance of this interface is limited to about 2Gbps. Another performance limitation will be 1Gbps Ethernet interface between Host CPU and external Linux System.

3.1.8 Flow Matching Limitations

The NP-3 processor architecture does not allow matching every flow and executing every set of actions. NP-3 is equipped with not large enough TCAM memory (Ternary Content Addressable Memory) which is the only entity capable of wildcarded matches. EZchip NP-3 developer doesn't have direct access to TCAM memory but has to use instead search structures implemented by vendor like binary tree or hash table. Max length of search key is 38 bytes what is not enough for all match fields required by OF. This limitation will results in the processing of certain flows in Software Datapath instead of Hardware Datapath.

3.1.9 Unsupported OpenFlow Functionalities

The first version of Hardware Specific Parts for EZappliance will not support OpenFlow functionalities showed in Table 4.

Table 4 Unsupported OpenFlow functionalities

Action	Description
Group	Process a packet through the specified group in group table. This is not supported in current

	version of the EZdriver specification but generally EZappliance device can support this action.

3.1.10 Flow Tables

EZappliance is capable of containing multiple OpenFlow tables and packet processing through those tables. Other issues related to multiple tables are that NP-3 limited size of processor memory is divided between each table so, if more tables will be created, a single table can contain less entries. The last, very important impact factor of network packets processing through many OpenFlow tables is that packet has to be looped 1-2 times within TOP processors pipeline per single OpenFlow table processing. This means that having many OpenFlow tables is reducing total available throughput of the device. These limitations can result in limited number of supported tables.

3.1.11 Virtualization

NP-3 network processor does not allow to be partitioned as a resource. There is no possibility to run multiple processes or maintain processing power allocation in other way. For this reason there can be only one instance of Hardware Datapath module. Technically it is possible to run multiple instances of Software Datapath, but it will not allow providing a performance isolation as traffic processing is always divided between both Hardware and Software Datapaths. The only way to virtualise OF switch built using the EZappliance is to partition a flow space like FlowVisor do. This kind of virtualization should be provided by Hardware Agnostic Part thus is out of scope of this document.

In summary, only a single Logical Switch Instance will be supported by EZappliance Hardware Specific Part implementation.

3.2 Hardware Specific Parts for ATCA (Cavium OCTEON Network Processor)

As described in deliverable D3.1 [2], EICT has access to an ATCA (Advanced Telecommunications Computing Architecture) platform produced by Kontron on which the partner develops ports of its eXtensible DataPath daemon (xDPd) [7]. For internal purposes another ATCA platform produced by Emerson depicts the upgrade path.

This chapter describes the architecture of the planned implementation. As of the time of writing, a first version of the software is in the testing phase and will still be developed further in the coming months.

The OM 8020 platform of Kontron consists of two AM8404 carrier boards that each host an Ethernet switch chip with 24*1GbE and 2*10GbE. One of the 10GbE ports of these carriers is wired back-to-back to the opposite carrier board. The 8404 carries up to 4 AMC (Advanced Mezzanine Cards), of which one is an AM4204 with a Cavium OCTEON Plus CN5650 on it.

The NPU has 12 MIPS64 R2 cores that run at 600MHz, and that can run either in a standalone mode (SE-S), in Linux user mode (SE-UM), or run under the control of SMP Linux (Synchronous Multi-Processing). A hybrid configuration of, e.g., a number of cores in Linux mode and the rest of the cores in SE-S, is possible and will be used in our implementation.

Standalone mode means that an ELF executable runs on the core without the burden of an entire operating system around it. Therefore standalone operation is promising a big speed-up in the processing performance compared to operation in Linux mode.

3.2.1 Architecture of Specific Parts

The architecture of xDPd as described in [7] and Section 2 foresees the separation of a protocol version dependent Control and Management Module (CMM) from a hardware-specific, but version-agnostic data path.

This separation is mapped onto different work groups on the OCTEON. One work group runs Linux and implements the CMM as well as the software parts responsible for initialization of the hardware pipeline, while the other cores run the actual hardware implementation of the pipeline. The OCTEON provides functions for the packet I/O, as well as hardware acceleration units (timers, Deterministic Finite Automatons, ZIP engine, per-core CRC engine).

The Linux core is responsible for initializing the bootmem memory that is shared between the two work groups. This shared memory is holding three parts:

- A state machine that is starting/stopping the standalone (SE) cores. The Linux core sets a variable that indicates that standalone cores have to hold operation until memory is initialized or entries in the FlowTable are changed. Afterwards it resets the status to “go”
- A buffer that lets SE cores store packets that do not match. These frames will be red out from Linux and transformed into *packet_in* messages to be sent upwards.
- The actual FlowTable. This is written by the Linux core and read (only) by the SE cores.

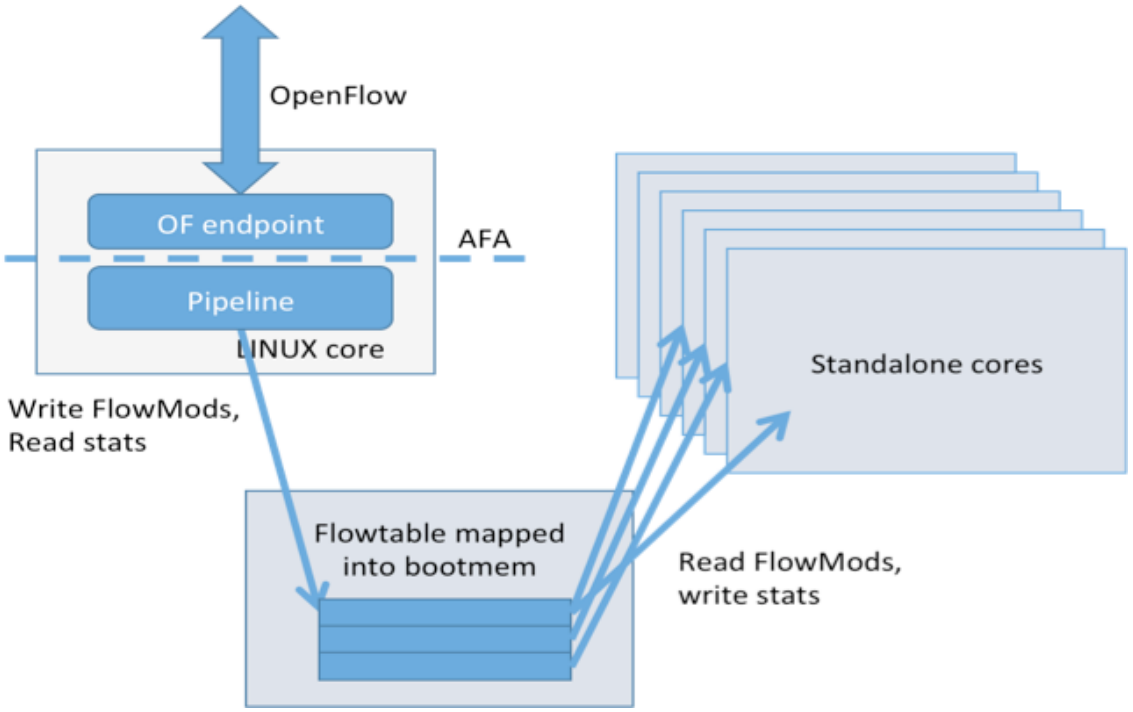

Figure 3.5 Relation between Linux core and SE cores in the OCTEON Plus implementation

Actual packet flow is going through the SE cores exclusively, except in the case when there is no match in the FlowTable.

The relation between Linux core and SE cores in the OCTEON Plus implementation is show in Figure 3.5.

3.2.2 Interaction with Device

The interaction of the controller with the device is taking place via OpenFlow. The OpenFlow endpoint is the one implemented as part of ROFL in the CMM. Inside the OCTEON processor itself another API is used to access the specific functions and registers of hardware accelerators. This API is called Simple Executive API (SE-API) or ***HAL*** in the OCTEON Users' Manual.

Linux core implements a pipeline that is a logical representation of the SE cores, and no packet actually passes through this (see Figure 3.6).

The SE core retrieves the WQE from the Work Queue (1). The header is matched against the FlowTable, and if no entry is found (2a), the WQE is entered into the *PACKET_IN_BUF*, where the Linux core can retrieve it. If a matching entry is found (2b), the frame is processed from and copied to L2 cache. The resulting WQE is then put into one of the output queues. In Figure 3.7 it could be seen workflow in and out if the SE cores.

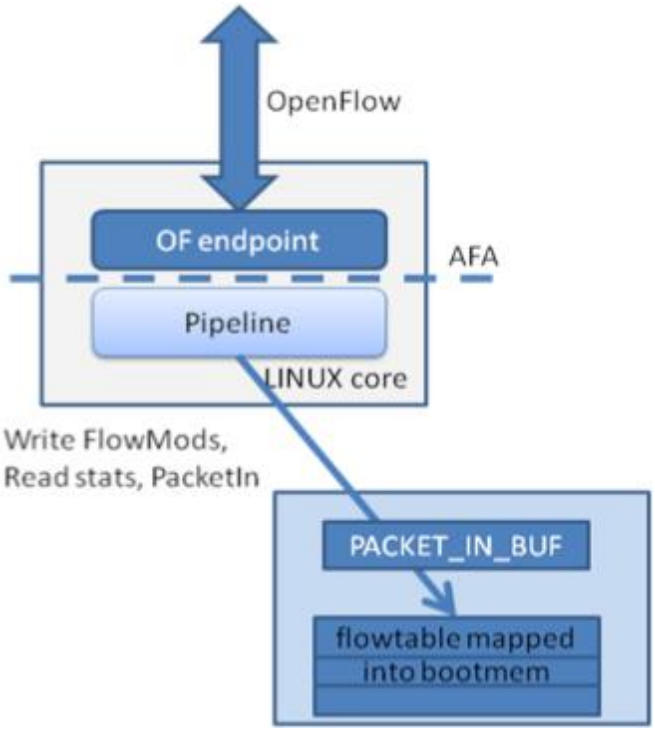

Figure 3.6 Linux core implements a pipeline that is a logical representation of the SE cores, and no packet actually passes through this

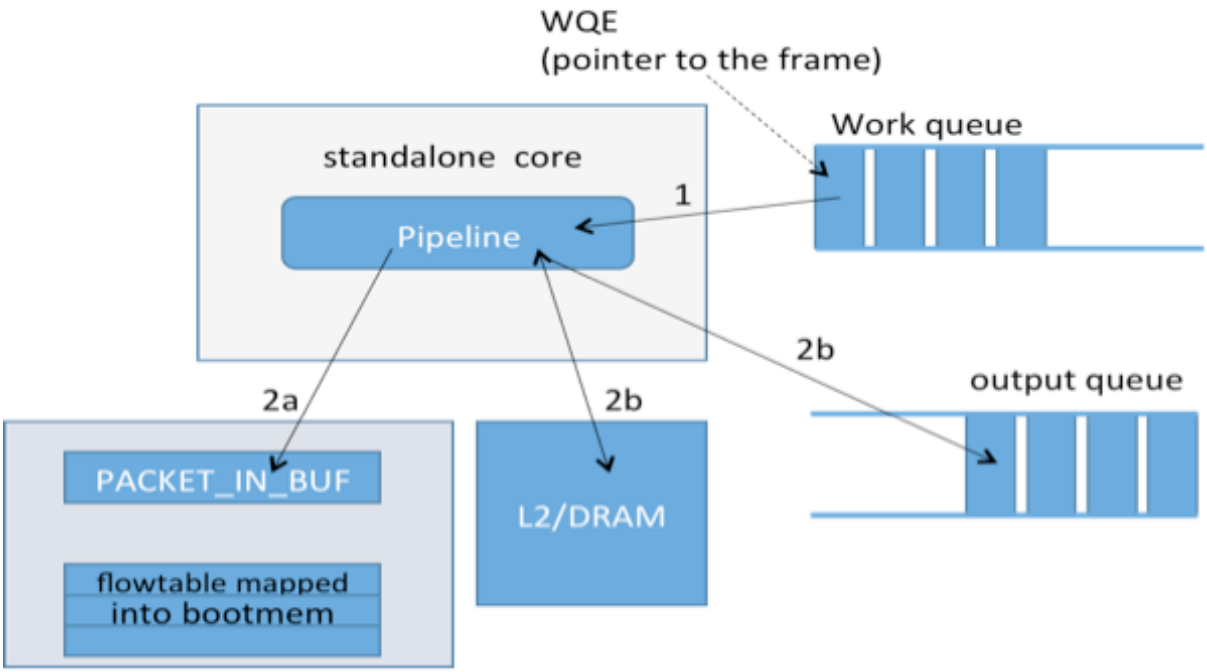

Figure 3.7 Work flow in and out the SE cores

3.2.3 Interaction with HAL

The HAL of the OCTEON provides access to low-level packet manipulation. The actual flow of a packet is rather complex and explained in detail in [9], but for the sake of further understanding of the limitations of the current implementation it is briefly recapitulated below.

Being a network processor, the unit of work for the OCTEON is a packet. The cores retrieve work from a *work queue (WQ)*, and the order of the incoming packets is not inherently maintained within the processor, as multiple cores may fetch packets from the WQ which may requires different processing effort. The WQ does however not contain entire frames, but so-called Work Queue Entries (WQE), which represents a pointer to the packet and a first evaluation of its header. In addition, the WQE carries the first 96 bytes of the packet header. This is comparable, though not identical, to OpenFlow, which matches on the first 128 bytes.

Packets are retrieved from the input queue by the Packet Input Processor (PIP) and Input Packet Data Unit (IPD). After initial checksum calculation of the entire L2 frame as well as the IP and TCP headers the PIP classifies the frame and sets the QoS values and group values (i.e. determines the group of cores that are eligible for packet processing. In our case, the group of cores that runs the Linux is not eligible to retrieve any packets from the WQ, hence the PIP sets the appropriate number 1.

After this initial processing of the packet header, the PIP obtains an empty WQE from a free buffer pool, writes the actual packet into the Level2 cache (DRAM), sets the WQE pointer to point to the memory address and writes out the WQE into the SSO (Schedule/Synchronization and Order Unit). This unit is responsible for the sequence maintenance of the packets. It does so by constructing on-the-fly virtual queues for packets belonging to a “flow”, where the flow is determined by the 5-tuple (IP Proto, Src/Dst, TCP/UDP port numbers).

Another shortcoming of the packet processing hardware units is the fact that the PIP only matches on the IP 5-tuple instead of the more elaborate OpenFlow match. For this reason, our implementation does not make use of the WQE, but instead matches the frame again and fully inside the SE cores. This is clearly decreasing forwarding performance, but we decided for this ‘double’ parsing of the frame in order to be fully OF capable.

3.2.4 Interface for Detection of the Hardware Capabilities

The interface for the detection of the hardware capabilities is the one implemented in the OpenFlow endpoint. This means for the currently installed version 1.2 of OpenFlow [3] that the *table_stats_reply* returns a bitmask with the actions supported by the pipeline.

3.2.5 Interface for Hardware Configuration

There are two ways of configuring the hardware: First, during boot process, the number of cores that execute Linux and the number of cores that run in SE mode are determined. A image name given as an argument to the *bootoct* command lets the bootloader install a certain pipeline image (an ELF executable) that is known to expose certain capabilities (because they have been programmed into). Additionally, a bitmask defines how many (and which) cores the images are installed to. Further hardware configuration takes place within the boundaries given by the pipeline implementation.

3.2.6 Interface for Statistic Collection

The usual (i.e. OpenFlow) commands *flow_stats_request/flow_stats_reply* and *table_stats_request/table_stats_reply* can be used to retrieve statistics of the flows processed in the OCTEON. Internally this is organized as shown in Figure 3.5. The SE cores update the statistics of the flow entries.

3.2.7 Limitations

The limitations of the current approach are quite obvious: OCTEON offers more hardware processing features than the current implementation is able to use. By circumventing blocks that could run pattern matching algorithms in hardware like the DFA (Deterministic Finite Automata) we gain easier implementation for the price of an only partial use of the features of OCTEON.

One opportunity is that the pipeline residing in the Linux core would be capable of exporting two or more tables, of which one could correspond to the limited parsing and matching capabilities of the OCTEON. Traffic that would use only Ethernet, IP and TCP would be directed into a work group (set of cores) with limited capabilities of the pipeline, but increased speed.

All in all the current experience with the Software Development Kit (SDK) of Cavium taught us that the HAL implementation and the limitation of shared memory access between cores made the implementation cumbersome. The very recent release 2.3 of the SDK appears to be a lot more stable and will be used for further development.

3.3 Hardware Specific Parts for NetFPGA Cards

3.3.1 Description of Physical Architecture

The physical architecture of the NetFPGA card has been described previously in details in deliverable D3.1 [2].

3.3.2 Description of Logical Architecture

The software for NetFPGA cards is composed of at least two main components. One on them is related to the hardware part, the second one for operating system hosting the card. The hardware part is programmed in VHDL and it is run in hardware FPGA chip. The second one works as a driver and it is run in software part of typical PC. They are connected physically via PCI bus and logically by mechanism of registers (describer later).

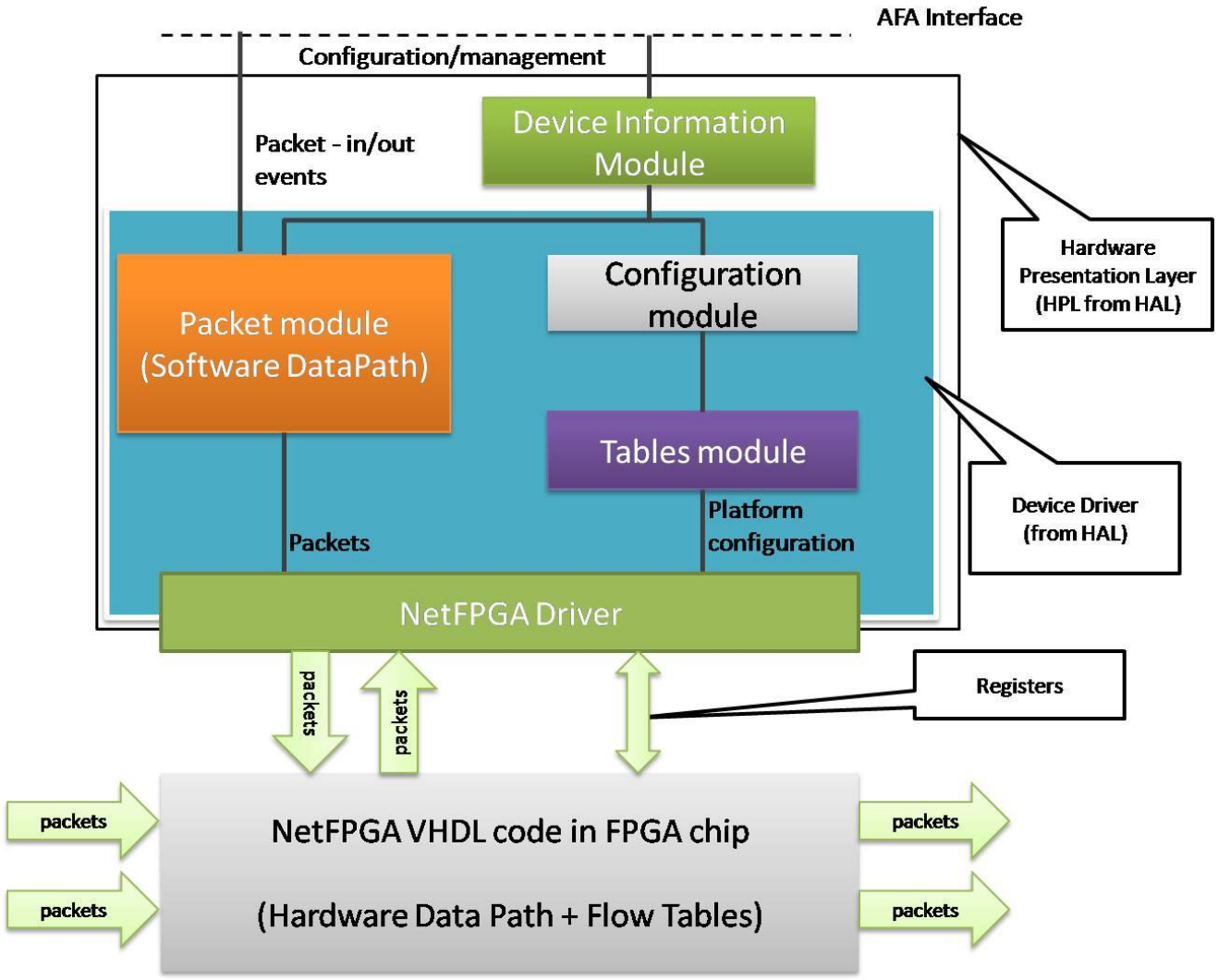

Figure 3.8 General architecture for HSP for NetFPGA

The software part of project for NetFPGA will contains following modules:

- **Device Information Module** represents the actual state of NetFPGA card, it offers information about active functionality and available resources.
- **Configuration Module** provides functionality for management of data path, reading and writing content in tables (tables of flows, counters).
- **Tables** stores information about active and past flows, counters, statistics and parameters.
- **Packet Module** serves packets that are not served by hardware part; these packets will be sent to/from OpenFlow controller.
- **NetFPGA Driver Module** is responsible for sending and receiving information to and from NetFPGA card.
- **NetFPGA VHDL Code** is the program compiled for specific FPGA hardware chip. It realizes all implemented functionality in hardware, it's very fast, but its functionality has to be implemented at very low level of abstraction. It's controlled by the NetFPGA driver.

In Figure 3.8 there is presented general architecture of system with the NetFPGA card. Detailed information about functionality of cards can be found in D3.1 [2]. The card is controlled by its driver which provides communication with the system and interfaces for modules of HAL.

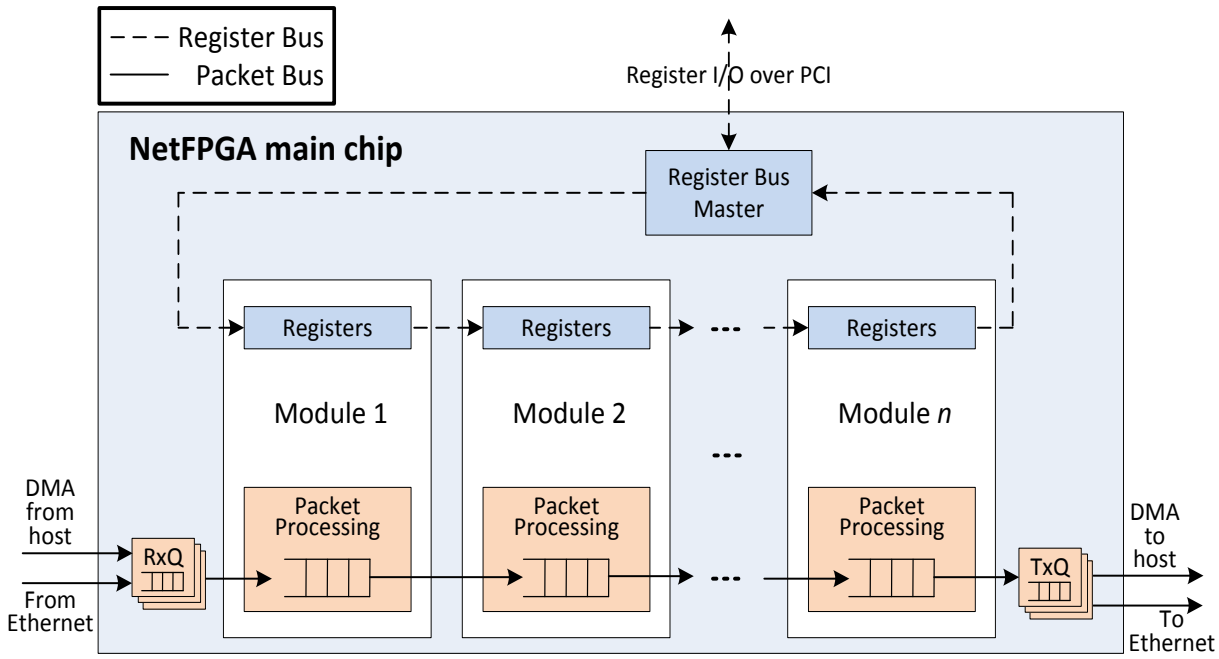

Figure 3.9 Two busses (for packets and control signals – registers) as a general pipeline for NetFPGA card

The general structure of hardware part of software for NetFPGA card is presented in the Figure 3.9. Incoming frames are placed in receiving queues (RxQ), next they are processed by modules arranged in serial pipeline. After processing the frames are placed in output queues (TxQ) and sent to their next hop or destination. The control information is sent to the card (and particular modules) via registers. The registers are implemented in both (hardware and software) parts of

system. They have logical connections and during the code synthesis specific memory addresses in the host are mapped to the appropriated variables in hardware part. This is the mechanism for control hardware part from software part.

Typical usage of NetFPGA card assumes that frames/packet enter the system via physical port and also leave it through another physical port. It is possible to send packets direct to the software part of system by directing it to proper output queues. Sometimes, the packet should be sent to the OpenFlow controller, hence, the NetFPGA driver will also be able to access the network interfaces which map to the physical ports.

The implementation of OpenFlow controlled switch requires the implementation of all tables and control structures in hardware part of system. It will be realized on the FPGA chip on NetFPGA card. General structure of such realization is presented in the Figure 3.10. It will contain flow tables and modules for OpenFlow actions. All these structures will have its representation in software part of the driver.

The NetFPGA driver contains two parts. One of them serves the communication between card and operating system, the second one is project dependent. In this second part all functionalities as tables for counters, flows, data path representation, control structures and functionality of interface to xDPd/ROFL will be implemented. Detailed information about particular modules and interfaces is given by next subsection.

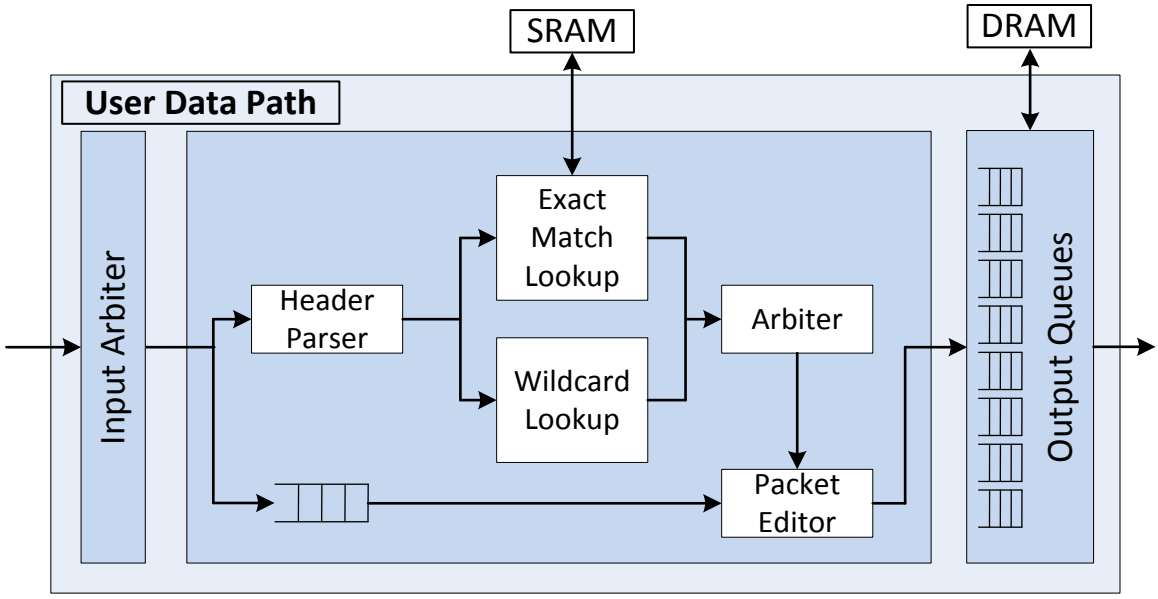

Figure 3.10 The Pipeline of OpenFlow implementation in NetFPGA card

3.3.3 NetFPGA VHDL Code

The executive module for whole system is the program running on the FPGA chip. It's functionality which can operate at line rate. With a proper configuration of internal modules it can achieve 100% of throughput of physical interfaces. The structure of code for hardware part of system suggested by NetFPGA provider is serial pipeline of modules (see Figure 3.9). This is the starting point to add new functionality. To implement OpenFlow enabled switches with ALIEN functionality, existing modules will be reused (i.e.: input queues, arbiters, tables and example of OpenFlow implementation) and some additional modules to add functionalities will be developed. The communication between program run in hardware chip and the rest of the system is realized by NetFPGA driver.

3.3.4 NetFPGA Driver

The NetFPGA card can be visible in the operating system (typicaly Linux, suggested version is Fedora 14 with kernel 2.6.35.14-106, but higher versions are also possible to be used). The driver code provided by NetFPGA cards developer allows configuring interfaces of this card as a typical network interfaces and implements mechanism of registers. With registers it is possible to write and read information to and from program running in the FPGA chip. This mechanism is used for sending control information. The data path for packet should be integrated with network interfaces.

3.3.5 Device Information Module

This module represents NetFPGA card to higher layers of HAL model. It connects NetFPGA HSP software with xDPd/ROFL system. It contains and offers the information about actual system parameters and available capabilities.

3.3.6 Configuration Module

This module contains all implemented management functionality. It implements functionality described by AFA interface i.e. functions from Datapath Management, Appendix A.2, Data Path Configuration - Appendix A.3). These functions are committed by receiving commands to set new actions. For every action type, proper parameters are sent to the tables' module and NetFPGA hardware module via NetFPGA driver.

3.3.7 Tables

The Module of Tables stores information about active and past flows, theirs parameters and counters assigned to measured values. This module has logical connection with its representation in hardware module run in FPGA chip. Basing on its information Forwarding Table is created and it is used for populating CAM memory. With this functionality control functionality is realized in hardware. It is also possible to measure some parameters and gather statistics. This module is dedicated for storing definition of measured parameters and its values. Proper functionality for updating and reporting will be implemented in this module.

3.3.8 Packet Module

Ideally, all packets will be treated in hardware (by NetFPGA hardware module). But some OpenFlow functionalities may require sending packet to the controller. In this case such a packet will be sent via control interfaces to the software part and consequently to the Controller. All operations on packet in software part will be realized by packet module. This module will also store packets which will be served by software data path.

3.3.9 Virtualization and Resource Isolation

In NetFPGA cards traffic from all ports go through the same pipeline (see Figure 3.9). It is possible to prepare multiple modules, however, it has to be done during the design stage. General idea assumes one common pipeline and there is no possibility to use multiple modules. Hence, all virtualization has to be done only logically, by assignment throughput for particular VLANs or other virtualized resources. It can be implemented in the control plane, through mechanisms like Flowvisor or advanced configuration. Since there is no possible to divide resources into independent sets, there is no resource isolation on physical layer. The main pipeline can process incoming frames with speed which is a consequence of word width and clock frequency (assuming there is no extra delay during processing), this mechanism is known as an internal speed-up. In case of NetFPGA cards, all traffic incoming to physical ports can be served in line-rate. So, if there is no additional delay caused by time-consuming processing, the resource isolation is native. It is not necessary to

implement advance admission control, because all incoming traffic can be served. But this implies a careful deign and coding to cope with time dependencies between modules in the main pipeline.

3.3.10 Limitations

Assumed implementation of OpenFlow is based on reference implementation (available at openflowswith.org). Due to hardware realization of main program, all the operation will be implemented with limited software support. Assumed implementation will be enabling to serve about 32 000 exact-match flow entries and it is capable of running at line-rate across all ports [10]. Each flow can be created basing on numerous parameters, such as source interface, destination and source MAC, protocol type, tagging parameters (VLAN id or priority), layer 3 addresses (destination and source IP), higher layer parameters, such as UDP/TCP port, type of service, type of protocols. To realize matching in hardware, typical TCAM will be used. It can have different length of word. This module can be designed and implemented according to wide range of request.

The control software will be implemented in the hosting PC. Hence, its performance will depend on processor speed, amount of RAM and machine load, although the main limitation will be the throughput of bus, which connect the NetFPGA card and operating system. It is PCI with 32 bit and 66 MHz, hence, the total throughput (for control and data plane) is about 1 Gbps. This parameter should be taken into account during planning operations on interface between driver and card.

It is hard to implement sophisticated multilevel queuing mechanisms in hardware for wide range of queues length, hence the advanced queuing mechanisms of OpenFlow will not be implemented. Only basic mechanisms will be deployed, it is probable that software support will be used for them.

3.4 Hardware Specific Parts for L0 Switch

3.4.1 Architecture of Specific Parts

As it has been explained previously in deliverable D3.1 [2], the ADVA DWDM ROADM Layer-0 switch (it can be seen in Figure 3.11) falls into circuit switched platform category. In general, circuit switch platforms have separated data and control plane. i.e. the control plane resides in another entity e.g. SOC on a board and controls the data plane entities without any visibility on data that passes through the date plane. ADAVA DWDM ROADM devices are no exception and have similar architecture.

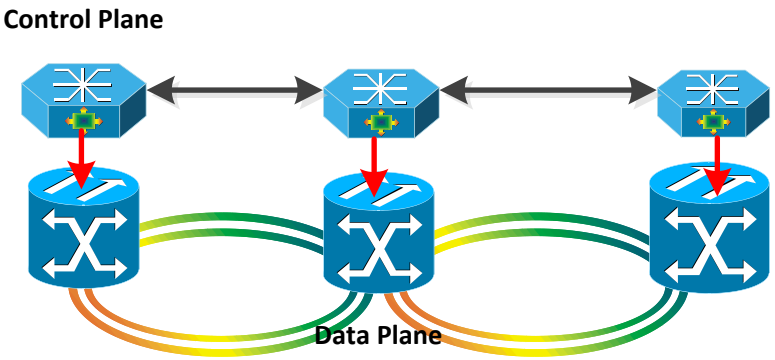

Figure 3.11 Multiple connected Layer-0 switch

ADVA Company has developed Management Information Base (MIB) for their hardware platforms which help the development of programs using SNMP protocol to manage and control ADVA devices. It's important to note that the MIBs are proprietary and for internal use for ADVA Co.

ADVA control plane program cannot be reprogrammed or flashed with a new firmware by third-party user on the device. Therefore, the only option to interact with the device is via control plane provided by the vendor. However, by using the SNMP MIBs library provided by ADVA Company, a third party user can develop a program exploiting the SNMP library to manage and control the device from another machine e.g. a virtual machine (see Figure 3.12).

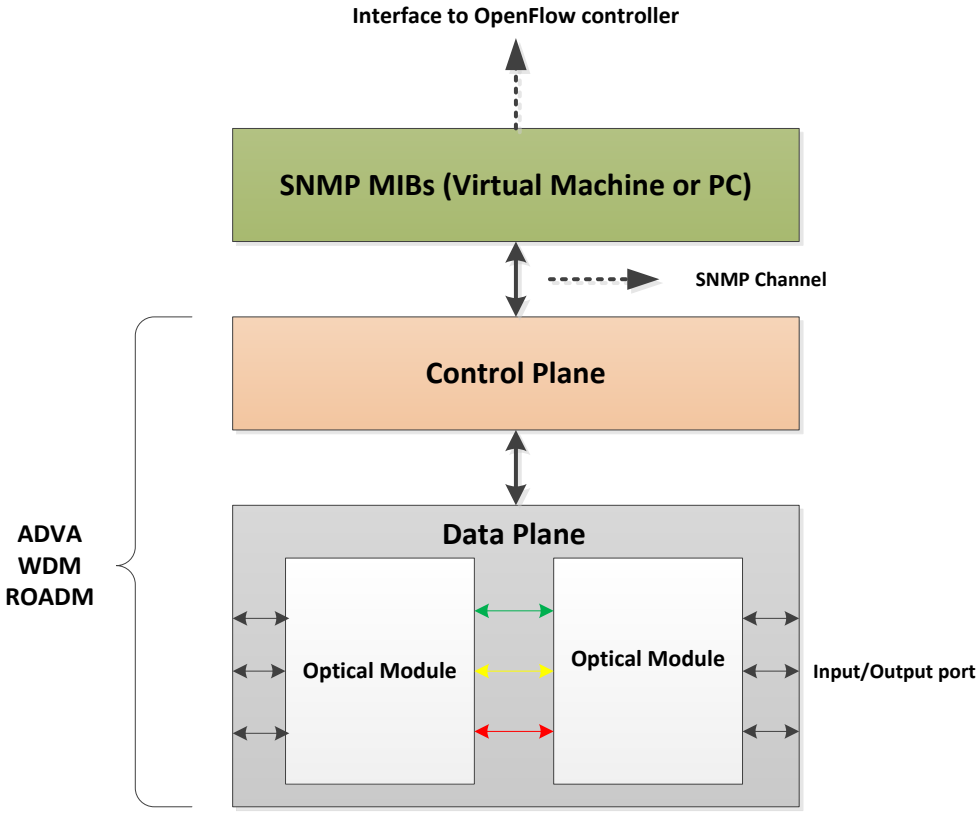

Figure 3.12 Logical architecture of controlling ADVA WDM ROADM by SNMP MIBs

3.4.2 Interaction with Device

Considering the device's architecture, technology and interfaces available to interact with it, an OpenFlow-enabled ADVA ROADM switch has to be logical switch which replicates the cross-connections on the actual device and is able to add or remove those cross-connections on the device. To build such logical switch, the OpenFlow API has to be extended according to OpenFlow addendum 0.3 [10] specifications to support circuit switch functionality.

The interaction i.e. information and functions needed to manage and control the ADVA layer-0 switch via SNMP protocol is defined in "resource.h" file in the SNMP library. That includes functions for hardware module discovery, port discovery, port status etc. Therefore, a third-party user by calling the header file in his program can develop functions to get access to the device status, query information or configure a modules and ports on the device. All the interactions between the device and the user's program happen over an SNMP channel i.e., the user's program has to implement SNMP trap and send and receive SNMP messages over the SNMP channel in order to be able to use SNMP library. The SNMP channel is a TCP connection which contains SNMP messages in its payload.

3.4.3 Interaction with HAL

As the SNMP library has already been developed for ADVA ROADM Layer-0 switch, according to the draft version of HAL [11], the SNMP library can be seen as device driver library in Hardware Presentation Layer (HPL) in the HAL architecture. This assumption is based on the fact that the SNMP library is the fundamental component for developing device driver and other components directly interact with the device for ADVA ROADM Layer-0 switch. By making this assumption, the

HAL could interact with the device using this library. To be specific, an SNMP channel has to be established between HAL and the control plane on the device.. The translator in the HAL acts as a glue logic between the OpenFlow messages from the controller and SNMP messages from the device. That means it translates from/to Network Element's (NE) syntax (ADVA's syntax) to/from the OpenFlow syntax (see Figure 3.13).

The Orchestrator module is not necessary for ADVA because the hardware components used to be represented as an OpenFlow switch are not acting separately and therefore no orchestration is needed.

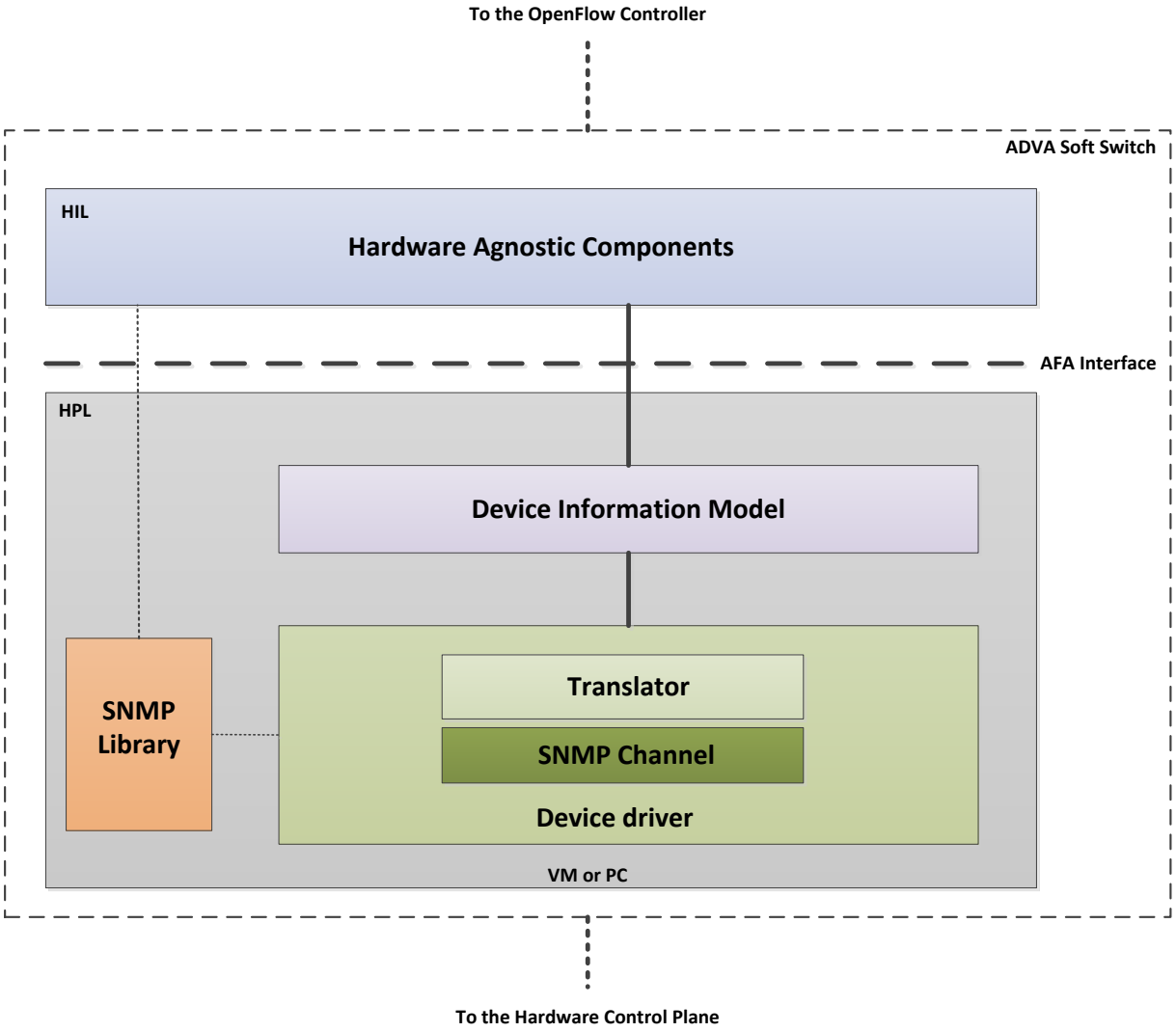

Figure 3.13 ADVA OpenFlow logical soft switch

3.4.4 Device Interfaces

Following the instructions provided by ADVA, to interact with the device through a third party program, an SNMP channel has to be established between the third party software and the control plane. The SNMP channel provides a

platform to implement SNMP trap listener and SNMP trap handler. The SNMP trap listener and handler will send and receive SNMP messages and act upon them based on the functions defined in the third party program.

3.4.5 Interface for Detection of the Hardware Capabilities

The interface is as usual defined in the "resources.h" header file. This file contains data types to query about the modules and their ports and addresses.

The module data type supports the following modules:

- Optical Lightpath,
- Transponder,
- Filter,
- Cross –Connect,
- Amplifier.

The port data type supports the following ports on modules:

- N and C ports on filter module,
- N and C port on transponder module.

N and C ports are network and circuit ports, respectively. Network port is for device discovery in a network topology and circuit port is the lightpath port on the device. Each module has a unique address and a number of ports on it and therefore each port is accessible by its unique address.

The DMI components build a repository of available modules and ports and their features. The repository gets updated upon any feature or configuration changes. The repository feeds all the information needed to build and OpenFlow endpoint as explained in the HAL white paper.

3.4.6 Interface for Hardware Configuration

The general instruction of resource library has been built based on the ADVA hardware platform. That includes of different data types for different part of the hardware platform. Each data type has different objects according to what it is defined for.

As said before, the "resource.h" file provides data types for all module types on the hardware. The data types include key features of the modules such as address, ports, network ports etc., which provide a method to control device's module. The same method is applied for module's ports which have their own features. ADVA's notation of port on modules is different to OpenFlow ports so before configuring or any other operation on ports they need to be represented in OpenFlow format according to OpenFlow Addendum [10] for circuit switching.

Considering the OpenFlow addendum for circuit switched platforms, the flow table implemented for ADVA ROADM switch has to have the following features:

- **Circuit key** – this is a key for each flow in flowtable for flow (circuit) identification,
- **Input port** – the port that light enters to,
- **Output port** – the port that light exits from,

- **Lambda** – the lambda (frequency) that light needs to switch to. That means the frequency of incoming light is changed according to Lambda value when it exits from the output port.
- As the module or port abstraction is already done in SNMP library, the developer needs to use them properly for initialization and, after that, configuration of the device.

3.4.7 Interface for Statistic Collection

Despite the packet switched platforms, in circuit switched platforms because of no visibility on data, the volume of traffic cannot be measured but instead, power equalization needs to be done on the device. The power equalization is necessary in optical switches when a created lightpath has to maintain a certain optical power throughput the path. This happens when a wavelength gets distorted entering to an input port and exits from an output port. Therefore, the power level has to be calculated and set on optical module to compensate the distortion and power loss.

A data type is provided for ports holding features such as:

- **Address** – this is port's unique address,
- **Connection** – shows connections to other ports,
- **Module** – identifies the parent module,
- **Features** – this is for port's frequency.

By using the information that port data type provides, the developer could be enable to implement power equalization function to check the power constraints required by the applications in controller.

3.4.8 Limitations

Regarding the fact that circuit switched platforms have completely different logic for dealing with the data they get on their ingress/egress ports and also the different medium (fiber, wavelength) used for transmitting data, most of the functions and actions that are executed in a generic OpenFlow switch could not be applied on circuit switched platforms. This means, the extended OpenFlow controller has to distinguish the device type (packet switch or circuit switch) after receiving feature message reply from the devices and change its behavior prior to sending any other messages to the device. By extending the OpenFlow controller to support circuit switched platform, some actions and functionalities will not be supported on the platform like packet switched platforms. For example, considering that control plane on circuit switched platform has no visibility on data in control plane and cannot react and reconfigure data plane based on incoming data, a flow modification has to be defined and executed prior of receiving the data.

3.5 Hardware Specific Parts for Dell/Force10 Switch

The Dell Force10 / Split Data Plane platform is constituted of two main physical parts: a Top of the Rack Ethernet Switch PowerConnect 7024 (PC7014) and a Split Data Plane Module (SDPM). The PC7024 is a 24 port 1Gbit/s with 2 back panel extension slot.

The SPDM is a pluggable module, which is inserted in one of the extension slot of the PC7024. The SDPM provides one administration 1Gbit/s port and a 10Gbit/s XAUI (X as 10 and Attachment Unit Interface). The XAUI interface is directly connected to the PC7014 Forwarding Broadcom ASIC chip BCM56630. The SDP module has 1GB DDR memory and a microSD flash of 16GB. The Network Processor Unit on the SDP module is a Cavium OCTEON CN5230 with 4 core MIPS CPU running Debian 2.6 with the Cavium SDK 2.3.0.427. The OCTEON CN5230 cores can run in two different mode: Linux user mode (SE-UM) and Standalone (SE-S). The standalone mode allow Executable and Linkable Format binary to run directly without operating system and then dramatically performance improve The 4 cores can be configured to run at the same time a mix of those two modes, for example one core in SE-E and the three others on SMP (Linux Synchronous Multi-Processing) SE-UM mode. Most Cavium specific comments, limitations and approaches are shared with section 3.2 ATCA, and are thus repeated instead of just pointed out for the sake of clarity. Differences come from chip variations and architecture of the whole system.

3.5.1 Architecture of Specific Parts

The Pipeline platform interface will be using xDPd/ROFL libraries inside the SDP Module this pipeline module will have full access to incoming network packets and will be able to control packet processing within the SDP module. They will contain the Control and Management Module (CMM) and the specific management API AFA controlling the specific hardware forwarding pipeline. The Network Processor provide hardware accelerate units to support the pipeline.

To take advantages of the SDPM NPU OCTEON CN5230 Multi Core MIPS hardware architecture and its acceleration hardware units (encryption, compression, TCP throughput), one or two MIPS cores can run as described in the XDPD design the Control and Management Module on top of Linux SE-UM. These two cores will be also responsible of the software initialization of the hardware pipeline.

The two or three remaining core of the OCTEON CN5230 are running in standalone mode SE-S and will be dedicated for the pipeline.

A Bootmen memory shared is necessary between the Linux SE-UM cores and the Standalone SE-S, here is the three different section shared :

- A state machine that is starting/stopping the standalone (SE) cores. The Linux core sets a variable that indicates that standalone cores have to hold operation until memory is initialized or entries in the FlowTable are changed. Afterwards it resets the status to “go”.
- A buffer that lets SE cores store packets that do not match. These frames will be red out from Linux and transformed into *packet_in* messages to be sent upwards.
- The actual FlowTable. This is written by the Linux core and read (only) by the SE cores.

Actual packet flow is going through the SE cores exclusively, except in the case when there is no match in the FlowTable.

The OCTEON is an Network Processor Unit and its work unit is a packet. The cores get their work from work queue (WQ) and the packets are not store entirely in the queue, only a pointer to the packet a Work Queue Entry (WQE). The WQE also include a part of the packet header, this could help to do pre-treatment on the packet.

The packet are move from the input queue by the PIP Packet Input Processor and Input Packet Data Unit IPD and Layer 2 and Layer 3 checksums are computed for verification as QoS and classification tasks.

After this initial processing of the packet header, the PIP obtains an empty WQE from a free buffer pool, writes the actual packet into the Level2 cache (DRAM), sets the WQE pointer to point to the memory address and writes out the WQE into the SSO (Schedule/Synchronization and Order Unit). This unit is responsible for the sequence maintenance of the packets. It does so by constructing on-the-fly virtual queues for packets belonging to a "flow", where the flow is determined by the 5-tuple [IP Proto, Src/Dst, TCP/UDP port numbers].

Another shortcoming of the packet processing hardware units is the fact that the PIP only matches on the IP 5-tuple instead of the more elaborate OpenFlow match. For this reason, our implementation does not make use of the WQE, but instead matches the frame again and fully inside the SE cores. This is clearly decreasing forwarding performance, but we decided for this 'double' parsing of the frame in order to be fully OF capable.

3.5.2 Interaction with Device

The device will be accessible using the OpenFlow protocol. The OpenFlow controller is talking directly with the OpenFlow endpoint included in the ROFL Control Management Module in the Linux core.

All the communication with the acceleration hardware blocks running the pipeline and the CMM is based on the API from the Octeon SDK.

The standalone core read the Work Queue Entry in the work queue (1). The header is matched against the FlowTable, and if no entry is found (2a), the WQE is entered into the PACKET_IN_BUF, where the Linux core can retrieve it. If a matching entry is found (2b), the frame is processed from and copied to L2 cache. The resulting WQE is then put into one of the output queues. It can be seen in Figure 3.14.

3.5.3 Interaction with HAL

The OCTEON SDK provides lot of functionalities and please refers to the SDK OCTEON manual for more detail. The OCTEON SDK has its own HAL functions below are the basic units and intermediates.

The Simple Executive API is used to access the hardware units:

- Basic units:
  - FPA: The Free Pool Allocator Unit manages up to 8 pools of free buffers which may be requested by other hardware units. The most common uses of the buffers are for Packet Data Buffers and Work Queue Entry Buffers.
  - IPD: The Input Packet Data Unit works together with the PIP to allocate needed buffers, and process the packet data. The IPD fills in the Work Queue Entry Buffer and the Packet Data Buffer. It then

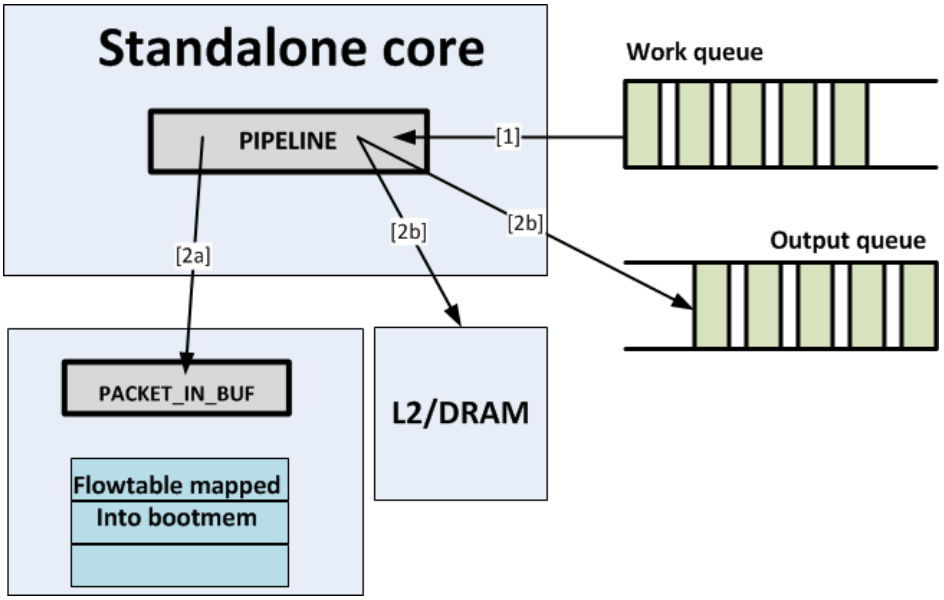

Figure 3.14 The communication with the acceleration hardware blocks running the pipeline and the CMM

submits the Work Queue Entry Buffer to the SSO's QoS Input Queues. Requires FPA Packet Input Buffers and Work Queue Entry Buffers.

- PIP: The Packet Input Processing Unit receives the packet data from the Packet Interfaces, and perform basic error checking on the data.
- SSO: The Schedule/Synchronization/Order Unit maintains the QoS Input Queues, and manages scheduling work to cores. It also maintains the work order, and provides the support needed for packet-linked atomic locking.
- PKO: The PKO manages packet output. Cores submit command words to its Output Queues. These command words include a pointer to the packet data to be transmitted. The cores then "ring" a door bell to notify the PKO how many command words were written to the Output Queue. The PKO DMAs the packet data from the Packet Data Buffer to its internal memory, and sends it from there to the Packet Interfaces. This operation requires an FPA pool of Command Buffers.

- Intermediate units:
  - FAU: Fetch and Add Unit - a 2 KB register file supporting read, write, atomic fetch and add, and atomic update operations. This unit can be accessed from both the cores and the PKO. The cores use the FAU for general synchronization purposes. Packet-management Acceleration: Packet receive/transmit is automated by software-configurable.
  - TIM: Timers - requires FPA timer pool.

Convenient access to these hardware units is provided by Simple Executive function calls and macros.

3.5.4 Interface for Detection of the Hardware Capabilities

The interface for the detection of the hardware capabilities is the one implemented in the OpenFlow endpoint. This means for the currently installed version 1.2 of OpenFlow [3] that the *table_stats_reply* returns a bitmask with the actions supported by the pipeline.

3.5.5 Interface for Hardware Configuration

At the boot process the cores need to be allocated using the specific bitmask for configuration how they will be running in SE-S standalone mode or Linux SE-UM. The right images need to be configured using the command *bootoct* for the SE-S and SE-UM cores and additional hardware configuration is done during the boot process.

3.5.6 Interface for Statistic Collection

The usual OpenFlow commands *flow_stats_request/reply* and *table_stats_request/reply* can be used to retrieve statistics of the flows processed in the OCTEON.

3.5.7 Limitations

The SDP Module has a single 10Gbits / XAUI connection to the OCTEON it limits the way interacting with the SDPM because we need to get the flow in and out of the same physical interface.

The PC7024 can run an OpenFlow implementation on the Broadcom chipset BCM56630, but it is limited and today is not that stable as expected.

We can manipulate the 'Flow' using ACL policy base routing or forwarding on the switch running in a not OpenFlow mode but we are limited with ACL scope and the number of ACL entries as the way to configure them.

We will not take all the advantages of the OCTEON hardware Network Processor and the OCTEON SDK still need to more develop.

The OCTEON we have on the SDPM is not the most recent and it is not getting lot of software improvement from Cavium.

3.6 Hardware Specific Parts for GEPON

The GEPON (Gigabit Ethernet Passive Optical Network) at UCL consists of several different physical components. The whole GEPON system consists of the OLT, a passive optical splitter and several ONU. The OLT is the "intelligent" part of the system and in the normal deployment is the part which is Internet facing. It is a point-to-multipoint device and in normal deployment is connected to a splitter which multiplexes the signal to the ONU which are typically situated in end user premises. The exact product numbers and detail on the device are given in Section 4 of ALIEN project deliverable D3.1 [2].

The GEPON is controlled by a proprietary chip which cannot be modified or flashed with new code. Commands can be sent to modify GEPON behavior including some limited packet matching and rewriting capability. The OpenFlow solution for the GEPON must, therefore, by necessity, involve at least one helper box.

The ONU are meant to be low-cost end-user devices deployed in user premises. A solution which enables OpenFlow on the GEPON but which has boxes associated with each ONU would not realistically be deployable without a redesign of the ONU itself or an increase to the price of the ONU deployment (these are low cost devices). In this case, therefore, we create a solution which requires fewer end-user boxes but at the expense of "reserving" a number of VLAN tags which cannot be used by the OpenFlow enabled GEPON. The architecture described here maps the GEPON to a single physically distributed OpenFlow switch. We will refer to this in this section as the virtual GEPON OpenFlow switch.

3.6.1 Architecture of OpenFlow GEPON

The GEPON consists of several devices. The OLT is the "head" end of the device which connects to the wider network and eventually the internet. It contains an Ethernet port and a management port. It feeds to an optical splitter (not pictured) which in turn sends data to the ONU each of which contains an Ethernet port. Between OLT and ONU transmission is passic optical and the encoding is IEEE 802.3ah which uses Logical Layer ID (LLID) tags to identify traffic destined for each ONU (underneath the system is one of time division multiplexing as explained in a previous deliverable). The OLT and ONU present externally as standard 802.3u GbE ports (and a management port at the OLT). Every packet sent between ONU, in fact, traverses the external internet facing switch before being returned to the OLT. All traffic between devices on the GEPON is, in fact, going via this switch. Together, the OLT, ONU and switch can be viewed as being a single switch distributed in space. Each ONU and the internet facing exit to the associated switch should be viewed as ports on a single conceptual switch. It is these pieces of hardware together (along with a helper box) which will become the virtual GEPON OpenFlow switch.

The Figure 3.15 shows the GEPON expanded by the addition of two extra boxes, the HAL (which will be described in more detail later) and an OpenFlow switch which sits on the data path. The OpenFlow Controller (OFC) connects to the HAL not the OLT or OpenFlow switch. These two boxes (the HAL and the OpenFlow switch) together should be sufficient for the GEPON to provide the majority of OpenFlow functionality. In our implementation it is currently planned that this switch runs xPDd (the eXtensible Data Path Daemon) [7] but any OpenFlow switch which can sustain the correct line rate will work. The contents of the HAL box are expanded later. The HAL has OpenFlow northbound and connects to the Openflow switch (also speaking OpenFlow) and to the OLT (using its proprietary command language) southbound.

The core concept is extremely simple. The OLT communicates with the ONU via LLID tags as previously stated. While the OLT is relatively limited in its matching capabilities it can match on VLAN tags and it can convert those matches to LLID tags which hence correspond to physical ONU. The idea, therefore, is to create a mapping between ports on the OpenFlow enabled GEPON and ONU (or OLT). This is achieved using temporary VLAN tags between the OpenFlow Switch

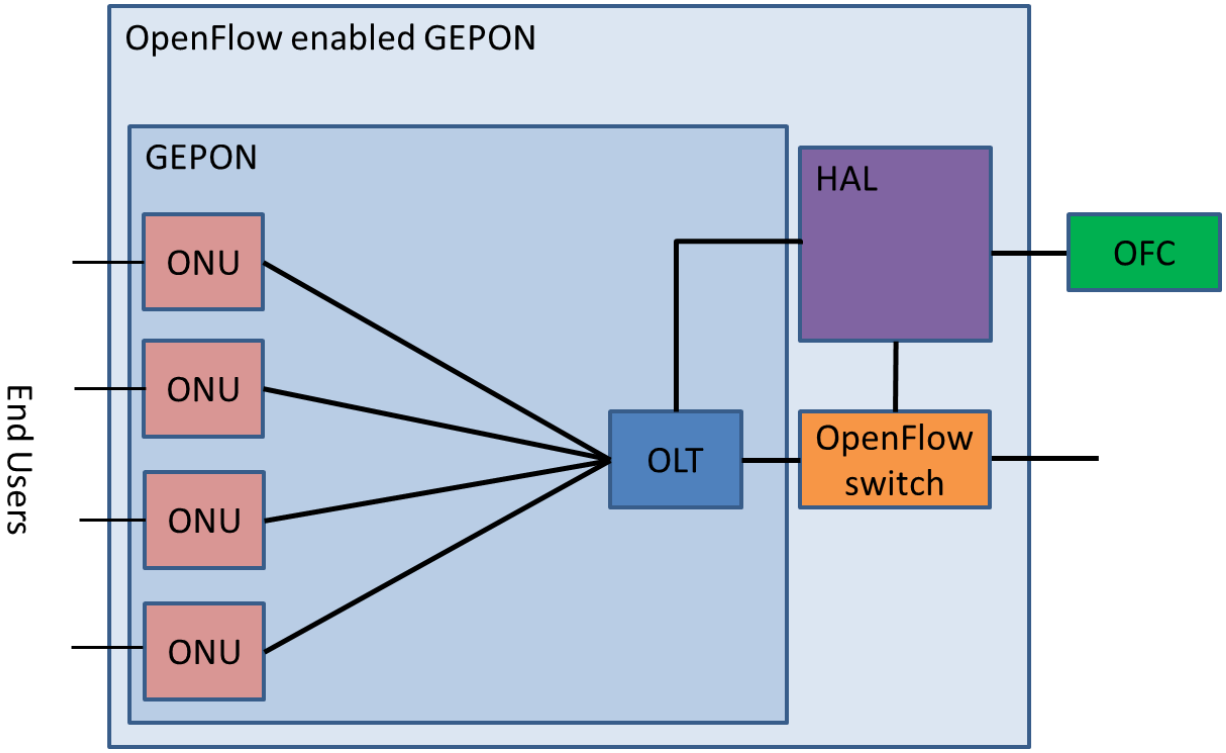

Figure 3.15 The GEPON expanded to provide an OpenFlow capable GEPON

and the OLT. The HAL creates a mapping between virtual ports and VLAN tags/LLID. These tags are inserted by the OpenFlow switch and removed by the OLT and replaced with LLID tags.

Consider traffic moving left to right from an ONU to the Internet. The LLID of the sending ONU is, at the OLT, replaced with the mapped VLAN tag. An OpenFlow rule which matches a source port is replaced on the OpenFlow switch with a rule which matches this VLAN tag. Every OpenFlow action in this direction (left to right on the OpenFlow switch) must be prefixed with an action which removes the VLAN tag and replaces the source port with the corresponded source port on the abstracted virtual switch. Similarly for traffic moving from right to left (from Internet to ONU), the xDPd's OpenFlow controller must cause it to operate as a learning switch to know which traffic has come from which ONU. Flow rules must be written from "output to port N" to instead tag with the appropriate VLAN tag and output to the port facing the OLT. For traffic moving from one ONU to another then a match rule to match against a specific incoming port would match on a given incoming VLAN and include an action to tag with a second VLAN.

3.6.2 Interaction with HAL

As expected the Hardware Independent Layer of the HAL takes incoming commands from the OpenFlow controller. These commands are interpreted into AFA and sent to the internal OpenFlow Controller. It can be seen in Figure 3.16.

New match/action rules and queries are passed on via AFA and must be translated to slightly modified rules or queries to the OpenFlow switch.

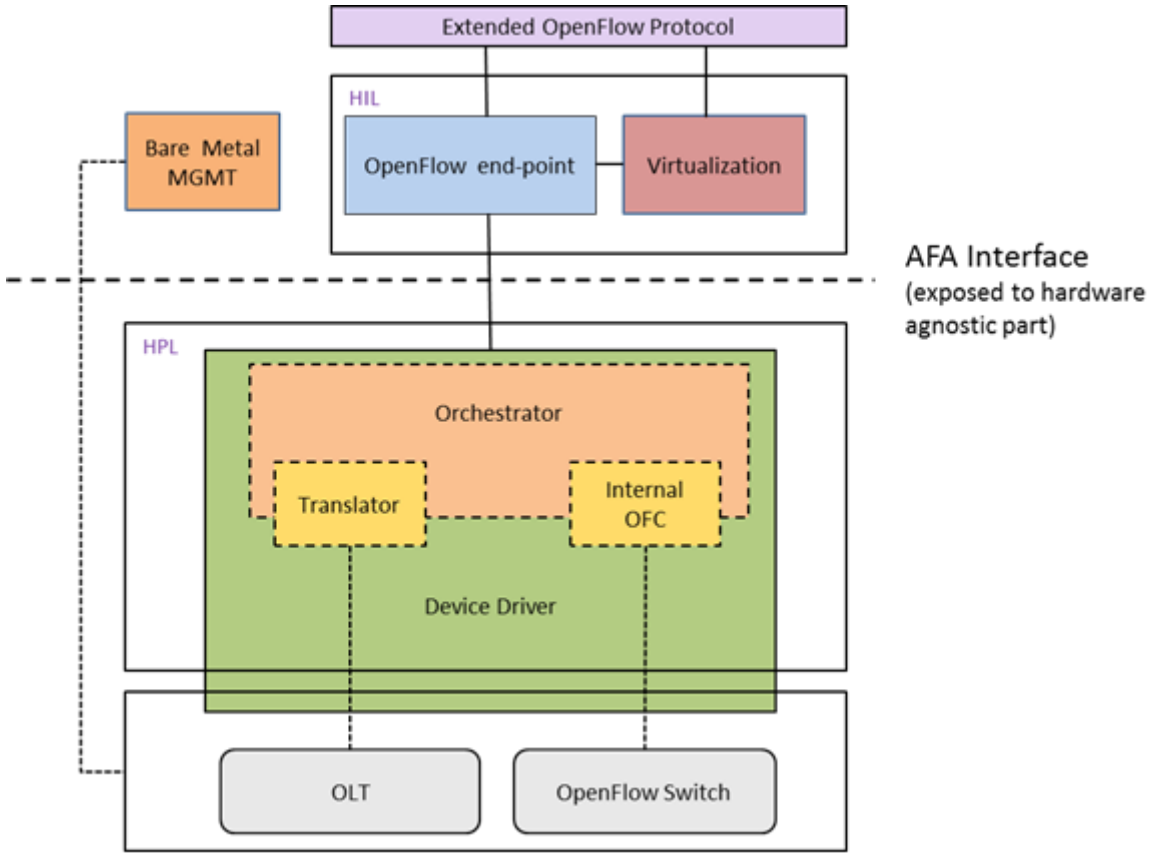

Figure 3.16 Expansion of the GEPON HAL

3.6.3 Interfaces to Device Driver

The hardware specific part of the GEPON proxy contains its own OpenFlow Controller which controls the xDPd as a learning switch. MAC addresses from each ONU are learned in order to be able to tag traffic appropriately with the correct VLAN tag. The device driver develops a model which associates each ONU with an LLID and with a VLAN tag. The mapping from ONU to LLID is done via queries to the OLT management console. If new ONU are connected this will be seen because packets with no VLAN tag will be seen at the xDPd from ports other than the internet facing port. The device driver has three external interfaces. It has a northbound interface via AFA to the Hardware Agnostic part and interfaces to the xDPd (via its inbuilt open flow controller) and to the OLT management console using the OLT's proprietary management language.

The OLT management console interface is used by the device specific part of the proxy to set up mappings between VLAN and LLID. When a new ONU is discovered by the OpenFlow Controller then it is assigned a virtual port number as part of the virtual GEPON OpenFlow switch and also a VLAN number. This is handled by the OFC within the Helper box which controls the xDPd. The OLT management console is used to assign the VLAN with the appropriate LLID.

3.6.4 Interface for Statistic Queries

There should be a one-to-one mapping between the rules installed on the OpenFlow switch and the rules the user installed on the "GEPON as switch". In this way then, statistics queries made by the external OpenFlow controller can be translated to slightly different queries which can be made by the internal OpenFlow controller.

3.6.5 Limitations

The GEPON is a closed-source, proprietary system and so cannot be programmed in the sense that some other devices in the ALIEN project can be. For this reasons, the system has to use auxiliary devices to help. By using up to 32 VLAN tags to implement a mechanism for mapping ONU to ports on a virtual distributed OpenFlow switch, those tags are not usable for other operations and reserved by the switch itself. It may also be necessary to configure one more VLAN tag for flood actions.

The architecture here should work for any GEPON or GEPON-like device which has the ability to translate a VLAN tag into an LLID and, hence, is much more general than working for just this specific GEPON. The exception is the interface required to send rules to the management console of the GEPON to associate a VLAN with an LLID. This will vary with each different device (and some devices may lack this capability). If this capability is present, however, then potentially any GEPON can be converted to an OpenFlow switch using this technique and with only minor changes to the function system suggested.

It is not yet clear how *output_all* and similar actions which output to many ports will be handled by the extended GEPON switch – it seems likely some GEPON mechanism will be present to do this and it is hoped this can also be mapped to a VLAN tag but this remains ongoing work at this time.

Actions like *enque*/*set-queue* present problems. The OLT has this facility but it is slow to set up new queues. It may be that traffic constraints at the OpenFlow switch can be set to mirror the queues in the OLT in such a way that queuing limits on the OpenFlow switch mirror queue requests on the OLT but this, again, remains an open question.

3.7 Hardware Specific Parts for DOCSIS Hardware

3.7.1 Architecture of Specific Part

3.7.2 Logical Architecture

The DOCSIS (Data Over Cable Service Interface Specification) platform at UPV/EHU consists of several components: the Cable Modem Termination System (CMTS), the Hybrid Fibre-Coaxial (HFC) distribution network and set of Cable Modems (CM). The CMTS is located at the premises of the provider and is the “intelligent” part of the system which controls the share medium (i.e. the HFC network). It is also considered the head-end of a point-to-multipoint system. The CMs are located at user’s premises and are considered the tail of such a system. More details about this platform can be found in Section 6 of ALIEN project deliverable D3.1 [2].

The CMTS is a closed-box (in this case, provided by Cisco) that cannot be modified or flashed with our own code. This means that we need to adapt our development to the standard interfaces provided by this vendor (e.g. the PCMM interface or configuration files). DOCSIS presents some limitations in order to fully implement the OpenFlow specification including some packet matching capabilities, the implementation of some actions and the generation of some events. Therefore, some helper boxes are needed to overcome these limitations.

With the aim of designing the most general solution for DOCSIS, which could be also adapted to other technologies, both the CMTS and the CMs have a helper box attached to them as shown in Figure 3.17. The Aggregator Helper is connected to the CMTS in the upstream interface towards the aggregation/core network, whereas the Residential Gateway (RG) Helper is connected to the CM in the downstream interface towards the LAN. Both helper boxes provide the additional features not supported by DOCSIS.

To this end, both the CTMS and CMs need to be configured as a bridge. Regarding the CM, it can be configured as router or as a bridge by changing one parameter in the configuration file. However, with regard to the CMTS, by default it is basically a router and additional configuration need to be done to be setup as a bridge. As a result, each CM can be mapped into a different VLAN tag at the upstream interface of the CMTS. Consequently, every packet that comes from the same CM is tagged with a specific VLAN identifier by configuration. The mapping between the CM and VLAN tag is needed by the DOCSIS proxy in order to properly configure the different resources.

The DOCIS proxy is responsible for orchestrating all the components and translating the OpenFlow protocol to the adequate interface towards each of these components. Both the aggregator and the RG helpers expose an OpenFlow interface, whereas the DOCSIS platform typically exposes the PCMM to dynamically configure the resources. Both the CMTS and the aggregator helper can be placed at the same location. In such a case, the DOCSIS proxy can directly interface both components. However, the RG helper will be always located remotely at the user’s premises, which means that a special connection is needed to remotely control this helper. Therefore, an extra service flow must be provided by the DOCSIS system to transport the OpenFlow protocol from the DOCSIS proxy to the RG helper.

In the end, the result is a set of components, i.e. the aggregator helper, the CMTS, the CMs and the RG helpers, acting as a single OpenFlow enabled device.

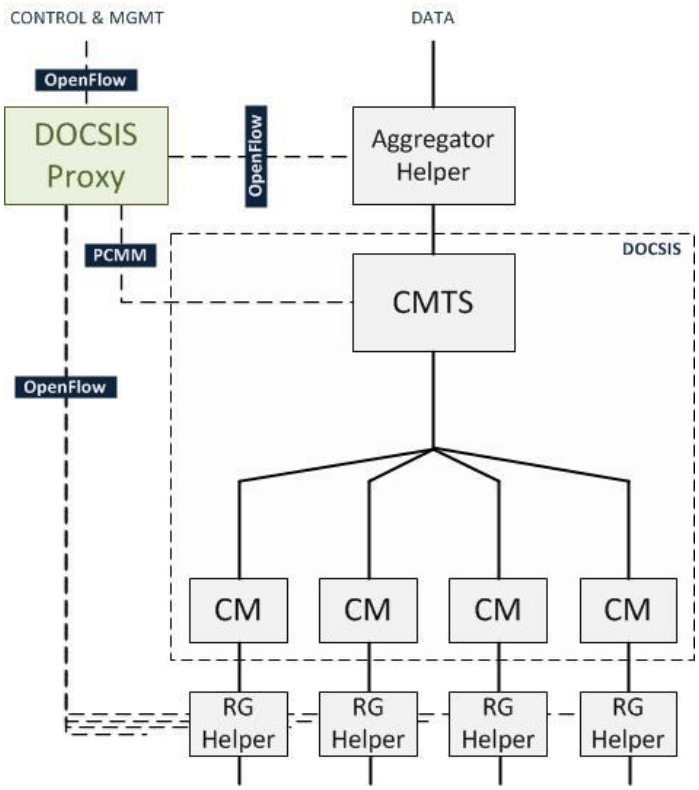

Figure 3.17 DOCSIS system architecture with an OpenFlow interface

3.7.3 DOCSIS Proxy

The DOCSIS proxy acts as an OpenFlow enabled device on behalf of the whole system towards the OpenFlow Controller. The northbound interface is the OpenFlow protocol and it has different southbound interfaces depending on the nature of the component below. Both the aggregator and the RG helpers expose an OpenFlow interface. However, the DOCSIS platform exposes other standardized interfaces defined by this technology, such as the PCMM interface. The Cisco CMTS does not expose any other interface or even a set of commands to interact with the DOCSIS resources.

Figure 3.18 shows the high level overview of the DOCSIS proxy and how the HAL has been extended with several platform specific modules. On the top, there is a Hardware Agnostic Part which is common for all the ALIEN hardware. The AFA interface, defined by ROFL [8], provides an abstract API to implement the hardware dependent part. The modules defined for this latter part are described below.

Device Information Model

This component stores the device state. It is a model of the whole system, which consists of several components, represented as a single OpenFlow device. It does not perform any processing.

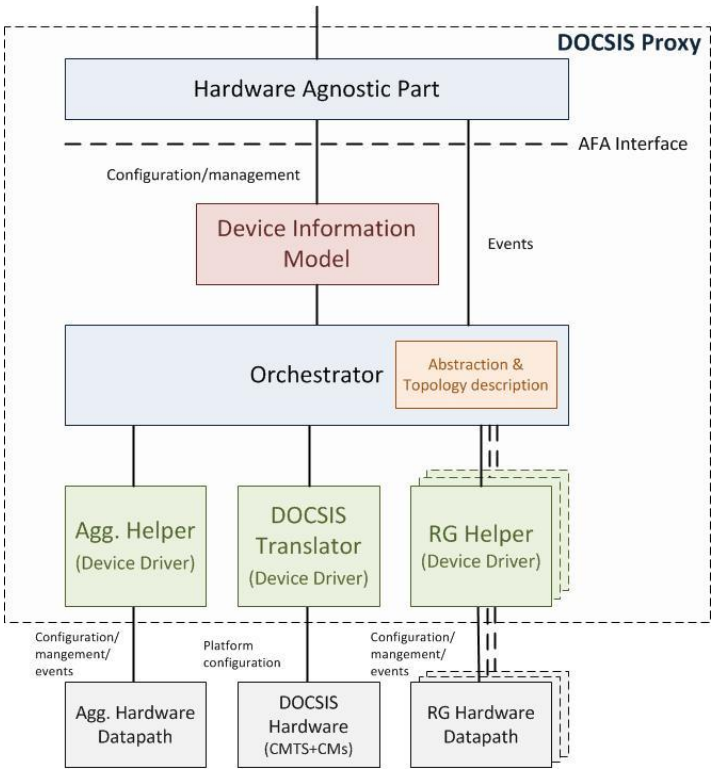

Figure 3.18 High level overview of Hardware Specific part for DOCSIS

Orchestrator

This component is responsible for exposing several devices as a single OpenFlow enabled device. Therefore, it must coordinate the helpers and the DOCSIS platform to simulate the behavior modeled by DIM. In a typical case, a data packet at least goes through one RG helper, one CM, the CMTS and the aggregator helper. This means that the four components should be properly configured to transport the packet from one point to the other. In this case, a single flow entry sent from the controller to the DOCSIS proxy is spitted into several flows: one OpenFlow flow entry at RG helper, a DOCSIS service flow between the CM and the CMTS, and one OpenFlow flow entry at the aggregator helper. It must be clarified that both the flow entry at the RG and the one at the aggregator are different and also different from the original flow entry sent to the DOCSIS proxy. For instance, the actions present at the original flow entry are applied at the outgoing helper, the RG or the aggregator depending on the direction of the packets.

A more complex scenario involves a packet sent from one CM (or RG) to another, because the packet must pass through the aggregator in its path to the destination CM/RG. This means that the packet goes from the source RG, the source CM, the CMTS and the aggregator in the upwards direction, and then the CMTS, the destination CM and the destination RG in the downwards direction. Finally, it involves seven elements.

It must be highlighted that the mapping between the original flows and the corresponding dissection depends on the case, which means that it must be done case by case.

Abstraction & Topology description

This module is located inside the orchestrator. It covers two important functionalities. On the one hand, it describes the topology and interconnection between the different elements. This information is basic for proper topology configuration of the elements to provide the services by their orchestration. This description should be provided somehow (e.g. file) since there is no mechanism to automatically discover the topology, since it involves DOCSIS to OpenFlow interfaces. On the other hand, the abstraction is needed to present a group of virtual ports (1 to N) from a set of physical ports from different physical devices. Some physical ports from these physical devices are used for internal connectivity without being exposed as virtual ports. Moreover, the physical identifiers are renamed in a common set, e.g. 1 to N.

Aggregator Helper – Device Driver

The device driver of the aggregator helper acts as the OpenFlow controller for the aggregator datapath. Once the orchestrator separates the original flow entry into several components, the flow entry generated for the aggregator helper must be installed. This device drive is in charge of installing this flow entry. The ROFL libraries are used to create this device driver.

DOCSIS Translator – Device Driver

The device driver for DOCSIS equipment needs to perform some kind of translation in order to transform the OpenFlow related information (i.e. stored in DIM) into DOCSIS aware configuration. This device driver depends completely on the interfaces exposed by the DOCSIS hardware, i.e. the CMTS from Cisco. This means that it could even depend on the model and vendor of the CMTS. Cisco CMTS is a closed-box with very limited access to the dynamic configuration of the DOCSIS resources. There is one standard interface to dynamically provision service flows in DOCSIS, the PCMM (PacketCable Multimedia) interface. Modifying the configuration files downloaded by the CMs could be another option to update the service flows; however, this implies the reset of the CMs. Anyway, the first thing to be defined is the mapping between the OpenFlow flow entries and the service flows defined by DOCSIS. Additionally, the service flows will add the enforcement of a configurable QoS to the flow entries.

RG Helper – Device Driver

The device driver of the RG helper also acts as the OpenFlow controller for the RG datapath located at user's premises. This device driver is in charge of installing the flow entry generated for the RG by the orchestrator as part of the original flow entry sent to the DOCSIS proxy. In this case, the ROFL libraries are also used to create this driver. There are several RG in the whole system, and consequently, there is one device driver per RG.

3.7.4 Device Interfaces

This section describes the interfaces exposed by the ALIEN hardware devices. In this case, the whole system consists of different resources: the aggregator helper, the DOCSIS equipment and the RG helper. Since both the aggregator and RG helpers expose a standard OpenFlow interface, they are not described in more detail in this section. Therefore, this section focuses on the DOCSIS elements: the CMTS and the CMs.

On the one hand, the CMs must be configured by the standard procedure defined by DOCSIS. First of all, the CM needs to get connectivity to the CMTS. Once this step is fulfilled, some additional services must be provided externally by a Provisioning System, such as DHCP, NTP and TFTP. In a nutshell, the CM obtains an IP address for management and download its configuration file with the initial configuration (i.e. service flows). This file configures the CM as a bridge.

On the other hand, the CMTS has a Cisco IOS with a CLI similar to any other Cisco device. The CMTS must be properly configured and, in this case, it behaves as a switch for packets coming from users behind the CMs. As a result, each CM is tagged with a different VLAN on egress port to the aggregation/core network (upwards). This means that the packets from each CM are tagged before accessing the aggregator helper. This mapping is relevant for the DOCSIS proxy to properly configure the flows dynamically.

3.7.5 Interfaces Provided by the CMTS

The CMTS is the key element to achieve the dynamic provisioning of services. As any other Cisco device it is a closed-box with limited access. Basically, there are two common ways of configuring and interfacing the CMTS: the Command-Line Interface (CLI) and SNMP. However, none of them provides an interface to dynamically update the service flows defined in the DOCSIS side. Only some management actions can be performed by these interfaces.

In DOCSIS, the PCMM architecture, shown in Figure 3.19, defines a mechanism to dynamically provide new service flows. To this end, another interface is exposed by the CMTS, which is based on the Common Open Policy Service (COPS) protocol [12]. In PCMM, this COPS-based interface enables the management of the QoS life-cycle by maintaining the service flow creation, deletion and modification functions.

Therefore, the DOCSIS translator (device driver) needs to implement a COPS interface to dynamically create and delete service flows in the DOCSIS network. The main challenge is the mapping of the OpenFlow flow entry parameters to COPS related parameters in order to create the corresponding service flow.

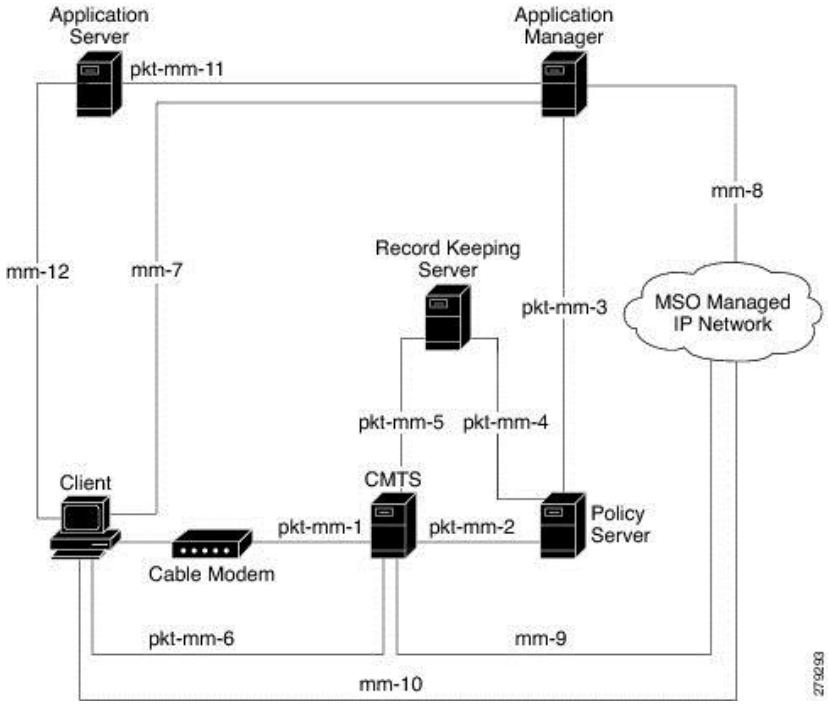

Figure 3.19 PCMM architectural overview [13]

3.7.6 Limitations

The main limitation of DOCSIS hardware is that it is based on closed-box from Cisco, which means that we cannot change its behaviour by adding new code with extra functionalities. Therefore, we need to adapt our solution to the available interfaces.

Another limitation is the fact that OpenFlow and DOCSIS are not fully compatible. For instance, the DOCSIS technology is connected-oriented, which means that we need to configure the connection between the CM and CMTS in advance. Additionally, the classifiers available at DOCSIS are not fully compatible with the matching structures used by OpenFlow. Furthermore, not all the actions defined by OpenFlow can be implemented with DOCSIS. Due to all these reasons, we have added new helper boxes attached to the CM and the CMTS in order to implement those capabilities not supported by DOCSIS and which are mandatory for OpenFlow.

Finally, the additional tagging needed to configure the CMTS as a bridge has some implications. Although this tagging can be used internally between the CMTS and the aggregation helper, this implies some restrictions when trying to provide VLAN support to end users. Since Q-in-Q is not supported by the system, there is not possible to fully support VLAN tags passing through the DOCSIS equipment.

4 Software Development Strategy

Hardware Specific Parts of the HAL developed for different platforms will be realized in a form of separated software packages. The actual implementation of the HSP for each hardware platform will be prepared by an individual partner and will not be integrated into a single software repository. For this particular reason the project has decided to weaken the software governance guidelines and let partners make decisions related to their own company's software development strategy.

4.1 Software Development Methodology and Languages

Each hardware platform imposes different requirements on the Hardware Specific Part implementation. The AFA interface is defined as a C-language interface but languages and technologies needed to implement all required functionalities will depend on a specific hardware platform. The use of agile software development methodologies is a good and common practice in research projects. In research projects system requirements are often not strictly defined at the beginning of the software development process. The agile methodology allows adopting the software development strategy to the emerging needs identified at further stages of the software development process. When talking about research projects, they usually assume a short time for releases of early prototypes, therefore giving researchers time to validate these developments and re-formulate some requirements, as a result of this analysis.

Based on the above, the proposal agreed by all development teams is to choose one of many agile development methodologies, e.g. Extreme Programming or Scrum, and use it in the HSP development process.

4.2 Software Quality Assurance

The primary software quality assurance method used in the HSP development will be testing. In ALIEN HSP development we will use three types of testing:

- **unit testing** – The aim of unit testing is to find faults at the lowest (class, function) level. Unit tests are prepared and performed by developers of given module. We recommend all teams to use unit testing during development of a HSP for its platforms wherever possible. The best way to do this is to use frameworks like JUnit for java code and CppUnit for C++ code.

- **integration testing** – In Integration testing, developers aggregate the modules in larger groups and apply test cases that exercise whether all components interact correctly. In ALIEN project integration testing will be performed on integrated hardware specific and agnostic parts (developed respectively in WP3 and WP2). OF Test software will be used as an automatic and standardized set of tests for OpenFlow devices.

- **system testing** – System (functional) testing evaluates the system's compliance with its specified requirements and are performed by testers. In ALIEN project HSP will be tested together with all other software components using prepared use cases. Part of system tests related to device management will be performed in WP4 (Alien Hardware Integration within OFELIA Control Framework). Part of system tests related to device operation will be performed in WP5 (Experiments on OFELIA) using CoNet software.

4.3 Development Plan

WP3 efforts during development phase will focus on implementation of drivers' prototypes (HSPs) for different hardware platforms. The implementation phase started in month 7 of the project. Two releases are planned within WP3:

- **Release 1** – in month 14 – it will be a preliminary hardware specific part prototype. After this release first demonstrations of project results will be possible.

- **Release 2** – in month 20 – it will be a final hardware specific part prototype. This release ends task T3.3 and WP3.

In the meantime in month 15 the HSP prototypes are going to be integrated and validated using the CCN application.

During the development phase some feedbacks from WP5 are expected in month 12 and 18 respectively. A beta version of common part of data element should be delivered by WP2 for integration with HSP.

HSP development plan is presented on Figure 4.1.

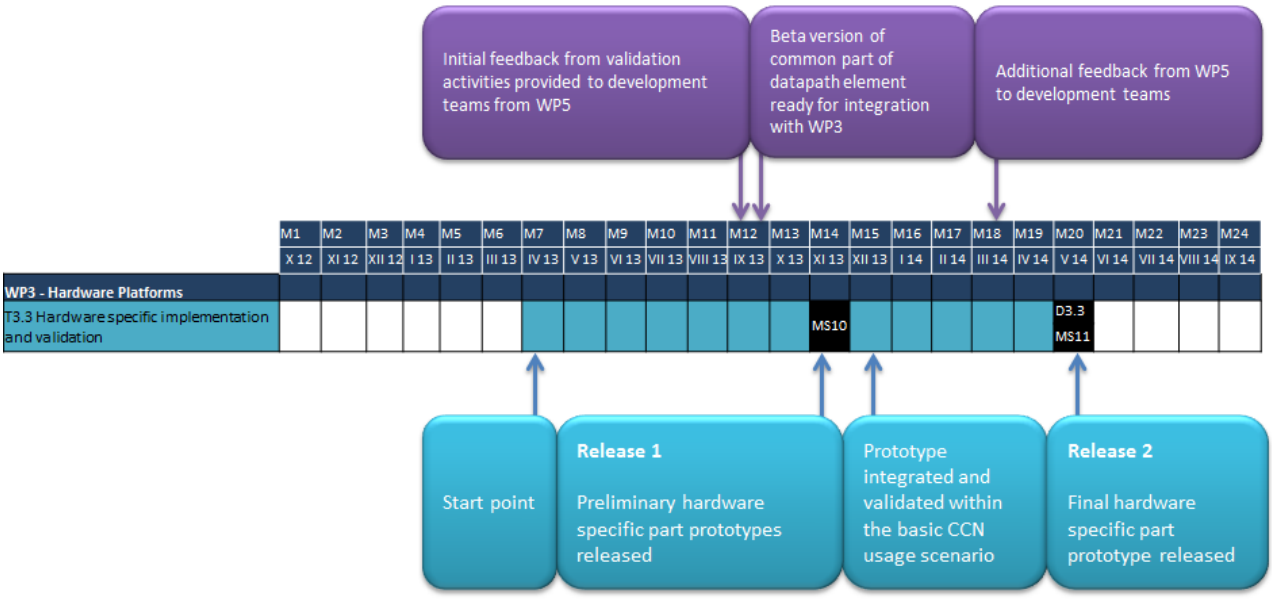

Figure 4.1 HSP development plan with dependencies between WPs

4.4 Requirements Definition

The HSP as a software instance has to fulfill a set of requirements that define its functionality. ALIEN project decided to use OpenFlow switch as a device abstraction and to expose it through the AFA interface. The OpenFlow switch is described in details in OpenFlow protocol specification published by Open Networking Foundation. This specification provides a stable base for the HSP development to be implemented for each ALIEN platform taking into account device specific constraints (summarized in chapter 10). The AFA interface will be specified by WP2 in month 16 (Specification of Hardware Abstraction Layer) and development teams should take into account potential changes in the HSP. A contingency plan should include a close cooperation between WP2 and WP3 members to minimize the impact of the final version of the HAL architecture (WP2) on the HSP developments in WP3.

4.5 Software Repository

Due to the independence of the individual implementations of the HSP all partners are allowed to choose a version control software and location of their software repository. PSNC as a project coordinator will provide repositories in the central ALIEN repository for all partners that will decide not to use its own locations. Code repositories provided by PSNC will support Git, a distributed open source version control system. Repositories provided by PSNC will not be open to public and will require authentication and authorization to realize an access to the repository.

Proposed directory structure for ALIEN HSP repositories is presented on Figure 4.2.

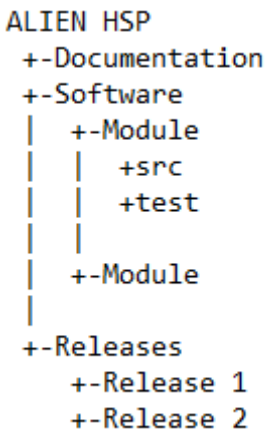

Figure 4.2 HSP repository directory structure

5 Conclusions and Future Steps Towards the HAL Implementation

This deliverable presents the details of the Hardware Abstraction Layer modules and interfaces (specifications provided by the Task 2.3) and specifies how these interfaces will be utilized within Hardware Specific Parts in order to integrate "alien" network hardware elements with an OpenFlow control framework. In this project, a wide range of different types of "alien" hardware is involved and includes: programmable packet switching network elements (EZappliance, ATCA, NetFPGA, Dell switch), non-packet switching network elements (Layer 0 switch) and point-to-multipoint systems (GEPON, DOCSIS). The specification of each HSP contains information how common set of the functionalities will be realized within given hardware platform, i.e.:

- packet forwarding,
- packet processing,
- flow table pipeline support,
- virtualization.

Packet forwarding and packet processing are represented by OpenFlow actions and OpenFlow instructions, respectively. Flow table pipeline supports indicate an ability to contain multiple OpenFlow tables by hardware and possibility of interacting with those OpenFlow tables.

The overview of the possible to implement functionalities for different hardware platform is presented in Table 5. This set of common functionalities is strictly defined by the AFA interface of the HAL and reflects functionalities defined by ONF within different versions of the OpenFlow protocol and switch specifications (v1.0, v1.2 and v1.3).

Table 5 Set of Common Functionalities for Different Hardware

OpenFlow functionalities (actions)	Device						
	EZappliance platform (NP-3 Network Processor)	ATCA (Cavium OCTEON Network Processor)	NetFPGA	L0 Switch	Dell/Force10 Switch	GEPON	DOCSIS
Forward to physical port (Output)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Forward to all physical port (Output ALL)	Yes	Yes	Yes	No	Limited	Limited	Limited
Forward along spanning tree ports (Output FLOOD)	Yes	Yes	Yes	N/A	Limited	Limited	Limited
Forward to controller (Output CONTROLLER)	Yes	Yes	Yes	N/A	Yes	Yes	Yes
Forward to a table (Output TABLE) (Goto-Table)	Yes	Yes	Yes	N/A	Yes	Yes	Yes
Forward to input port (Output IN_PORT)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multiple forwards	Yes	Yes	Yes	No	Yes	Limited	Yes
Enqueue (Set-Queue)	Yes	Yes	No	No	Yes	Limited	Limited
Drop	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Modify-Field actions (Set-Field)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Apply-Actions	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Write-Actions	Yes	Yes	Yes	No	Yes	Yes	Yes

Clear-Actions	Yes	Yes	Yes	No	Yes	Yes	Yes
Group	No	Limited	No	No	No	Limited	Limited
Flow Tables	Limited	Limited	Yes	Yes	Yes	Yes	Limited
Virtualization	Limited	No	Limited	Limited	No	No	Limited

Table 5 shows that most of the selected functionalities will be available on each "alien" hardware platform. It should be noted that not all hardware platforms will be able to implement all common functionalities or some features will be implemented in a limited or simplified way. Presented limitations result from hardware limitations and unsupported hardware functions are as follows:

- not all flow entries matching are possible in hardware because of limited memory size and wildcard operation availability in hardware (i.e.: EZappliance, ATCA);
- problems in implementing of the full support of OpenFlow queues (i.e.: NetFPGA, GEPON);
- non-packet hardware cannot perform packet oriented OpenFlow functionalities (i.e.: L0 switch);
- constraints regarding VLANs operations (i.e.: GEPON, DOCSIS);
- adopting close-box functionalities to fulfill OpenFlow capabilities is challenging (i.e.: L0 switch, GEPON, DOCSIS);
- specific network processor architectures impose some problems in generic OpenFlow packet matching and pipelining implementation – the development process will be complex due to the:
    - shared memory (in OCTEON platform),
    - pipelining of specific internal entities of network processor (in EZappliance platform),
    - very low level silicon circuit designing (in NetFPGA cards)
- available functionality (and also stability) of SDK libraries for network processors makes some features harder to implement (i.e.: Dell/Force10 switch).

In order to overcome some of the above limitations, additional network elements (i.e.: Open-Flow switches) could be connected to the hardware platform and orchestrated together with the platform by single OpenFlow endpoint instance (i.e.: GEPON or DOSICS systems). Regardless the lack of some functionalities within given platforms, the important conclusion is that "alien" platforms should be able to support all usage scenarios defined by Task 5.1 within deliverable D5.1.

Task 2.3 in the ALIEN project has provided a first version of the common part (hardware independent layer) of the HAL concept in form of ROFL library and xDPd framework for creating datapath element. The Hardware Specific Parts design assumes that each hardware platform will implement AFA and/or Hardware Pipeline interfaces defined by the set of header files (included in ROFL libraries [8]). The hardware specific parts (implementing HPL) are exposing the capabilities of the "alien" network device to hardware independent HAL layer and interacting with lower layers represented by a network device. Although, hardware specific parts are different for each "alien" network device and they will be realized in a form of separated software packages by partners responsible for particular platforms, we can recognize common functional blocks, which have to appear in HSP. This deliverable defines the HSP by software modules and interfaces descriptions as well as presents software deployment plans for each "alien" network device.

This document ends Task 3.2 and provides the main input for Task 3.3 (e.g.: high-level design of software components) where different teams will continue the work on low-level software design and implementation of software components of each single driver prototype (HSP). It must be clearly mentioned, that in moment of publishing of this deliverable, the

work on the HAL (hardware agnostic part) is still an ongoing activity in Task 2.2 as well as virtualization mechanisms are still analysed in Task 2.4. Thus, some further changes of the Hardware Specific Parts could be required in the future – it will be handled directly in Task 3.3. This deliverable contains also a more specific implementation, deployment and integration plans (with WP4 and WP5 activities) for HSPs.

References

[1] HAL whitepaper; ALIEN project; http://www.fp7-alien.eu/files/deliverables/ALIEN-HAL-whitepaper.pdf

[2] Deliverable D3.1; ALIEN project; http://www.fp7-alien.eu/files/deliverables/D3.1-ALIEN-final.pdf

[3] OpenFlow specification version 1.2; https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.2.pdf

[4] EZchip; 2008; EZdriver NP-3 API Reference Manual

[5] Corba specification ; OMG (n.d); retrieved 2013; http://www.omg.org/spec/index.htm

[6] omniORB; Pasphere Ltd. (n.d.); retrieved 2013; http://omniorb.sourceforge.net

[7] xDPd; https://www.codebasin.net/redmine/projects/xdpd/wiki/Architecture

[8] ROFL; https://www.codebasin.net/redmine/projects/rofl-core/wiki/Wiki?version=12

[9] OCTEON Programmer's Guide Introduction Version for Cavium Networks University Program; http://www.university.caviumnetworks.com/downloads/00-FUNDAMENTALS_ALL_CHAPTERS_EDU_July_2010.pdf

[10] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, Nick McKeown; *Implementing an OpenFlow Switch on the NetFPGA Platform*; ACM/IEEE Symposium on Architectures for Networking and Communications Systems; San Jose, CA; November 6-7, 2008

[11] OpenFlow Addendum version 0.3; http://archive.openflow.org/wk/images/8/81/OpenFlow_Circuit_Switch_Specification_v0.3.pdf

[12] COPS protocol; RFC 2748;January 2000; http://www.rfc-editor.org/rfc/pdfrfc/rfc2748.txt.pdf

[13] http://www.cisco.com/en/US/docs/ios/cable/configuration/guide/cmts_pktcable_mm.html

[14] https://www.codebasin.net/redmine/projects/rofl-core/repository/revisions/devel/entry/src/rofl/datapath/afa/fwd_module.h

[15] https://www.codebasin.net/redmine/projects/rofl-core/repository/revisions/devel/entry/src/rofl/datapath/afa/openflow/openflow12/of12_fwd_module.h

[16] https://www.codebasin.net/redmine/projects/rofl-core/repository/revisions/devel/entry/src/rofl/datapath/afa/cmm.h

[17] https://www.codebasin.net/redmine/projects/rofl-core/repository/revisions/devel/entry/src/rofl/datapath/pipeline/platform/packet.h

[18] https://www.codebasin.net/redmine/projects/rofl-core/repository/revisions/devel/entry/src/rofl/datapath/pipeline/platform/memory.h

[19] https://www.codebasin.net/redmine/projects/rofl-core/repository/revisions/devel/entry/src/rofl/datapath/pipeline/platform/lock.h

[20] https://www.codebasin.net/redmine/projects/rofl-core/repository/revisions/devel/entry/src/rofl/datapath/pipeline/platform/atomic_operations.h

[21] https://www.codebasin.net/redmine/projects/rofl-core/repository/revisions/devel/entry/src/rofl/datapath/pipeline/of_switch.h

Acronyms

AFA	Abstract Forwarding API
ATCA	Advanced Telecommunications Computing Architecture
AMC	Advanced Mezzanine Cards
CLI	Command-Line Interface
CM	Cable Modem
CMM	Control and Management Module
CMTS	Cable Modem Termination System
COPS	Common Open Policy Service
DFA	Deterministic Finite Automata
DIM	Device Information Module
GEPON	Gigabit Ethernet Passive Optical Network
HAL	Hardware Adaptation Layer
HFC	Hybrid Fibre-Coaxial
HIL	Hardware Interface Layer
HPL	Hardware Presentation Layer
HSP	Hardware Specific Part
IPD	Input Packet Data Unit
LLID	Logical Layer ID
NMS	Network Management System
NE	Network Element
NP	Network Processor
NPsl	Network Processor script language
NPU	Network Processor Unit
OF	OpenFlow
PIP	Packet Input Processor
PPC	Power PC
RG	Residential Gateway
ROFL	Revised OpenFlow Library
SDK	Software Development Kit
SDN	Software Defined Network
SDPM	Split Data Plane Module
SMP	Synchronous Multi-Processing
TCAM	Ternary Content Addressable Memory

TM	Traffic Management
WQ	Work Queue
WQE	Work Queue Entries
VA	Virtual Agent
xDPd	eXtensible DataPath Daemon

Appendix A *AFA Interface*

A.1 *Data Structures*

Structure	**of_switch_t**
Description	Openflow-enabled switch abstraction.
Attributes	• Name of the switch • Datapath identifier • Supported OpenFlow version • Number of ports • List of logical ports • State of hardware platform

Structure	**switch_port_t**
Description	Implements the switch port abstraction..
Attributes	• MAC address • Administrative state • Operational state • Port type (Physical, virtual or tunnel) • Port system name • Port capabilities (supported features: traffic rate, duplex, physical medium, auto-negotiation) • Flags (if packets must be dropped, if packets can be send out, if generate packet-in events) • Port statistics (Number of received/transmitted packets/bytes, packets dropped, received/transmitted/alignment/overrun/checksum errors and collisions) • Port speed • Attached queues • Logical switch to which port is attached

A.2 *AFA Interface Functions – Datapath Management [16]*

Forwarding module management:

Function	**fwd_module_init**
Description	Initializes hardware platform driver which covers all initial operations required for a

	particular hardware platform (e.g.: checking hardware accessibility, discovering hardware capabilities and resources, initialize device configuration as well). Only initialized driver can be used.
Declaration	*afa_result_t fwd_module_init(void);*
Arguments	None
Result	Success or failure

Function	**fwd_module_destroy**
Description	Destroy driver state. Allows platform to be properly cleaned.
Declaration	*afa_result_t fwd_module_destroy(void);*
Arguments	None
Result	Success or failure

Logical switch management:

Function	**fwd_module_create_switch**
Description	Instruct driver to create an OF logical switch.
Declaration	*of_switch_t* fwd_module_create_switch(char* name, uint64_t dpid, of_version_t of_version, unsigned int num_of_tables, int* ma_list/);*
Arguments	• Name of the switch • Datapath identifier • Supported OpenFlow version • Number of tables • Frame Matching algorithms
Result	Pointer to OF switch instance

Function	**fwd_module_get_switch_by_dpid**
Description	Retrieve the switch with the specified datapath identifier.
Declaration	*of_switch_t* fwd_module_get_switch_by_dpid(uint64_t dpid);*
Arguments	• Datapath identifier
Result	Pointer to OF switch instance

Function	**fwd_module_destroy_switch_by_dpid**
Description	Instructs the driver to destroy the switch with the specified datapath identifier.
Declaration	*afa_result_t fwd_module_destroy_switch_by_dpid(uint64_t dpid);*
Arguments	• Datapath identifier
Result	Pointer to OF switch instance

Port management:

Function	**fwd_module_get_port_by_name**
Description	Get a reference to the port by its name.
Declaration	*switch_port_t* fwd_module_get_port_by_name(const char *name);*
Arguments	• Port system name
Result	Pointer to the port instance

Function	**fwd_module_get_physical_ports** **fwd_module_get_virtual_ports** **fwd_module_get_tunnel_ports**
Description	Retrieve the list of the physical/virtual/tunnel ports of the platform.
Declaration	*switch_port_t** fwd_module_get_physical_ports(unsigned int* max_ports);* *switch_port_t** fwd_module_get_virtual_ports(unsigned int* max_ports);* *switch_port_t** fwd_module_get_tunnel_ports(unsigned int* max_ports);*
Arguments	None
Result	• Number of ports • Pointer to the first port instance

Function	**fwd_module_attach_port_to_switch**
Description	Attempts to attach a system's port to switch at *of_port_num* if defined, otherwise in the first empty OF port number.
Declaration	*afa_result_t fwd_module_attach_port_to_switch(uint64_t dpid, const char* name, unsigned int* of_port_num);*
Arguments	• Datapath identifier to which port will be attached • Port system name • Assigned port number (if zero then first available port number will be

	assigned)
Result	• Success or failure • Assigned port number

Function	**fwd_module_detach_port_from_switch**
Description	Detaches a port from the switch by port name.
Declaration	*afa_result_t fwd_module_detach_port_from_switch(uint64_t dpid, const char* name);*
Arguments	• Datapath identifier to which port will be attached • Port system name
Result	Success or failure

Function	**fwd_module_detach_port_from_switch_at_port_num**
Description	Detaches a port from the switch by port number.
Declaration	*afa_result_t fwd_module_detach_port_from_switch_at_port_num(uint64_t dpid, const unsigned int of_port_num);*
Arguments	• Datapath identifier to which port will be attached • Assigned port number
Result	Success or failure

Function	**fwd_module_enable_port** **fwd_module_disable_port**
Description	Brings up/down a system port identified by port system name
Declaration	*afa_result_t fwd_module_enable_port(const char* name);* *afa_result_t fwd_module_disable_port(const char* name);*
Arguments	• Port system name
Result	Success or failure

Function	**fwd_module_enable_port_by_num** **fwd_module_disable_port_by_num**
Description	Brings up/down a port from an OF logical switch (and the underlying physical interface) identified by assigned port number.

Declaration	*afa_result_t fwd_module_enable_port_by_num(uint64_t dpid, unsigned int port_num);* *afa_result_t fwd_module_disable_port_by_num(uint64_t dpid, unsigned int port_num);*
Arguments	• Datapath identifier to which port will be attached • Assigned port number
Result	Success or failure

Function	**fwd_module_list_matching_algorithms**
Description	Get a list of available matching algorithms.
Declaration	*afa_result_t fwd_module_list_matching_algorithms(of_version_t of_version, const char * const** name_list, int *count);*
Arguments	• OpenFlow version
Result	• Number of algorithms • List of algorithms names

A.3 *AFA Interface Functions – Datapath Configuration [14, 15]*

Port/pipeline/table configuration:

Function	**fwd_module_of12_set_port_drop_received_config**
Description	Instructs driver to drop or not drop all incoming packets on the port.
Declaration	*afa_result_t fwd_module_of12_set_port_drop_received_config(uint64_t dpid, unsigned int port_num, bool drop_received);*
Arguments	• Datapath identifier • Port number • Drop received packets flag
Result	Success or failure

Function	**fwd_module_of12_set_port_forward_config**
Description	Instructs driver to send or not send (output) packets on this port to the network link.
Declaration	*afa_result_t fwd_module_of12_set_port_forward_config(uint64_t dpid, unsigned int port_num, bool forward);*
Arguments	• Datapath identifier

	• Port number • Forward packets flag
Result	Success or failure

Function	**fwd_module_of12_set_port_generate_packet_in_config**
Description	Instructs driver to generate or not generate packet-in events for the port.
Declaration	*afa_result_t fwd_module_of12_set_port_generate_packet_in_config(uint64_t dpid, unsigned int port_num, bool generate_packet_in);*
Arguments	• Datapath identifier • Port number • Generate packet-in flag
Result	Success or failure

Function	**fwd_module_of12_set_port_advertise_config**
Description	Instructs driver to modify port advertise flags representing features being advertised by the port.
Declaration	*afa_result_t fwd_module_of12_set_port_advertise_config(uint64_t dpid, unsigned int port_num, uint32_t advertise);*
Arguments	• Datapath identifier • Port number • Features to be advertised
Result	Success or failure

Function	**fwd_module_of12_set_pipeline_config**
Description	Instructs driver to set pipeline configuration.
Declaration	*afa_result_t fwd_module_of12_set_pipeline_config(uint64_t dpid, unsigned int flags, uint16_t miss_send_len);*
Arguments	• Datapath identifier • Pipeline capabilities (flow statistics, table statistics, etc) • Length of packet to be send to controller if packet does not matched any flow entry

Result	Success or failure

Function	**fwd_module_of12_set_table_config**
Description	Instructs driver to set table configuration.
Declaration	*afa_result_t fwd_module_of12_set_table_config(uint64_t dpid, unsigned int table_id, of12_flow_table_miss_config_t config);*
Arguments	• Datapath identifier • Table identifier • Table reaction to not matched packet (send to controller, continue to next table, drop packet)
Result	Success or failure

Packet out functionality:

Function	**fwd_module_of12_process_packet_out**
Description	Instructs driver to send a packet from controller out through the datapath. This function covers two situations: 1) packet is already stored in local buffer, 2) the whole packet is passed in OpenFlow message.
Declaration	*afa_result_t fwd_module_of12_process_packet_out(uint64_t dpid, uint32_t buffer_id, uint32_t in_port, of12_action_group_t* action_group, uint8_t* buffer, uint32_t buffer_size);*
Arguments	• Datapath identifier • Buffer identifier if packet already stored locally • Buffer with packet (pointer to a buffer and size of packet in the buffer) if packet not stored locally (*buffer_id* must be equal OF12P_NO_BUFFER) • Packet's input port • Action group structure to be applied on packet (should contain OUTPUT action)
Result	Success or failure

Flows configuration:

Function	**fwd_module_of12_process_flow_mod_add**
Description	Instructs driver to add new flow entry.
Declaration	*afa_result_t fwd_module_of12_process_flow_mod_add(uint64_t dpid, uint8_t table_id, of12_flow_entry_t* flow_entry, uint32_t buffer_id, bool check_overlap, bool*

	reset_counts);
Arguments	• Datapath identifier • Table identifier • Flow entry structure • Buffer identifier • Flags (i.e. check for overlapping entries first, flow counters should be reseted if flow already present)
Result	Success or failure

Function	**fwd_module_of12_process_flow_mod_modify**
Description	Instructs driver to modify the existing flow entry.
Declaration	*afa_result_t fwd_module_of12_process_flow_mod_modify(uint64_t dpid, uint8_t table_id, of12_flow_entry_t* flow_entry, of12_flow_removal_strictness_t strictness, bool reset_counts);*
Arguments	• Datapath identifier • Table identifier • Flow entry structure • Buffer identifier • Flow modify strictness (strict or not strict) • Flag (flow counters should be reseted if flow already present)
Result	Success or failure

Function	**fwd_module_of12_process_flow_mod_delete**
Description	Instructs driver to delete the existing flow entry.
Declaration	*afa_result_t fwd_module_of12_process_flow_mod_delete(uint64_t dpid, uint8_t table_id, of12_flow_entry_t* flow_entry, uint32_t buffer_id, uint32_t out_port, uint32_t out_group, of12_flow_removal_strictness_t strictness);*
Arguments	• Datapath identifier • Table identifier • Flow entry structure • Buffer identifier • Output port identifier that entry must include • Output group identifier that entry must include • Flow modify strictness (strict or not strict)
Result	Success or failure

Flow statistics:

Function	**fwd_module_of12_get_flow_stats**
Description	Fetch the flow statistics on a given set of matches.
Declaration	*of12_stats_flow_msg_t* fwd_module_of12_get_flow_stats(uint64_t dpid, uint8_t table_id, uint32_t cookie, uint32_t cookie_mask, uint32_t out_port, uint32_t out_group, of12_match_t* matchs);*
Arguments	• Datapath identifier • Table identifier • Cookie to be applied • Mask for the cookie • Output port identifier that entry must include • Output group identifier that entry must include • Match structure
Result	Linked list containing all the individual flow statistics

Function	**fwd_module_of12_get_flow_aggregate_stats**
Description	Fetch the aggregated flow statistics on a given set of matches.
Declaration	*of12_stats_flow_aggregate_msg_t* fwd_module_of12_get_flow_aggregate_stats(uint64_t dpid, uint8_t table_id, uint32_t cookie, uint32_t cookie_mask, uint32_t out_port, uint32_t out_group, of12_match_t* matchs);*
Arguments	• Datapath identifier • Table identifier • Cookie to be applied • Mask for the cookie • Output port identifier that entry must include • Output group identifier that entry must include • Match structure
Result	Aggregated flow statistics

Group configuration:

Function	**fwd_module_of12_group_mod_add** **fwd_module_of12_group_mod_modify**
Description	Instructs driver to add/modify a new group.
Declaration	*rofl_of12_gm_result_t fwd_module_of12_group_mod_add(uint64_t dpid, of12_group_type_t type, uint32_t id, of12_bucket_list_t *buckets);*

	*rofl_of12_gm_result_t fwd_module_of12_group_mod_modify(uint64_t dpid, of12_group_type_t type, uint32_t id, of12_bucket_list_t *buckets);*
Arguments	• Datapath identifier • Group type (all multicast/broadcast group, select group, indirect group, fast failover group) • Group identifier • Group bucket list
Result	Success or failure

Function	**fwd_module_of12_group_mod_delete**
Description	Instructs driver to delete group.
Declaration	*rofl_of12_gm_result_t fwd_module_of12_group_mod_delete(uint64_t dpid, uint32_t id);*
Arguments	• Datapath identifier • Group identifier
Result	Success or failure

Function	**fwd_module_of12_fetch_group_table**
Description	Instructs driver to search if group with specific identifier already exists,
Declaration	*afa_result_t fwd_module_of12_fetch_group_table(uint64_t dpid, of12_group_table_t *group_table);*
Arguments	• Datapath identifier • Group identifier
Result	Success or failure

Group statistics:

Function	**fwd_module_of12_get_group_stats** **fwd_module_of12_get_group_all_stats**
Description	Instructs driver to fetch the group statistics.
Declaration	*of12_stats_group_msg_t * fwd_module_of12_get_group_stats(uint64_t dpid, uint32_t id);*
Arguments	• Datapath identifier

	• Group identifier
Result	Group statistics

A.4 *AFA Interface Functions – Notifications [16]*

Port notifications:

Function	***cmm_notify_port_add*** ***cmm_notify_port_delete*** ***cmm_notify_port_status_changed***
Description	Notifies port related changes (port added, removed, its attributes changed) within the platform.
Declaration	*afa_result_t cmm_notify_port_add(switch_port_t* port);* *afa_result_t cmm_notify_port_delete(switch_port_t* port);* *afa_result_t cmm_notify_port_status_changed(switch_port_t* port);*
Arguments	• Switch port
Result	Success or failure

OpenFlow notifications:

Function	**cmm_process_of12_packet_in**
Description	Handle packet from datapath to be send to controller.
Declaration	*afa_result_t cmm_process_of12_packet_in(const of12_switch_t* sw,* *uint8_t table_id,* *uint8_t reason,* *uint32_t in_port,* *uint32_t buffer_id,* *uint8_t* pkt_buffer,* *uint32_t buf_len,* *uint16_t total_len,* *of12_packet_matches_t matches);*
Arguments	• Logical switch • Table identifier that generated packet-in event • Reason for packet-in event • Incoming packet port • Buffer identifier of hardware buffer • Buffer containing the packet • Buffer length (may be shorter than the packet stored in buffer)

	• Total length of buffer • Packet matches
Result	Success or failure

Function	**cmm_process_of12_flow_removed**
Description	Process a flow removed event coming from the datapath
Declaration	*afa_result_t cmm_process_of12_flow_removed(const of12_switch_t* sw, uint8_t reason, of12_flow_entry_t* removed_flow_entry);*
Arguments	• Logical switch • Reason for packet-in event • Removed flow entry
Result	Success or failure

Appendix B ROFL Pipeline Interface

B.1 Packet Operations [17]

Structure	**datapacket_t**
Description	Abstraction that represents a data packet that may transverse one logical switch OpenFlow pipeline and keep reference to real packet stored in the device.
Attributes	• Successful matches (OpenFlow matches) • Operations to be applied on packet (write actions) • Platform specific state

Function	**platform_packet_get_size_bytes**
Description	Getting the size of the packet related to an instance of packet abstraction processed within the pipeline.
Declaration	*uint32_t platform_packet_get_size_bytes(datapacket_t *const pkt);*
Arguments	• Data packet abstraction instance
Result	The real packet size in bytes

Function	**platform_packet_get_port_in**
Description	Getting an identifier of the logical port (within the logical switch) where packet was received.
Declaration	*uint32_t platform_packet_get_port_in(datapacket_t *const pkt);*
Arguments	• Data packet abstraction instance
Result	Port number

Function	**platform_packet_get_phy_port_in**
Description	Getting an system identifier of the physical port of device where packet was received.
Declaration	*uint32_t platform_packet_get_phy_port_in(datapacket_t *const pkt);*
Arguments	• Data packet abstraction instance
Result	Physical port number

Function	**platform_packet_get_eth_dst** **platform_packet_get_eth_src** **platform_packet_get_eth_type** **platform_packet_get_vlan_vid** **platform_packet_get_vlan_pcp** **platform_packet_get_ip_ecn** **platform_packet_get_ip_dscp** **platform_packet_get_ip_proto** **platform_packet_get_ipv4_src** **platform_packet_get_ipv4_dst** **platform_packet_get_tcp_dst** **platform_packet_get_tcp_src** **platform_packet_get_udp_dst** **platform_packet_get_udp_src** **platform_packet_get_icmpv4_type** **platform_packet_get_icmpv4_code** **platform_packet_get_mpls_label** **platform_packet_get_mpls_tc** **platform_packet_get_pppoe_code** **platform_packet_get_pppoe_type** **platform_packet_get_ppp_proto**
Description	Getting specified header field value of OSI layers 2-4 protocols (802.3 Ethernet, ICMPv4, IPv4, MPLS, PPPoE and TCP/UDP).
Declaration	*//802* *uint64_t platform_packet_get_eth_dst(datapacket_t *const pkt);* *uint64_t platform_packet_get_eth_src(datapacket_t *const pkt);* *uint16_t platform_packet_get_eth_type(datapacket_t *const pkt);* *//802.1q VLAN outermost tag* *uint16_t platform_packet_get_vlan_vid(datapacket_t *const pkt);* *uint8_t platform_packet_get_vlan_pcp(datapacket_t *const pkt);* *//IPv4* *uint8_t platform_packet_get_ip_proto(datapacket_t *const pkt);* *uint8_t platform_packet_get_ip_ecn(datapacket_t *const pkt);* *uint8_t platform_packet_get_ip_dscp(datapacket_t *const pkt);* *uint32_t platform_packet_get_ipv4_src(datapacket_t *const pkt);* *uint32_t platform_packet_get_ipv4_dst(datapacket_t *const pkt);*

	//TCP *uint16_t platform_packet_get_tcp_dst(datapacket_t *const pkt);* *uint16_t platform_packet_get_tcp_src(datapacket_t *const pkt);* *//UDP* *uint16_t platform_packet_get_udp_dst(datapacket_t *const pkt);* *uint16_t platform_packet_get_udp_src(datapacket_t *const pkt);* *//ICMPv4* *uint8_t platform_packet_get_icmpv4_type(datapacket_t *const pkt);* *uint8_t platform_packet_get_icmpv4_code(datapacket_t *const pkt);* *//MPLS-outermost label* *uint32_t platform_packet_get_mpls_label(datapacket_t *const pkt);* *uint8_t platform_packet_get_mpls_tc(datapacket_t *const pkt);* *//PPPoE related extensions* *uint8_t platform_packet_get_pppoe_code(datapacket_t *const pkt);* *uint8_t platform_packet_get_pppoe_type(datapacket_t *const pkt);* *uint16_t platform_packet_get_pppoe_sid(datapacket_t *const pkt);* *//PPP related extensions* *uint16_t platform_packet_get_ppp_proto(datapacket_t *const pkt);*
Arguments	• Data packet abstraction instance
Result	Specified Field value

Function	**platform_packet_set_mpls_ttl** **platform_packet_set_nw_ttl** **platform_packet_set_eth_dst** **platform_packet_set_eth_src** **platform_packet_set_eth_type** **platform_packet_set_vlan_vid** **platform_packet_set_vlan_pcp** **platform_packet_set_ip_dscp** **platform_packet_set_ip_ecn** **platform_packet_set_ip_proto** **platform_packet_set_ipv4_src** **platform_packet_set_ipv4_dst** **platform_packet_set_tcp_src** **platform_packet_set_tcp_dst** **platform_packet_set_udp_src** **platform_packet_set_udp_dst** **platform_packet_set_icmpv4_type** **platform_packet_set_icmpv4_code** **platform_packet_set_mpls_label** **platform_packet_set_mpls_tc** **platform_packet_set_pppoe_type**

	platform_packet_set_pppoe_code **platform_packet_set_pppoe_sid** **platform_packet_set_ppp_proto**
Description	Setting (overwriting) specified header field value of OSI layers 2-4 protocols (802.3 Ethernet, ICMPv4, IPv4, MPLS, PPPoE and TCP/UDP). Should always be applied to the outermost-possible header of given type.
Declaration	*void platform_packet_set_mpls_ttl(datapacket_t* pkt, uint8_t new_ttl);* *void platform_packet_set_nw_ttl(datapacket_t* pkt, uint8_t new_ttl);* *void platform_packet_set_queue(datapacket_t* pkt, uint32_t queue);* *//Ethernet* *void platform_packet_set_eth_dst(datapacket_t* pkt, uint64_t eth_dst);* *void platform_packet_set_eth_src(datapacket_t* pkt, uint64_t eth_src);* *void platform_packet_set_eth_type(datapacket_t* pkt, uint16_t eth_type);* *//802.1q* *void platform_packet_set_vlan_vid(datapacket_t* pkt, uint16_t vlan_vid);* *void platform_packet_set_vlan_pcp(datapacket_t* pkt, uint8_t vlan_pcp);* *//IP, IPv4* *void platform_packet_set_ip_dscp(datapacket_t* pkt, uint8_t ip_dscp);* *void platform_packet_set_ip_ecn(datapacket_t* pkt, uint8_t ip_ecn);* *void platform_packet_set_ip_proto(datapacket_t* pkt, uint8_t ip_proto);* *void platform_packet_set_ipv4_src(datapacket_t* pkt, uint32_t ip_src);* *void platform_packet_set_ipv4_dst(datapacket_t* pkt, uint32_t ip_dst);* *//TCP* *void platform_packet_set_tcp_src(datapacket_t* pkt, uint16_t tcp_src);* *void platform_packet_set_tcp_dst(datapacket_t* pkt, uint16_t tcp_dst);* *//UDP* *void platform_packet_set_udp_src(datapacket_t* pkt, uint16_t udp_src);* *void platform_packet_set_udp_dst(datapacket_t* pkt, uint16_t udp_dst);* *//ICMPV4* *void platform_packet_set_icmpv4_type(datapacket_t* pkt, uint8_t type);* *void platform_packet_set_icmpv4_code(datapacket_t* pkt, uint8_t code);* *//MPLS* *void platform_packet_set_mpls_label(datapacket_t* pkt, uint32_t label);* *void platform_packet_set_mpls_tc(datapacket_t* pkt, uint8_t tc);* *//PPPOE* *void platform_packet_set_pppoe_type(datapacket_t* pkt, uint8_t type);* *void platform_packet_set_pppoe_code(datapacket_t* pkt, uint8_t code);* *void platform_packet_set_pppoe_sid(datapacket_t* pkt, uint16_t sid);* *//PPP* *void platform_packet_set_ppp_proto(datapacket_t* pkt, uint16_t proto);*

Arguments	• Data packet abstraction instance • Specified field value
Result	None

Function	**platform_packet_copy_ttl_in** **platform_packet_copy_ttl_out**
Description	Copy the TTL from outermost to next-to-outermost header with TTL or 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.
Declaration	*void platform_packet_copy_ttl_in(datapacket_t* pkt);* *void platform_packet_copy_ttl_out(datapacket_t* pkt);*
Arguments	• Data packet abstraction instance
Result	None

Function	**platform_packet_dec_nw_ttl** **platform_packet_dec_mpls_ttl**
Description	Decrement the TTL field: • IPv4 TTL and up-date the IP checksum • IPv6 Hop Limit field and up-date the IP checksum. • MPLS TTL. Only applies to Pv4 and IPv6 packets and packets with an existing MPLS shim header.
	void platform_packet_dec_nw_ttl(datapacket_t pkt);* *void platform_packet_dec_mpls_ttl(datapacket_t* pkt);*
Arguments	• Data packet abstraction instance
Result	None

Function	**platform_packet_pop_vlan**
Description	Pop the outer-most VLAN header from the packet.
Declaration	*void platform_packet_pop_vlan(datapacket_t* pkt);*
Arguments	• Data packet abstraction instance
Result	None

Function	**platform_packet_pop_pppoe** **platform_packet_pop_mpls**
Description	Pop the outer-most header/tag of specified protocol from the packet. Push a new MPLS shim header onto the packet. The Ethertype is used as the Ethertype for the resulting packet.
Declaration	*void platform_packet_pop_mpls(datapacket_t* pkt, uint16_t ether_type);* *void platform_packet_pop_pppoe(datapacket_t* pkt, uint16_t ether_type);*
Arguments	• Data packet abstraction instance • Ethertype value
Result	None

Function	**platform_packet_push_pppoe** **platform_packet_push_mpls** **platform_packet_push_vlan**
Description	Push a new MPLS shim header onto the packet. The Ethertype is used as the Ethertype for the resulting packet.
Declaration	*void platform_packet_push_pppoe(datapacket_t* pkt, uint16_t ether_type);* *void platform_packet_push_mpls(datapacket_t* pkt, uint16_t ether_type);* *void platform_packet_push_vlan(datapacket_t* pkt, uint16_t ether_type);*
Arguments	• Data packet abstraction instance • Ethertype value
Result	None

Function	**platform_packet_drop**
Description	Drop a real packet pointed by a packet abstraction instance.
Declaration	*void platform_packet_drop(datapacket_t* pkt);*
Arguments	• Data packet abstraction instance
Result	None

Function	**platform_packet_output**
Description	Output packet to the port.
Declaration	*void platform_packet_output(datapacket_t* pkt, switch_port_t* port);*

Arguments	• Data packet abstraction instance • Port instance
Result	None

B.2 *Memory Allocation [18]*

Function	**platform_malloc** **platform_malloc_shared**
Description	Allocates a block of dynamic memory of size length.
Declaration	*void* platform_malloc(size_t length);* *void* platform_malloc_shared(size_t length);*
Arguments	• Length
Result	Allocated memory block.

Function	**platform_free** **platform_free_shared**
Description	Frees a block of dynamic memory/
Declaration	*void platform_free(void* data);* *void platform_free_shared(void* data);*
Arguments	• Allocated memory block.
Result	None

Function	**platform_memcpy** **platform_memmove**
Description	Copies/moves a content of memory block to another memory block.
Declaration	*void* platform_memcpy(void* dst, const void* src, size_t length);* *void* platform_memmove(void* dst, const void* src, size_t length);*
Arguments	• Destination memory block • Source memory block • Length of memory block
Result	None

Function	**platform_memset**
Description	Sets bytes of the block of memory to the specified value
Declaration	*void* platform_memset(void* src, int c, size_t length);*
Arguments	• Memory block • Value set • Length of memory block
Result	Memory block

Function	**platform_packet_replicate**
Description	Creates a copy of the Data packet abstraction instance including platform specific state (e.g.: a reference to the packet buffer in the platform).
Declaration	*datapacket_t* platform_packet_replicate(datapacket_t* pkt);*
Arguments	• Data packet abstraction instance
Result	Data packet abstraction instance

B.3 ***Lock Operations [19]***

Function	**platform_mutex_init** **platform_rwlock_init**
Description	Allocate the memory for the mutex and then perform the mutex initialization. Separated mutex type distinguishing read/write operations could be initialized.
Declaration	*platform_mutex_t* platform_mutex_init(void* params);* *platform_rwlock_t* platform_rwlock_init(void* params);*
Arguments	• Params
Result	Mutex

Function	**platform_mutex_destroy** **platform_rwlock_destroy**
Description	First destroy the lock and then release the memory allocated for the mutex.

Declaration	*void platform_mutex_destroy(platform_mutex_t* mutex);* *void platform_rwlock_destroy(platform_rwlock_t* rwlock);*
Arguments	• Mutex
Result	None

Function	**platform_mutex_lock** **platform_mutex_unlock** **platform_rwlock_rdlock** **platform_rwlock_rdunlock** **platform_rwlock_wrlock** **platform_rwlock_wrunlock**
Description	Lock/unclock mutex (also for only reading or writing operations).
Declaration	*void platform_mutex_lock(platform_mutex_t* mutex);* *void platform_mutex_unlock(platform_mutex_t* mutex);* *void platform_rwlock_rdlock(platform_rwlock_t* rwlock);* *void platform_rwlock_rdunlock(platform_rwlock_t* rwlock);* *void platform_rwlock_wrlock(platform_rwlock_t* rwlock);* *void platform_rwlock_wrunlock(platform_rwlock_t* rwlock);*
Arguments	• Mutex
Result	None

B.4 ***Atomic Counters Operations [20]***

Function	**platform_atomic_inc64** **platform_atomic_inc32** **platform_atomic_dec32** **platform_atomic_add64** **platform_atomic_add32**
Description	Performs an atomic increment/decrement of the counter (32 or 64 bit type).
Declaration	*void platform_atomic_inc64(uint64_t* counter, platform_mutex_t* mutex);* *void platform_atomic_inc32(uint32_t* counter, platform_mutex_t* mutex);* *void platform_atomic_dec32(uint32_t* counter, platform_mutex_t* mutex);* *void platform_atomic_add64(uint64_t* counter, uint64_t* value, platform_mutex_t* mutex);* *void platform_atomic_add32(uint32_t* counter, uint32_t* value, platform_mutex_t* mutex);*
Arguments	• Counter

	• Mutex
Result	None.

Function	**platform_atomic_add64** **platform_atomic_add32**
Description	Performs an atomic addition of given value of the counter (32 or 64 bit type).
Declaration	*void platform_atomic_add64(uint64_t* counter, uint64_t* value, platform_mutex_t* mutex);* *void platform_atomic_add32(uint32_t* counter, uint32_t* value, platform_mutex_t* mutex);*
Arguments	• Counter • Value to be added • Mutex
Result	None.

B.5 ***Notifications [21]***

Function	**of_process_packet_pipeline**
Description	Processes a packet abstraction through the Openflow pipeline.
Declaration	*rofl_result_t of_process_packet_pipeline(const of_switch_t* sw, datapacket_t *const pkt);*
Arguments	• The switch which has to process the packet • Packet abstraction
Result	Success of failure.