OCTEON PCI/PCIe Base Driver Reference Guide

Released Version: 2.2
Last modified: 11/21/2011 3:01 PM

Chapter 1: Overview

OCTEON processors from Cavium Networks can be designed into a host system using PCI, PCI-X or PCI-E bus technology. The OCTEON CN58xx can be used in systems that use PCI-X bus. The OCTEON CN56xx supports PCI-Express Gen1 while CN63xx, CN66xx & CN68xx family supports PCI-Express Gen2 technology.

Note: *This document uses PCI as a general term to refer to all variants of PCI bus technology including PCI-X and PCI-Express. The document makes the distinction between the technologies where required.*

OCTEON is used as a PCI target device when it is used to offload cpu-intensive tasks or when it is used in an inline mode where it sends & receives packets from the external world using its built-in interfaces and transfers those packets to the host like a network device.

The PCI block of all processors in the OCTEON family provide separate hardware queues for transferring data between OCTEON and the host. It also provides several general purpose DMA engines that allow software running on OCTEON to asynchronously transfer data between the host memory and OCTEON memory.

The OCTEON PCI driver runs on the host and is used to configure and communicate with an OCTEON device in PCI target mode. OCTEON has been designed into platforms with all types of host processors.

Cavium Networks ships 3 different linux driver packages that provide different functionality in an OCTEON PCI target device. They are:

1. **OCTEON-PCI-BASE:** The base driver package that configures the target OCTEON device. The base driver sets up its input and output queues, DMA engines and provides various hooks to other kernel modules.

2. **OCTEON-PCI-NIC:** This package makes use of hooks provided by the base package to use OCTEON as a network device.

3. **OCTEON-PCI-CNTQ:** This package makes use of hooks provided by the base package to provide software queues to host applications by using OCTEON's general purpose DMA engines. This package is part of the TCP Offload package offered by Cavium Networks.

Note: *This document covers the OCTEON-PCI-BASE package only for Release version 2.2.0.*

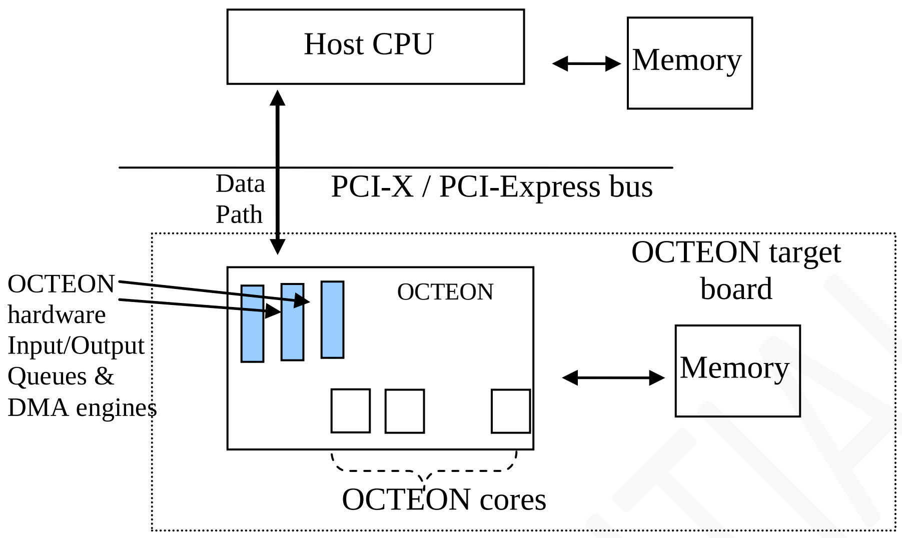


Figure 1: OCTEON target device used for offload operation

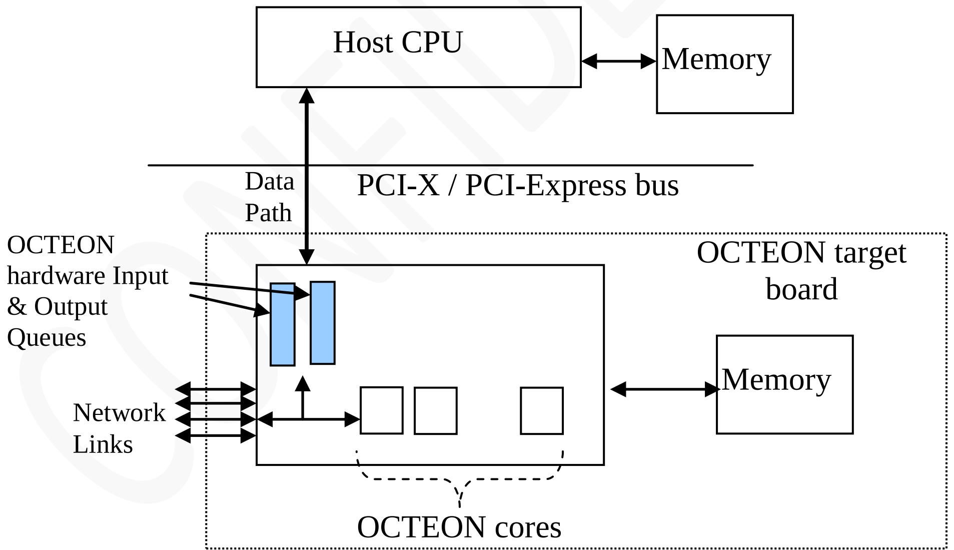


Figure 2: OCTEON target device used for inline operation (NIC)

OCTEON provides the following mechanisms for data transfer across PCI:

1.1. Input rings

Input rings allow data to be pushed from host to OCTEON. The host is required to create a ring in host memory that holds input commands. Commands are 32 or 64-byte instructions that hold address of the data in host memory and a 64-bit field that gives information like length, tag value etc for the data. The first two fields of the command are in a format specified by the OCTEON input queue hardware and hold the data pointer and the Instruction Header (IH). The remaining bytes in a command (16 bytes for a 32-byte command and up to 48 bytes in a 64-byte command) are free to be used by software. The DPTR field can point to a single buffer in host memory or to a gather list which in turn points to one or more buffers in host memory.

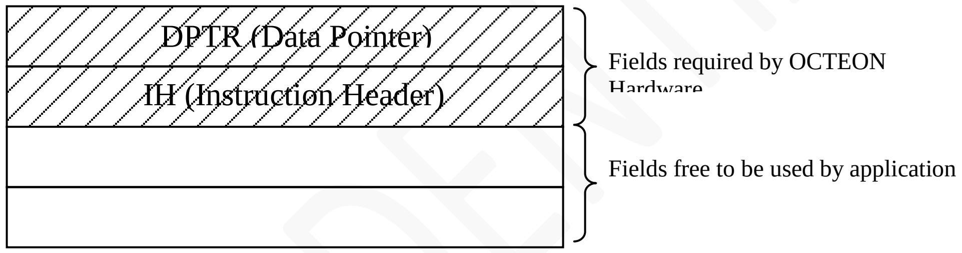

DPTR (Data Pointer)
IH (Instruction Header)
Fields required by OCTEON Hardware
Fields free to be used by application

Figure 3: PCI Input Queue Command structure (32-byte format)

Tip: *The instruction header format and gather list format is described in*
Section 10.3.1/10.3.2 – CN58xx HRM
Section 9.4.1/9.4.2 – CN56xx HRM
Section 19.1.2/19.1.3 – CN63xx HRM
Section 12.1.1/12.1.2 – CN68xx/CN66xx HRM

The input queue ring in host memory is handled as a circular buffer by OCTEON. After fetching the command at the end of the ring, OCTEON automatically circles to the beginning of the ring to fetch the next command. Based on the information in the instructions written by the host, OCTEON can, without further assistance from host, read data from host memory and create work queue entries that can be scheduled and processed on the OCTEON cores. There are 4 input rings in CN58XX and 32 rings available in the CN56XX & newer OCTEON processors. The PCI input ports 32 to 35 correspond to the PCI input rings. For CN58XX, there is one Input ring per PCI input port, while there can be up to 8 rings per port in CN56XX, CN66XX & CN63XX. For CN68xx, the concepts of ports and queues have changed with port numbers 0x100 to 0x11F used as PCI ports with each port associated with a PCI input ring. Note that CN68xx introduces the concepts of Pkinds which are used to identify ports of a common type for programmability. Please refer to the CN68xx HRM for additional details.

Note: *The PCI Input* ***command*** *is also called an* ***instruction*** *in this document. Both refer to the hardware-specified 32 or 64-byte command format. A* ***soft instruction****, on the other hand, refers to a software structure used internally by the driver.*

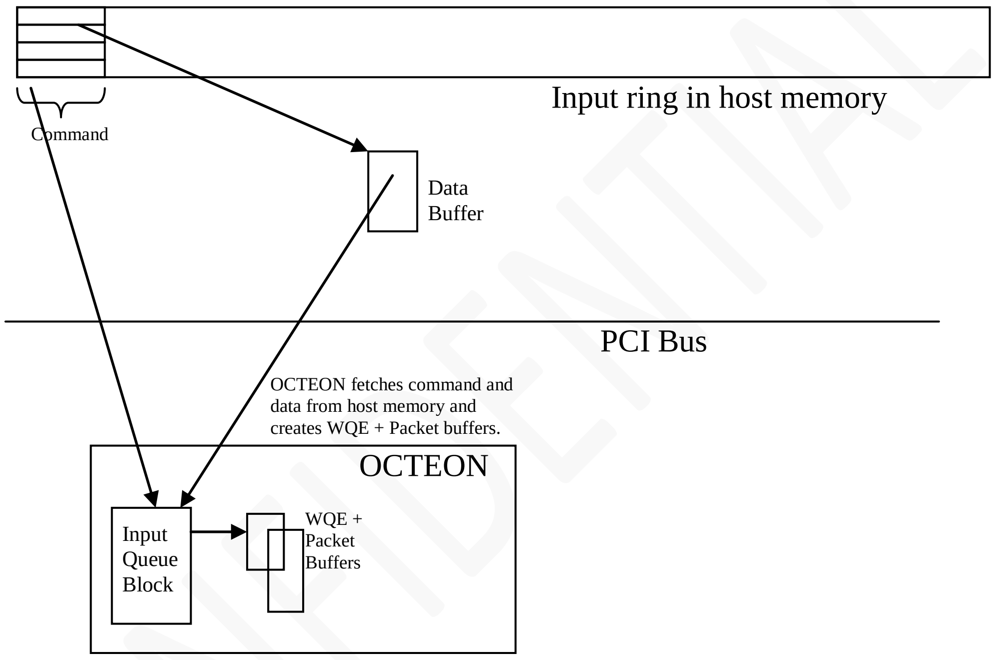

Input ring in host memory
Command
Data Buffer
PCI Bus
OCTEON fetches command and data from host memory and creates WQE + Packet buffers.
OCTEON
Input Queue Block
WQE + Packet Buffers

Figure 4: OCTEON PCI Input mechanism

1.2. Output rings

OCTEON Output rings allow data to be sent from OCTEON to host. The host is required to pre-allocate fixed size buffers and use a descriptor ring to store the address of these buffers. OCTEON PCI output hardware block fetches the descriptor ring contents and uses the buffer addresses to push data in OCTEON memory to these buffers.

Each descriptor in the output descriptor ring is 16-bytes in length. The first 8 bytes of the descriptor holds the address of the buffer in host physical memory. The next 8 bytes holds the address of an info pointer when the descriptor ring is in Info-Pointer mode. OCTEON can use buffers from one or more descriptors to transfer a single packet to host memory.

OCTEON allows descriptor rings to be in one of two different formats:

1. **Info pointer**: In this mode, both the buffer pointer and info pointer are valid and used by the OCTEON output queue mechanism. OCTEON copies the size of the data to the length field in the info pointer of the first descriptor used for a packet. OCTEON can also be configured to write a programmed number of data bytes from the start of the packet to the Info bytes. This allows application to incorporate classification data at the start of a packet when sending it to the host and handle them separately from the data in the host buffers.

Tip: The *format for data bytes written by OCTEON at the Info pointer can be found at*
Section 10.4.1 – CN58xx HRM
Section 9.5.1 - CN56xx HRM
Section 19.2.1 – CN63xx HRM
Section 12.2.1 – CN68xx/CN66xx HRM

2. **Buffer-Only**: In this mode, the descriptor uses the same format as Info-Pointer but the info bytes are ignored by the OCTEON output queue mechanism. OCTEON writes the length to the first 8 bytes of the buffer in the first descriptor used to transfer the data.

Buffer Pointer
Info Pointer

Figure 5: Output Queue descriptor format

The driver always configures the output queue to use Info-Pointer mode by default.

There are 4 output rings in CN58XX and 32 rings available in the CN56XX, CN6XX & CN63XX OCTEON processors. The OCTEON output ports 32 to 35 are used to send packets on the PCI output rings. For CN58XX, there is one output rings per PCI output port, while there can up to 8 rings per port in CN56XX, CN66XX & CN63XX. For CN68xx, the concepts of ports and queues have changed with port numbers 0x100 to 0x11F used as PCI ports with each port associated with a PCI output ring.

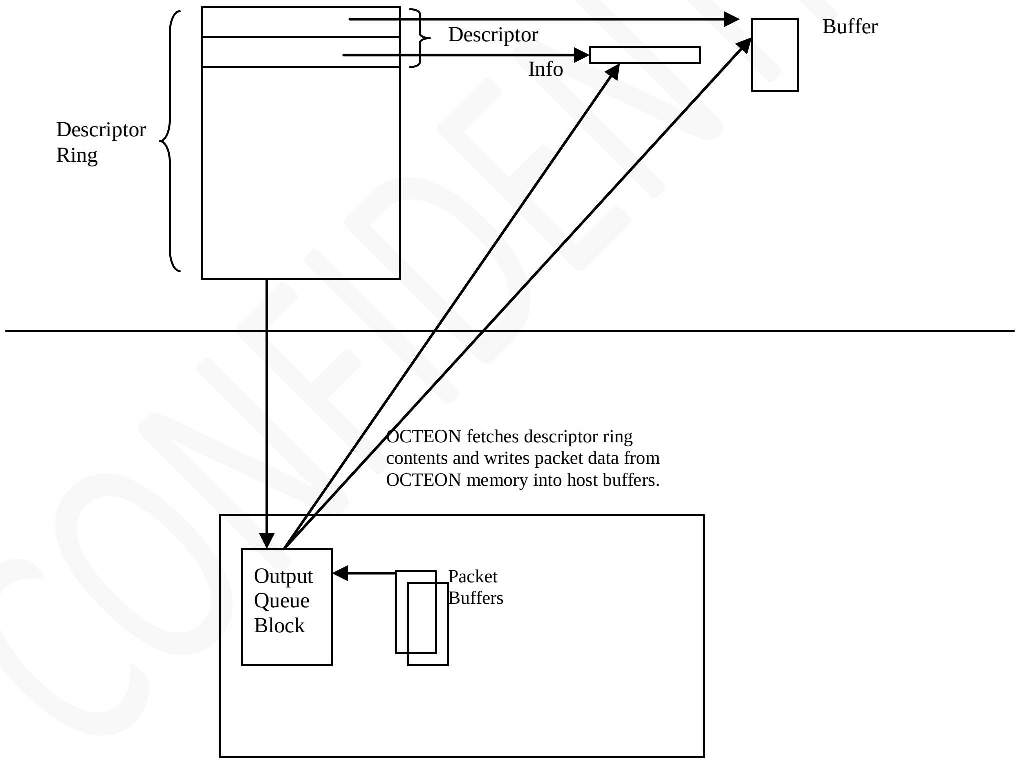

Descriptor
Buffer
Info
Descriptor Ring
OCTEON fetches descriptor ring contents and writes packet data from OCTEON memory into host buffers.
Output Queue Block
Packet Buffers

Figure 6: OCTEON Output queue mechanism

1.3. DMA engines

OCTEON DMA engines allow software running on the cores to directly transfer data between host memory and OCTEON L2/DRAM. The cores submit PCI DMA instructions to the engines which perform the required PCI operations to transfer the data. Unlike the Input & Output queues, the host is not required to do any setup for the OCTEON cores to be able to use the DMA engines. The DMA engines can be instructed to generate interrupts on the host on a command-by-command basis, in which case the host is expected to handle the interrupts.

There are 2 DMA engines in CN58XX, 5 in CN56XX and 6 DMA engines in CN63XX, CN66XX & CN68XX.

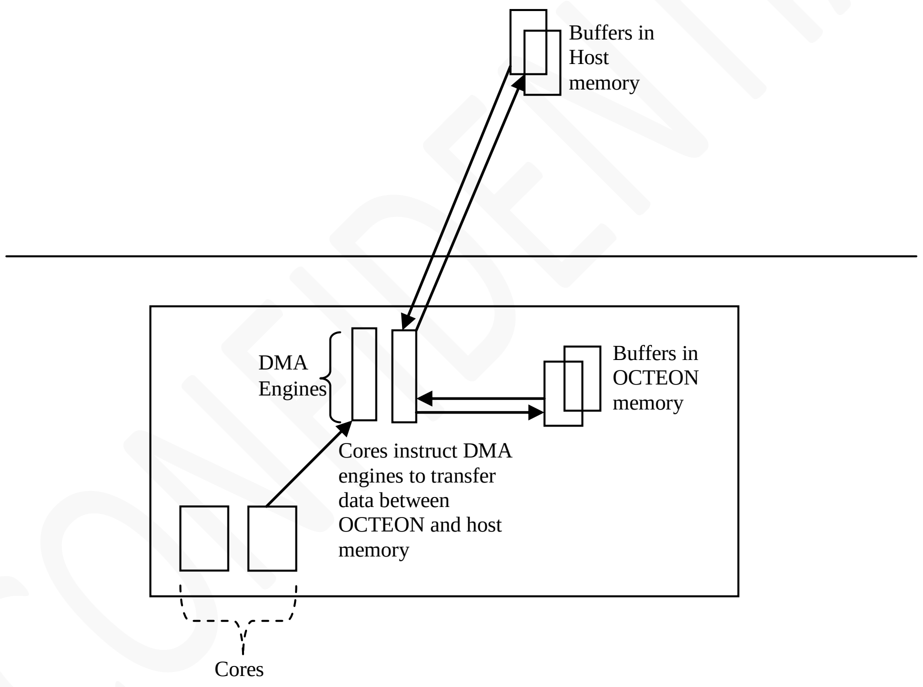

Buffers in
Host
memory
DMA
Engines
Buffers in
OCTEON
memory
Cores instruct DMA engines to transfer data between OCTEON and host memory
Cores

Figure 7: OCTEON DMA engine

The DMA engines in CN58XX allowed two modes of operation:

1. Inbound – where data is read into OCTEON memory from host.
2. Outbound – where data is read from OCTEON memory and written into host.

All processors from CN56XX onwards adds two more modes:

3. Internal – where data is moved internally in OCTEON DRAM from one location to another.
4. External – where data is moved externally from one PCI address region to

another.

CN63XX & CN68XX supports all 4 modes.

Chapter 2: PCI Driver Sources

The OCTEON PCI BASE driver supports the CN58XX, CN56XX, CN63XX, CN66XX & CN68XX OCTEON processors in PCI target mode. It initializes the input and output rings and provides a range of API's that lets kernel and user level applications gain access to the OCTEON device. It also provides hooks that other driver modules use to register their services.

Note: *An end-of-life notice has been issued for CN38XX. While the current release supports CN38XX, all features may not be tested on CN38XX.*

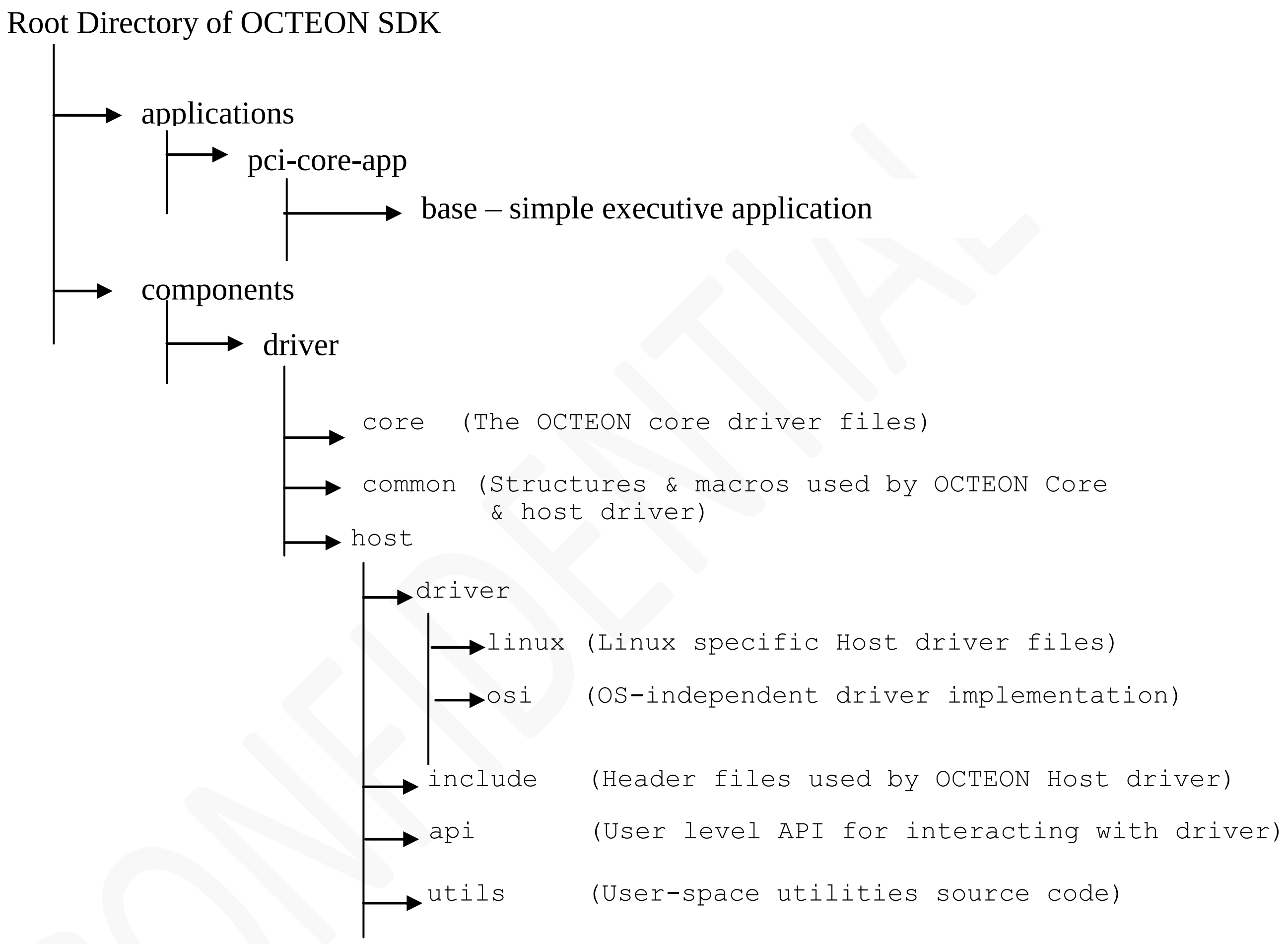
The sources are arranged in a directory structure that allows for easy porting of the driver to different hardware platforms and operating systems. The sources are also arranged in a fashion that easily identifies software that runs on OCTEON cores and those that run at kernel or user level in the host. Most operating system dependent features are accessed using macros that can be easily over-ridden when porting to a different operating system. The portion of the driver sources that runs on the OCTEON cores are bound to a particular version of the OCTEON SDK released separately by Cavium Networks.

Note: *The host and core software also have a handshake mechanism that may change from one release of the driver to another. Using the simple executive application (core software) and host driver from different releases of the PCI driver may lead to incompatibilities.*

The package includes a set of API that can be compiled into a library that user-space applications can use to interact with the OCTEON device driver. The package also provides kernel level API's that can be used by linux kernel mode applications to interact with the OCTEON device driver. The kernel API also includes hooks into the driver to develop additional functionality using the underlying base driver features.

The driver package also provides a set of API (referred to as the core driver in this document) that applications designed to run on the OCTEON cores can use to communicate with the PCI blocks. The OCTEON core driver API's make use of the simple executive routines provides by the OCTEON-SDK package for all interactions with the PIP/IPD and PKO blocks.

The sources are usually installed within the OCTEON SDK source directory which usually resides at /usr/local/Cavium_Networks/OCTEON-SDK.
The directory hierarchy of the driver sources is shown in the figure below.


Root Directory of OCTEON SDK
applications
pci-core-app
base – simple executive application
components
driver
core (The OCTEON core driver files)
common (Structures & macros used by OCTEON Core & host driver)
host
driver
linux (Linux specific Host driver files)
osi   (OS-independent driver implementation)
include   (Header files used by OCTEON Host driver)
api   (User level API for interacting with driver)
utils   (User-space utilities source code)

Chapter 3: Initialization

The OCTEON Host PCI driver is responsible for:

- Initialization of various OCTEON PCI blocks including Input/Output rings and DMA engines.
- Receiving requests from user and kernel space applications and forwarding them to OCTEON.
- Managing different requests types while they wait being fetched by OCTEON or till a response arrives from OCTEON.
- Receiving packets from OCTEON via the output rings and dispatch them to kernel applications.
- Managing various hooks that other kernel modules (like NIC) can call to change the default packet processing behavior.

The OCTEON driver detects the OCTEON devices present in the system by making OS-dependent calls. The device driver is capable of handling multiple OCTEON devices within a single instance of the driver. The maximum number of OCTEON devices supported is controlled by a compile-time definition (MAX_OCTEON_DEVICES).

The driver currently claims the following OCTEON devices:

- CN38XX Pass2
- CN38XX Pass3
- CN58XX Pass1 & Pass2
- CN56XX Pass2.1
- CN63XX Pass1.2
- CN66XX Pass1.1
- CN68XX Pass1.1

For each OCTEON device it finds in the system, it maps the PCI BAR required to access OCTEON's CSR registers into virtual memory and performs a soft reset of the OCTEON chip by writing to an OCTEON internal register. A successful reset brings all the OCTEON hardware blocks and all PCI registers to a known state.

After a soft reset, the driver sets various registers that affect OCTEON's behavior as a PCI target and sets up the PCI Input & Output rings. The DMA engines are setup by the core driver component that runs as part of the application running on the OCTEON cores.

The driver also sets up various software queues and lists as part of its infrastructure to manage kernel and user-space application interactions. The next few sections will cover the initialization of each of the blocks.

3.1 Device configuration information

See OCTEON_config.h/OCTEON_main.c

The configuration for an OCTEON device is kept in a statically initialized structure. There are separate structures for each OCTEON processor family. Many of the attributes that are common to all families are kept in a common structure defined by ***OCTEON_common_config_t***. Parameters unique to each processor family are kept in separate configuration structures. For e.g. cn56xx_config_t has the configuration for the CN56xx processor family.

The parameters that can be configured are:

- number of input and output rings
- the size of each input and output ring,
- size of the output buffers
- size of the PCI instruction in the input ring
- parameters that affect input & output ring processing and interrupt handling.

Tip: *The set of registers accessed during initialization of the PCI blocks can be found in*
Section 10.12 "PCI Checklist" - CN58xx HRM
Section 9.14 "PCIe Configuration Summary" - CN56xx HRM
Section 19.9 "Configuration Summary" – CN63xx HRM
Section 12.9 "Configuration Summary" – CN68xx/CN66xx HRM

3.2 Input ring initialization

See request_manager.c - OCTEON_init_instr_queue()

The driver reads in the input ring configuration from a configuration structure that describes among other things, the number of input rings to set up, the number of instructions in each ring, the instruction type to use for each ring. A structure of type ***OCTEON_instr_queue_t*** is maintained by the driver to manage each input ring. For each input ring, the driver allocates a block of physically contiguous memory that represents the instruction ring in host memory. The size of this block depends on the instruction type and the number of entries for this ring. By default, the driver uses a 32-byte instruction type for all input rings and keeps input ring 0 at 1024 entries and all other rings to 128 entries. It programs the base address of the instruction ring and its size (in number of entries per ring) into OCTEON CSR registers.

The input rings are enabled by writing a bit mask to a register. Setting a bit to 1 enabled the corresponding input ring for operation. The instruction size is also entered as a bit mask in a separate register. Setting a bit to 1 instructs OCTEON that the ring uses 64-byte instruction. For 32-byte instructions, the bit is set to 0. The driver uses 32-byte instructions by default.

For all OCTEON devices, the driver also sets up the Endian-swap, Relaxed-Ordering and NoSnoop settings. These settings are not per-ring but affect all the rings. The default setting by the driver enables 64-bit endian swapping and disables relaxed ordering and NoSnoop operations.

For CN56XX & newer processor families the driver has to setup the PCI-E port that the input rings are attached (the default is port 0).

3.3 Output Ring Initialization

See OCTEON_droq.c - OCTEON_init_droq()

The driver reads in the output ring configuration from a configuration structure that describes the number of output rings to set up, the number of descriptors in each ring, the buffer size to use and the interrupt coalescing settings. A structure of type ***OCTEON_droq_t*** is maintained by the driver to manage each output ring. For each output ring, the driver allocates a block of physically contiguous memory that represents the descriptor ring in host memory. The size of this block depends on the number of entries for this ring. By default, the driver uses info pointer mode for all output rings and keeps output ring 0 at 1024 entries and all other rings to 128 entries. It programs the base address of the descriptor ring and its size (in number of entries per ring) into OCTEON CSR registers.

Buffer management for each output ring is also the driver's responsibility. At initialization time, the driver allocates a buffer for each entry in the descriptor ring using OS-specific buffer allocation routines (for Linux the driver uses skbuff for output ring buffers). Each buffer is 2048 bytes by default.

The driver configures OCTEON devices to interrupt the host when packets have been posted to the descriptor ring buffers in host memory. Though OCTEON devices support both packet-count & time based interrupt, the driver programs the devices to interrupt the host using time based interrupt at intervals of 100 microseconds. This allows predictable packet processing times on the host.

3.4 DMA Engine Initialization

The DMA engines are initialized by the core driver during the global initialization step of the OCTEON application. The host driver does not participate in the initialization process though it does setup the interrupt registers for CN38xx/CN58xx. For CN56XX, the core driver takes care of this task as well.

Tip: *The chunked list buffer management for OCTEON DMA engines is explained in*
Section 10.5.4 - CN58xx HRM
Section 9.6.4 - CN56xx HRM
Section 19.3.4 – CN63xx HRM
Section 12.3.4 – CN63xx/CN66xx HRM

The DMA engines fetch their instruction from a chunk list of buffers. The core driver allocates the required buffers and programs the start address of the first buffer in the list and the size of the buffer for each DMA engine. It also creates internal structures to maintain the state of each DMA engine.

The DMA engines can notify the completion of a DMA command in two ways:

- Issue a work-queue entry to the POW.
- Write 0 to a memory location.

The core driver supports both modes. For the first mode, the driver expects the caller to provide the work-queue entry. The driver provides a pool of completion words that applications can use to allocate a completion byte if it chooses to use the second mode. This pool of completion bytes is also allocated at initialization time.

For CN58xx, there were only 2 DMA engines and the driver would use the DMA engine number as the counter index. CN56xx & newer processor families have only 2 DMA counters though they have more DMA engines. For these processors, the driver continues to support use of the DMA counters for the first two DMA engines only.

3.5 Other Initialization tasks

There are additional data structures maintained by the driver for different tasks. They are described in brief below and will be covered in more detail in the following sections.

Internal Lists
The driver also creates internal lists for managing requests from user & kernel applications. All requests that require a response from the application on OCTEON target are added to a pending list. The driver also uses several response lists to manage requests of different types (like ORDERED, UNORDERED).

Poll Lists
The driver creates a kernel thread for managing tasks that should run at regular intervals. It provides a set of functions that allows kernel applications to register functions that should be called by the poll thread. The driver has its own tasks that are also registered with the poll thread.

Dispatch Management
The driver maintains a lookup table of functions and opcodes for packets arriving on the output rings. Kernel applications can register a function to be called by the driver when a packet arriving on the output ring with a particular opcode.

Note: The kernel API (section 7.2) covers the different hooks provided by the driver for kernel applications.

Chapter 4: Input ring packet processing

The PCI driver builds PCI commands on behalf of applications that have data to send to the OCTEON target device. By default, the PCI command used by the driver is 32 bytes in length.

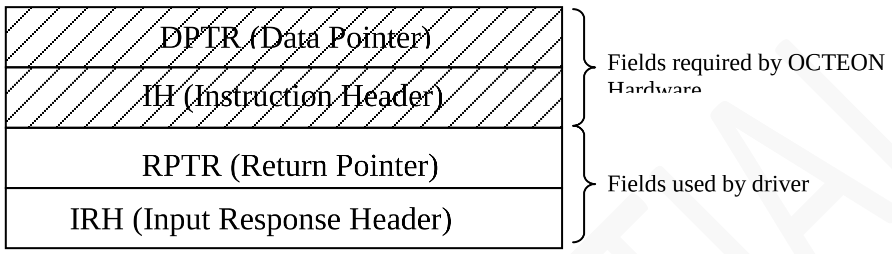


Figure 8: Input Queue Command structure used by driver.

For each command, the driver sets up the 32 bytes as follows:

1. First eight bytes to pass address of data in host memory. The address can be of a gather list if the data is being passed in multiple buffers. The address points to the data buffer directly if data is in a single buffer.

2. Second eight bytes to pass the instruction header (in a hardware defined format) by filling in the tag, tag-type, QOS, group settings as directed by the host application (more details in the section below). The gather bit is set by the driver if multiple buffers are used and the application requested the use of gather support available in the OCTEON Input rings. The dlengsz field holds the total bytes of data, if data exists in a single buffer or the number of gather list buffers if multiple buffers are used.

   Note: CN63xx introduces a new mode of gather where up to 4 gather pointers can be transferred in the Input ring command itself. In this mode, the gather bit is 1, but dlengsz is 0. The length of each gather buffer is kept within a 64-bit word (with 16 bits for each length field). The first 8-byte word in the Input Ring command points to the first gather buffer. More details can be found in Section 19.1.1 - **DPI Instruction Formats** of the CN63xx HRM.

3. Third eight bytes to pass an 8-byte address of host memory (called RPTR) where the result data can be written by the OCTEON target. If no response is expected, this field is left empty.

4. Fourth eight bytes to pass information regarding the response. The format of this 8 byte field (called IRH) is shown below. The rlenssz field contains the size of the data buffer available at RPTR above if a single buffer is used for response. If multiple buffers are used for response, the rlenssz field contains the number of buffers. The request id contains the index in the pending list of the driver where the request is held by the driver till the OCTEON target application responds.

```
Opcode | param | destination port| rlenssz| scatter| request id

63    48|47   40|39              34|33    20|   19   |18        0
```

Figure 9: Format of Input Response Header used by driver.

Tip: *The format used for sending the address and length of response buffers by the driver when the response is expected in multiple buffers is the same as the gather list format used in Input Rings (See section 1.1 above or click here).*

4.1 Sending requests to OCTEON target

Host applications make use of the Input rings to send data to the application running on OCTEON target device. Applications running in kernel space make use of the **OCTEON_send_request()** API exported by the driver to send data on the OCTEON Input rings. The driver provides an ioctl for user-space applications to send requests to OCTEON.

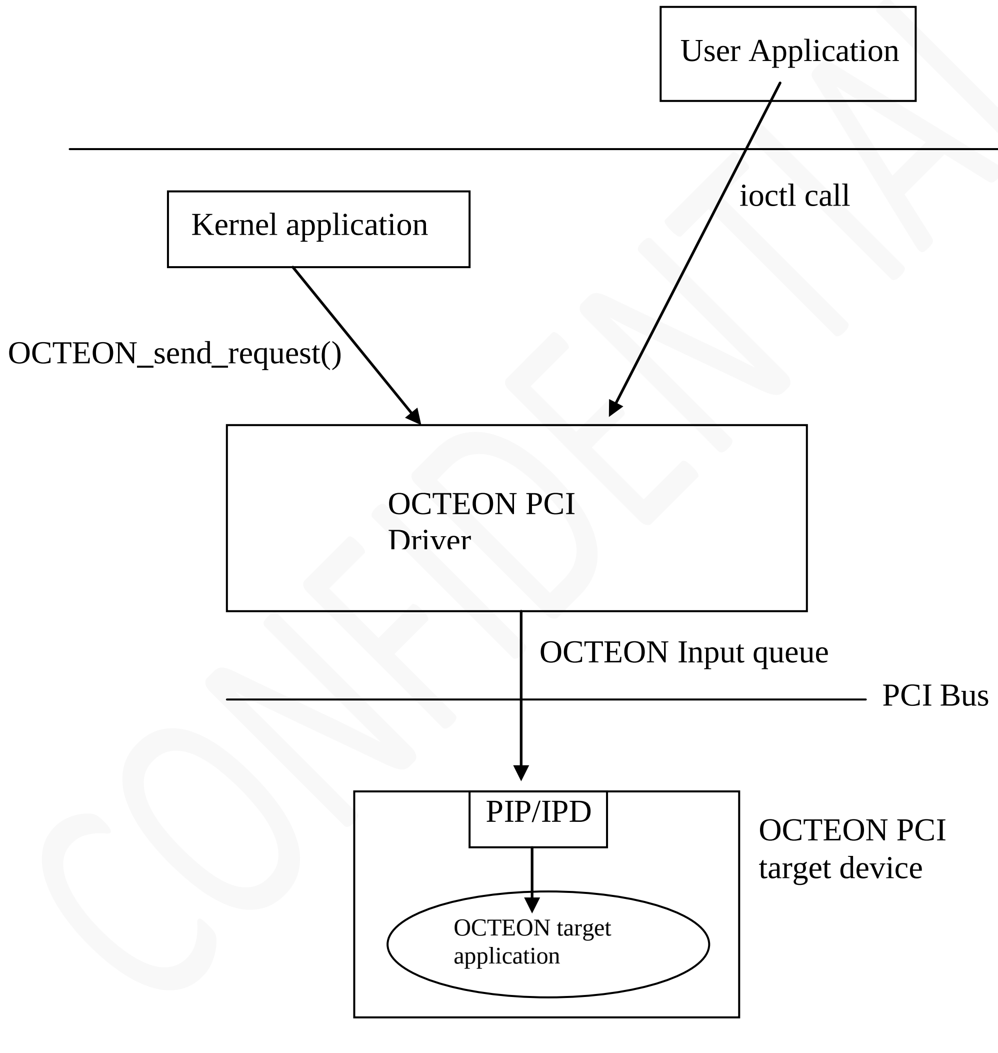


Figure 9: Flow of request from host application to OCTEON target application.

4.2 Software Request Attributes

Requests from applications have two attributes that identify the type of processing that the driver should do with the request:

1. Response Order:
Response Order determines if and how the driver should wait for a response from the OCTEON target application for this request.

Response Order can be:
A. Unordered – Requests of this type can complete execution in any order i.e. a request can get a response and therefore complete execution ahead of requests that were sent before it.
B. Ordered – Requests of this type always completes in order i.e. requests always complete execution in the order they were received by the driver.
C. Noresponse – Requests of this type do not expect a response from the OCTEON target application and completes execution as soon as its data is sent to the OCTEON target application.

2. Response Mode:
UNORDERED requests have a Response Mode attribute that determines if the application's call to the driver returns before a response arrives for the request.

Response Mode can be:
A. Blocking – This type of call returns only after a response arrives from the OCTEON target application.
B. NonBlocking – This type of call can return before a response arrives for the request. The application will have to query the driver later to check if a response arrived.

ORDERED requests are always considered non-blocking. Each request has a callback that is called by the driver as the requests complete in-order. Since a callback is required, ORDERED requests are only sent from kernel applications. The callbacks are guaranteed to be called in the order that the corresponding requests were posted to the driver.

UNORDERED requests can be sent by both kernel and user space applications, though usually only user-space applications make use of them. Kernel space applications run at higher priority than user-space applications. They should avoid using UNORDERED requests since the mechanism for request completion use up a lot of CPU time waiting for response (in BLOCKING mode) or in multiple queries to the driver (in NON-BLOCKING mode). UNORDERED requests are efficient for user-space applications. In NON-BLOCKING mode they allow for user space applications to send multiple requests while awaiting a response for previously sent requests. User-space requests can make blocking requests for faster response times.

NORESPONSE requests can also be sent by both user and kernel space applications. NORESPONSE requests from kernel applications are allowed to have a callback. The callback is called by the driver only after the data for the request has been fetched by the OCTEON hardware and allows applications to complete house-keeping tasks with the knowledge that the data buffers won't be accessed by the hardware.

4.3 Software request DMA modes

Applications can use several data transfer modes to send and receive data. The DMA mode in the soft request determines the way the input data is presented to the OCTEON Input Queue hardware and also how the output buffer pointers are made available as part of the instruction sent to OCTEON. There are 4 DMA modes that determine how the Instruction Header (IH), DPTR, Input Request Header (IRH) and RPTR fields are generated.

1. **DIRECT:** Driver uses a single input buffer. The DPTR field carries the address of the input data buffer. The dlengsz field in IH carries the length of data in input buffer. The RPTR field carries the address of the output buffer. The rlenssz field in IRH carries the size of the output buffer.

2. **GATHER:** Driver allows multiple input buffers. For CN56xx and older processors, the DPTR field carries the address of a gather list for indirect gather which in turn holds the address and size of each input buffer. The dlengsz field in IH carries the count of input buffers. The RPTR field carries the address of the output buffer. The rlenssz field in IRH carries the size of the output buffer.

3. **SCATTER:** Driver uses a single input buffer. The DPTR field carries the address of the input data buffer. The dlengsz field in IH carries the length of data in input buffer. Driver allows multiple output buffers. The RPTR field carries the address of a gather list which in turn holds the address and size of each output buffer. The rlenssz field in IRH carries the count of output buffers.

4. **SCATTER_GATHER:** There are multiple input and output buffers. The DPTR field carries the address of a gather list which in turn holds the address and size of each input buffer. The dlengsz field in IH carries the count of input buffers The RPTR field carries the address of a

gather list which in turn holds the address and size of each output buffer. The rlenssz field in IRH carries the count of output buffers.

User-space and kernel-space applications can use all 4 modes in their requests but there are some restrictions on how the buffers are presented to the driver.

The following restrictions apply to buffers sent by user-space applications for each of the above DMA modes.

Note: *Look at components/driver/host/include/cavium_defs.h for all constants used below.*

1. **DIRECT DMA:** Multiple input buffers are allowed only if the sum of the sizes does not exceed OCT_MAX_DIRECT_INPUT_DATA_SIZE. Multiple output buffers are supported provided the sum of the sizes does not exceed OCT_MAX_DIRECT_OUTPUT_DATA_SIZE.

2. **GATHER DMA:** Multiple Input buffers are allowed and the sum of the sizes should not exceed OCT_MAX_GATHER_DATA_SIZE. Multiple output buffers are supported provided the sum of the sizes does not exceed OCT_MAX_DIRECT_OUTPUT_DATA_SIZE.

3. **SCATTER DMA:** Multiple input buffers are allowed only if the sum of the sizes does not exceed OCT_MAX_DIRECT_INPUT_DATA_SIZE. Multiple output buffers are allowed and the sum of the sizes should not exceed OCT_MAX_SCATTER_DATA_SIZE.

4. **SCATTER_GATHER DMA:** Multiple Input buffers are allowed and the sum of the input buffer sizes should not exceed OCT_MAX_GATHER_DATA_SIZE. Multiple output buffers are allowed and the sum of the output buffer sizes should not exceed OCT_MAX_SCATTER_DATA_SIZE.

The fact that the driver allows multiple input/output buffers even when the DMA mode allows for single buffers only is a side-effect of an optimization in the driver. When copying data in and out of user-space, the driver attempts to allocate a single buffer in kernel memory. So, as long as the size restrictions are satisfied, the driver can manage the multiple data buffers and still create the PCI command fields in the manner requested by the DMA mode in the request.

Let's see a couple of examples to understand this behavior.

Example 1:
If the request used GATHER dma mode and presented two buffers, the first of size 8000 bytes and the second of size 12000 bytes, the driver would allocate one large buffer of 20000 bytes in kernel memory to copy in the input data, but still create a gather list, with the first address at the start of the kernel buffer and the second at a 8000 byte offset in the kernel buffer. The dptr field would point to the gather list (which, too, was allocated in kernel memory by the driver).

Example 2:
If DIRECT dma mode was used and two buffers of size 4000 and 8000 were used, the driver would allocate a single buffer again and copy in all 12000 bytes from user-space, but dptr in this case would point directly to the start of the buffer in kernel memory.

For kernel mode applications, the restrictions are more stringent.

1. **DIRECT DMA:** Multiple Input buffers are allowed only if the sum of the sizes does not exceed OCT_MAX_DIRECT_INPUT_DATA_SIZE. Multiple output buffers are not supported. Maximum Output data size is OCT_MAX_DIRECT_OUTPUT_DATA_SIZE.

2. **GATHER DMA:** Multiple Input buffers are allowed. Input buffers are put into a gather list and the sum of their sizes cannot exceed OCT_MAX_GATHER_DATA_SIZE. Multiple output buffers are not supported. Maximum output data size cannot exceed OCT_MAX_DIRECT_OUTPUT_DATA_SIZE.

3. **SCATTER DMA:** Multiple Input buffers are not allowed. Maximum input data cannot exceed OCT_MAX_DIRECT_INPUT_DATA_SIZE. Output buffers are put into a scatter list and the sum of their sizes cannot exceed OCT_MAX_SCATTER_DATA_SIZE.

4. **SCATTER_GATHER DMA:** Multiple Input or Output buffers are allowed. Input buffers are put into a gather list and the sum of their sizes cannot exceed OCT_MAX_GATHER_DATA_SIZE. Output buffers are put into a scatter list and the sum of their sizes cannot exceed OCT_MAX_SCATTER_DATA_SIZE.

Note: *Kernel applications are required to reserve 8 bytes at start of the output buffer for Response Header and 8 bytes at end for status bytes. Thus, the actual output buffer size constraint is the maximum data size mentioned above + 16 bytes. For example, to receive the maximum DIRECT DMA data size of 16367 bytes, the application should allocate a buffer of size 16383 bytes.*

4.4 The software request structure

Applications make use of the ***OCTEON_soft_request_t*** structure to encapsulate information about the request when they call the driver. Both user and kernel applications make use of the same request structure when sending a request to the driver. The structure definition can be found in components/driver/host/include/cavium_defs.h.

The following fields must be filled by the application calling **OCTEON_soft_request():**

- **inbuf** - set the number of input buffers, address and size of each input buffer. It is legal to have 0 input buffers. Applications can send 0 bytes when all the information they need to send is encapsulated in the software defined fields, IRH and RPTR.

- **outbuf** - set the number of output buffers, address and size of each output buffer. It is legal to have 0 output buffers if no response is expected. Output buffers must be allocated if a response is expected. Kernel applications must allocate an additional 16 bytes to accommodate the response header and status of a response. For user-space applications, driver will allocate the extra bytes.

- **ih** - instruction header bits. Refer to the OCTEON Hardware Manual (Section 10.3.1 of CN58XX HRM, Section 9.4.1 of CN56XX HRM, Section 19.1.1 of CN63xx HRM, Section 12.1.1 of CN68xx HRM) for the format and usage.

- **irh** - The opcode field should be set. Optionally the param and dport fields can also be set. All other fields are set by the driver and should not be set by the application. See driver/common/OCTEON-opcodes.h for opcodes reserved for use by driver and applications released by Cavium.

- **exhdr** – up to 4 additional 64-bit values can be passed which will be added to the front of the input data bytes.

- **exhdr_info** - only the exhdr_count field needs to be set. It gives a count of the number of 64-bit values present in the exhdr field.

- **req_info** - The following field needs to be set by application.

- **OCTEON_id** - OCTEON device id.

- **req_mask** - Set the response order, response mode, dma mode and instruction ring to use.

- **timeout** - time in millisecs to wait for response.
- **callback** - should not be set by user space application. Kernel applications can set it to a function pointer.
- **callback_arg** - additional argument to be passed to callback by driver. Should not be set by user application.
- **status** - Should not be set by any application. Driver will return status in this field.
- **request_id** - Should not be set by any application. Driver will return request id after request is posted to OCTEON.

If the call returns success, the current state of the request will be in the req_info.status field of the request.

4.5 Response format

The response sent by the core application to any request is expected in the following format:

```
+----------------------------------+
| Response Header (8 bytes)        |
|                                  |
+----------------------------------+
| Response Data (0 or more         |
| bytes)                           |
+----------------------------------+
| Status word (8 bytes)            |
|                                  |
+----------------------------------+
```

Figure 10: Response buffer format

Kernel applications should take care that the response (output) buffer is created in the above format. For user-space application, the driver builds the buffer in the required format.

```
 Opcode | Source Port | Destination Port | Reserved |   Request ID
63   48 47         42 41               20      19      18           0
```

Figure 11: Response Header format

```
 Reserved    | Application Identifier | Request Status  | DMA status
63        32 31                     16 15              8  7          0
```

Figure 12: Status word format

Only the lower 4 bytes of the status word currently hold the completion status. The upper 4 bytes are reserved. Of the lower 2 bytes, the most significant 2 bytes is used as an application identifier. The least significant 2 bytes hold a 16-bit status value.

Note: *The last 8 bits of status word is used as a DMA completion indicator by the driver. Applications have the 24-bits as shown above for their* *use. If more than 255 error conditions can occur in an application, it can reserve multiple application identifiers. The application identifier value 0 is reserved for the driver.*

The OCTEON host driver looks at the 8 byte status word for completion of a request. When a request is sent, the driver initializes it to all f's. The driver keeps checking that byte[0] of status word changes from 0xff to any other value to consider the request completed.

Note: *It is illegal for core applications to set byte[0] of the status word to 0xff.*

4.6 Response lists

The driver maintains response lists to manage requests of different types. There is one list for each of the following:

1. Noresponse.
2. Ordered
3. Unordered-blocking
4. Unordered-nonblocking.

With the exception of the Noresponse list, all other response lists are doubly-linked lists that keep track of requests of the corresponding type while they await a response from the OCTEON target application. The Noresponse list is an array, one for each instruction ring, with space equal to the number of entries in the instruction ring. Entries are cleared from the Noresponse list as soon as the data for the request is fetched by the OCTEON hardware.

To speed up the response time, the core driver library (compiled as part of the OCTEON target application) may use interrupts when a response is sent from the target application. The host driver will check the unordered-blocking and ordered response lists when the interrupt arrives. This reduces the time the threads spend in kernel-mode awaiting a response and improves system response time at the cost of additional interrupt handling.

4.7 OCTEON device file

Before any user applications can start using the OCTEON driver, a device file should be created. The OCTEON device file is a character device with major number 127 named /dev/OCTEON_device. The OCTEON user api library (liboctapi.a) looks for this device file.

There is only one device file irrespective of the number of OCTEON devices present. The device id of the OCTEON device to which a user-space API is directed is passed as an argument to the user-space API. The first OCTEON device in the system has a device id of 0, the second has device id 1 and so on.

Once the device file has been opened, there are several ioctl's that provide read/write access to the OCTEON registers and allow requests to be sent to the application running on OCTEON.
The OCTEON_SDK/components/driver/host/include/OCTEON_ioctl.h file lists the ioctls supported by the driver. The octapi library provides wrappers for all ioctls. The preferred method though is to use the octapi library. The header file OCTEON_SDK/components/driver/api/OCTEON_user.h lists the API functions available from the library.

The API's are described in the section 7.1.

Chapter 5: Output Ring Packet Processing

Output rings are a mechanism of pushing data out of OCTEON to a host device. The OCTEON output rings are configured in info-pointer mode by default. Output ring packet processing is triggered by an interrupt from OCTEON. The driver sets up OCTEON to interrupt at 100 micro-second interval if at least one packet has been sent to the host output rings.

The interrupt handler code is unique to each OCTEON processor families, but in all cases, the interrupt would schedule a tasklet after performing all device-dependent operations and making a note of the number of packets to process in each output ring. The tasklet routine is the same for all OCTEON devices.

The driver configures OCTEON's registers to notify the host about the number of packets it has dispatched on each output ring.

Note: *It is also possible to program OCTEON's registers to notify the host about the number of bytes (instead of packets) it has sent. The driver does not use this option.*

Packets can span multiple buffers. The driver looks at the length field written by OCTEON in the info pointer of the first descriptor to determine the number of descriptors that were used for this packet.

Receiving packets from OCTEON

The driver processes one output ring at a time starting with ring 0, followed by ring 1 and so on. The content of the packet count register (PCI_PKTS_SENT for CN58XX; NPEI_PKTn_CNTS for CN56XX; SLI_PKTn_CNTS for CN63xx/CN68xx) determines the number of packets that the driver attempts to process in an output ring. The driver guarantees ordering of packets only within a given ring. It is possible for packets sent to different rings to be dispatched out of order with respect to packets from other rings. To guarantee ordering of packets within the host application, packets of the same type should be sent to the same ring from the core application.

The driver uses an 8-byte field in the info-pointer to allow applications on the OCTEON target to specify the packet type. This is the default mechanism provided by the driver. The 8-byte field has the following format:

```
Opcode |Source Port | Destination Port | Reserved |Request Id

63   48 47        42 41               20      19     18          0
```

Figure 13: Info Pointer packet header format

The Info Pointer header is represented by the ***OCTEON_resp_header_t*** structure in the driver sources. The driver allows kernel mode applications to register a callback function for a particular opcode. The driver will call the function whenever a packet arrives with that opcode in the packet header. The packet buffers and its related information are passed to the callback function after being wrapped up in an ***OCTEON_recv_info_t***. The driver also allows packets to be processed on a fast path by allowing kernel applications to register a function pointer with the driver for each output ring. The driver skips looking up the opcode and creating a recv_info wrapper for the packet for all packets arriving on this output ring once the appropriate API to enable fast path processing has been called. It is the responsibility of the application to free the packets that it receives from the driver. The driver provides appropriate wrappers that applications can call to free the packet buffers.

5.1 Packet Processing

Output packet processing can take one of two paths:

5.2.1. Slow path packet processing.

This is the default path taken by the driver. For each incoming packet, the driver does a lookup of the opcode found in the Info-pointer header to check if a function is registered for the opcode. If there is no function, the packet is not processed further and the buffers are re-used by the driver. If a function exists, the buffers are removed from the output ring and dispatched to the registered function by encapsulating them in an ***OCTEON_recv_info_t*** structure. The descriptors are marked empty so that the refill procedure can allocate new buffers for these descriptors.

The ***OCTEON_recv_info_t*** structure has the following format:

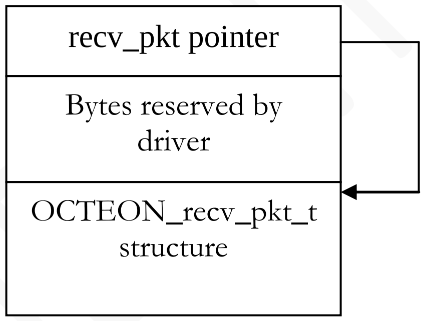


Figure 15: Format of the data structure pushed by driver to application.

As seen in above figure, ***OCTEON_recv_info_t*** acts as a wrapper to the ***OCTEON_recv_pkt_t*** structure. Applications should always look at the recv_pkt structure. The remaining fields in ***OCTEON_recv_info_t*** are for the driver's internal use and can be changed without notice.

```
typedef struct {

  uint16_t          OCTEON_id;
  uint16_t          buffer_count;
  uint32_t          length;
  uint32_t          buf_type;
  OCTEON_resp_hdr_t      resp_hdr;
  void             *buffer_ptr[MAX_RECV_BUFS];
  uint32_t          buffer_size[MAX_RECV_BUFS];

} OCTEON_recv_pkt_t;
```

Figure 16: Format of the OCTEON_recv_pkt_t structure

Explanation of the fields

- **OCTEON_id** - holds the id of the OCTEON device on which the packet was received.
- **buffer_count** - number of buffers used by this packet.
- **length** - total number of bytes in this packet.
- **buf_type** - type of buffer contained in the buffer ptr's for this packet.
- **resp_hdr** - this is the 8 bytes from the info pointer that includes the opcode for the packet.
- **buffer_ptr** - Pointer to the OS-specific packet buffer
- **buffer_size** - Size of the buffers pointed to by pointer's in buffer_ptr

Depending on the type (given by the **buf_type** field), the buffer pointer may or may not point to actual data. If the **buffer_ptr** field does not point to data, then the required wrappers should be used. For example, on linux the driver uses skbuff for output ring buffers. The **buffer_ptr** holds the skb pointer. Applications should call the get_recv_buffer_data() function to get the skb->data pointer.

Packets are never dropped in the slow path. If a function is registered for an opcode, it is guaranteed to receive all packets with this opcode in the order they were sent by the core application.

5.2.2. Fast path packet processing.

Fast path packet processing is triggered when an application registers a fast path function using the **OCTEON_register_droq_ops()** kernel mode API. Application can register a fast path function for each output ring from which it is interested in receiving packets.

Once a fast path function is registered, the driver ignores the opcode in the info pointer and all packets arriving on the ring is sent directly to the fast path function.

The fast path function is called with the following parameters:

1. **OCTEON_id** - id of OCTEON device that sent this packet.
2. **buffer** - pointer to a buffer. This buffer is OS-dependent. For example, on linux this pointer points to a skb. Application should call the get_recv_buffer_data() function to get the skb->data pointer.
3. **size** - size of data in the buffer
4. **resp_hdr** - The 8 bytes in the info pointer. The driver does not interpret this field, but forwards it nevertheless to the fast path function. This allows the core application to attach information identifying the contents of the packet.

Only one buffer is pushed to the function. If a packet spans multiple buffers, the driver copies the data from all buffers into a new buffer and sends the new buffer to the function.

The driver may not process all the packets in the fast path. One of the parameters passed by the application calling the OCTEON_register_droq_ops() tells the driver if it should limit the number of packets that will be processed in every invocation of the tasklet. If this parameter is set & a certain number of packets (determined by the output ring configuration in oct_config_data.h) are reached, the remaining packets may be dropped. The driver will re-use the descriptors of the packets that were dropped to save time in refilling the ring.

5.2 Refilling the output ring descriptors

At regular intervals, the driver attempts to refill the buffers consumed during normal packet processing. It does this for each output ring when the packet processing has traversed a certain number of descriptors since the last refill (that number is given by the refill threshold in the configuration for this ring in oct_config_data.h). Not all descriptors may be in need of a new buffer, since descriptors for packets which have been dropped (in fast path) or for which no registered functions were found (in the slow path) would still have the old buffer address that can be re-used. For all descriptors in need of a buffer, it allocates new buffers and updates the descriptor ring with the physical address of the new buffers. Once all descriptors have a buffer, the driver asks OCTEON to fetch the descriptors by ringing the doorbell for the respective ring.

Note: *Also see the section in the core driver reference on how a simple executive application running on OCTEON can use the PKO to push packets to the PCI output rings.*

Chapter 6: Using PCI Interface from an OCTEON application

Simple executive (SE) applications running on OCTEON use API's provided by the OCTEON core driver to interact with the PCI block. The core driver is linked in as a library during compilation by simple executive applications.

6.1 Initializing the PCI block in a SE application

The PCI block needs to be initialized before the SE can start sending or receiving data across the PCI bus. While most of the initialization is handled by the host driver, the core driver is responsible for preparing some portion of the PCI block from within OCTEON. The SE is required to follow a series of steps highlighted below to setup the PCI block:

1. Allocate memory resources for the application globally i.e. one of the cores does the allocation on behalf of all cores. This includes setting up of FPA pools, arena allocation and configuring the IO interfaces.
2. Each core then allocates resources it requires like scratchpad location and memory allocation for local structures.
3. Setup the application type.
4. Send notification to host that it is ready.
5. Finally, it starts its application specific chores in a loop.

In step (1), the application should call **cvm_drv_init()**. This will perform initialization of the PCI ports and also allocate resources for the PCI input and output interfaces. It also sets up the chunked list for the DMA engines and programs the appropriate registers. This action is done globally by one of the cores in the application.

In step (2), the application should call **cvm_drv_local_init()**. This reserves scratchpad locations for use by the core driver. These scratchpad locations serve as storage for memory pointers for driver async allocation routines. This is done on each core that the application runs on.

In step (3), the application should call **cvm_drv_setup_app_mode()** to setup the application type. The header file components/driver/common/OCTEON-drv-opcodes.h defines all application types by the driver. The sample application in the OCTEON-PCI-BASE package uses application type CVM_DRV_BASE_APP.

In step (4), the application should call **cvm_drv_start()**. This routine sends a packet on PCI output ring that tells the host that core initialization is complete and PCI packet transfer can begin. The notification also includes the application type. This step is also performed by only one of the cores on behalf of all the cores running the application.

Note: *To see an example of the initialization process, look at the cvmcs.c file in applications/pci-core-app/base/ directory.*

6.2 Receiving packets from PCI Input rings

The host driver can use the PCI input ports to push packets into OCTEON. Packets from the PCI port are queued in the POW just as packets from any other interface. No support is necessary from the core driver at run time to receive packets from PCI. The SE application does not need to be aware of the input ring mechanism. Each packet from PCI has a WQE and one or more packet buffers representing it in the POW. The only difference is that packets arriving from PCI are in raw format and include an additional 24 bytes before the start of data. These 24 bytes are derived from the PCI instruction posted by the host driver. The IH field is processed into the Packet Instruction Header and forms the first 8 bytes. The remaining 16 bytes are the 16 software-defined bytes from the PCI instruction. The PCI instruction may have specified direct or gather mode DMA transfer but in both cases, the data arrives in a continuous stream in a list of linked packet buffers.

Tip: *The Packet Instruction Header format is covered in*
Section 7.2.1 - CN58xx & CN56xx HRM
Section 8.2.1 – CN63x HRM
Section 9.4.1 – CN68xx
Section 9.2.1 – CN66xx HRM

Tip: *The conversion from PCI Instruction Header to Packet Instruction Header is covered in*
Section 7.2.2 - CN58xx & CN56xx HRM
Section 8.2.2 – CN63xx HRM
Section 9.4.2 – CN68xx HRM
Section 9.2.2 – CN66xx HRM

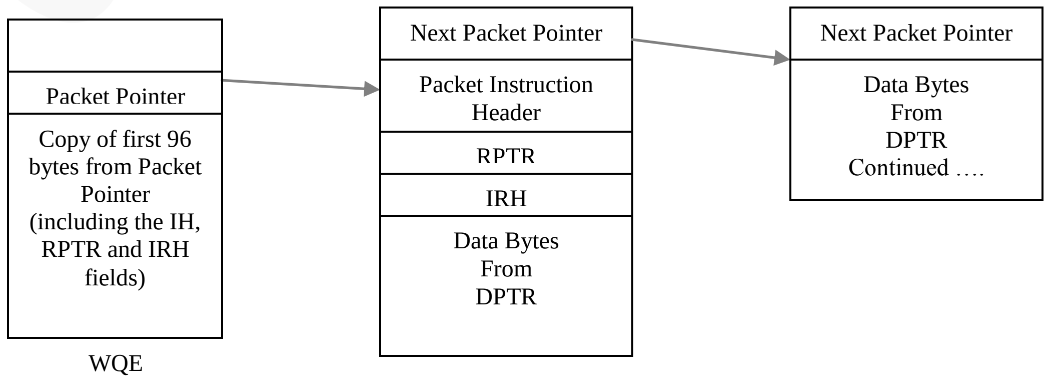

Packet Pointer
Copy of first 96 bytes from Packet Pointer (including the IH, RPTR and IRH fields)
WQE
Next Packet Pointer
Packet Instruction Header
RPTR
IRH
Data Bytes From DPTR
Next Packet Pointer
Data Bytes From DPTR Continued ….

Figure 17: Work queue entry for data sent on PCI

Packets arriving from PCI will have its port number (the ipprt field in the work-queue entry) set to a value between 32 and 35 depending on the input ring used by the host driver. For CN58xx, there is one ring per port, whereas for CN56xx & CN63xx there are 32 rings with 0,4,8..28 associated with port 32, rings 1,5,9..29 associated with port 33 and so on for all the other rings. Packets sent by the host driver on ring 0 or ring 4 in a CN56xx/CN63xx will have the same port number.

CN68xx introduces a different concept of port numbers for the PCI rings. The PCI port numbers 0x100 to 0x11F correspond to input rings 0 to 31.

The SE application can also distinguish raw packets from PCI from any other packet by looking at the software bit in the WQE. The core driver configures OCTEON to set this bit for all raw packets from PCI.

The SE application fetches packets that arrived from the PCI ports by calling the SE API for getting work from POW as it would for any other interface in OCTEON. For raw packets arriving from PCI, the software-defined IRH field usually carries information in the opcode field that identifies the packet type to the SE application. The application is responsible for processing the data appropriately and freeing the WQE and packet buffers.

6.3 Sending packets to host via PCI Output Rings

SE applications for all OCTEON’s before CN68XX use ports 32 to 35 to send data to the host via OCTEON's PCI output ring. The application does not need to know anything about the architecture of the PCI output ring. It can send a command to PKO in the same way that it would if it were sending a packet on a packet interface (RGMII, SPI etc).

For CN68XX, the PCI port numbers 0x100 to 0x11F correspond to output rings 0 to 31.

Each PKO command is considered as one complete packet by OCTEON for the PCI block. OCTEON will push the data from this command to one or more host buffers. It will write the packet length into the info field of only the first descriptor used for this packet in host memory.

The OCTEON hardware would perform the necessary tasks to transmit this packet including:

1. Fetching the descriptors from host memory to determine the address of the next available host buffers. It does this every time the host writes to the credit register for an output ring.
2. Split the packet received from PKO such that the contents fit into one or more host buffer as required.
3. Write the packet length into the info pointer of the first descriptor used for this packet.
4. Generate an interrupt if enabled (based on the packet or time threshold) set by the driver.

All the above steps are performed in a transparent manner by OCTEON for each packet sent by the SE application.

While OCTEON simple executive applications can directly use the PKO API provided by SDK, the driver also provides API's for using the PKO. These API's require different parameters than the SDK API and include additional checks for size and core driver state. The API's are described in section 7.3.

6.4 Sending data to host via PCI DMA engines.

The DMA engines are initialized by the core driver. They use a chunked list of buffers to receive commands from the SE application. Applications use the core driver API's to post a DMA command to OCTEON. The API's allow data from multiple local buffers to be transferred to one or more memory locations in the host. The number of local and remote (host) pointers do not need to be the same, but the sum of the sizes in the local and remote pointers must be the same.

Tip: T*he format of the DMA instruction header can be found in*
Section 10.5.1 - CN58xx HRM
Section 9.6.1 - CN56xx HRM
Section 19.3.1 – CN63xx HRM
Section 12.3.1 – CN68xx/CN66xx HRM

The core driver API makes checks for appropriateness of the pointers including the address and sizes. It sets up the DMA command to identify the number of local and remote pointers and the DMA engine to transfer this data. Applications can also pass a WQE pointer or an address in physical memory that would be used by the application to check for the completion of the DMA. The core driver would set the required fields in the DMA command on behalf of the application. The commands and the local and remote pointers are copied into the chunked list of buffers and the DMA engines doorbell is hit by the core driver to indicate the arrival of new commands to the DMA hardware. The core driver takes care of writing the command words into multiple linked buffers in the chunked list if the current buffer does not have space for the DMA command and pointers. It is also responsible for allocating new buffers to keep the chunked list populated.

The API's allow for Inbound (data is transferred from host to OCTEON memory) and Outbound (data is transferred from OCTEON to host memory). For CN56xx & newer processors, two additional modes are supported – Internal transfer where data is transferred from within OCTEON memory from one location to another and External, where data is transferred from one external PCI location to another external PCI location.

Note: The DMA API's are covered in section 7.3.

The core driver also provides API's that can be used to send responses for requests received from the host in the format specified in section 4.5 of this document. These API's also use the DMA engines for transferring the response data to the host. The internal routines used to send the response data are the same as those used for normal DMA transfers described above. These API's are also covered in section 7.3.

Chapter 7: Driver API

This chapter describes the API that the driver provides to user-space & kernel applications running on the host in the next two sections. The last section describes the API provided by the core driver that can be used by applications running on OCTEON.

7.1 API for user-space applications

(See components/driver/host/api/OCTEON_user.h)

int OCTEON_initialize()

Arguments: None.
Return Value:
Success - 0.
Failure - value of errno.
Description: Opens the OCTEON device file and enables a channel of communication with the OCTEON device driver. This is the first step that any user-space application must do before it can interact with the OCTEON device driver.

int OCTEON_shutdown()

Arguments: None.
Return Value: 0.
Description: Closes the OCTEON device file and stops the channel of communication with the OCTEON device driver.

int OCTEON_hot_reset(int oct_id)

Arguments:
oct_id - OCTEON device id
Return Value:
Success - 0.
Failure - value of errno.
Description: Do a Hot Reset of the OCTEON device. A hot reset shuts down all PCI activity and applies a soft reset to OCTEON. After a hot reset, a new application can be loaded on the OCTEON cores without having to restart the OCTEON driver module.

int OCTEON_get_dev_count()

Arguments: None.
Return Value:
Success - Count of OCTEON devices.
Failure - 0.
Description: Get the number of OCTEON devices currently managed by the driver.

int OCTEON_send_request(int oct_id, OCTEON_soft_request_t *soft_req)

Arguments:
oct_id - id of OCTEON device to which the request is sent.
soft_req - pointer to the request.
Return Value:
Success - 0.
Failure - value of errno.
Description: Send a request to OCTEON. The request can be in UNORDERED or NORESPONSE mode. For UNORDERED mode, both blocking (the function call blocks till it gets a response from OCTEON) or non-blocking (the function returns immediately) are supported. Noresponse type requests, as the name suggests does not expect a response and the call return as soon as the request is queued with the driver. For all types, the request is guaranteed to be queued up with the driver only if the return value is 0.

int OCTEON_query_request(int oct_id, OCTEON_query_request_t *query)

Arguments:
oct_id - id of OCTEON device to which the query is sent.
query - pointer to a query structure which holds the request id. The driver copies the status of the request into this structure when the call returns.
Return Value:
Success - 0.
Failure - value of errno.
Description: Query the status of a previously posted UNORDERED Non-blocking request. A successful query returns with value 0 and the driver returns the state of the request can be found in the query structure that was passed as a parameter.

int OCTEON_get_stats(int oct_id, oct_stats_t *stats)

Arguments:
oct_id - id of OCTEON device.
stats - pointer to the OCTEON statistics structure. The statistics are copied into this structure by the driver.
Return Value:
Success - 0.
Failure - value of errno.
Description: Read the Input and Output ring and general device statistics for an OCTEON device. On success, the call returns 0 and the driver would have copied the statistics into the **stats** structure that was passed as a parameter.

int OCTEON_read_pcicfg_register(int oct_id, uint32_t offset, uint32_t *data)

Arguments:
oct_id - id of OCTEON device.
offset - offset into the config space (register address).
data   - Pointer to a user space location where the driver copies the register contents.
Return Value:
Success - 0.
Failure - value of errno.
Description: Read the OCTEON PCI config register at the specified offset.

int OCTEON_write_pcicfg_register(int oct_id, uint32_t offset, uint32_t data)

Arguments:
oct_id - id of OCTEON device.
offset - offset into the config space (register address).
data   - 32-bit value to be written.
Return Value:
Success - 0.
Failure - value of errno.
Description: Write to the OCTEON PCI config register at the specified offset.

int OCTEON_get_mapping_info(int oct_id, OCTEON_reg_type_t type, unsigned long *mapped_address, uint32_t *mapped_size)

Arguments:

oct_id - id of OCTEON device.
type - the type of mapping (BAR0, BAR1, BAR2)
mapped_address - the address where the driver returns the mapped physical address.
mapped_size - the address where the driver returns the size of the mapped region.

Return Value:

Success - 0.
Failure - value of errno.

Description: Get the physical mapped address for different PCI address spaces.

int OCTEON_read32(int oct_id, unsigned long address, uint32_t *data);
int OCTEON_read16(int oct_id, unsigned long address, uint16_t *data);
int OCTEON_read8(int oct_id, unsigned long address, uint8_t *data);

Arguments:

oct_id - id of OCTEON device.
address - the OCTEON PCI BAR mapped memory address to read.
data - address where the driver returns the value read from OCTEON.

Return Value:

Success - 0.
Failure - value of errno.

Description: Read an 8/16/32 bit value from an OCTEON PCI mapped address. The **address** argument passed to these functions is computed by getting the physical address of OCTEON BAR memory by calling **OCTEON_get_mapping_info()** and then adding the offset for the location you need to read.

int OCTEON_write32(int oct_id, unsigned long address, uint32_t data);
int OCTEON_write16(int oct_id, unsigned long address, uint16_t data);
int OCTEON_write8(int oct_id, unsigned long address, uint8_t data);

Arguments:
oct_id - id of OCTEON device.
address - the OCTEON PCI BAR mapped memory address to write.
data - data to write to OCTEON BAR mapped memory.
Return Value:
Success - 0.
Failure - value of errno.
Description: Write an 8/16/32 bit value from an OCTEON PCI mapped address. The **address** argument passed to these functions is computed by getting the physical address of OCTEON BAR memory by calling **OCTEON_get_mapping_info**() and then adding the offset for the location you need to write.

int OCTEON_win_write(int oct_id, uint64_t address, uint64_t data);

Arguments:
oct_id - id of OCTEON device.
address - the OCTEON PCI windowed (64-bit) address to write.
data - 64-bit data.
Return Value:
Success - 0.
Failure - value of errno.
Description: Write a 64-bit value to a 64-bit address in the OCTEON PCI windowed access region.

int OCTEON_win_read(int oct_id, uint64_t address, uint64_t *pdata)

Arguments:
oct_id - id of OCTEON device.
address - the OCTEON PCI windowed (64-bit) address to read.
pdata - address where driver will returned 64-bit data read from OCTEON.
Return Value:
Success - 0.
Failure - value of errno.
Description: Read a 64-bit value from a 64-bit address in the OCTEON PCI windowed access region.

int OCTEON_read_core_direct(int oct_id, uint64_t core_addr, uint32_t len, void *data, uint32_t swap_mode)

Arguments:

oct_id - the OCTEON device to read.
core_addr - the address in the OCTEON memory to read.
len - number of bytes to read.
data - driver returns the data read from OCTEON into user-space starting at this address.
swap_mode - byte-swap mode to use. Usually is 0 (no-swap) or 1 (64-bit swap).

Return Value:

Success - 0.
Failure - value of errno.

Description: Read **len** bytes from OCTEON memory at address **core addr** and copy it into user-space memory. The byte-swap mode is specified by **swap_mode**.

int OCTEON_write_core_direct(int oct_id, uint64_t core_addr, uint32_t len, void *data, uint32_t swap_mode)

Arguments:

oct_id - the OCTEON device to write.
core_addr - the address in the OCTEON memory to write.
len - number of bytes to write.
data - driver copies data from this user-space address.
swap_mode - byte-swap mode to use. Usually is 0 (no-swap) or 1 (64-bit swap).

Return Value:

Success - 0.
Failure - value of errno.

Description: Copy **len** bytes from user-space memory pointed by **data** into OCTEON memory at address **core addr**. Use the byte-swap mode specified by **swap_mode**.

int OCTEON_read_core(int oct_id, OCTEON_user_datatype_t datatype, uint64_t address, void *data)

Arguments:

oct_id - the OCTEON device to read.
datatype - specifies the number of bytes to read.
address - the address in the OCTEON memory to read.
data - driver returns the data read from OCTEON into user-space starting at this address.

Return Value:

Success - 0.
Failure - value of errno.

Description: Read 1/2/4 bytes from OCTEON memory at address given by **address** and copy it into user-space memory pointed by **data**. The **datatype** argument specifies whether the read is 1/2/4 bytes wide.

int OCTEON_write_core(int oct_id, OCTEON_user_datatype_t datatype, uint64_t address, void *data)

Arguments:

oct_id - the OCTEON device to write.
datatype - specifies the number of bytes to write.
address - the address in the OCTEON memory to write.
data - driver copies the data from this user-space address into OCTEON.

Return Value:

Success - 0.
Failure - value of errno.

Description: Copy 1/2/4 bytes from user-space memory pointed by **data** into OCTEON memory at address given by **address**. The **datatype** argument specifies whether the read is 1/2/4 bytes wide.

7.2 API for kernel-space applications

(See components/driver/host/include/cavium_defs.h & components/driver/host/include/cavium_kernel_defs.h)

OCTEON_instr_status_t OCTEON_process_request(uint32_t OCTEON_id, OCTEON_soft_request_t *soft_req)

Arguments:

OCTEON_id - id of OCTEON device
soft_req - pointer to the request structure.

Return Value:

Return value is in a structure which is a union of a 64-bit value and 2 32-bit values for status and request id. If request was successful, the driver returns the status and request id in the 32-bit fields. On failure, the 64-bit field is set to OCTEON_REQUEST_FAILED.

Description: This is the entry point for kernel applications to send requests to OCTEON device. All requests for OCTEON arrive as a soft request to this routine. This routine formats the request to the driver's internal format and posts it to OCTEON Input ring. When the routine returns successfully, the request should have been scheduled for fetching by the OCTEON Input queue hardware.

uint32_t OCTEON_query_request_status(uint32_t OCTEON_id, OCTEON_query_request_t *query)

Arguments:

OCTEON_id - id of OCTEON device
query - pointer to the query structure.

Return Value:

Success - 0.
Failure - 1.

Description: Query the status of a previously posted request. The caller passes the request id in the **query** structure. The driver returns the status in the status field of the **query** structure. The status value is valid only if the driver returns from the call successfully.

uint32_t OCTEON_register_dispatch_fn(uint32_t OCTEON_id, OCTEON_opcode_t opcode, OCTEON_dispatch_fn_t fn, void *fn_arg)

Arguments:

OCTEON_id - id of the OCTEON device to register with.
opcode - the opcode for which the dispatch will be registered.
fn - pointer to the dispatch function.
fn_arg - caller specified argument that will be passed along with the dispatch function by the driver.

Return Value:

Success - 0.
Failure - 1.

Description: Register a dispatch function for an **opcode**. When the driver receives a packet with the given opcode in its output rings it encapsulates the packet buffer and related information in a ***OCTEON_recv_info_t*** structure and calls the dispatch function with the ***OCTEON_recv_info_t*** structure and the caller specified argument as parameters.

uint32_t OCTEON_unregister_dispatch_fn(uint32_t OCTEON_id, OCTEON_opcode_t opcode)

Arguments:

OCTEON_id - id of the OCTEON device.
opcode - the opcode to be unregistered.

Return Value:

Success - 0.
Failure - 1.

Description: Remove registration for an **opcode**. This will delete the mapping for an opcode. The dispatch function will be unregistered and will no longer be called if a packet with the opcode arrives in the driver output rings.

int OCTEON_register_poll_fn(int oct_id, OCTEON_poll_ops_t *ops)

Arguments:
OCTEON_id - OCTEON device id
ops - pointer to a structure with the poll function details.
Return Value:
Success - 0.
Failure - -ENODEV, -EINVAL or -ENOMEM.
Description: Register a poll function with the driver. The **ops** parameter points to a ***OCTEON_poll_ops_t*** structure that holds a pointer to the poll function, an optional argument that the driver should call if the poll function is called and the time interval in ticks that the poll function should be called.

int OCTEON_unregister_poll_fn(int oct_id, OCTEON_poll_fn_t fn, unsigned long fn_arg)

Arguments:
OCTEON_id - OCTEON device id
fn - the poll function to be unregistered.
fn_arg - the poll function argument.
Return Value:
Success - 0.
Failure - -ENODEV.
Description: Unregister a poll function. The **fn** and **fn_arg** used when **OCTEON_register_poll_fn()** was called is passed as arguments to uniquely identify the instance of the poll function to be unregistered.

int OCTEON_register_droq_ops(int OCTEON_id, int q_no, OCTEON_droq_ops_t *ops)

Arguments:
OCTEON_id - OCTEON device id
q_no - OCTEON output ring number (0 <= q_no <= MAX_OCTEON_DROQ-1)
ops - the droq_ops settings for this ring
Return Value:
Success - 0.
Failure - -ENODEV or -EINVAL.
Description: Register a change in droq operations. The **ops** argument is a pointer to a function which will be called by the DROQ handler for all packets arriving on output rings given by **q_no** irrespective of the type of packet. The **ops** field also has a flag which if set tells the

DROQ handler to drop packets if it receives more than what it can process in one invocation of the packet handler routine.

int OCTEON_unregister_droq_ops(int OCTEON_id, int q_no)

Arguments:
OCTEON_id - OCTEON device id
q_no - OCTEON output ring number (0 <= q_no <= MAX_OCTEON_DROQ-1)
Return Value:
Success - 0.
Failure - -ENODEV or -EINVAL.
Description: Resets the function pointer and flag settings made by **OCTEON_register_droq_ops().** After this routine is called, the DROQ handler will go to default mode where it looks up dispatch function for each arriving packet on the output ring given by **q_no**.

int OCTEON_register_module_handler(OCTEON_module_handler_t *handler);

Arguments:
handler - structure with "start" and "stop" routines and app type
Return Value:
Success - 0.
Failure: -EINVAL if the handler arguments are invalid or if a handler was already registered; -ENOMEM if no space is available to add this handler.
Description: Register module handler functions to be called by the OCTEON base driver when an OCTEON device is being initialized. The **start** function in the handler may be called in this function call's context if this function is called after an OCTEON device was initialized. The **app_type** field is checked by the base module against the application type given by the core application in its start notification. If it matches, the base module calls the start handler for that OCTEON device.

int OCTEON_unregister_module_handler(uint32_t app_type);

Arguments:
app_type - the handler for this applicatin type should be removed.
Return Value:
Success – 0
Failure: -ENODEV if no handler was found for the app_type.
Description: Called by a module when it is being unloaded to unregister start and stop functions. The **stop** function will be called in this function call's

context. The **stop** routine will also be called if the OCTEON device if being removed (due to a hot-swap operation).

7.3 Core Driver API for OCTEON applications

(See components/driver/core/cvm-drv.h; components/driver/core/cvm-pci-pko.h; components/driver/core/cvm-pci-dma.h)

int cvm_drv_init(void)

Arguments: None.
Return Value:
Success: 0; Failure: -1.
Description: The application should call this routine during global initialization from only one of the cores. This routine sets up the PCI ports and initializes the DMA engines including allocation of chunked lists for each ring.

void cvm_drv_setup_app_mode(int app_mode)

Arguments: None.
Return Value: None.
Description: Sets the application type. Application types are defined in driver/common/OCTEON-drv-opcodes.h

int cvm_drv_start(void)

Arguments: None.
Return Value:
Success: 0; Failure: -1.
Description: This routine sends a notification to the host driver that the application is ready to accept instructions from host. The notification also includes the application type set up by calling **cvm_drv_setup_app_mode().** The application should call this from only one of the cores.

int cvm_drv_register_op_handler(uint16_t opcode, int (*handler)(cvmx_wqe_t *))

Arguments:
opcode - register function for this opcode.
handler - the function to be registered for opcode.
Return Value:
Returns 0 if function is successfully registered with driver, else returns 1.

Description: Register a handler function for an opcode. When the driver receives a WQE with the matching opcode, it will call the function registered for the opcode with the WQE as argument.

int cvm_drv_process_instr(cvmx_wqe_t *wqe)

Arguments:
wqe - work queue entry that contains the host instruction.

Return Value:
Returns 0 if a function wqe is successfully processed, else frees the wqe and returns 1.

Description: Common routine to parse all driver instructions. This routine looks up an internal table for a function for the opcode in the instruction and passes the wqe to the function if it exists. If no function is registered for the opcode, the work queue entry is freed.

int cvm_pko_send_data(cvmx_resp_hdr_t *user_resp_hdr, cvmx_buf_ptr_t dptr, cvm_ptr_type_t ptr_type, uint32_t total_bytes, uint32_t segs, uint32_t port)

Arguments:
user_resp_hdr - Pointer to 8 bytes of response header.
ptr_type - Describes the type of buffer available at dptr.
dptr - Pointer to data or gather buffer in OCTEON's local memory.
total_bytes - Sum of data bytes from all buffers.
segs - Number of buffers.
port - The PCI port to send this data.

Return Value:
Success: 0; Failure: 1

Description: Send data in one or more buffers in OCTEON local memory to the next available buffers in the host's output ring. The 8 byte response header will appear in the info pointer of the first buffer used for this data. The opcode in the response header is used by the host driver to determine the application that will receive this data.

Depending on the pointer type (given by **ptr_type**), **dptr** is interpreted differently. The **ptr_type** field can be:
PKO_DIRECT_DATA: **dptr** points to a single data buffer.
PKO_GATHER_DATA: **dptr** points to an address that holds a list of ***cvmx_buf_ptr_t*** type entries that hold the address and and size information for multiple data buffers.

PKO_LINKED_DATA: **dptr** points to the first buffer in a linked list of buffers. The next buffer pointer in the ***cvmx_buf_ptr_t*** format is available at 8 bytes above the **dptr** address.

int cvm_pko_send_direct(cvmx_buf_ptr_t lptr, cvm_ptr_type_t ptr_type, uint32_t segs, uint32_t total_bytes, uint32_t port)

Arguments:

lptr - Pointer to data or gather buffer in OCTEON's local memory.
ptr_type - Describes the type of buffer available at dptr.
segs - Number of buffers.
total_bytes - Sum of data bytes from all buffers.
port - The PCI port to send this data.

Return Value:

Success: 0; Failure: 1.

Description: The arguments have the same meaning as the **cvm_pko_send_data()** API above. This API assumes that 8 bytes of response header is available at the start of the first buffer pointed by **lptr**.

int cvm_pko_send_direct_flags(cvmx_buf_ptr_t lptr, cvm_ptr_type_t ptr_type, uint32_t segs, uint32_t total_bytes, uint32_t port, uint32_t flags, uint64_t flag_data)

Arguments:

lptr - Pointer to data or gather buffer in OCTEON's local memory.
ptr_type - Describes the type of buffer available at dptr.
segs - Number of buffers.
total_bytes - Sum of data bytes from all buffers.
port - The PCI port to send this data.
flags - additional flags.
flag_data – data associated with the flags.

Return Value:

Success: 0; Failure: 1

Description: The first 5 parameters have the same meaning as the **cvm_pko_send_data()** API above. This API also assumes that 8 bytes of response header is available at the start of the first buffer. In addition, the **flags** parameter is used to pass additional flags. The flags_data parameter provides additional information when **flags** is non-zero. Currently **flags** is reserved for use by the driver and should always be set to zero.

int cvm_send_pci_pko_direct(cvmx_buf_ptr_t lptr, cvm_ptr_type_t ptr_type, uint32_t segs, uint32_t total_bytes, uint32_t oq_no)

Arguments:

lptr - Pointer to data or gather buffer in OCTEON's local memory.
ptr_type - Describes the type of buffer available at dptr.
segs - Number of buffers.
total_bytes - Sum of data bytes from all buffers.
oq_no - The PCI output ring to send this data.

Return Value:

Success: 0; Failure: 1.

Description: The arguments have the same meaning as the **cvm_pko_send_direct()** API above. The oq_no corresponds to the PCI output ring number (not the PCI port) and ranges from 0 to 31. This API can be used when you have more than 1 output queue per PCI port to target the output ring directly. The previous API's always sent to the first queue in the PCI port. Additional setup is required before using this API. See the section on Output Rings above.

int cvm_pci_dma_raw(cvmx_oct_pci_dma_inst_hdr_t *dma_hdr, cvmx_oct_pci_dma_local_ptr_t *lptr, cvm_dma_remote_ptr_t *rptr)

Arguments:

dma_hdr - the DMA instruction header.
lptr - list of local data pointers.
rptr - list of remote (host/peer) addresses.

Return Value:

Success: 0; Failure: -ENOMEM, -EINVAL.

Description: This routine creates a PCI DMA instruction based on information in the DMA header and local and remote pointers and buffer sizes passed to it. Applications have complete control over the DMA operation when using this API. This function can be called for INBOUND & OUTBOUND operations only for all variants of OCTEON.

int cvm_pcie_dma_raw(int q_no, cvmx_oct_pci_dma_inst_hdr_t *dma_hdr, void *firstptrs, void *lastptrs);

Arguments:

q_no - The PCI DMA queue (0 to 4) to use for this DMA operation.
dma_hdr - the DMA instruction header.
firstptrs - first set of pointers.
lastptrs - second set of pointers.

Return Value:

Success: 0; Failure: -ENOMEM, -EINVAL.

Description: This routine can be used for all OCTEON's but also includes support for the DMA features in CN56xx & CN63xx. It creates a PCI DMA instruction based on information in the DMA header and local and remote pointers and buffer sizes passed to it. Applications have complete control over the DMA operation when using this API. This function can be called for INBOUND & OUTBOUND operations. For CN56xx & CN63xx, an application can also perform INTERNAL-ONLY or EXTERNAL-ONLY operations using this function. Based on the DMA type specified, the firstptrs and lastptrs will be interpreted differently.

int cvm_pci_dma_send_data(cvm_pci_dma_cmd_t *cmd, cvmx_buf_ptr_t *lptr, cvm_dma_remote_ptr_t *rptr)

Arguments:

cmd - DMA command options.
lptr - list of local data pointers.
rptr - list of remote (host/peer) addresses.

Return Value:

Success: 0; Failure: -ENOMEM, -EINVAL.

Description: Write data in local buffers pointed by **lptr** to host or PCI-Express memory pointed by **rptr** using OCTEON PCI DMA engine.
See ***cvm_pci_dma_cmd_t*** (in components/driver/core/cvm-pci-dma.h) for description of fields in the **cmd** parameter.

int cvm_pci_dma_recv_data(cvm_pci_dma_cmd_t *cmd, cvmx_buf_ptr_t *lptr, cvm_dma_remote_ptr_t *rptr)

Arguments:
cmd - DMA command options.
lptr - list of local data pointers.
rptr - list of remote (host/peer) addresses.
Return Value:
Success: 0; Failure: -ENOMEM, -EINVAL.
Description: Read data from host or PCI-Express memory pointed by **rptr** into local buffers pointed by **lptr** using OCTEON PCI DMA engine.
See ***cvm_pci_dma_cmd_t*** (in components/driver/core/cvm-pci-dma.h) for description of fields in the **cmd** parameter.

cvm_dma_comp_ptr_t * cvm_get_dma_comp_ptr(void)

Arguments: None.
Return Value:
Success: Pointer to completion byte; Failure: NULL.
Description: Function to get a completion pointer from core driver pool. The application would use the address of the **comp_byte** field in the ***cvm_dma_comp_ptr_t*** structure returned to it from this call as a completion word for PCI DMA.

void cvm_release_dma_comp_ptr(cvm_dma_comp_ptr_t *ptr)

Arguments: ptr – pointer to a completion pool node.
Return Value: None.
Description: Return a completion pool node back to the core driver.

Chapter 8: Test Programs & Utilities

The PCI driver package includes several additional programs that can be used for reference or debug purposes. They include:

1. Test programs that can be used as reference for building your own user-space or kernel space application.
2. User-space utilities to display statistics and to read/write OCTEON registers.
3. Sample OCTEON application to get more details into how data flow across PCI is handled in an OCTEON Simple Executive application.

8.1 Test Programs

8.1.1 oct_req - test request/response from user space

The user space test program **oct_req** uses the user space library api to send RAW PCI instructions to OCTEON. It requires the core application **cvmcs** to be running on OCTEON cores for its operation.

The source for oct_req is at components/driver/host/test.

oct_req can be used to send requests of UNORDERED and NORESPONSE types. The operations can be blocking or non-blocking. The test parameters can be modified in oct_req.c. The type of request and request mode can also be specified on the command line. The command line arguments override any settings made in oct_req.c. The default setting in oct_req.c is to send NORESPONSE/NON-BLOCKING type requests in DIRECT DMA mode.

Command line options:

-d : Change the DMA mode to use for the request. Supported values are "direct", "gather", "scatter" and "scatter_gather".
-u : Send requests of UNORDERED response order.
-b : Send a blocking mode request.
-s : Do not print per-packet messages.
-y : Answer "yes" to all prompts.
[**count**] - Number of requests to send. If specified, the test will attempts to send count requests and stop, else it will continue to run till stopped (Press ^C to stop).

To compile the **oct_req** program, do **make** in components/driver/host/test. A symbolic link is created from components/driver/bin to the application.

To test, load the driver and core application by following the steps in components/driver/README.txt and then run the test program with the required options.

Note: The OCTEON device file must be created before running the test program.

8.1.2 req_resp – kernel space request/response test program

The kernel space test program **req_resp**, calls driver API's to send requests to OCTEON. It requires the core application **cvmcs** to be running on the OCTEON cores for its operation.

The source for **req_resp** is at components/driver/host/test/kernel/req_resp.

The **req_resp** program is a linux kernel module. It can be used to send requests of ORDERED, UNORDERED and NORESPONSE types. It can send requests in both BLOCKING and NON-BLOCKING mode, though it's not recommended to use the UNORDERED/NON-BLOCKING type. The default setting is to send ORDERED/BLOCKING requests. The test program provides a good reference to writing kernel mode applications that need to interface with the OCTEON PCI driver.

The setting for the application can be changed in OCTEON_req_resp.c. Each parameter is documented in this file.

To compile the req_resp program, do **make** in components/driver/host/test/kernel/req_resp. A symbolic link is created from components/driver/bin to the application.

To test, load the driver and core application by following the steps in components/driver/README.txt and then load the test program using insmod command.

8.1.3 droq_test – kernel space program to process output ring packets

The kernel space test program **droq_test** registers dispatch functions to receive packets arriving on the output rings. The **droq_test** program requires the **cvmcs** core application to be running on the OCTEON cores. The **cvmcs** application must be compiled with the CVMCS_TEST_PKO flag enabled.

The source for **droq_test** is at components/driver/host/test/kernel/droq_test.

The core application, when set accordingly, will generate packets with opcodes DROQ_PKT_OP1 & DROQ_PKT_OP2, and send them on the OCTEON output ring. The OCTEON PCI driver fetches these packets in its interrupt handler and looks for a dispatch function for these opcodes. If a function exists, the packet are sent to the dispatch function, else the packets are discarded.

The settings for **droq_test** can be modified in the OCTEON_droq_test.c file. If DROQ_TEST_SEC_OPCODE is enabled, the application registers dispatch function for opcodes DROQ_PKT_OP1 and DROQ_PKT_OP2 with the driver, else a dispatch function is registered only for DROQ_PKT_OP1.

To compile the **droq_test** program, do make in components/driver/host/test/kernel/droq_test. A symbolic link is created from components/driver/bin to the application.

To test, load the driver and core application by following the steps in components/driver/README.txt and then load the test program using insmod command.

8.2 Sample OCTEON Application

All the test programs running on the PCI host require the **cvmcs** core application. The source for this application is located at applications/pci-core-app. The main initialization routines can be found in pci-core-app/base and pci-core-app/common. All test routines are available from pci-core-app/test.

The application can be compiled to perform one or more type of tests at a given time. The type of tests that the application runs is determined by the definitions in pci-core-app/test/cvmcs-test.h file.

By default, the application will process instruction from the host with opcode CVMCS_REQRESP_OP. The test programs described in the previous section use this opcode for all the requests they generate. If a response is required (the rptr field in PCI instruction will be non-zero), the application copies the input bytes into the output buffers and sends a response back to the host. If the output buffer is bigger than the input, it fills the remaining output bytes with a signature value (0xFA currently). The host application (either the **req_resp** kernel application or **oct_req** user-space application) can compare the input and output buffer contents to see if the operation was successful.

The application also supports scatter mode responses. The cvmcs-reqresp.c file has routines to read a scatter list in blocking and non-blocking mode and to send back a scatter response. The application exercises all scatter mode API depending on the flags enabled in cvmcs-test.h. The source is well-documented and should be self-explanatory. See section 7.3 for more details on the core-driver API used by this application.

The following definitions in cvmcs-test.h control the settings for sending data on the PCI Output Rings:

- CVMCS_TEST_PKO - Enable this to send test packets on the output rings with opcodes DROQ_PKT_OP1 and DROQ_PKT_OP2. This option is useful when running the **droq_test** kernel space program.

- CVMCS_FIXED_SIZE_TEST - If enabled, all packets sent on the output ring are of fixed size, else each packet has a randomly assigned size.

- CVM_MAX_DATA - The packets sent to output ring will not be greater than CVM_MAX_DATA in size.

To compile the **cvmcs** program, do make in applications/pci-core-app/base. A symbolic link is created from components/driver/bin to the core application.

8.3 User-space utilities

8.3.1 oct_dbg - OCTEON debug utility

oct_dbg is a menu driven utility that allows the user to reset the OCTEON device (without unloading the driver) and to read or write OCTEON registers and core memory locations. This utility requires the OCTEON user-space API library. **oct_dbg** can read the PCI config-space registers, registers mapped to BAR0 and windowed NPI registers. It can also read or write OCTEON core memory locations using BAR1 mappings.

Command line options:
OCTEON_id - OCTEON device id (0 for the first OCTEON device).

The utility prompts for device id if it finds multiple OCTEON's in the system and a device id was not passed on the command line.

Menu Options:
t: to reset OCTEON
r: to read OCTEON BAR0 registers
R: to read OCTEON config registers.
i: to read Windowed registers
o: to write Windowed registers
w: to write OCTEON BAR0
W: to write PCI config register
l: to read from OCTEON DRAM
s: to write to OCTEON DRAM
d: to display PCI mapping info.

8.3.2 oct_stats - OCTEON statistics display utility

oct_stats allows the user to check on the input, output and DMA queue statistics for packet count and byte count. It also displays 10-second average byte count for input and output rings. It has options to display the driver buffer pool usage. The utility can be used to display statistics once or at user-defined intervals. The **OCTEON_id** argument is always required. If no other options are given, **oct_stats** will print statistics for the input and output ring refreshed at 1 second interval

Command line options:

OCTEON_id - OCTEON device id (0 for the first OCTEON device) [required]
interval - refresh the statistics at this interval (in seconds)
-i : display input ring statistics
-o : display output ring statistics
-c : display DMA queue statistics
-d : display Direct Data ring statistics
-b : display buffer pool statistics
-a : display statistics for all queues
-r : refresh statistics at regular intervals
-v : verbose output
-h : print usage messages