Packet Input Processor (PIP) and Input Packet Data (IPD) Units

Revision History
Revision2 - Nov. 19, 2010
1. Changed chapter number from 4 to 6.
2. Changed references to the *Configuration and Advanced Topics* chapter: these are two separate chapters now.
3. Added references to new *Essential Topics* and *Advanced Topics* chapters.
4. Added Revison History.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

In the *Packet Flow* chapter of the *OCTEON Programmer's Guide (The Fundamentals*), the actions performed by the Packet Input Processor (PIP) and Input Packet Data (IPD) are shown with minimal details. This chapter provides in depth detail of the processing performed by these two units, and how to customize their configuration to optimize throughput.

Together, PIP/IPD:

1. Parse the packet, checking for errors in L2/L3 headers
2. Provide congestion control: drop packet and/or backpressure as needed
3. Create a Work Queue Entry (WQE) for the packet if it is not dropped
4. Determine packet properties which affect subsequent scheduling actions by the SSO (POW) (Group, QoS, Tag Type, Tag Value)
5. Store the packet data
6. Send the WQE to the SSO for scheduling

The PIP/IPD provides a tremendous amount of configuration flexibility. Correct configuration of the PIP/IPD requires a clear view of the desired software architecture and specifics of the target application. Before reading this chapter, it is essential to understand the contents of the *Packet Flow* chapter. It is also helpful to read the *Software Overview* chapter (especially the *Software Architecture* section), the *Essential Topics*, *Configuration*, *Advanced Topics*, and the *Free Pool Allocator (FPA)* chapters These chapters help the user visualize the overall system necessary to develop a customized solution which will best fit the target application. The PIP/IPD is a central component of that customized solution.

Because of the feature-richness flexibility of the PIP and IPD units, the chapter describing these features is quite extensive. The chapter is designed so that readers can select the relevant section their specific application, and ignore sections discussing unused features.

Although the PIP and IPD are separate units, they are so closely associated that they are collected into a pseudo-block, the Packet Input (PKI) block. This pseudo-block is only used in high-level diagrams, and is not used in the text of this chapter.

The PIP/IPD works closely with:

- The Packet Input Interfaces
- The Free Pool Allocator (FPA)
- The Schedule Synchronization Order (SSO) unit. (The SSO unit is referred to as the Packet Order Work unit, or POW in the *Hardware Reference Manual*.)

The PIP/IPD receives the packet data from a traffic ingress port (for example, GMII). By using configuration information from PIP/IPD Configuration and Status Registers (CSRs), and from parsing the packet header, PIP/IPD determines the essential packet scheduling information: QoS level, Work Group ID, Tag Value, and Tag Type. The PIP/IPD creates a Work Queue Entry (WQE) and forwards it to the SSO. PIP/IPD stores the packet data in L2/DRAM, using a dedicated bus to the I/O Bridge (IOB). If necessary, congestion control is managed via backpressure or RED.

The FPA maintains two pools of buffers: the Packet Data buffer pool and the Work Queue Entry buffers. Packet Data buffers and WQE buffers are automatically allocated from the appropriate FPA pool by PIP/IPD. The size and quantity of buffers, and the WQE buffer pool number are configured when Simple Executive is configured at build time. The Packet Data buffer pool is required by hardware to be FPA pool 0. The pool number not a configurable option. See the *Essential Topics*, *Configuration*, and *FPA* chapters for details.

The IPD is responsible for:

- If required by PIP/IPD congestion control mechanisms, backpressure ports
- If no backpressure, IPD is responsible for receiving the packet from the input ports
- If required by configurable congestion control mechanisms, drop the packet, otherwise continue packet processing
- Allocating the Work Queue Entry buffer from the WQE buffer pool maintained by the FPA
- Allocating the Packet Data buffers from the Packet Data buffer pool in maintained by the FPA
- Storing information in the Work Queue Entry, including the QoS queue, Work Group, Tag Type, and Tag Value which are computed by the PIP
- Storing information in the Packet Data Buffer(s) as needed
- Writing the Packet Data Buffer(s) to L2/DRAM
- Performing the `add_work` operation to add the Work Queue Entry to the appropriate Quality of Service (QoS) queue in the SSO

The PIP is responsible for:

- Packet parsing
- Perform optional checks on the packet
- Compute the QoS queue, Work Group, Tag Type, and Tag Value
- Provide QoS, Work Group, Tag Type, and Tag Value information to IPD

The OCTEON Software Development Kit supports the PIP/IPD units with a thin layer of software designed to serve as a base for more complex customized development. Because of the rich features provided by the PIP/IPD units, the API does not cover all possible uses. This chapter provides an overview of the API and the details needed to get started with customization.

The *CN54/55/56/57 Hardware Reference Manual* (*HRM*) was used to create many of the examples in this document. Different processors have slight differences in implementation, primarily in ports and interfaces supported. Whenever information in the *HRM* conflicts with information in this chapter, it is assumed that the *HRM* is more correct.

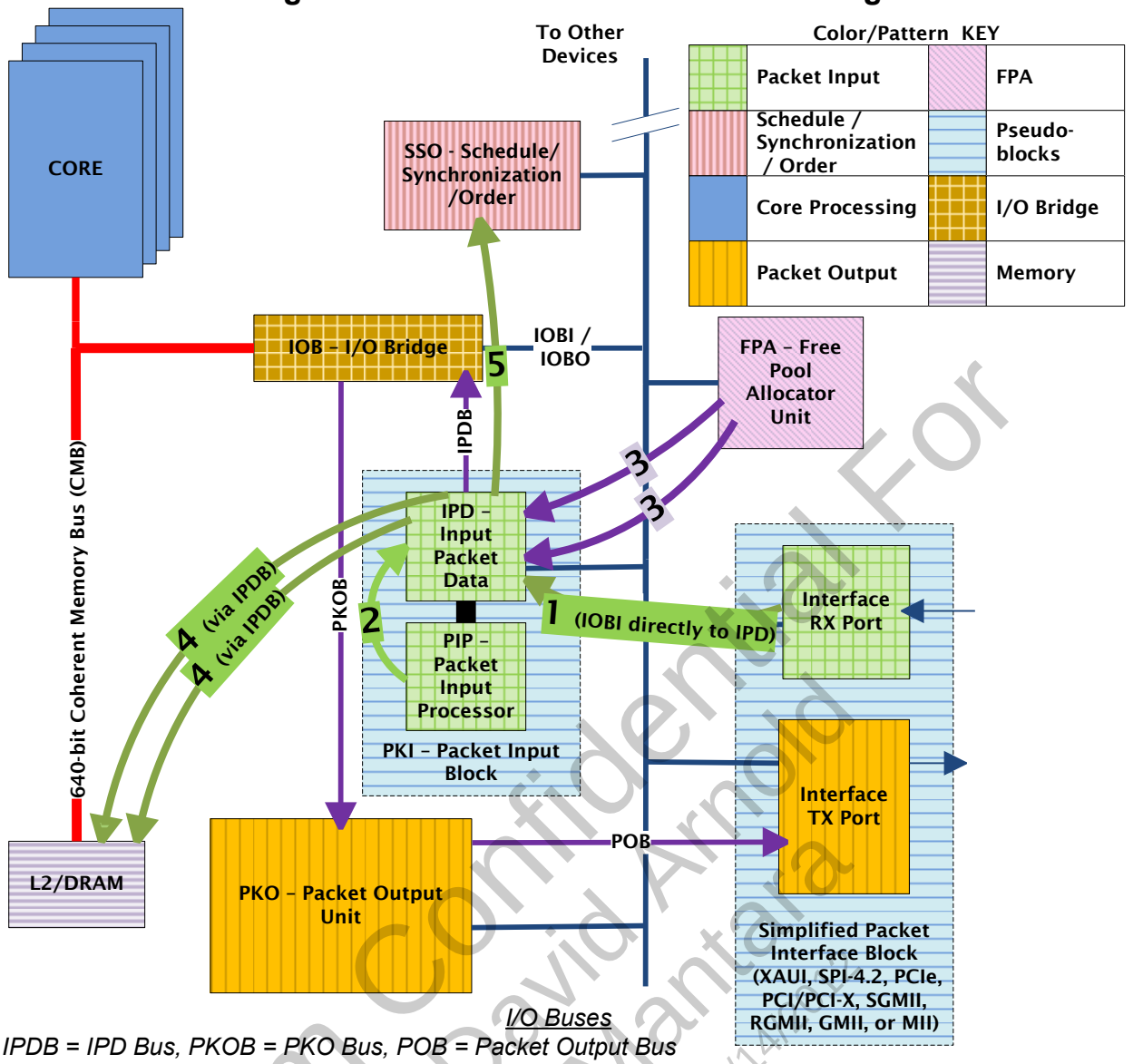
The *HRM* is an essential reference when writing customized software for the PIP/IPD units. This chapter is not intended to replace the *HRM*.

In this chapter, most register information matches the OCTEON CN55/55/56/57XX processor *HRM*, with some additions for CN63XX.

Note that in most cases the format `REGISTER[FIELD]` in this chapter refers to a hardware register and field combination, not a software `ARRAY[INDEX]`.

The following figure is from the *Packet Flow* chapter. This figure provides a high-level view the packet flow through the system. Proper configuration of PIP/IPD is essential for high-performance systems.

Figure 1: Overview of PIP/IPD Processing



IPDB = IPD Bus, PKOB = PKO Bus, POB = Packet Output Bus
The I/O Bus consists of two buses: IOBI (input) and IOBO (output). Received packet data goes directly from Interface RX to IPD on IOBI without going through IOB. (IPD is a second sink on the bus.)

1. After the Interface Rx Port receives the packet and checks it for errors, it passes the packet to the Input Packet Data (IPD) Unit (via the IOBI). The IPD shares the data with the Packet Input Processor (PIP). These two units work together to process the input packet.
2. After the PIP performs the packet parsing, including any checks configured by software, it computes the data needed by the IPD for the Work Queue Entry (WQE) Fields (Group, Tag Type, Tag Value, and QoS).
3. If IPD does not drop the packet, it allocates a WQE buffer and Packet Data buffer from the Free Pool Allocator (FPA) Unit. (The FPA manages the free buffers.)
4. The IPD writes the WQE fields to the WQE Buffer, and writes the packet data to the Packet Data buffer in L2/DRAM (DMA via IPDB).
5. The IPD performs the **`add_work`** operation to add the WQE Pointer to the appropriate QoS queue in the Schedule Synchronization Order (SSO) Unit.

2 Simple Executive Configuration and APIs

The PIP/IPD API supplied with the SDK does not cover all possible uses of the two units. This section provides an overview of the SDK 1.9.0 API, with some new features from SDK 2.0.

Most users take the default configuration provided by Simple Executive. The configuration can be customized (see the directions in the *Configuration* chapter, the *Free Pool Allocator (FPA)* chapter, and in this chapter).

An example of using the API functions is in Section 18 – "Appendix E: Example Code (`linux-filter`)".

Most applications will use the API as follows:

1) Define `CVMX_HELPER_ENABLE_IPD=0` // allows user to control when initialization is considered to be complete
2) Call `cvmx_helper_initialize_fpa()` to setup the FPA pools.
3) Call `cvmx_helper_initialize_packet_io_global()` once on only one core
4) Call `cvmx_helper_initialize_packet_io_local()` on each core. This will get all packet IO running.
5) Call the `cvmx_pip*` or `cvmx_ipd*` functions only to change (modify) the IPD/PIP defaults as needed. (For example, call `cvmx_pip_config_port()`.)
6) Call `cvmx_helper_ipd_and_packet_input_enable()`

Table 1: Summary of Relevant Functions

Function	Description
Helper Functions	
`cvmx_helper_initialize_packet_io_global()`	Initialize global PIP/IPD variables. This function calls `cvmx_ipd_config()` using values defined in `executive-config.h`.
`cvmx_helper_initialize_packet_io_local()`	Each core calls this after global initialization routine is complete
`cvmx_helper_ipd_and_packet_input_enable()`	Call once all initialization is complete
`cvmx_helper_setup_red()`	Configure Per-QoS RED for congestion control (all queues will have the same pass and drop thresholds).
`cvmx_helper_setup_red_queue()`	Configure Per-QoS RED or WRED for congestion control (each queue can have different pass and drop thresholds). Call `cvmx_helper_setup_red()` first, then call this function to modify queue thresholds as needed.

Function	Description
`cvmx_helper_shutdown_packet_io_global()`	(New in SDK 2.0). This function is used to shutdown the packet handling units, including IPD.
`cvmx_helper_shutdown_packet_io_local()`	(New in SDK 2.0.) This function does a core-local shutdown of packet I/O after the global shutdown is complete.
PIP Functions	
`cvmx_pip_config_port()`	Configure a PIP/IPD input port.
`cvmx_pip_config_crc()`	Configure the hardware CRC engine (on some processors).
`cvmx_pip_tag_mask_clear ()`	Clear all bits in a tag mask.
`cvmx_pip_tag_mask_set()`	Set bits in the selected tag mask (used to create tag value)
`cvmx_pip_config_vlan_qos()`	Configure VLAN-to-QoS Table 0
`cvmx_pip_config_diffserv_qos()`	Configure Diffserv-to-QoS table
`cvmx_pip_get_port_status()`	Get port statistics
IPD Functions	
`cvmx_ipd_config()`	Configure global settings for IPD.
`cvmx_ipd_enable(void)`	This function is used to enable the IPD if Simple Executive is configured to not enable IPD.
`cvmx_ipd_disable(void)`	Instead of calling this function, use the new SDK 2.0 function `cvmx_helper_shutdown_packet_io_global()`, which calls `cvmx_ipd_disable()` at the right time.

2.1 Simple Executive Configuration

PIP and IPD depend on proper configuration of Simple Executive defines and FPA pool configuration.

In addition to the pools, the following Simple Executive Configuration variables are applicable to PIP/IPD:

Table 2: Simple Executive PIP/IPD Configuration Variables

Define	Purpose	Default Value
PIP/IPD Configuration variables defined in `executive-config.h` *(alphabetical order)*		
CVMX_ENABLE_HELPER_FUNCTIONS	Enables essential functions such as `cvmx_helper_initialize_fpa()`. We strongly recommend use of the helper functions.	Un-defined
CVMX_ENABLE_LEN_M8_FIX	Enable fix for the known issue PKI-100 ("Size field is 8 too large in the WQE and next pointers"). If this variable is set to 0, the fix for this known issue will not be enabled.	1
CVMX_HELPER_ENABLE_BACK_PRESSURE	We strongly recommend use of this backpressure feature.	1
CVMX_HELPER_ENABLE_IPD	This will cause the IPD to be enabled after initialization. Once IPD is enabled, the hardware will start accepting packets. If configuration changes are made from the default, then set this configuration variable to 0 and, after custom changes are complete, then call `cvmx_ipd_enable()`.	1
CVMX_HELPER_FIRST_MBUFF_SKIP	The number of bytes to reserve before the start of the packet in the MBUF.	184 (See Note1)
CVMX_HELPER_INPUT_PORT_SKIP_MODE	Select either skip-to-L2, skip-to-IP, or uninterpreted.	See Note2
CVMX_HELPER_INPUT_TAG_INPUT_PORT	Use input port value in tag value creation.	1
CVMX_HELPER_INPUT_TAG_IPV4_DST_IP	Use IPv4 Destination IP address field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_IPV4_DST_PORT	Use IPv4 Destination Port field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_IPV4_PROTOCOL	Use IPv4 Protocol field in tag value creation.	0

Define	Purpose	Default Value
CVMX_HELPER_INPUT_TAG_IPV4_SRC_IP	Use IPv4 Source IP address field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_IPV4_SRC_PORT	Use IPv4 Source Port field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_IPV6_DST_IP	Use IPv6 Destination IP address field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_IPV6_DST_PORT	Use IPv6 Destination Port field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_IPV6_NEXT_HEADER	Use IPv6 Next Header field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_IPV6_SRC_IP	Use IPv6 Source IP address field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_IPV6_SRC_PORT	Use IPv6 Source Port field in tag value creation.	0
CVMX_HELPER_INPUT_TAG_TYPE	Either ORDERED, ATOMIC, or NULL.	See Note3
CVMX_HELPER_NOT_FIRST_MBUFF_SKIP	The number of bytes to reserve in each chained packet buffer (MBUF) after the first MBUF.	0

Notes

Note1: The default `CVMX_HELPER_FIRST_MBUFF_SKIP` value in the base SDK is set to 184 for compatibility with IPSEC to allow header expansion.

Note2: The default value for `CMVX_HELPER_INPUT_PORT_SKIP_MODE` is `CVMX_PIP_PORT_CFG_MODE_SKIPL2`. See `cvmx_pip_port_parse_mode_t` in `cvmx-csr-enums.h`.

Note3: The default value for `CVMX_HELPER_INPUT_TAG_TYPE` is `CVMX_POW_TAG_TYPE_ORDERED`.

2.1.1 About FPA Pools

In most applications, two FPA pools are used by PIP/IPD:

- The FPA Pool used for Packet Data Buffers is always FPA Pool 0
- The FPA pool used for Work Queue Entry (WQE) buffers is configurable, but is typically FPA Pool 1

FPA pool configuration information is provided in the *Free Pool Allocator (FPA)* chapter. If the default configuration will be changed, it is essential to read the *FPA* chapter. Pool information is summarized in this section.

The default pool configuration used in the SDK is shown in the following table.

Table 3: Default FPA Pool Configuration

Item	Default Value
Packet Data Buffers	
Pool Name	`CVMX_FPA_PACKET_POOL`
Description String	"Packet buffers"
Pool Number (Default value)	0 (cannot be changed)
Default Buffer Size	16 * cache line size (2048 bytes) - See Note1, Note2
Default Number of Buffers	Configurable via `cvmx_helper_initialize_fpa()`- See Note3
Protected / Permanent	1 (TRUE)
Work Queue Entry Buffers	
Name	`CVMX_FPA_WQE_POOL`
Description String	"Work queue entries"
Pool Number (Default value)	1 (This can be any number; it is set to 1 by convention)
Default Buffer Size	1 * cache line size (128 bytes) – See Note1, Note2
Default Number of Buffers	Usually the same as the number of Packet Data Buffers
Protected / Permanent	1 (TRUE)
Notes	
Note1: Buffer Size must be a minimum of 128 bytes (cache line size), and must be a multiple of 128 bytes (`CVMX_FPA_MIN_BLOCK_SIZE, CVMX_FPA_ALIGNMENT`).	
Note2: The default buffer size is configured in `cvmx-resources.config`	
Note3: See the `passthough` example code.	

The following tables provide the PIP/IPD perspective on the Packet Data Buffers and Work Queue Entry buffers.

Table 4: Packet Data Buffers Information

Packet Data Buffers
Unit Allocating Buffer
The IPD automatically allocates Packet Data Buffers. Packet Data Buffers are always in FPA pool 0: this is not configurable.
What controls the buffer allocation and use?
In the Simple Executive, the function `cvmx_helper_global_setup_ipd()` sets the value of `IPD_PACKET_MBUFF_SIZE[MB_SIZE]`. This value must match the size of the buffers in FPA pool 0. The IPD always allocates Packet Data Buffers from pool 0.

Packet Data Buffers
Recommended Buffer Size
Up to 2048 bytes (sixteen cache lines) (MTU of 1500 bytes).
Recommended Number of Buffers
Either 4096 or the maximum number of in-flight packets.
Unit Freeing Buffer
PKO or core software
How does the system know which pool the buffer should be returned to?
The originating FPA pool number is stored automatically in the Work Queue Entry data structure by the IPD. The PKO will optionally free the buffer to the specified pool (always Pool 0 for Packet Data Buffers). The core may also optionally free the Packet Data Buffer.

Table 5: Work Queue Entry Buffers Information

Work Queue Entry (WQE) Buffers
Unit Allocating Buffer
The IPD or the core (via software). The IPD automatically allocates WQE buffers.
What controls the buffer allocation and use?
In the Simple Executive, the function `cvmx_helper_global_setup_ipd()` sets the value of `IPD_WQE_FPA_QUEUE[WQE_QUE]`. This register field is used to specify which FPA pool the Work Queue Entry comes from.
Recommended Buffer Size
128 bytes (one cache line)
Recommended Number of Buffers
At least as many as Packet Data Buffers. If dynamic shorts are enabled, then packets which can fit entirely in the space reserved in the WQE will not also have a duplicate copy in the Packet Data Buffer. In the case where the WQE is in a WQE buffer (the option to have it in the Packet Data Buffer is not enabled), then the Packet Data Buffer will not exist for dynamic shorts, and more WQE Buffers will be needed than Packet Data Buffers.
Unit Freeing Buffer
Core software is responsible for freeing the buffer.
How does software know which pool the buffer should be returned to?
When freeing a WQE Buffer, use the define provided by the Simple Executive: `CVMX_FPA_WQE_POOL`

2.2 Helper Functions

These functions simplify configuring and using the PIP/IPD.

Table 6: Helper Functions

Helper API Functions
`int cvmx_helper_initialize_packet_io_global(void)`
Initialize global PIP/IPD variables. It initializes the PIP, IPD, and PKO hardware to support simple priority based queues for the Ethernet ports. Each port is configured with a number of priority queues based on `CVMX_PKO_QUEUES_PER_PORT_*` where each queue is lower priority than the previous. Returns 0 on success, otherwise returns non-zero.This function calls `cvmx_ipd_config()` using values defined in `executive-config.h`.
`int cvmx_helper_initialize_packet_io_local(void)`
Each core calls this after global initialization routine is complete. Returns 0 on success, otherwise returns non-zero.
`int cvmx_helper_ipd_and_packet_input_enable(void)`
Called after all internal packet IO paths are setup. This function enables IPD/PIP and begins packet input and output. Returns 0 on success, otherwise returns non-zero.
`int cvmx_helper_setup_red(int pass_thresh, int drop_thresh)`
Configure Per-QoS RED for congestion control (all queues will have the same pass and drop thresholds). The arguments are: `pass_thresh`: the HIGH watermark (if the number of available Packet Data Buffers is > `pass_thresh`, the packet is admitted) `drop_thresh`: the LOW watermark (if the number of available Packet Data buffers is <= drop_thresh, all incoming packets are dropped If `pass_thresh` >= number of available buffers > `drop_thresh`, packets are randomly dropped.
`cvmx_helper_setup_red_queue(int queue, int pass_thresh, int drop_thresh)`
Configure Per-QoS RED or WRED for congestion control (each queue can have different pass and drop thresholds). Call `cvmx_helper_setup_red()` first, then call this function to modify queue thresholds as needed. The arguments are: `queue`: which QoS queue's watermarks to set `pass_thresh`: the HIGH watermark (if the number of available Packet Data Buffers is > `pass_thresh`, the packet is admitted) `drop_thresh`: the LOW watermark (if the number of available Packet Data buffers is <= drop_thresh, all incoming packets are dropped If `pass_thresh` >= number of available buffers > `drop_thresh`, packets are randomly dropped.

Helper API Functions
`int cvmx_helper_shutdown_packet_io_global(void)`
New in SDK 2.0. This function is used to undo the the initialization performed in `cvmx_helper_initialize_packet_io_global()`. After calling this routine and the local version on each core, packet IO for the OCTEON processor will be disabled and placed in the initial reset state. It will then be safe to call the initialization function later on. Note that this routine does not empty the FPA pools. It frees all buffers used by the packet IO hardware to the FPA so a function emptying the FPA after shutdown should find all packet buffers in the FPA. Returns 0 on success, negative on failure.
`int cvmx_helper_shutdown_packet_io_local(void)`
New in SDK 2.0. This function does a core-local shutdown of packet I/O and should be called on each core after calling `cvmx_helper_shutdown_packet_io_global()`. Returns 0 on success, negative on failure.

2.3 PIP Functions

These functions can be used to change the default configuration created by the helper routines.

Table 7: PIP Functions

PIP API Functions (cvmx-pip.h)
`void cvmx_pip_config_port(uint64_t port_num, cvmx_pip_port_cfg_t port_cfg, cvmx_pip_port_tag port_tag_cfg)`
Configure an Ethernet input port. The arguments are: `port_num`: The port number to configure `port_cfg`: a data structure containing the configuration information `port_tag_cfg`: a data structure containing the port's tag configuration information
`void cvmx_pip_config_crc(uint64_t interface, uint64_t invert_result, uint64_t reflect, uint32_t initialization_vector)`
Configure the hardware CRC engine. The arguments are: `interface`: Interface to configure (0 or 1) `invert_result`: Invert the result of the CRC `reflect`: Reflect `initialization_vector`: CRC initialization vector
`cvmx_pip_tag_mask_clear (uint64_t mask_index)`
Clear all bits in a tag mask. This function should be called on startup before any calls to `cvmx_pip_tag_mask_set()`. Each bit set in the final mask represents a byte used in the packet for tag generation. The argument is: `mask_index`: Which tag mask to clear (0..3)
`cvmx_pip_tag_mask_set (uint64_t mask_index, uint64_t offset, uint64_t len)`
The tag mask is used when the `cvmx_pip_port_tag_cfg_t tag_mode` is non zero. There are four separate masks that can be configured. The arguments are: `mask_index`: which tag mask to modify (0..3) `offset`: offset into the bitmask to set bits at. Use the GCC macro `offsetof()` to determine the offsets into packet headers. For example, `offsetof(ethhdr, protocol)` returns the offset of the ethernet protocol field. The bitmask selects which bytes to include the tag, with bit offset X selecting byte at offset X from the beginning of the packet data. `len`: Number of bytes to include. Usually this is the `sizeof()` the field.
`void cvmx_pip_config_vlan_qos(uint64_t vlan_priority, unit64_t qos)`
Configures the VLAN priority to QOS mapping for VLAN-to-QOS Table0. Note there is no function to configure VLAN-to-QOS Table1. The arguments are: `vlan_priority`: 0-7 `qos`: QOS value to assign to incoming packets with VLAN priority matching this VLAN priority.

PIP API Functions (cvmx-pip.h)
`void cvmx_pip_config_diffserv_qos(unit64_t diffserv, unit64_t qos)`
Configures the Diffserv to QOS mapping. Note this function does not enable Diffserv QOS for the port. The arguments are: `diffserv`: diffserv field value (0-63) `qos`: QOS value to assign to incoming packets with Diffserv value matching this Diffserv field value.
`void cvmx_pip_get_port_status(unit64_t port_num, uint64_t clear, cvmx_pip_port_status_t *status)`
Get the statistics for a port. The arguments are: `port_num`: the port number `clear`: whether to clear the values after reading them (1=clear, 0=do not clear) `status`: the data structure used to store the status On success, this function retrieves the port status and stores it in the `status` data structure.

2.4 PIP Data Structures and Defines

2.4.1 The `cvmx_pip_port_cfg_t` Data Structure

This data structure is used to specify the configuration parameters for each port. The contents of the data structure vary with the processor model. See `cvmx_pip_port_cfgx_t` in `cvmx-csr-typedefs.h` in the SDK for details.

2.4.2 The `cvmx_pip_port_tag_cfg_t` Data Structure

This data structure is used to specify the tag configuration parameters for each port. The contents of the data structure vary with the processor model. See `cvmx_pip_port_tag_cfgx_t` in `cvmx-csr-typedefs.h` in the SDK for details.

2.4.3 The `cvmx_pip_parse_mode_t` Defines (Parse Modes for Incoming Packets)

These defines (enumerated in `cvmx_pip_parse_mode_t`) are used to set the parse mode for the incoming packet:

```
CVMX_PIP_PORT_CFG_MODE_NONE = 0ull,   // Uninterpreted
CVMX_PIP_PORT_CFG_MODE_SKIPL2 = 1ull, // Skip-to-L2
CVMX_PIP_PORT_CFG_MODE_SKIPIP = 2ull  // Skip-to-IP
```

2.4.4 The `cvmx_pip_tag_mode_t` Defines (control the initial SSO Tag Value)

These defines (enumerated in `cvmx_pip_tag_mode_t`) are used to set the initial Tag Value for the incoming packet:

```
CVMX_PIP_TAG_MODE_TUPLE = 0ull,          // Always use tuple tag algorithm.
CVMX_PIP_TAG_MODE_MASK = 1ull,           // Always use mask tag algorithm
CVMX_PIP_TAG_MODE_IP_OR_MASK = 2ull,     // If packet is IP, use tuple else
```

```
                                        // use mask
CVMX_PIP_TAG_MODE_TUPLE_XOR_MASK = 3ull // tuple XOR mask
```

2.4.5 The `cvmx_pow_tag_type_t` Defines (control the initial SSO Tag Type)

These defines (enumerated in `cvmx_pow_tag_type_t`) are used to set the initial Tag Type for the incoming packet:

```
CVMX_POW_TAG_TYPE_ORDERED   = 0L,   // ORDERED
CVMX_POW_TAG_TYPE_ATOMIC    = 1L,   // ATOMIC
CVMX_POW_TAG_TYPE_NULL      = 2L,   // NULL
```

2.4.6 The `cvmx_pip_port_status_t` Data Structure

PIP statistics registered are accessed via the `cvmx_pip_get_port_status()` function, which returns the information in the `cvmx_pip_port_status_t` data structure. This information is the same for all processors.

For register-level details, see Table 30 – "Statistics Register Fields".

```
typedef struct
{
    uint32_t    dropped_octets;         // Inbound octets marked to be
                                        // dropped by the IPD
    uint32_t    dropped_packets;        // Inbound packets marked to be
                                        // dropped by the IPD
    uint32_t    pci_raw_packets;        // RAW PCI Packets received by
                                        // PIP per port
    uint32_t    octets;                 // Number of octets processed by PIP
    uint32_t    packets;                // Number of packets processed by PIP
    uint32_t    multicast_packets;      // Number of indentified
                                        // L2 multicast packets.
                                        // (Does not include broadcast packets.
                                        // Only includes packets whose
                                        // parse mode is SKIP_TO_L2)
    uint32_t    broadcast_packets;      // Number of indentified L2 broadcast
                                        // packets.  Does not include multicast
                                        // packets.  Only includes packets whose
                                        // parse mode is SKIP_TO_L2
    uint32_t    len_64_packets;         // Number of 64B packets
    uint32_t    len_65_127_packets;     // Number of 65-127B packets
    uint32_t    len_128_255_packets;    // Number of 128-255B packets
    uint32_t    len_256_511_packets;    // Number of 256-511B packets
    uint32_t    len_512_1023_packets;   // Number of 512-1023B packets
    uint32_t    len_1024_1518_packets;  // Number of 1024-1518B packets
    uint32_t    len_1519_max_packets;   // Number of 1519-max packets
    uint32_t    fcs_align_err_packets;  // Number of packets with FCS or
                                        // Align opcode errors
    uint32_t    runt_packets;           // Number of packets with length < min
    uint32_t    runt_crc_packets;       // Number of packets with
                                        // length < min and FCS error
    uint32_t    oversize_packets;       // Number of packets with length > max
    uint32_t    oversize_crc_packets;   // Number of packets with
                                        // length > max and FCS error
    uint32_t    inb_packets;            // Number of packets without
                                        // GMX/SPX/PCI errors received by PIP
    uint64_t    inb_octets;             // Total number of octets from all
                                        // packets received by PIP,
                                        // including CRC
    uint16_t    inb_errors;             // Number of packets with GMX/SPX/PCI
                                        // errors received by PIP
} cvmx_pip_port_status_t;
```

2.4.7 The `cvmx_pip_err_t` Data Structure

The opcode field in WQE WORD2 is used to report the error details. The meaning of opcode depends on which of WORD2[RE] (L1/L2 receive error), WORD2[LE] (L2 receive error), or WORD2[IE] (L4 receive error) is set in WQE WORD2.

```
/**
 * This defines the err_code field errors in the Work Queue Entry
**/
typedef union
{
    cvmx_pip_l4_err_t  l4_err;  // L3 receive error (WORD2[LE]==1)
    cvmx_pip_ip_exc_t  ip_exc;  // L4 receive error (WORD[IE]==1)
    cvmx_pip_rcv_err_t rcv_err; // L1/L2 receive error (WORD2[RE]==1)
} cvmx_pip_err_t;
```

2.4.8 The Packet Instruction Header Data Structure

This data structure is defined in Section 4.3.1 – "The `cvmx_pip_inst_hdr_t` Data Structure".

2.4.9 L1/L2 Receive Error Codes (WQE WORD2[RE] ==1)

If there is a receive error, then Work Queue Entry (WQE) WORD2[RE] field is set to 1, and WORD2[opcode] contains the error code. When using the SDK, the following list of error codes apply. These definitions are made in the `cvmx_pip_rcv_err_t` data structure. For more details about the error codes, see the *HRM*.

Note: Late collisions (data received before collision) cannot be detected by the receiver because they would appear as `JAM` bits which would appear as bad FCS or carrier extend error which is `CVMX_PIP_EXTEND_ERR`.

```
    CVMX_PIP_RX_NO_ERR          // no error
    CVMX_PIP_PARTIAL_ERR        // RGMII+SPI4: partially received packet
                                // (buffering/bandwidth) not adequate
    CVMX_PIP_JABBER_ERR         // RGMII+SPI4: receive packet too
                                // large and truncated
    CVMX_PIP_OVER_FCS_ERR       // RGMII: max frame error
                                // (pkt len > max frame len) (with FCS error)
    CVMX_PIP_OVER_ERR           // RGMII+SPI4:  max frame error
                                // (pkt len > max frame len)
    CVMX_PIP_ALIGN_ERR          // RGMII: nibble error (data not byte
                                // multiple - 100M and 10M only)
    CVMX_PIP_UNDER_FCS_ERR      // RGMII:  min frame error
                                // (pkt len < min frame len) (with FCS error)
    CVMX_PIP_GMX_FCS_ERR        // RGMII:  FCS error
    CVMX_PIP_UNDER_ERR          // RGMII+SPI4: min frame error
                                // (pkt len < min frame len)
    CVMX_PIP_EXTEND_ERR         // RGMII:  Frame carrier extend error
    CVMX_PIP_LENGTH_ERR         // RGMII: length mismatch (len did
                                // not match len in L2 length/type)
    CVMX_PIP_DAT_ERR            // RGMII: Frame error (some or all data
                                // bits marked err)
    CVMX_PIP_DIP_ERR            // SPI4: DIP4 error
    CVMX_PIP_SKIP_ERR           // RGMII: packet was not large enough to pass
                                // the skipper - no inspection could occur
```

```
CVMX_PIP_NIBBLE_ERR       // RGMII: studder error (data not repeated -
                          // 100M and 10M only)
CVMX_PIP_PIP_FCS          // RGMII+SPI4: FCS error
CVMX_PIP_PIP_SKIP_ERR     // RGMII+SPI4+PCI: packet was not large enough
                          // to pass the skipper - no inspection
                          // could occur
CVMX_PIP_PIP_L2_MAL_HDR=  // RGMII+SPI4+PCI: malformed L2 (packet
                          // not long enough to cover L2 header)
```

2.4.10 L3 (IP) Error Codes (WQE WORD2[IE]==1)

If the WQE WORD2[IE] field is set to 1 (IP error), the following error codes apply. These error codes are defined in the `cvmx_pip_ip_exc_t` data structure. For more details about the error codes, see the *HRM*.

```
CVMX_PIP_IP_NO_ERR        // no error
CVMX_PIP_NOT_IP           // not IPv4 or IPv6
CVMX_PIP_IPV4_HDR_CHK     // IPv4 header checksum violation
CVMX_PIP_IP_MAL_HDR       // malformed (packet not long enough to
                          // cover IP header
CVMX_PIP_IP_MAL_PKT       // malformed (packet not long enough to
                          // cover length specified in IP header)
CVMX_PIP_TTL_HOP          // TTL / hop count equal zero
CVMX_PIP_OPTS             // IPv4 options / IPv6 early extension headers
```

2.4.11 L4 Error Codes (WQE WORD2[LE]==1)

L4 Error codes are shown in the following list. These error codes are defined in the `cvmx_pip_l4_err_t l4_err` data structure. For more details about the error codes, see the *HRM*.

```
CVMX_PIP_L4_NO_ERR      // No error
CVMX_PIP_L4_MAL_ERR     // TCP (UDP) packet not long enough to cover the
                        // TCP (UDP)header
CVMX_PIP_CHK_ERR        // TCP/UDP checksum failure
CVMX_PIP_L4_LENGTH_ERR  // TCP/UDP length check (TCP/UDP length does not
                        // match IP length)
CVMX_PIP_BAD_PRT_ERR    // illegal TCP/UDP port (either source or dest
                        // port is zero)
CVMX_PIP_TCP_FLG8_ERR   // TCP flags = FIN only
CVMX_PIP_TCP_FLG9_ERR   // TCP flags = 0
CVMX_PIP_TCP_FLG10_ERR  // TCP flags = FIN+RST+*
CVMX_PIP_TCP_FLG11_ERR  // TCP flags = SYN+URG+*
CVMX_PIP_TCP_FLG12_ERR  // TCP flags = SYN+RST+*= 12ull,
                            CVMX_PIP_TCP_FLG13_ERR  // TCP flags = SYN+FIN+*
```

2.5 IPD Functions

Table 8: IPD API Functions

IPD API Functions
`static inline void cvmx_ipd_config(uint64_t mbuff_size, uint64_t first_mbuff_skip, uint64_t not_first_mbuff_skip, uint64_t first_back, uint64_t second_back, uint64_t wqe_fpa_pool, cvmx_ipd_mode_t cache_mode, uint64_t back_pres_enable_flag )`

Configure global settings for IPD. This function is called when `cvmx_helper_initialize_packet_io_global()` is executed, before IPD is enabled using values configured into Simple Executive (see Note1).
`mbuff_size`: Packets buffer size in 8-byte words. This size may be set to the same as or less than the size of the Packet Data Buffer.
`first_mbuff_skip`: Number of 8-byte words to skip in the first buffer
`not_first_mbuff_skip`: Number of 8-byte words to skip in each following buffer
`first_back`: Must be same as `first_mbuff_skip / 128`
`second_back`: Must be same as `not_first_mbuff_skip / 128`
`wqe_fpa_pool`: FPA pool to get work entries from
`cache_mode`: Select the style of write to the L2 Cache (`IPD_CTL_STATUS[OPC_MODE]`)
Cache mode can be any of:
`CVMX_IPD_OPC_MODE_STT /* All blocks DRAM, not cached in L2 */`
`CVMX_IPD_OPC_MODE_STF /* All blocks into L2 */`
`CVMX_IPD_OPC_MODE_STF1_STT /* 1st block L2, rest DRAM */`
`CVMX_IPD_OPC_MODE_STF2_STT /* 1st, 2nd blocks L2, rest DRAM */`
`back_pres_enable_flag`: Enable or disable port back pressure (`IPD_CTL_STATUS[PBP_EN]`)
 	
Note: When `cvmx_ipd_config()` is called using the default values configured into Simple Executive, the values are:
`mbuff_size`: `CMVX_FPA_PACKET_POOL_SIZE / 8 // the entire Packet Data Buffer`
`first_mbuff_skip`: `CVMX_HELPER_FIRST_MBUFF_SKIP / 8`
`not_first_mbuff_skip`: `CMVX_HELPER_NOT_FIRST_MBUFF_SKIP / 8`
`first_back`: `(CVMX_HELPER_FIRST_MBUFF_SKIP + 8) / 128` (+8 is for next ptr)
`second_back`: `(CVMX_HELPER_NOT_FIRST_MBUFF_SKIP + 8)/128` (+8 is for next ptr)
`wqe_fpa_pool`: `CVMX_FPA_WQE_POOL`
`cache_mode`: `CVMX_IPD_OPC_MODE_STT`
`back_pres_enable_flag`: `CVMX_HELPER_ENABLE_BACK_PRESSURE`

`static inline void cvmx_ipd_enable(void)`
This function is used to enable the IPD if Simple Executive is configured to not enable IPD (CVMX_HELPER_ENABLE_IPD is defined to 0). This is done if the user will add customizations after Simple Executive configuration functions complete. Note: Configuration changes after the IPD is enabled will result in a race condition, specifically "invalid" packet parsing results for those packets which arrived before the configuration changes.

IPD API Functions
`static inline void cvmx_ipd_disable(void)`
Instead of calling this function, use the `cvmx_helper_shutdown_packet_io_global()`, which calls `cvmx_ipd_disable()` at the right time. This function can be used to shutdown ALL packet reception. This function is called when reinitializing the packet interface. It is used by the cavium-ethernet Linux driver when the module is removed.

2.6 IPD Defines

2.6.1 The `cvmx_ipd_mode_t` Defines (How data is stored)

These defines (enumerated in `cvmx_ipd_mode_t`) are used to set how packet data is written the L2 cache.

```
typedefs enum {
   CVMX_IPD_OPC_MODE_STT = 0LL;       // Write all blocks DRAM, none are
                                      // cached in the L2
   CVMX_IPD_OPC_MODE_STF = 1LL;       // Write all blocks into L2
   CVMX_IPD_OPC_MODE_STF1_STT = 2LL; // Write first cache block to L2 cache,
                                      // others to DRAM
   CVMX_IPD_OPC_MODE_STF2_STT = 3LL; // Write first two cache blocks to
                                      // L2 cache, others to DRAM
} cvmx_ipd_mode_t
```

2.7 Beyond the SDK: Custom Software

Starting at Section 4 – "Incoming Packet Formats", technical details about PIP/IPD are provided to help the user customize Simple Executive or write custom software.

The user should also refer to the *HRM* to get precise technical details for the specific model of OCTEON being used.

3 IPD Input Ports

When using the PIP/IPD, many registers names include the IPD port number. The IPD port numbering conventions vary depending on the processor and the specific hardware configuration.

Port numbers follow these conventions:

- Packet interface 0: ports 0-15 (See note below)
  - XAUI: port 0
  - SGMII/RGMII: ports 0-3
  - SPI-4.2: ports 0-15
- Packet interface 1: ports 16-31
  - For XAUI config: port 16
  - For SGMI/RGMII: ports 16-19
  - For SPI-4.2: 16-31
- PCI/PCIe/DPI (sRIO Memory Accesses): ports 32-35
- Loopback: ports 36-39
- sRIO Messages: ports 40-43

Note: The processor which is most confusing on port numbering is the CN54XX/CN55XX because packet interface 0 port numbers start at 16.

This section provides information on CN56XX and CN57XX IPD input ports first because this processor contains a super-set of most options. This information is followed by CN54XX and CN55XX information. Before reading the CN54XX and CN55XX information, it is worthwhile to skim the CN56XX and CN57XX information. The configuration is similar, and easier to understand on the CN56XX/CN57XX.

IPD port information for other processors is located in Section 19 – "Appendix F: Input Port Configuration".

3.1 CN56XX and CN57XX IPD Input Ports

The following figures illustrate the input ports for the CN54/55/56/57XX processors.

For example, the CN56XX and CN57XX processors offer the following hardware configuration options:

Table 9: CN56XX and CN57XX Packet Input Configuration Options

Choice	PCIe Port 0	Packet Interface 0	PCIe Port 1	Packet Interface 1
1	4 lanes	SGMII	4 lanes	SGMII
2	4 lanes	SGMII	4 lanes	XAUI
3	4 lanes	XAUI	4 lanes	SGMII
4	4 lanes	XAUI	4 lanes	XAUI
5	8 lanes	not available	4 lanes	SGMII
6	8 lanes	not available	4 lanes	XAUI
7	4 lanes	SGMII	8 lanes	not available
8	4 lanes	XAUI	8 lanes	not available
9	8 lanes	not available	8 lanes	not available

Notes

When PCIe port 0 is configured to use 8 lanes (instead of 4 lanes), QLM0 & QLM1 have been dedicated for PCIe Port 0 use only; Packet Interface 0 is not available.

When PCIe port 1 is configured to use 8 lanes (instead of 4 lanes), QLM2 & QLM3 have been dedicated for PCIe Port 1 use only; Packet Interface 1 is not available.

The following figure illustrates the resultant IPD port numbers. For example, if the processor is configured with two packet interfaces, both in SGMII mode, and two PCIe controllers with of 4 PCIe lanes each:

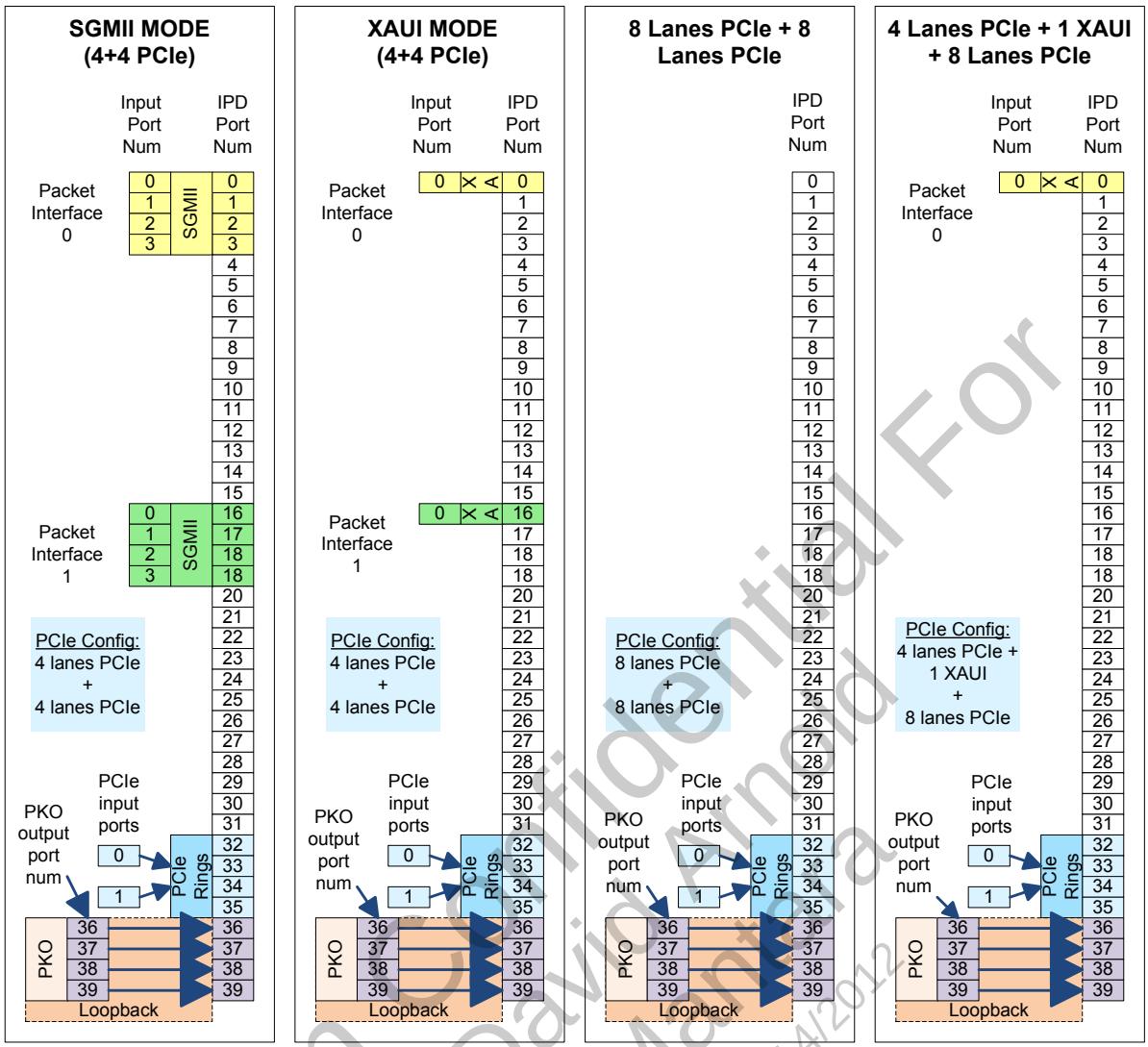
- Packet Interface 0 SGMII ports = 0-3
- Packet Interface 1 SGMII ports = 16-19
- PCIe ports = 32-35
- Loopback ports = 36-39 (PKO output ports 36-39 are connected to IPD input ports 36-39)

In another example, if the processor is configured with one packet interface in XAUI mode, and one PCIe controller configured to be 4 lanes, while the other PCIe controller is configured to be 8 lanes (packet interface 1 is unused in this configuration):

- Packet Interface 0 XAUI port = 0
- PCIe ports = 32-35
- Loopback ports = 36-39 (PKO output ports 36-39 are connected to IPD input ports 36-39)

Note that from an IPD port point of view, the number of PCIe lanes is invisible.

Figure 2: CN56XX and CN57XX IPD Input Ports



Note1: Input is provided by 4 Quad-Lane Input Modules (QLMs), each able to provide 4 lanes of SerDes.
QLM0: dedicated to PCIe, connected to PCIe controller 0.
QLM1: optionally either connected to PCIe controller 0 or to Packet Interface 0
QLM2: dedicated to PCIe, connected to PCIe controller 1.
QLM3: optionally either connected to PCIe controller 1 or to Packet Interface 1
If QLM1 is configured as PCIe, it is combined with QLM0 to provide 8 lanes of SerDes on PCIe controller 0.
If QLM3 is configured as PCIe, it is combined with QLM2 to provide 8 lanes of SerDes on PCIe controller 1.

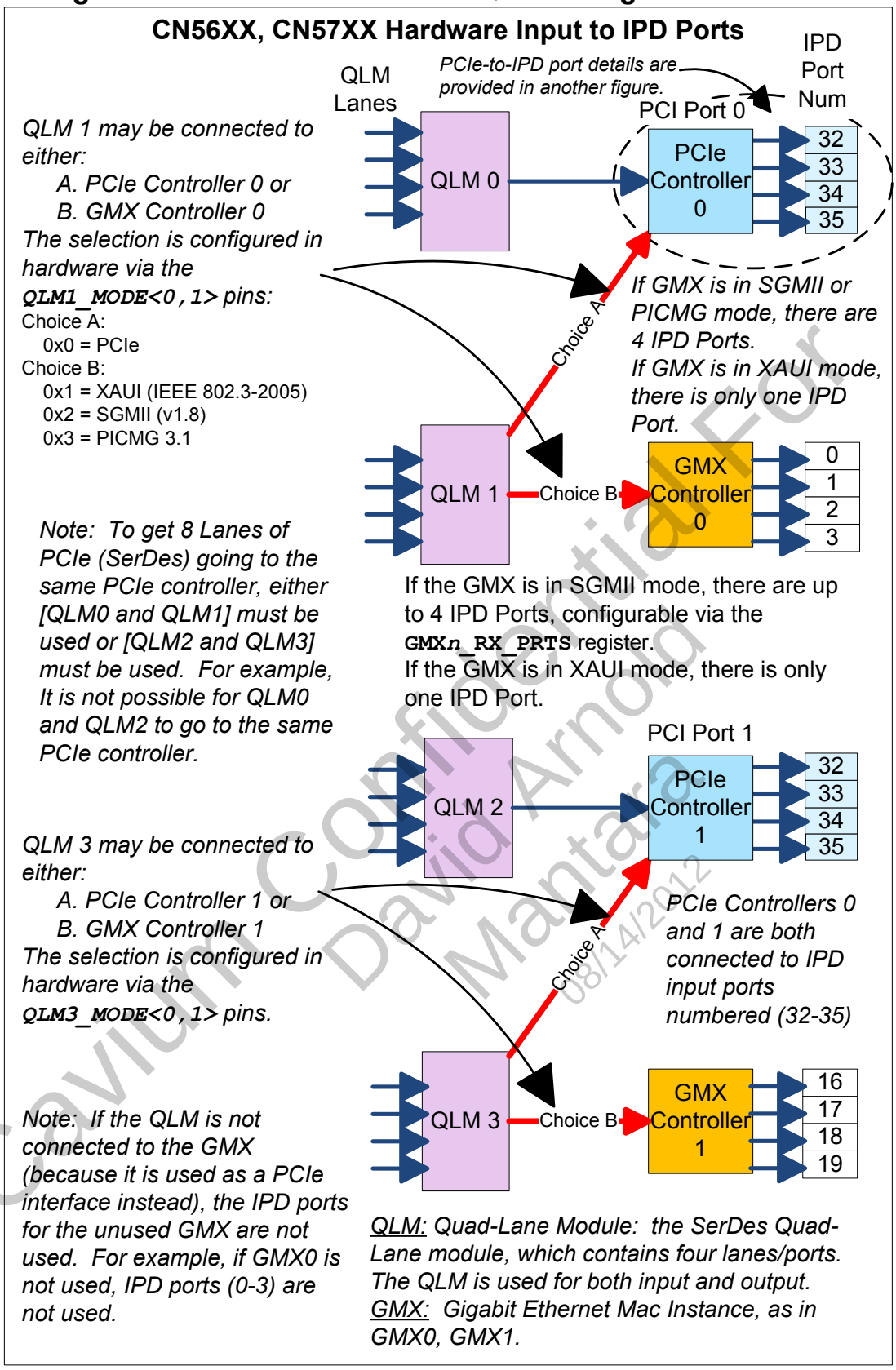
Note2: If the QLM is configured as a packet interface, its packet interface type may be configured to be SGMII, XAUI (shown as "XA" in the figure), or PICMG.. QLM mode by configured in hardware.

Note3: IPD ports which shown in white are unused. This information is provided to emphasize the gap in IPD port numbers.

Note4: The Media-Independent Interface (MII) does not supply packets to IPD. MII packets traverse memory ring buffers outside of the IPD/PIP/SSO path.

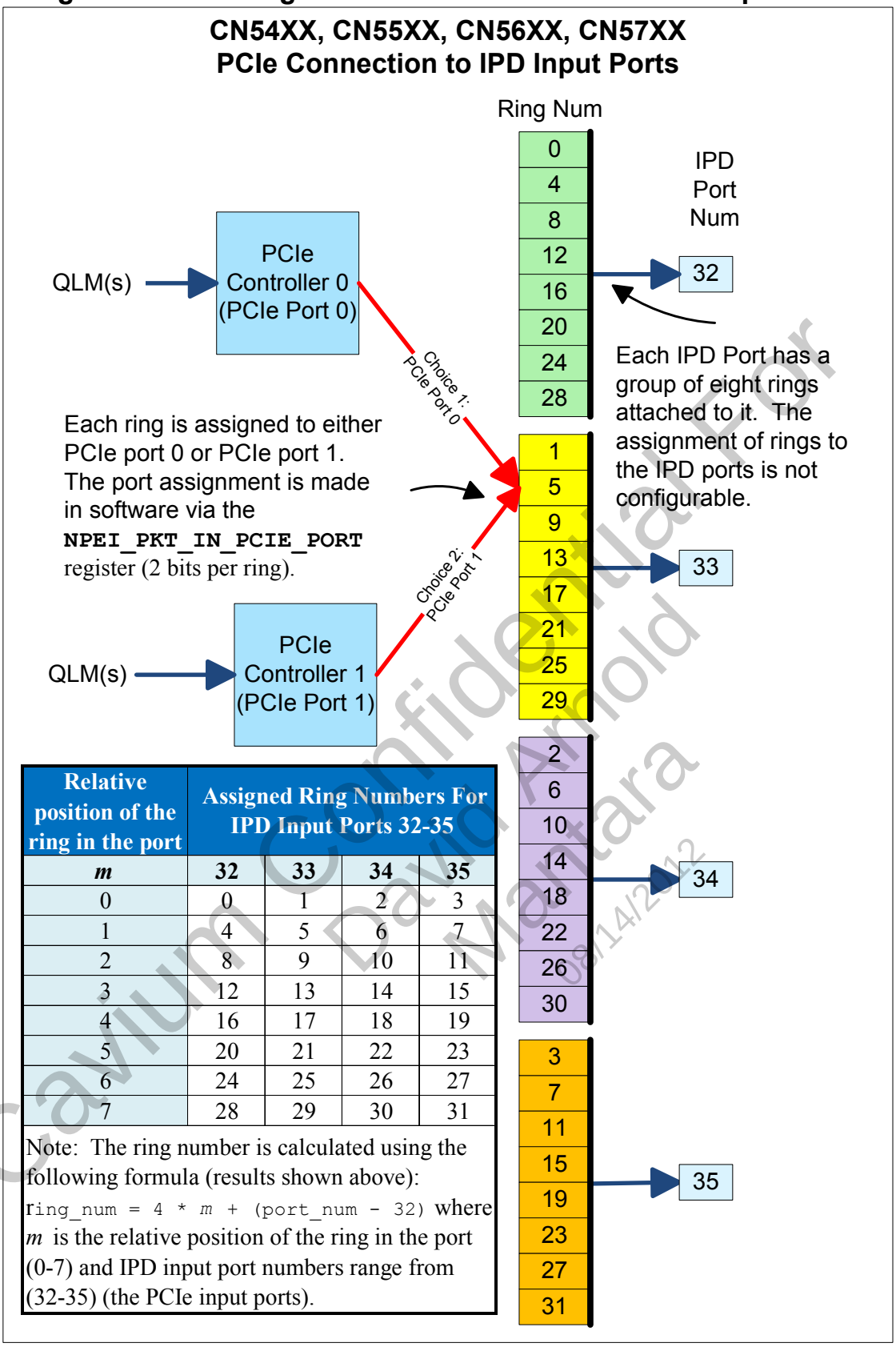
For the CN56XX andCN57XX, there are four Quad-Lane Modules (QLMs) which can be configured in hardware in different ways. Although this is hardware-level information, it may be useful to software engineers to visualize the system. Understanding this figure is also helpful for understanding PCIe ring configuration. This figure shows the option of connecting a QLM to a Gigabit Ethernet MAC Instance (GMX) controller, or a PCIe controller. On this processor, the GMX can be configured in hardware to be in either SGMII or XAUI mode.

Figure 3: CN56XX and CN57XX QLM Configuration Choices



The connection from the two PCIe ports to the IPD port is via PCIe rings. There are eight PCIe rings assigned to each IPD port. The assignment of the PCIe rings to the IPD ports is not configurable. Software can configure which of the two PCIe ports provides input to which PCIe ring, as shown in the following figure:

Figure 4: PCIe Rings: PCIe Port Connection to IPD Input Ports



Relative position of the ring in the port	Assigned Ring Numbers For IPD Input Ports 32-35			
m	**32**	**33**	**34**	**35**
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19
5	20	21	22	23
6	24	25	26	27
7	28	29	30	31

Note: The ring number is calculated using the following formula (results shown above):
`ring_num = 4 * m + (port_num - 32)` where *m* is the relative position of the ring in the port (0-7) and IPD input port numbers range from (32-35) (the PCIe input ports).

3.2 CN54XX and CN55XX IPD Input Ports

The CN54XX and CN55XX processors provide different hardware configuration options. The essential information is the packet interface 0 is not available.

Table 10: CN54XX and CN55XX Packet Input Configuration Options

Choice	PCIe Port 0	Packet Interface 0	PCIe Port 1	Packet Interface 1
1	8 lanes	not available	4 lanes	SGMII
2	8 lanes	not available	4 lanes	XAUI
3	8 lanes	not available	8 lanes	not available
Notes				
QLM0 & QLM1 have been dedicated for PCIe Port 0 use only; Packet Interface 0 is not available. *When PCIe port 1 is configured to use 8 lanes (instead of 4 lanes), QLM2 & QLM3 have been dedicated for PCIe Port 1 use only; Packet Interface 1 is not available.*				

The following figure illustrates the resultant IPD port numbers. Note that Packet Interface 0 is missing, so that IPD port numbers start at "16".

For example, if the processor is configured with one packet interfaces in SGMII mode, the first PCIe controller configured with 8 PCIe lanes, and the second PCIe controller configured with of 4 PCIe lanes:

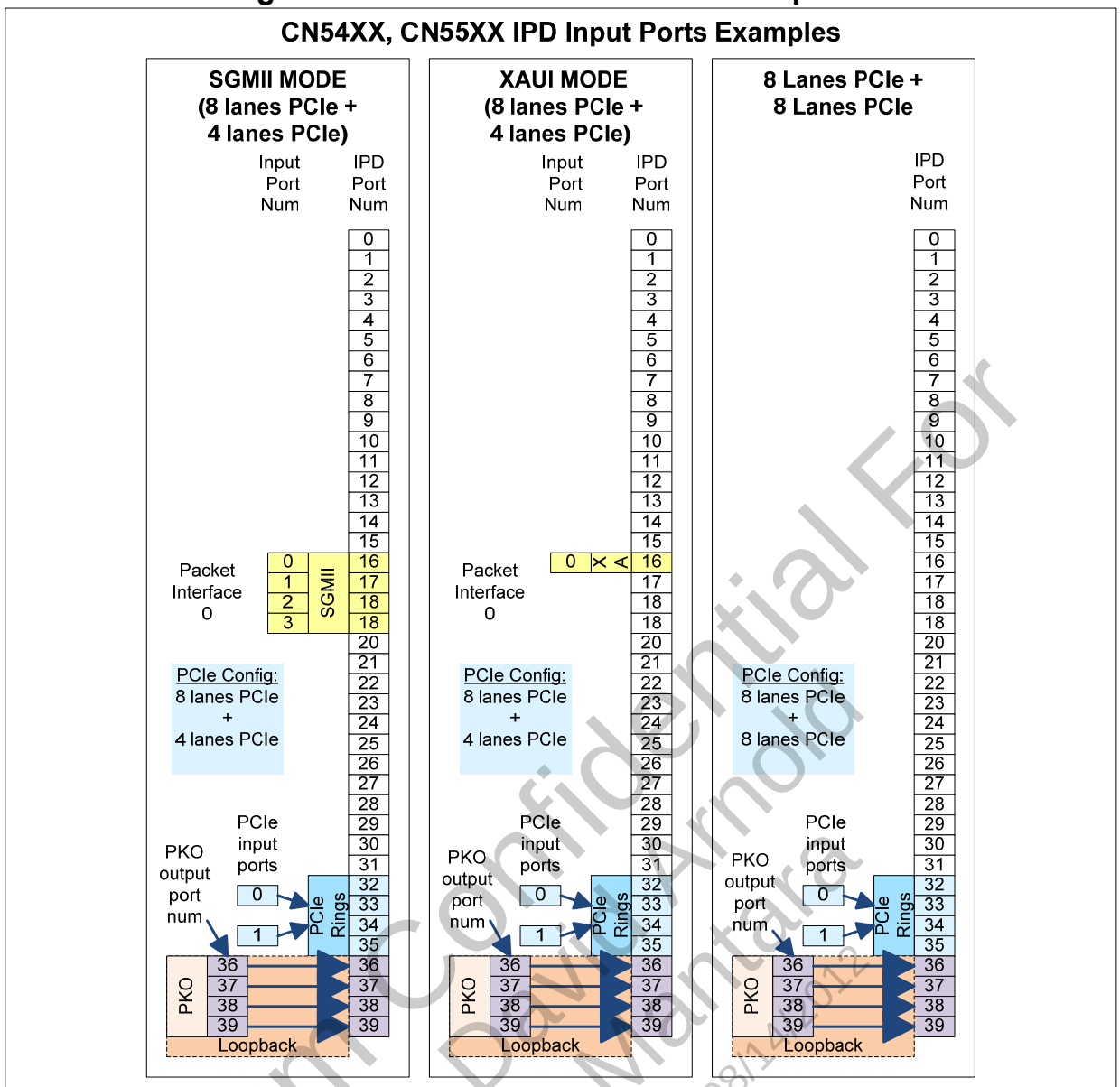
- Packet Interface 1 SGMII ports = 16-19
- PCIe ports = 32-35
- Loopback ports = 36-39 (PKO output ports 36-39 are connected to IPD input ports 36-39)

In another example, if the processor is configured with no packet interfaces, and both PCIe controllers configured for 8 PCIe lanes:

- PCIe ports = 32-35
- Loopback ports = 36-39 (PKO output ports 36-39 are connected to IPD input ports 36-39)

Note that from an IPD port point of view, the number of PCIe lanes is invisible.

Figure 5: CN54XX and CN55XX IPD Input Ports



Note 1: Input is provided by 4 Quad-Lane Modules (QLMs), each able to provide 4 lanes of SerDes.
QLM0 and QLM1: dedicated to PCIe, connected to PCIe controller 0 (8 lanes of SerDes)
QLM2: dedicated to PCIe, connected to PCIe controller 1 (4 lanes of SerDes)
QLM3: optionally either connected to PCIe controller 1, or to Packet Interface 1
If QLM3 is configured as PCIe, it is combined with QLM2 to provide 8 lanes of SerDes on PCIe controller 1

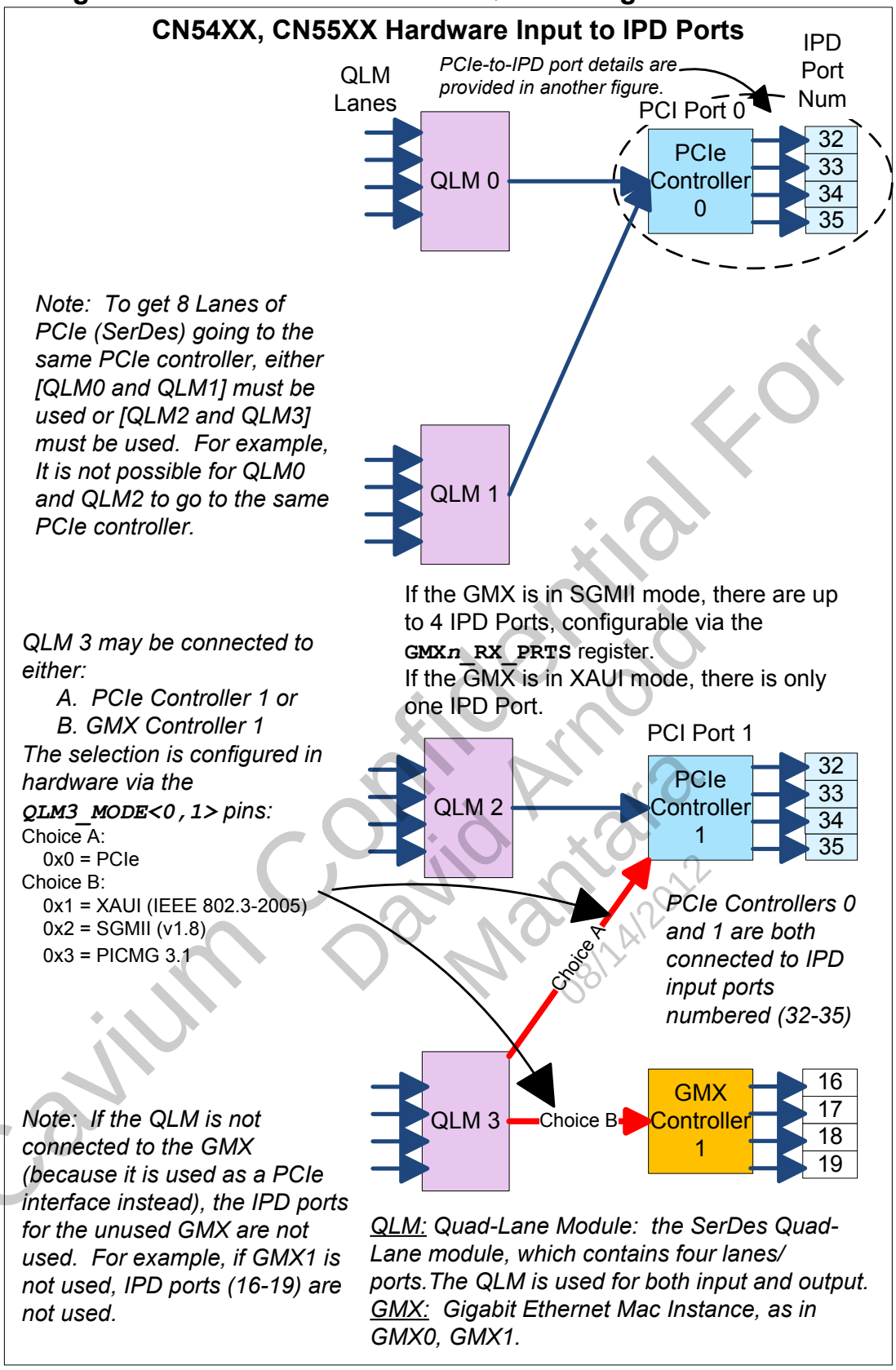
Note 2: If QLM3 is configured as a packet interface, its packet interface type may be configured to be SGMII, XAUI (shown as "XA" in the figurer), or PICMG. QLM mode is configured by hardware.

Note 3: IPD ports which shown in white are unused. This information is provided to emphasize the gap in IPD port numbers.

Note 4: The Media-Independent Interface (MII) does not supply packets to IPD. MII packets traverse memory ring buffers outside of the IPD/PIP/SSO path.

For the CN54XX andCN55XX, there are four Quad-Lane Modules (QLMs) which can be configured in hardware in different ways. The first two QLMS (QLM0 and QLM1) are dedicated to PCIe. Although this is hardware-level information, it may be useful to software engineers to visualize the system. Understanding this figure is helpful for understanding PCIe ring configuration. This figure shows the option of connecting a QLM to a Gigabit Ethernet MAC Instance (GMX) controller, or a PCIe controller. On this processor, the GMX can be configured in hardware to be in either SGMII or XAUI mode.

Figure 6: CN54XX and CN55XX QLM Configuration Choices



The connection from the two PCIe ports to the IPD port is via PCIe rings. There are eight PCIe rings assigned to each IPD port. The assignment of the PCIe rings to the PCIe ports is not configurable. Software can configure which of the two PCIe ports provides input to which PCIe ring, as shown in Figure 4 – "PCIe Rings: PCIe Rings: PCIe Port Connection to IPD Input Ports".

4 Incoming Packet Formats

Incoming packets can have a variety of formats. The most common format is simply an IP packet with an L2 header, TCP/UDP header, data, and a CRC. This section introduces the supported formats.

Whatever format is used, PIP/IPD has an overall goal which must be achieved.

4.1 Overall Processing Goal

PIP/IPD will attempt to receive the packet, and perform error checks on it. It will create a WQE and save the packet data. When the WQE is created, the following information will be included:

- Hardware checksum
- Scheduling information needed by the SSO:
  - QoS
  - Group
  - Tag Value
  - Tag Type
- The total packet length
- The physical address of the start of packet data in the Packet Data Buffer (not the same as the start of the Packet Data Buffer), unless the packet is entirely contained in the Work Queue Entry (dynamic short)
- Packet information such as:
  - Errors and error codes
  - IP information (IPv4 or IPv6, TCP or UDP, Fragment, etc)
  - VLAN information (VLAN, VLAN STACKED, VLAN ID, VLAN CFI bit)
  - User-defined information

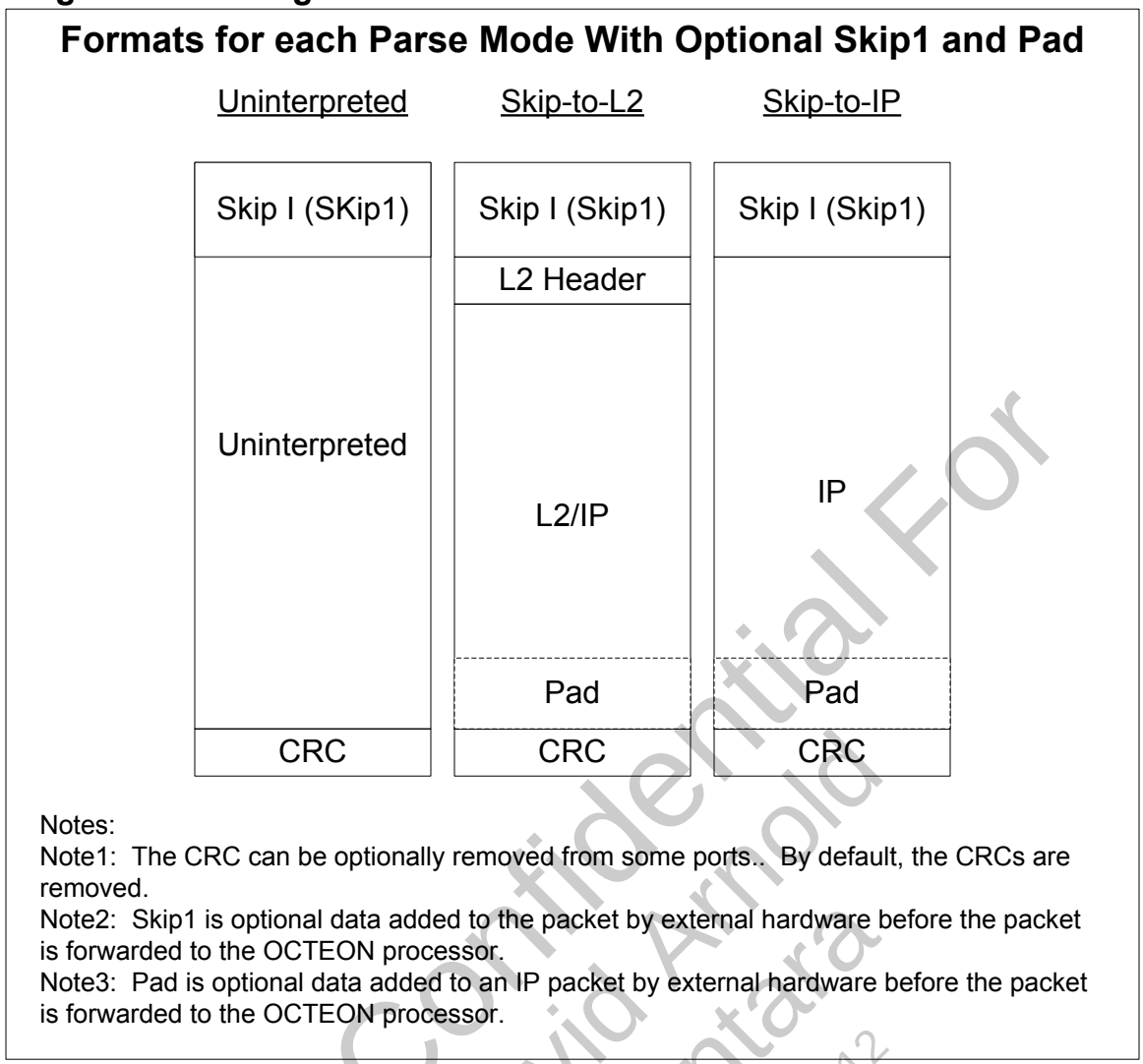
4.2 Parsing Modes

The exact packet information included in the WQE depends on the configured packet parse mode.

PIP/IPD supports three parse modes:

- Skip-to-L2 which parses the packet's L2 header (L2 error check, VLAN information provided, if packet is IP, IP information is provided)
- Skip-to-IP which parses the packet's IP header (No L2 error check, IP error check, no VLAN information, but IP information provided)
- Uninterpreted which does not parse the packet (no additional error checks, no VLAN or IP information)

Figure 7: Parsing Mode Choices Without Packet Instruction Header



The parse mode is set in one of two ways:

1. All packets on this port have the same parse mode, which is set via the SDK configuration variable `CVMX_HELPER_INPUT_PORT_SKIP_MODE` or the register configuration variable (`PIP_PRT_CFGn[MODE]`).
2. The parse mode can vary per-packet. In this case, the packet has a Packet Instruction Header or PCIe Instruction Header. The parse mode is one of the fields in the instruction header. Instruction Headers are discussed later in this section.

Customers may also add customized data to the start of the packet (Skip1) and the end of an IP packet (Pad). This is done by using external hardware which adds the customized data, then sends the packet to OCTEON. All packet data is stored, allowing the customer to access the customized data from software after the packet is received. Note: if Pad is added to the end of the IP packet, set the register field `PIP_PRT_CFGn[PAD_LEN]=1`, to disable the length check.

The number of bytes in the Skip1 region is configured per-port using the configuration register (PIP_PRT_CFG*n*[SKIP]). PIP/IPD skips over the specified number of bytes before beginning packet parsing. Skip1 must always be less than the number of bytes of packet data or WORD2[RE] will be set to 1. See the *HRM* for details.

The Pad field can be added to an IP packet. For example, a 40-byte IP packet arriving via a packet interface may have been padded out to the minimum-defined packet size of 64 bytes. If any input packet contains padding beyond the end of the IP packet, the PIP/IPD receives the pad and buffers it along with the other packet data.

See the *HRM* for details on how to configure the optional Skip1 and Pad correctly.

4.2.1 Optionally removing the CRC (FCS) (CRC stripping)

The packet's hardware CRC (Frame Check Sum (FCS)) can be removed (stripped) by IPD before the packet is buffered. This option does not apply to PCIe ports.

Note: Software should *not* remove the CRC from ports for which Work Queue Entry's hardware checksum field (HW_Chksum) may be used by software. This is because the CRC bytes are included in the hardware checksum, and software will probably need to reference the CRC value to use the hardware checksum.

See Section 10 – "Packet Storage" for more information on optional FCS stripping.

4.3 Optional Packet Instruction Headers

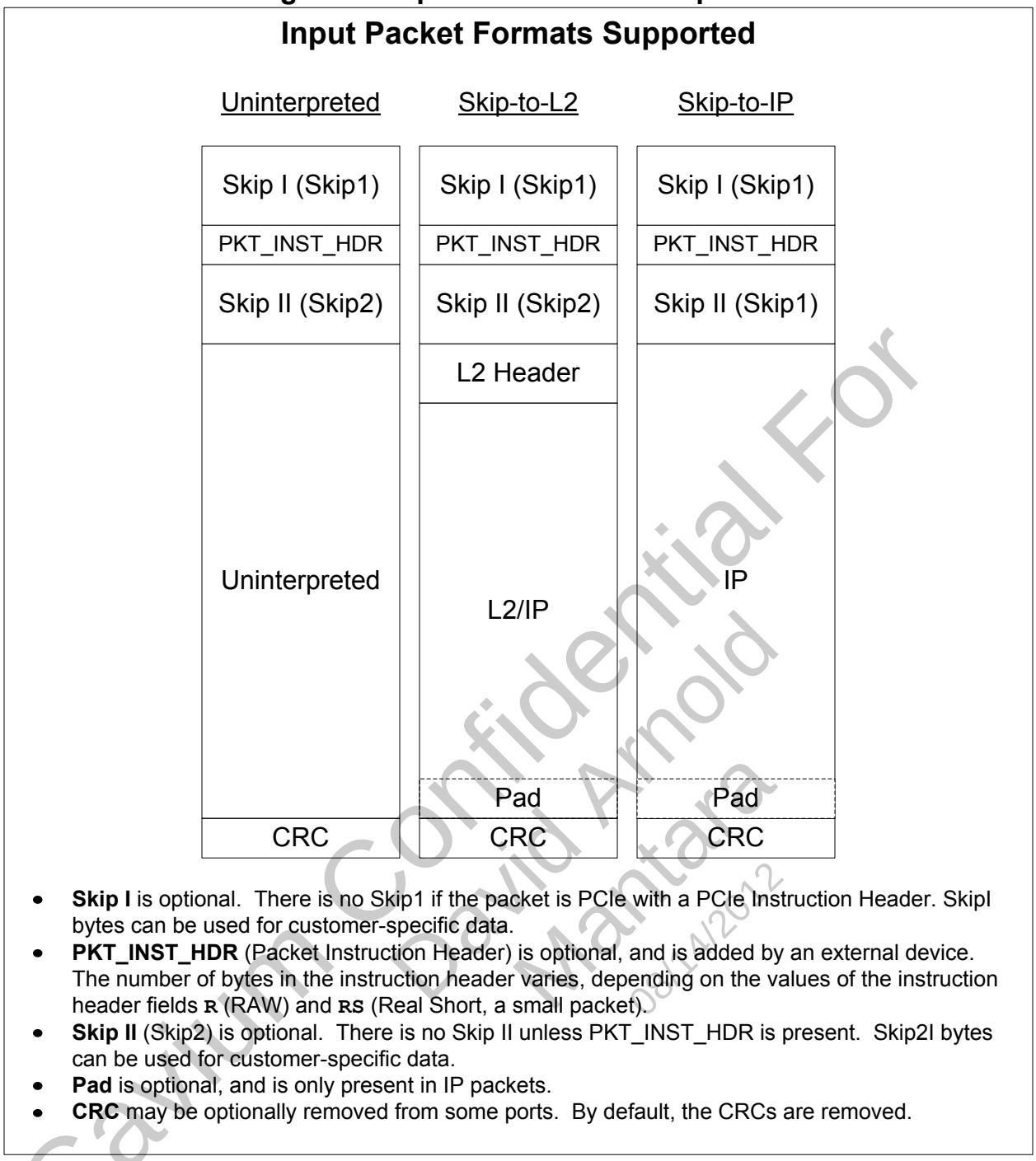
PIP/IPD also supports incoming packets which have variable-length Packet Instruction Headers. These headers are added by external hardware. The Packet Instruction Header specifies the packet's parse mode and may include the packet's scheduling information: QoS Value, Work Group ID, Tag Value, and Tag Type. Packet Instruction Headers allow an external device to control packet scheduling and parsing on a packet-by-packet basis. Packet Instruction Headers may be 2, 4, or 8 bytes long.

If a Packet Instruction Header is included with the packet, customized data may be added before and after the Packet Instruction Header. In this case, PIP/IPD needs to know how many bytes to skip before the Packet Instruction Header (Skip1), and how much to skip after the Packet Instruction Header (Skip2). Skip2 is only used if a Packet Instruction Header is included.

The Skip2 region is the number of bytes of customized data added after the Packet Instruction Header.

The total number of Skip bytes is the sum of the bytes in Skip1 + the number of bytes in the Packet Instruction Header + the number of bytes in Skip2. The number of bytes in the Skip1 region is specified in a configuration register. The remaining Skip bytes are provided via the SL (skip length) field in the Packet Instruction Header. The SL (skip length) field is the number of bytes in the Packet Instruction Header (2, 4, or 8 bytes), plus the number of bytes of customized data added after the Packet Instruction Header.

Figure 8: Input Packet Format Options



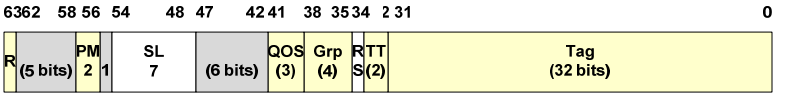
- **Skip I** is optional. There is no Skip1 if the packet is PCIe with a PCIe Instruction Header. SkipI bytes can be used for customer-specific data.
- **PKT_INST_HDR** (Packet Instruction Header) is optional, and is added by an external device. The number of bytes in the instruction header varies, depending on the values of the instruction header fields **R** (RAW) and **RS** (Real Short, a small packet).
- **Skip II** (Skip2) is optional. There is no Skip II unless PKT_INST_HDR is present. Skip2I bytes can be used for customer-specific data.
- **Pad** is optional, and is only present in IP packets.
- **CRC** may be optionally removed from some ports. By default, the CRCs are removed.

If Packet Instruction Headers are used for incoming packets on a port, set the port's `PIP_PRT_CFGn[INST_HDR]` to 1. The default value is 0 (no packets will contain Packet Instruction Headers). When this variable is set to 1, *all* packets received on the port must include a Packet Instruction Header. This variable is not used for PCIe ports.

Figure 9: Packet Instruction Header – Hardware View

Simplified Packet Instruction Header

Packet Instruction Header (8-byte version)



Note: Grayed out fields (reserved) must be set to zero.

R: RAW mode. Packet will be either RAWFULL or RAWSCH depending on the value of PM.

PM: Parse Mode. One of: (0=uninterpreted), (1=skip-to-L2), or (2=skip-to-IP)

SL: Skip Length: The number of bytes to skip from the start of the Packet Instruction Header to either the L2 or the IP section of the packet. (This value is the sum of (size of Packet Instruction Header) + (size of Skip2 region)).

QOS: Set the **QOS** value in the Work Queue Entry to this value.

Grp: Set the Group (**Grp**) value in the Work Queue Entry to this value.

RS: Real Short. Packet data will fit into the WQE. No Packet Buffer is used. See details on dynamic shorts in the text.

TT: Set the Tag Type (**TT**) in the Work Queue Entry to this value.

Tag: Set the **Tag** in the Work Queue Entry to this value.

If R=1 (RAW), then the Parse Mode (PM) determines the scheduling type:

- If **PM == 0** (unscheduled), then Packet type = **RAWFULL**. **PIP_RAW_WORD[WORD]** is used to create WORD2 of the WQE
- If **PM == 1** (skip-to-L2) or 2 (skip-to-IP), then Packet type = **RAWSCH**. PIP/IPD create the contents of WORD2 of the WQE by examining the packet information.

4.3.1 The `cvmx_pip_inst_hdr_t` Data Structure

This data structure is defined in `cvmx-pip.h`:

```
/**
 * Definition of the PIP custom header that can be pre-pended
 * to a packet by external hardware.
**/
typedef union
{
    uint64_t    u64;
    struct
    {
        uint64_t rawfull     : 1;   // Documented as R - Set if the Packet is
                                    // RAWFULL.  If set, this header must be
                                    // the full 8 bytes
        uint64_t reserved0   : 5;   // Must be zero
        cvmx_pip_port_parse_mode_t  parse_mode  : 2;   // PIP parse mode for
                                    // this packet
        uint64_t reserved1   : 1;   // Must be zero
        uint64_t skip_len    : 7;   // Skip amount, including this header,
                                    // to the beginning of the packet
        uint64_t reserved2   : 6;   // Must be zero
        uint64_t qos         : 3;   // POW input queue for this packet
        uint64_t grp         : 4;   // POW input group for this packet
        uint64_t rs          : 1;   // Flag to store this packet in the
                                    // work queue entry, if possible
        cvmx_pow_tag_type_t tag_type    : 2;   // POW input tag type
        uint64_t            tag         : 32;  // POW input tag
    } s;
} cvmx_pip_pkt_inst_hdr_t;
```

4.3.2 RAW, RAWFULL, RAWSCH

The *HRM* and this chapter mention the options "RAW", "RAWFULL", and "RAWSCH".

A packet is considered to be "RAW" if it has a Packet Instruction Header and the RAW bit is set in the instruction header. (The Packet Instruction Header data structure is shown in Figure 9 – "Packet Instruction Header".)

Depending on the parse mode, a RAW packet is either RAWSCH or RAWFULL. Both types of raw packets provide scheduling information (QoS, Group, Tag Value, and Tag Type) for the packet. The difference between the two types is in how the WQE WORD2 fields are created.

RAWSCH:

- RAW and
- Parse Mode = Skip-to-L2 or Skip-to-IP

In this case only the scheduling information comes from the header. The packet is parsed to create the Work Queue Entry (WQE) WORD2 fields.

RAWFULL:

- RAW and
- Parse Mode = Uninterpreted

Because the parse mode is "uninterpreted", WORD2 data cannot be derived from parsing the packet. In this case "FULL" WORD2 data comes from the port configuration register: `PIP_RAW_WORD[WORD]`. Note that in this case, there is only one configuration of WORD2 for the system, not one per port.

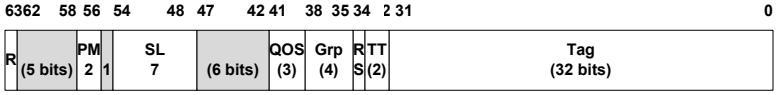
WQE WORD2 is discussed in more detail in Section 6 – "How Parse Mode Affects WQE WORD2 ".

Figure 10: WQE Information Copied From the Packet Instruction Header

Optional Packet Instruction Header

Note: If `PIP_PKT_CFGn[INST_HDR]` is set, then all packets arriving on the port must have a Packet Instruction Header

Packet Instruction Header (variable length)



Note: Grayed out fields (reserved) must be set to zero.

`R`: RAW mode. Packet will be either RAWFULL or RAWSCH depending on the value of PM.

`PM`: Parse Mode. One of: (0=uninterpreted), (1=skip-to-L2), or (2=skip-to-IP)

`SL`: Skip Length: The number of bytes to skip from the start of the Packet Instruction Header to either the L2 or the IP section of the packet. (This value is the sum of (size of Packet Instruction Header) + (size of Skip2 region)).

`QOS`: Set the `QOS` value in the Work Queue Entry to this value.

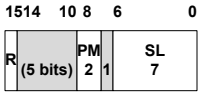
`Grp`: Set the Group (`Grp`) value in the Work Queue Entry to this value.

`RS`: Real Short. Packet data will fit into the WQE. No Packet Buffer is used. See details on dynamic shorts in the text.

`TT`: Set the Tag Type (`TT`) in the Work Queue Entry to this value.

`Tag`: Set the `Tag` in the Work Queue Entry to this value.

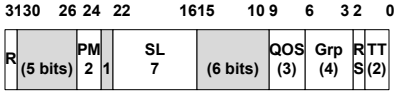
1.



If `R==0` and `RS` is *not* used

RS is *not used* if any of the following conditions is true:
1. `PIP_PRT_CFGn[DYN_RS]` = 1 for the given port
2. `PIP_GBL_CFG[IGNRS]` = 1 AND the port is NOT a PCIe port
3. The packet is not a dynamic short (does NOT fit entirely into the WQE)

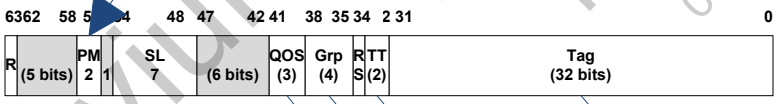
2.



If `R==0` and `RS` *is* used

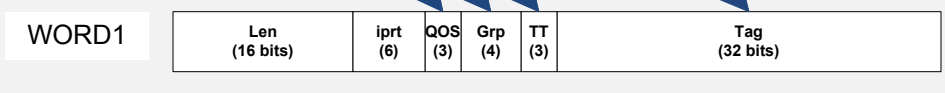
If `R==1` (RAW), then the Parse Mode (PM) determines the scheduling type:
A. If `PM==0` (unscheduled), then
Packet type = RAWFULL. `PIP_RAW_WORD[WORD]` is used to create WORD2 of the WQE
B. If `PM==1` (skip-to-L2) or 2 (skip-to-IP), then
Packet type = RAWSCH. PIP/IPD create the contents of WORD2 of the WQE by examining the packet information.

3.



If `R==1`

Work Queue Entry (WQE)



4.4 Optional PCIe Instruction Headers

When packets arrive via PCIe ports, they have a PCIe Instruction Header if either:

- The `PCIE_INST_HRD[R]` bit is set to 1 OR
- The `NPEI_PKT(0-31)_INSTR_HEADER[USE_IHDR]` bit is set for the PCIe ring the packet arrived on

Skip1 does not apply to PCIe ports. There is no customized data allowed before the PCIe Instruction Header.

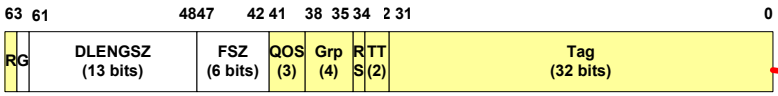
PIP/IPD converts the PCIe Instruction Header into a Packet Instruction Header, as shown in the next figure.

Figure 11: PCIe Instruction Header Conversion to Packet Instruction Header

Creating Packet Instruction Header From PCIe Instruction Header

For PCIe packets, the Packet Instruction Header is created from the PCIe Instruction Header plus per-ring register Parse Mode and Skip Length values. This header is pre-pended to the PCIe packet, replacing the **DPTR** and **PCIE_INST_HDR** fields which were at the start of the PCIe packet.

PCIe Instruction Header



Note: Grayed out fields (reserved) must be set to zero.

R: RAW mode. If **R==1**, packet will be either RAWFULL or RAWSCH depending on the value of the parse mode. If **R==0**, the **QOS**, **Grp**, **TT**, and **Tag_Value** fields are ignored.

G: Gather is used

DLENGSX: If **G==1** and **DLENGSZ!=0** (indirect gather instruction): **DLENGSZ** is the number of entries in the gather list.

- If **G==1** and **DLENGSZ==0** (direct gather instruction): **DLENGSZ** is only used to select the instruction mode
- If **G==0** (no gather): **DLENGSZ** must be nonzero. It represents the length of the packet data (length in bytes) directly pointed at by **DPTR**.

FSZ: Front-data size: the number of bytes of packet data before the **DPTR** data and after the optional Packet Instruction Header

QOS: If **R==1**, set the **QOS** value in the Work Queue Entry to this value.

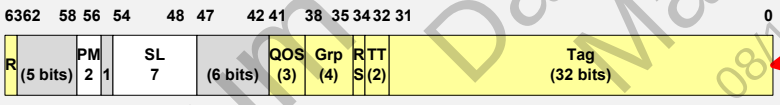
Grp: If **R==1**, set the Group (**Grp**) value in the Work Queue Entry to this value.

RS: Real Short. Packet data will fit into the WQE. See details on dynamic shorts in the text.

TT: If **R==1**, set the Tag Type (**TT**) in the Work Queue Entry to this value.

Tag: If **R==1**, set the **Tag** in the Work Queue Entry to this value.

Packet Instruction Header (variable length)



Note: Grayed out fields (reserved) must be set to zero.

R, QOS, Grp, RS, TT, and Tag: are the same as for the PCIe Instruction Header.

PM: Parse Mode. One of: (0=uninterpreted), (1=skip-to-L2), or (2=skip-to-IP).
If **NPEI_PKTr[PBP]==1** (packet-by-packet mode) is set, this field is set to the value of **NPEI_PKTr[RPARMODE]** (the raw parse mode set for this PCIe ring), otherwise the value is set to **NPEI_PKTr[PAR_MODE]**.

SL: Skip Length: The number of bytes to skip from the start of the Packet Instruction Header to either the L2 or the IP section of the packet. (This value is the sum of (size of Packet Instruction Header) + (size of Skip2 region)).
If **NPEI_PKTr[PBP]==1** (packet-by-packet mode) is set this field is set to the value of **NPEI_PKTr[RSKP_LEN]** (the raw skip length set for this PCIe ring), otherwise the value is set to **NPEI_PKTr[SKP_LEN]**.

4.5 Registers to Configure Input Packet Format

These per-port configuration variables control PIP/IPD expectations about incoming packet content, and how it should be handled. Note that `PIP_PRT_CFGn[MODE]` is not examined if `PIP_PRT_CFGn[INST_HDR]==1`. In that case, the parse mode is determined by the `PM` field in the packet's Packet Instruction Header.

Note that Skip2 is specified in the Packet Instruction Header, and Pad is not specified, the extra bytes are simply received. When Pad is used, the length check must be disabled.

Table 11: Registers to Configure Input Packet Format

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Parse Mode: Parse mode (Skip-to-L2 (1), Skip-to-IP (2), or Uninterpreted (0))	`PIP_PRT_CFGn` (one per port)	`MODE`	0	1 (See Note1)
Packet Instruction Header Present: When set, the Packet Instruction Header is present on all packets (except PCIe ports 32-35)	`PIP_PRT_CFGn` (one per port)	`INST_HDR`	0	0 (H/W Default)
SKIP 1 Amount: Optional SKIP 1 amount: the number of bytes PIP/IPD will skip before parsing the packet.	`PIP_PRT_CFGn` (one per port)	`SKIP`	0	0 (H/W Default)
Broadcom HiGig: Enable Broadcom HiGig parsing	`PIP_PRT_CFGn` (one per port)	`HIGIG_EN`	0	0 (H/W Default) (See Note2)
Notes				
Note1: Configured via `executive-config.h`: `CVMX_HELPER_INPUT_PORT_SKIP_MODE = CVMX_PIP_PORT_CFG_MODE_SKIPL2`				
Note2: Can be configured via `cvmx_higig_initialize()`				

5 The Work Queue Entry Data Structure (WQE)

The Work Queue Entry (WQE) can be located either in a Work Queue Entry Buffer (most common case) or in the first 128 bytes of the Packet Data Buffer (this feature is supported on selected OCTEON models).

The following table shows the registers used to select either the FPA pool used to supply the Work Queue Entry buffers, or the variable to set to use the Packet Data Buffer instead.

If the WQE is in the first 128 bytes of the Packet Data Buffer, when the work is added to the SSO, the WQE pointer is simply a pointer to the Packet Data Buffer. (See the `passthrough` example for code that uses this configuration.) See Section 10.2.1 – "Storing WQE in Packet Data Buffer instead of WQE Buffer".

Table 12: Registers to Configure Work Queue Entry Details

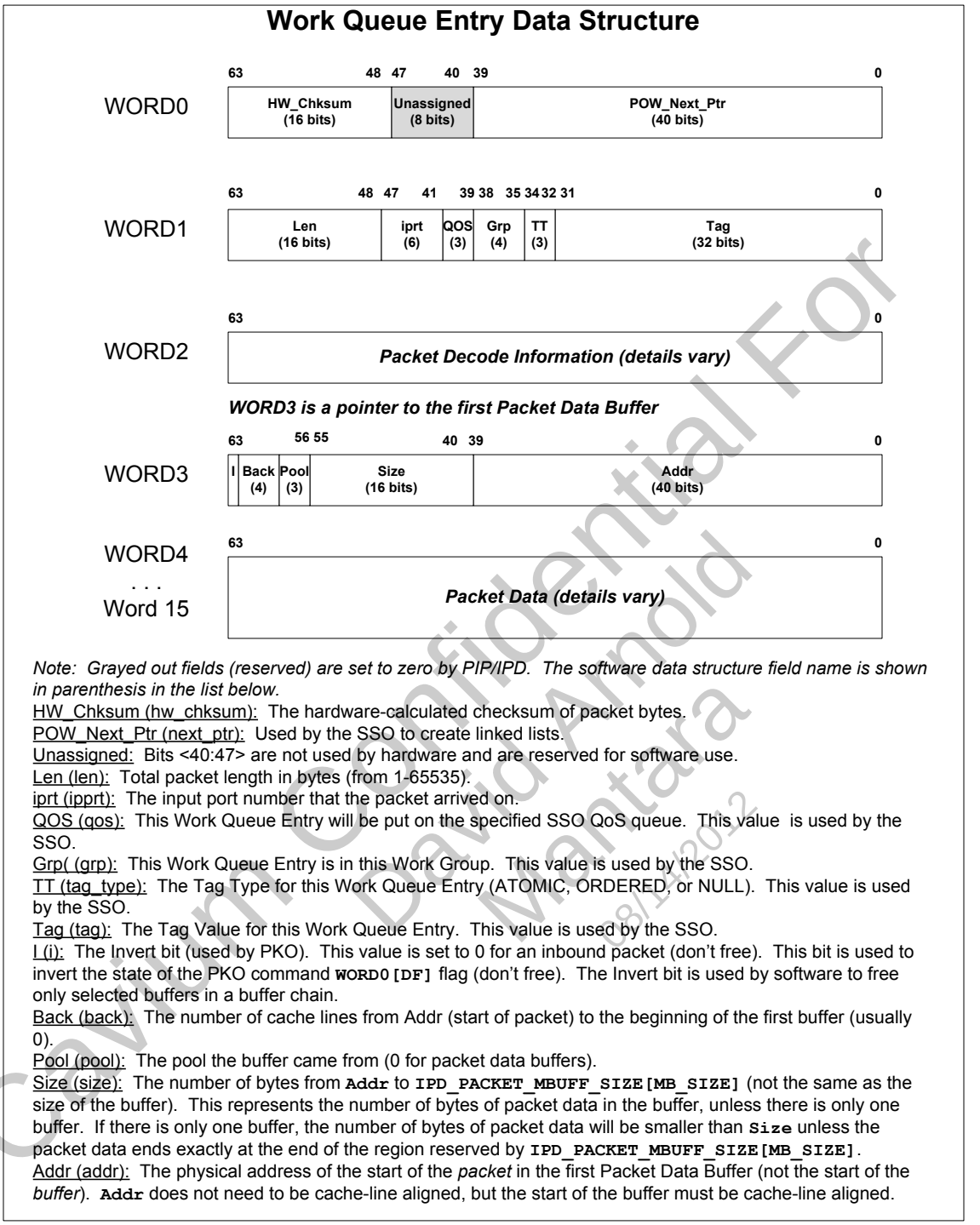
Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Select FPA Pool to Use for Work Queue Entry Buffers				
Select WQE Pool: Select FPA Pool to Use for Work Queue Entry Buffers. This field is not used when `IPD_CTL_STATUS[NO_WPTR]` is set.	`IPD_WQE_FPA_QUEUE`	`WQE_QUE`	0	1 (See Note1)
Store WQE in first 128 bytes of Packet Data Buffer				
Omit WQE Buffer: When set to 1, Work Queue Entry buffers are not used. The WQE data is located in the first 128 bytes of the Packet Data Buffer. Space must be reserved using `IPD_1ST_MBUFF_SKIP[SKIP_SZ]`. See the *HRM* register field description for details.	`IPD_CTL_STATUS`	`NO_WPTR`	0	0 (H/W Default)
WQE Endianness				
Work Queue Entry Endian specification. If set to 1, WQE is written in little Endian.	`IPD_CTL_STATUS`	`WQE_LEND`	0	0 (H/W Default)
Notes				
Note1: The pool WQE pool number is configured automatically by Simple Executive. See the *Configuration* chapter for details.				

5.1 Work Queue Entry Data Structure

The Work Queue Entry (WQE) data structure is shown in the following figure. The format is dictated by hardware requirements. Notice that details for WORD2 are not provided. The WORD2 fields depend on the parsing results. The different WORD2 data structures are shown as "CASE1, CASE2, CASE3" in Figure 13 – "Parsing Cases". The "CASE" notation is used in this

section to help the reader match the figures and tables to the appropriate data structure. (See Section 6 – "How Parse Mode Affects WQE WORD2 " for more information.)

Figure 12: Work Queue Entry Data Structure – Hardware View



Note: Grayed out fields (reserved) are set to zero by PIP/IPD. The software data structure field name is shown in parenthesis in the list below.

HW_Chksum (hw_chksum): The hardware-calculated checksum of packet bytes.

POW_Next_Ptr (next_ptr): Used by the SSO to create linked lists.

Unassigned: Bits <40:47> are not used by hardware and are reserved for software use.

Len (len): Total packet length in bytes (from 1-65535).

iprt (ipprt): The input port number that the packet arrived on.

QOS (qos): This Work Queue Entry will be put on the specified SSO QoS queue. This value is used by the SSO.

Grp((grp): This Work Queue Entry is in this Work Group. This value is used by the SSO.

TT (tag_type): The Tag Type for this Work Queue Entry (ATOMIC, ORDERED, or NULL). This value is used by the SSO.

Tag (tag): The Tag Value for this Work Queue Entry. This value is used by the SSO.

I (i): The Invert bit (used by PKO). This value is set to 0 for an inbound packet (don't free). This bit is used to invert the state of the PKO command `WORD0[DF]` flag (don't free). The Invert bit is used by software to free only selected buffers in a buffer chain.

Back (back): The number of cache lines from Addr (start of packet) to the beginning of the first buffer (usually 0).

Pool (pool): The pool the buffer came from (0 for packet data buffers).

Size (size): The number of bytes from `Addr` to `IPD_PACKET_MBUFF_SIZE[MB_SIZE]` (not the same as the size of the buffer). This represents the number of bytes of packet data in the buffer, unless there is only one buffer. If there is only one buffer, the number of bytes of packet data will be smaller than `Size` unless the packet data ends exactly at the end of the region reserved by `IPD_PACKET_MBUFF_SIZE[MB_SIZE]`.

Addr (addr): The physical address of the start of the *packet* in the first Packet Data Buffer (not the start of the *buffer*). `Addr` does not need to be cache-line aligned, but the start of the buffer must be cache-line aligned.

5.2 Software WQE Data Structures

WQE data structures are:

- WQE: the WQE data structure is defined in `cvmx_wqe_t` in `cvmx-wqe.h`
- WORD0: the fields are defined in `cvmx_wqe_t` in `cvmx-wqe.h`
- WORD1: the fields are defined in `cvmx_wqe_t` in `cvmx-wqe.h`
- WORD2: the fields are defined in `cvmx_pip_wqe_word2` in `cvmx-wqe.h`, a union of:
  - `uint64_t u64`
  - `struct s` – used if the hardware determines the packet is IP (CASE 2 shown in figures below).
  - `struct svlan` – used to access the 16 VLAN bits
  - `struct snoip` – used if the hardware could not determine whether the packet is IP (CASE 3 shown in figures below)
  - The CASE 1 data structure is not defined
- WORD3: The fields are defined in the `cvmx_buf_ptr_t` data structure in `cvmx-packet.h`

Note: the "CASE 1, CASE 2, CASE 3" notation is explained in Section 6 – "How Parse Mode Affects WQE WORD2 Data Structure".

The data structures used in the Work Queue Entry are shown below.

Note: hardware field names tend to be short, such as "`VV`". This short name fits well into the figures showing the hardware data structure. Software field names are longer to help code readability, such as "`vlan_valid`" instead of "`VV`". To help cross-connect the HRM with the SDK, both the hardware and software field names are shown when possible:

- In the software data structures below, the hardware name is shown as the first part of the field comment. See Section 5.2.1 – "WQE The `cvmx_wqe_t` Data Structure" for an example.
- In the tables which accompany the figures showing hardware data structures, the software field names are shown in parenthesis after the hardware field name. See Table 13 – "Fields: WQE WORD2 Fields if L1/L2 Error (CASE 3C)" for an example.

5.2.1 WQE The `cvmx_wqe_t` Data Structure

The Work Queue Entry software data structure is defined in `cvmx-wqe.h`:

```
/**
 * Work queue entry format
 *
 * must be 8-byte aligned
 */
typedef struct
{
    // WORD0
    uint16_t            hw_chksum;          // HW_Chksum - hardware checksum
    uint8_t             unused;             // Unassigned - available for
                                            // software use
    uint64_t            next_ptr      : 40; // POW_Next_Ptr - used by the
                                            // SSO (POW) to create lists
    // WORD 1
    uint64_t            len           :16;  // Len - total bytes in the packet
    uint64_t            ipprt         : 6;  // iprt - input port
    uint64_t            qos           : 3;  // QOS - calculated QoS value
    uint64_t            grp           : 4;  // Grp- calculated Group value
    cvmx_pow_tag_type_t tag_type      : 3;  // TT - calculated tag type
    uint64_t            tag           :32;  // Tag_value (Tag) - calculated
                                            // tag value
    // WORD 2
    cvmx_pip_wqe_word_t word2;              // status and error conditions

    // WORD 3
    cvmx_buf_ptr_t      packet_ptr;         // pointer to first packet
                                            // data buffer
    // WORD4 to WORD15

    /**
     *   HW WRITE: Hardware will fill in a programmable amount from the
     *             packet, up to (at most, but perhaps less) the amount
     *             needed to fill the work queue entry to 128 bytes
     *   If the packet is recognized to be IP, the hardware starts (except that
     *   the IPv4 header is padded for appropriate alignment) writing here
     *   where the IP header starts.
     *   If the packet is not recognized to be IP, the hardware starts writing
     *   the beginning of the packet here.
     */
    uint8_t packet_data[96]; // WORD4 to WORD15 = 96 bytes

    /**
     * The WQE is usually 128 bytes (one cache line).  Software can make the
     * WQE any length, but the hardware only manages the first 128 bytes.
     * (Making the WQE larger will not change the amount of packet data
     * stored in the WQE).
     */
}  CVMX_CACHE_LINE_ALIGNED cvmx_wqe_t;
```


5.2.2 WQE WORD2: The `cvmx_pip_wqe_word2` Data Structure

The contents of the WQE WORD2 data structure depend on the results of the hardware parsing. There are three possible data structures (a union).

There are three different possible data structures:

- CASE 1 (Uninterpreted, RAW, and No receive error). The SDK does not supply this data structure since this is not the usual case.
- CASE 2 (IP)
- CASE 3 (Not IP)

CASE 2 and CASE 3 are contained in the `cvmx_pip_wqe_word2` data structure shown below, and also appear in the following figures.

```
typedef union
{
    uint64_t                u64;

    // Use this structure if the hardware determines that the packet
    // is IP (CASE 2) */
    struct
    {
        uint64_t              bufs          : 8; // Bufs - number of buffers
        uint64_t              ip_offset     : 8; // IP_offset - offset to
                                                 // start of IP packet
        uint64_t              vlan_valid    : 1; // VV - VLAN or VLAN STACKED
        uint64_t              vlan_stacked  : 1; // VS - VLAN STACKED
        uint64_t              unassigned    : 1; // is set to all 0
        uint64_t              vlan_cfi      : 1; // VC - VLAN CFI bit
        uint64_t              vlan_id       :12; // VLAN_id - VLAN ID
        uint64_t              pr            : 4; // PR - PCIe ring position
                                                 // [0-7]
        uint64_t              unassigned2   : 8; // is set to all 0
        uint64_t              dec_ipcomp    : 1; // CO - IP decompression
                                                 // needed
        uint64_t              tcp_or_udp    : 1; // TU - TCP or UDP packet
        uint64_t              dec_ipsec     : 1; // SE - decryption needed
        uint64_t              is_v6         : 1; // V6 - set if packet is IPv6
        uint64_t              software      : 1; // Reserved for software use
        uint64_t              L4_error      : 1; // LE - L4 error
        uint64_t              is_frag       : 1; // FR - fragment
        uint64_t              IP_exc        : 1; // IE - IP exception
        uint64_t              is_bcast      : 1; // B - broadcast
        uint64_t              is_mcast      : 1; // M - multicast
        uint64_t              not_IP        : 1; // NI - not IP
        uint64_t              rcv_error     : 1; // RE - L1/L2 receive error
        uint64_t              err_code      : 8; // opcode - error code (see
                                                 // cvmx_pip_err_t)
    } s;  // packet is IP (CASE 2)

    // VLAN view of the structure - use this structure to get at the
    // 16 VLAN bits */
    struct
    {
        uint64_t              unused1       :16;
        uint64_t              vlan          :16;
        uint64_t              unused2       :32;
    } svlan;
```

```
    // use this structure if the packet
    // is NOT IP, including if an L1/L2 error occurs (CASE 3)
    // Note this data structure is used of the hardware cannot determine
    // that the packet is IP.
    struct
    {
        uint64_t              bufs          : 8; // Bufs - number of buffers
        uint64_t              unused        : 8; // is set to 0
        uint64_t              vlan_valid    : 1; // VV - VLAN or VLAN STACKED
        uint64_t              vlan_stacked  : 1; // VS - VLAN STACKED
        uint64_t              unassigned    : 1; // is set to 0
        uint64_t              vlan_cfi      : 1; // VC - VLAN CFI bit
        uint64_t              vlan_id       :12; // VLAN_id - VLAN ID
        uint64_t              pr            : 4; // PR - PCIe ring position
                                                 // [0-7]
        uint64_t              unassigned2   :12; // is set to 0
        uint64_t              software      : 1; // reserved for software use,
                                                 // hardware will clear on
                                                 // packet creation
        uint64_t              unassigned3   : 1; // is set to 0
        uint64_t              is_rarp       : 1; // IR - RARP
        uint64_t              is_arp        : 1; // IA - ARP
        uint64_t              is_bcast      : 1; // B - broadcast
        uint64_t              is_mcast      : 1; // M - multicast
        uint64_t              not_IP        : 1; // NI - Not IP
        uint64_t              rcv_error     : 1; // RE - L1/L2 receive error
        uint64_t              err_code      : 8; // opcode - error code (see
                                                 // cvmx_pip_err_t)

    } snoip;  // structure if NOT IP (CASE 3)
} cvmx_pip_wqe_word2;
```

5.2.3 WQE WORD3: The `cvmx_buf_ptr_t` data structure

The `cvmx_buf_ptr_t` data structure (WQE WORD3) is defined in `cvmx-packet.h`:

```
typedef union
{
    void*           ptr;
    uint64_t        u64;
    struct
    {
        uint64_t    i    : 1; // set to 0 for inbound packet
        uint64_t    back : 4; // Back - Indicates the number of cache lines
                              // to back up to access the start of the buffer
                              // relative to addr.  In most cases the amount to
                              // back up is less than a complete cache line, so
                              // this value is set to 0
        uint64_t    pool : 3; // Pool - The pool the buffer came from (pool
                              // 0 for packet data buffers)
        uint64_t    size :16; // Size - The size of the segment pointed to
                              // by addr (in bytes)
        uint64_t    addr :40; // Addr - Pointer to the first byte of data
    } s;
} cvmx_buf_ptr_t;
```

6 How Parse Mode Affects WQE WORD2 Data Structure

The packet's parse mode is essential to how the packet is parsed. Parsing affects the information stored in the WQE WORD 2 data structure and field values. The parsing mode does not change the packet data (all received bytes are stored). This section presents the different parse modes and the resultant WQE WORD2 data structures. The options are shown in the figure below. To navigate this section most easily, use the next figure to locate the case applicable to your specific application, then use the specific "CASE" notation to locate the relevant figure and table.

Parsing details are shown in the flow chart in section 16 – "Appendix C: Input Packet Parsing".

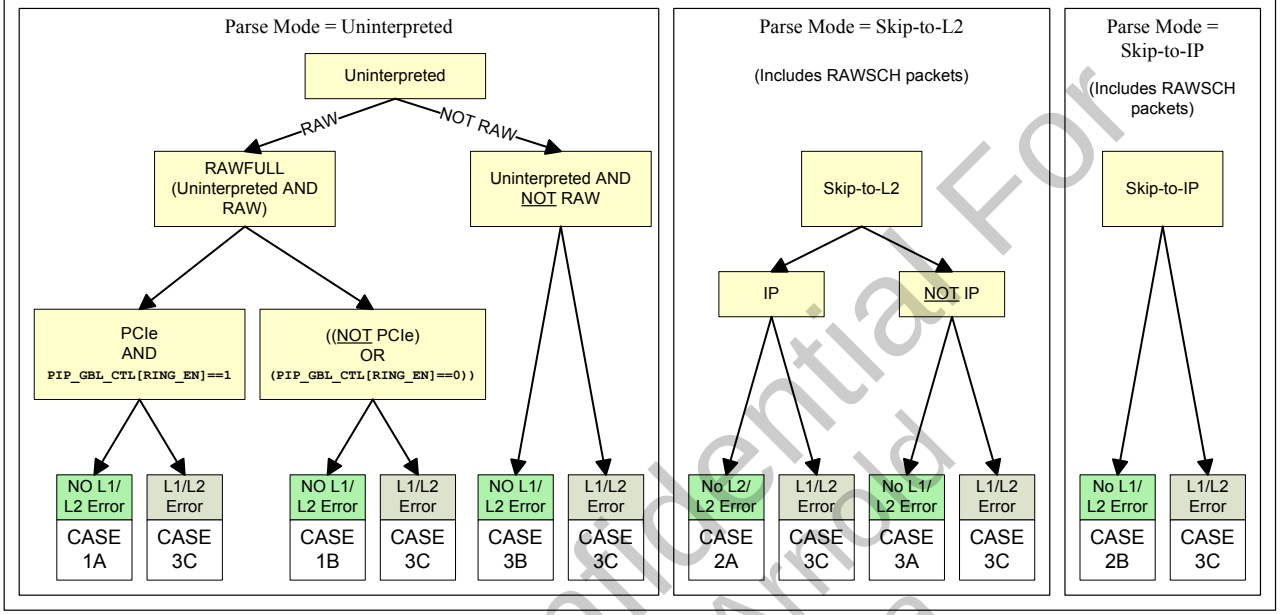
Figure 13: Parsing Cases

Packet Parsing and WQE WORD2 Cases

There are three parsing modes available: "Skip-to-L2" where the packet's L2 header is parsed, "Skip-to-IP" which skips directly to the IP portion of the packet, and "Uninterpreted" which does not examine the packet contents.

A packet with the parse mode "skip-to-L2" is further classified as either being an IP packet or Non-IP packet. The packet is an IP packet if the L2 header's type field contains either 0x800 (for IPv4) or 0x86DD (for IPv6).

There are three different data structures used for WORD2, depending on the parsing results: CASE 1, 2, and 3. Within each case, field values depend on parsing results (A, B, C). Each of these WQE WORD2 variations are shown in other figures. Cases which do not have L1/L2 receive errors may be found on the Parsing flowchart.



6.1 All Parse Modes if L1/L2 Error Occurs

If there is an L1/L2 error during parsing, for any of the parsing modes (skip-to-L2, skip-to-IP, uninterpreted), the WQE WORD2 fields are set as shown in the following figure:

Figure 14: WORD2 if L1/L2 Error (CASE 3C)

Parse Mode = Skip-to-L2, Skip-to-IP, or Uninterpreted, and an L1/L2 Receive Error has occurred

WQE WORD2 Data Structure

This case applies if:

- An L1/L2 receive error occurred
- Any parse mode is specified: Skip-to-L2, Skip-to-IP, or Uninterpreted

In this case **RE** and **NI** are both set to 1 to indicate a receive error.

The fields **VV**, **VS**, **VC**, **VLAN_id**, **IR**, **IA**, **B**, and **M** are unpredicable (U) because when there is an L1/L2 receive error the packet may be corrupted.

Reserved fields in WQE are not named and are highlighted in gray. They are set to 0.

The different fields in the data structures are explained in the next table.

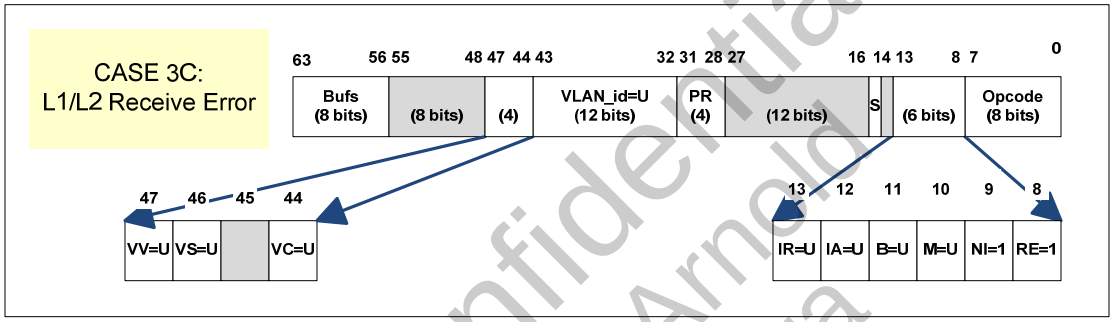


Table 13: Fields: WQE WORD2 Fields if L1/L2 Error (CASE 3C)

Field	Definition *(Fields are in alphabetical order.* *The SDK software field names are shown in parenthesis.)*
B (is_bcast)	Broadcast: set when the packet's destination MAC address field in the L2 header is the broadcast address (all ones). Note: B always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
Bufs (bufs)	Number of Buffers: The number of buffers used to store the packet data. A zero value means that a dynamic short packet is stored entirely in the WQE (there is no Packet Data Buffer).
IA (is_arp)	Is ARP: Set when the packet's L2 header type field ==0x0806 (an ARP packet). Note: IA=0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
IR (is_rarp)	Is RARP: Set when the packet's L2 header type field ==0x0835 (an RARP packet). Note: IA=0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
M (is_mcast)	Multicast: Set when the packet's destination MAC address field in the L2 header is a multicast address (the group bit is set, and at least one of the remaining bits is a zero). Note: M is always zero when NOT in skip-to-L2 mode.

Field	Definition *(Fields are in alphabetical order. The SDK software field names are shown in parenthesis.)*
NI (not_IP)	Not IP: Not an IP Packet or an L1/L2 receive error has occurred.
Opcode (err_code)	Error Code: Numeric code indicating specific error which occurred: If there is an error (any of WORD2[RE] or WORD2[IE] or WORD2[LE] is set), then Opcode contains an error code, otherwise Opcode=0. The error codes values depend on which error bit is set (RE, IE, or LE). See those specific errors for details.
PR (pr)	PCIe Ring: The relative position of the PCIe ring in the PKI input port [0-7]. PR is enabled if PIP_GBL_CTL[RING_EN]==1. If the packet was not received on a PCIe port OR PIP_GBL_CTL[RING_EN]==0, then PR=0. Note: Zero is both 1) a legal ring position value and 2) the value if the packet is not received on a PCIe port.
RE (rcv_error)	Receive error (L1/L2 error): For CASE 2, by definition RE==0 because CASE 2 only occurs there is NO L1/L2 error.
S (software)	Software Use: Reserved for software use.
VC (vlan_cfi)	VLAN CFI bit: The VC bit is the VLAN CFI bit (VLAN bit <12>). If VV==0 (NOT VLAN), then VC=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VC is set to the packet's VLAN CFI bit. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN CFI will be used (VLAN0 or VLAN1). If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 CFI is selected, otherwise VLAN1 CFI is selected. Note: VC is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VLAN_id (vlan_id)	VLAN_ID: VLAN_id is the VLAN ID field (VLAN bits <11:0>). If VV==0 (NOT VLAN), then VLAN_id=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VLAN_id is set to the packet's VLAN id. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN id will be used. If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 id is selected, otherwise VLAN1 id is selected. Note: VLAN_id is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VS (vlan_stacked)	VLAN STACKED: This bit is only set if VV==1 (VLAN) AND the packet is VLAN STACKED. Note: VS is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VV (vlan_valid)	VLAN Valid: This bit is only set if the packet is VLAN or VLAN STACKED. Note: VV is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.

6.2 Parse Mode = Skip-to-L2

When the parse mode is set to "skip-to-L2", the L2 header is analyzed and appropriate fields are set the WQE WORD2.

The industry-standard L2 Header options are shown in Figure 49 – "L2 Header Formats".

Figure 15: WORD2 if PM=Skip-to-L2, No L1/L2 Errors (CASE 2A, CASE 3A)

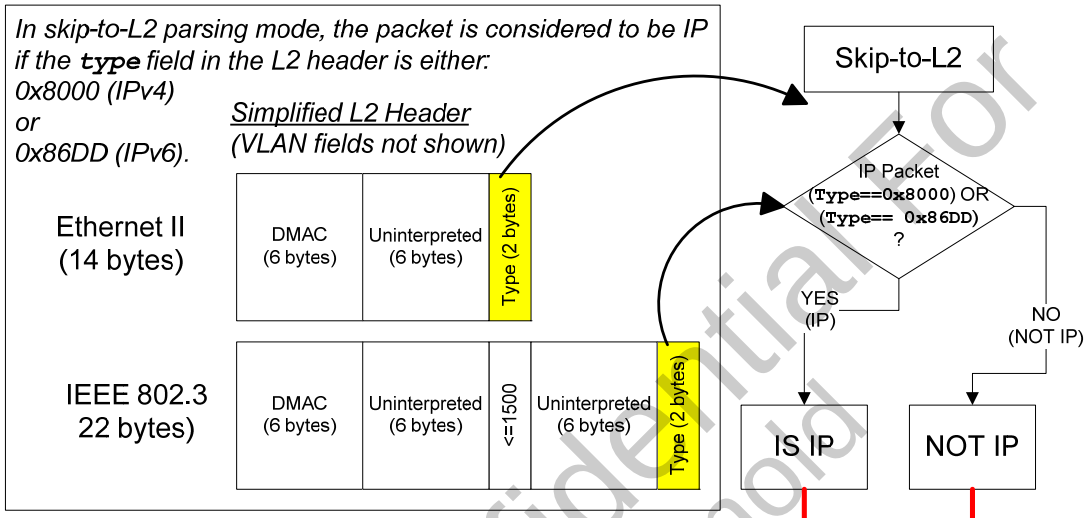
Parse Mode=Skip-to-L2: WQE WORD2 Data Structure

This case applies if:

- The packet's parse mode is Skip-to-L2
- There are no L1/L2 Errors

A packet with the parse mode "skip-to-L2" is classified as either being an IP packet or Not-IP packet based on the value of the L2 header **`type`** field. The **`NI`** bit in WQE WORD2 is set to 0 if IP and 1 if not IP.

Reserved fields in WQE are not named and are highlighted in gray. They are set to 0.



In the case of an IP packet, more information about the packet (such as whether the packet is TCP or UDP) can be determined by parsing the IP header.

Thus WORD2 of the Work Queue Entry can be one of two different data structures, depending on whether the skip-to-L2 packet is IP or not. Fields highlighted in green are contained in one or the other data structure, but not in both.
The different fields in the data structures are explained in the next table.

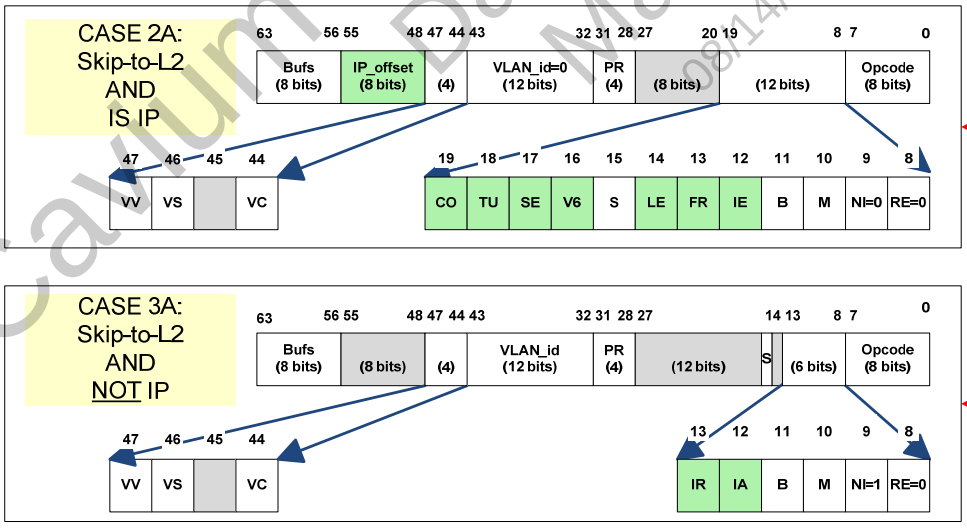


Table 14: WQE WORD2 Fields for Skip-to-L2 and Is_IP (CASE 2A)

Field	Definition *(Fields are in alphabetical order.* *The SDK software field names are shown in parenthesis.)*
B (is_bcast)	Broadcast: set when the packet's destination MAC address field in the L2 header is the broadcast address (all ones). Note: B always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
Bufs (bufs)	Number of Buffers: The number of buffers used to store the packet data. A zero value means that a dynamic short packet is stored entirely in the WQE (there is no Packet Data Buffer).
CO (dec_ipcomp)	IP Compression: The CO bit (IP compression protocol bit) is set when the packet is IPCOMP (the IPv4 header protocol field or the initial IPv6 next header field == 108.) This bit is always clear when WORD2[IE]==1 (IP error), and when WORD2[V6] and WORD2[FR] are both set. This bit indicates that the packet needs to be decompressed.
FR (is_frag)	Fragment: Set when the packet is a fragment. For IPv4, this bit is set when either the IPv4 header's MF (More Fragments) flag is set, or the IPv4 header fragment offset field is non-zero (the last fragment has the MF flag cleared and a non-zero fragment offset). For IPv6, this bit is set when the initial next header value is fragmentation (44). (For IPv6, FR is never set when WORD2[IE]==1 (IP error) .)
IE (IP_exc)	IP Error: Set when the packet has an IP exception condition. When the IE bit is set, WORD2[Opcode] contains an error code specific to this type of error. The exact error codes will be provided in a separate table. Note the bit only applies if (!RE) && (!NI).
IP_offset (ip_offset)	IP Offset: The number of bytes from the first byte of packet data to the first byte of the IP packet (the IP header).
LE (L4_error)	IP L4 Error: This bit is set when WORD2[TU] is set and the PIP/IPD hardware found an error in the TCP/UDP header and /or data. When the LE bit is set, WORD2[Opcode] contains an error code specific to this type of error. The exact error codes will be provided in a separate table. Note this bit only applies if only applies if (!RE) && (!NI) && (!IE) && (!FR).
M (is_mcast)	Multicast: Set when the packet's destination MAC address field in the L2 header is a multicast address (the group bit is set, and at least one of the remaining bits is a zero). Note: M is always zero when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
NI (not_IP)	Not IP: Not an IP Packet or an L1/L2 error has occurred. For CASE 2, by definition NI==0 because CASE 2 only occurs if the packet is IP. (Note this bit is set if the hardware cannot determine that the packet is IP. This does not necessarily mean that the packet is in fact not IP.)
Opcode (err_code)	Error Code: Numeric code indicating specific error which occurred: If there is an error (any of WORD2[RE] or WORD2[IE] or WORD2[LE] is set), then Opcode contains an error code, otherwise Opcode=0. The error codes values depend on which error bit is set (RE, IE, or LE). See those specific errors for details.
PR (pr)	PCIe Ring: The relative position of the PCIe ring in the PKI input port [0-7]. PR is enabled if PIP_GBL_CTL[RING_EN]==1. If the packet was not received on a PCIe port OR PIP_GBL_CTL[RING_EN]==0, then PR=0. Note: Zero is both 1) a legal ring position value and 2) the value if the packet is not received on a PCIe port.
RE (rcv_error)	Receive error (L1/L2 error): For CASE 2, RE==0 because CASE 2 only occurs there is NO L1/L2 error.
S (software)	Software Use: Reserved for software use.

TU (tcp_or_udp)	Is TCP or UDP: The TU bit is set when an IP packet is TCP or UDP (when the IPv4 protocol value or the IPv6 initial next header ==6 (TCP) or ==17 (UDP)). This bit is always 0 when WORD2[IE]==1, and when WORD2[V6] and WORD2[FR] are both set.
V6 (is_v6)	Is IPv6: Set if IP header version field ==6
VC (vlan_cfi)	VLAN CFI bit: The VC bit is the VLAN CFI bit (VLAN bit <12>). If VV==0 (NOT VLAN), then VC=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VC is set to the packet's VLAN CFI bit. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN CFI will be used (VLAN0 or VLAN1). If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 CFI is selected, otherwise VLAN1 CFI is selected. Note: VC is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VLAN_id (vlan_id)	VLAN ID: VLAN_id is the VLAN ID field (VLAN bits <11:0>). If VV==0 (NOT VLAN), then VLAN_id=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VLAN_id is set to the packet's VLAN id. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN ID will be used. If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 id is selected, otherwise VLAN1 ID is selected. Note: VLAN_id is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VS (vlan_stacked)	VLAN STACKED: This bit is only set if VV==1 (VLAN) AND the packet is VLAN STACKED. Note: VS is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VV (vlan_valid)	VLAN Valid: This bit is only set if the packet is VLAN or VLAN STACKED. Note: VV is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.

Table 15: WQE WORD2 Fields for Skip-to-L2 and NOT IP (CASE 3A)

Field	Definition *(Fields are in alphabetical order. The SDK software field names are shown in parenthesis.)*
B (is_bcast)	Broadcast: set when the packet's destination MAC address field in the L2 header is the broadcast address (all ones). Note: B always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
Bufs (bufs)	Number of Buffers: The number of buffers used to store the packet data. A zero value means that a dynamic short packet is stored entirely in the WQE (there is no Packet Data Buffer).
IA (is_arp)	Is ARP: Set when the packet's L2 header type field ==0x0806 (an ARP packet). Note: IA=0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
IR (is_rarp)	Is RARP: Set when the packet's L2 header type field ==0x0835 (an RARP packet). Note: IA=0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
M (is_mcast)	Multicast: Set when the packet's destination MAC address field in the L2 header is a multicast address (the group bit is set, and at least one of the remaining bits is a zero). Note: M is always zero when NOT in skip-to-L2 mode.
NI (not_IP)	Not IP: Not an IP Packet or an L1/L2 receive error has occurred.
Opcode (err_code)	Error Code: Numeric code indicating specific error which occurred: If there is an error (any of WORD2[RE] or WORD2[IE] or WORD2[LE] is set), then Opcode contains an error code, otherwise Opcode=0. The error codes values depend on which error bit is set (RE, IE, or LE). See those specific errors for details.
PR (pr)	PCIe Ring: The relative position of the PCIe ring in the PKI input port [0-7]. PR is enabled if PIP_GBL_CTL[RING_EN]==1. If the packet was not received on a PCIe port OR PIP_GBL_CTL[RING_EN]==0, then PR=0. Note: Zero is both 1) a legal ring position value and 2) the value if the packet is not received on a PCIe port.
RE (rcv_error)	Receive error (L1/L2 error): For CASE 2, by definition RE==0 because CASE 2 only occurs there is NO L1/L2 error.
S (software)	Software Use: Reserved for software use.
VC (vlan_cfi)	VLAN CFI bit: The VC bit is the VLAN CFI bit (VLAN bit <12>). If VV==0 (NOT VLAN), then VC=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VC is set to the packet's VLAN CFI bit. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN CFI will be used (VLAN0 or VLAN1). If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 CFI is selected, otherwise VLAN1 CFI is selected. Note: VC is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.

Field	Definition *(Fields are in alphabetical order. The SDK software field names are shown in parenthesis.)*
VLAN_id (vlan_id)	VLAN_ID: VLAN_id is the VLAN ID field (VLAN bits <11:0>). If VV==0 (NOT VLAN), then VLAN_id=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VLAN_id is set to the packet's VLAN id. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN id will be used. If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 id is selected, otherwise VLAN1 id is selected. Note: VLAN_id is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VS (vlan_stacked)	VLAN STACKED: This bit is only set if VV==1 (VLAN) AND the packet is VLAN STACKED. Note: VS is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VV (vlan_valid)	VLAN Valid: This bit is only set if the packet is VLAN or VLAN STACKED. Note: VV is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.

6.3 Parse Mode = Skip-to-IP

When the parse mode is "skip-to-IP", the IP header is analyzed, and appropriate fields in WQE WORD2 are set.

(For reference, the IPv4 and IPv6 headers, including the IPv4 TCP/IP combined header may be found in Section 15 – "Appendix B: Industry-Standard Reference Information".

Figure 16: WORD2 if PM=Skip-to-IP and No L1/L2 Errors (CASE 2B)

Parse Mode = Skip-to-IP and the WQE WORD2 Data Structure

This case applies if:

- The packet's parse mode is Skip-to-IP
- There are no L1/L2 errors

A packet with the parse mode "skip-to-IP" is always classified as an IP packet.

In the case of skip-to-IP parsing mode, the information from the L2 header, such as VLAN is not available. These fields are set to zero. The packet's IP information (such as whether the packet is TCP or UDP) is determined by parsing the IP header.

The data structure for WORD2 of the Work Queue Entry is shown below. Fields which require IP parsing are highlighted in green.

Reserved fields in WQE are not named and are highlighted in gray. They are set to 0.

The different fields in the data structures are explained in the next table.

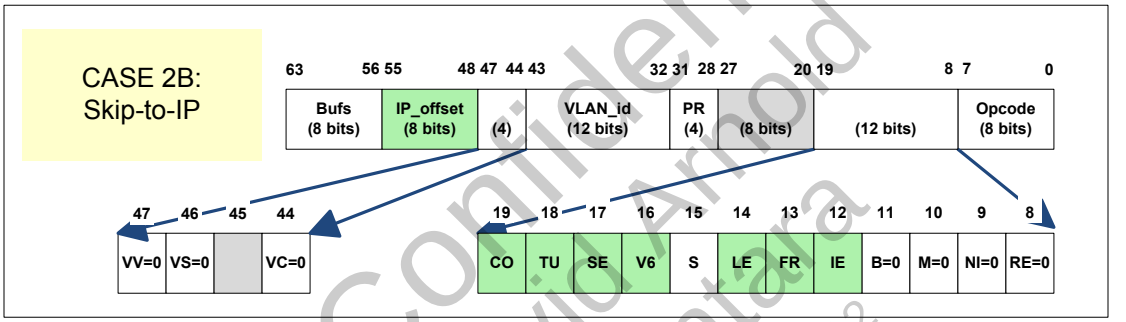


Table 16: WQE WORD2 Fields for Skip-to-IP (CASE 2B)

Field	Definition *(Fields are in alphabetical order. The SDK software field names are shown in parenthesis.)*
`B` (`is_bcast`)	Broadcast: set when the packet's destination MAC address field in the L2 header is the broadcast address (all ones). Note: `B` always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
`Bufs` (`bufs`)	Number of Buffers: The number of buffers used to store the packet data. A zero value means that a dynamic short packet is stored entirely in the WQE (there is no Packet Data Buffer).
`CO` (`dec_ipcomp`)	IP Compression: The `CO` bit (IP compression protocol bit) is set when the packet is IPCOMP (the IPv4 header protocol field or the initial IPv6 `next header` field `== 108`.) This bit is always clear when `WORD2[IE]==1` (IP error), and when `WORD2[V6]` and `WORD2[FR]` are both set. This bit indicates that the packet needs to be decompressed.

Field	Definition *(Fields are in alphabetical order. The SDK software field names are shown in parenthesis.)*
FR (is_frag)	Fragment: Set when the packet is a fragment. For IPv4, this bit is set when either the IPv4 header's MF (More Fragments) flag is set, or the IPv4 header fragment offset field is non-zero (the last fragment has the MF flag cleared and a non-zero fragment offset). For IPv6, this bit is set when the initial next header value is fragmentation (44). (For IPv6, FR is never set when WORD2[IE]==1 (IP error) .)
IE (IP_exc)	IP Error: Set when the packet has an IP exception condition. When the IE bit is set, WORD2[Opcode] contains an error code specific to this type of error. The exact error codes will be provided in a separate table. Note the bit only applies if (!RE) && (!NI).
IP_offset (ip_offset)	IP Offset: The number of bytes from the first byte of packet data to the first byte of the IP packet (the IP header).
LE (L4_error)	IP L4 Error: This bit is set when WORD2[TU] is set and the PIP/IPD hardware found an error in the TCP/UDP header and /or data. When the LE bit is set, WORD2[Opcode] contains an error code specific to this type of error. The exact error codes will be provided in a separate table. Note this bit only applies if only applies if (!RE) && (!NI) && (!IE) && (!FR).
M (is_mcast)	Multicast: Set when the packet's destination MAC address field in the L2 header is a multicast address (the group bit is set, and at least one of the remaining bits is a zero). Note: M is always zero when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
NI (not_IP)	Not IP: Not an IP Packet or a L1/L2 error has occurred. For CASE 2, by definition NI==0 because CASE 2 only occurs if the packet is IP. (Note this bit is set if the hardware cannot determine that the packet is IP. This does not necessarily mean that the packet is in fact not IP.)
Opcode (err_code)	Error Code: Numeric code indicating specific error which occurred: If there is an error (any of WORD2[RE] or WORD2[IE] or WORD2[LE] is set), then Opcode contains an error code, otherwise Opcode=0. The error codes values depend on which error bit is set (RE, IE, or LE). See those specific errors for details.
PR (pr)	PCIe Ring: The relative position of the PCIe ring in the PKI input port [0-7]. PR is enabled if PIP_GBL_CTL[RING_EN]==1. If the packet was not received on a PCIe port OR PIP_GBL_CTL[RING_EN]==0, then PR=0. Note: Zero is both 1) a legal ring position value and 2) the value if the packet is not received on a PCIe port.
RE (rcv_error)	Receive error (L1/L2 error): For CASE 2, RE==0 because CASE 2 only occurs there is NO L1/L2 error.
S (software)	Software Use: Reserved for software use.
TU (tcp_or_udp)	Is TCP or UDP: The TU bit is set when an IP packet is TCP or UDP (when the IPv4 protocol value or the IPv6 initial next header ==6 (TCP) or ==17 (UDP)). This bit is always 0 when WORD2[IE]==1, and when WORD2[V6] and WORD2[FR] are both set.
V6 (is_v6)	Is IPv6: Set if IP header version field ==6

Field	Definition *(Fields are in alphabetical order.* *The SDK software field names are shown in parenthesis.)*
VC (vlan_cfi)	VLAN CFI bit: The VC bit is the VLAN CFI bit (VLAN bit <12>). If VV==0 (NOT VLAN), then VC=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VC is set to the packet's VLAN CFI bit. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN CFI will be used (VLAN0 or VLAN1). If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 CFI is selected, otherwise VLAN1 CFI is selected. Note: VC is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VLAN_id (vlan_id)	VLAN ID: VLAN_id is the VLAN ID field (VLAN bits <11:0>). If VV==0 (NOT VLAN), then VLAN_id=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VLAN_id is set to the packet's VLAN id. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN ID will be used. If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 id is selected, otherwise VLAN1 ID is selected. Note: VLAN_id is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VS (vlan_stacked)	VLAN STACKED: This bit is only set if VV==1 (VLAN) AND the packet is VLAN STACKED. Note: VS is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VV (vlan_valid)	VLAN Valid: This bit is only set if the packet is VLAN or VLAN STACKED. Note: VV is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.

6.4 Parse Mode = Uninterpreted

When the parse mode is "uninterpreted", there are two cases for setting WQE WORD2:

1. The packet has a Packet Instruction Header and the RAW bit is set to 1 in the instruction header
2. Either the packet does not have a Packet Instruction Header, or the RAW bit is set to 0.

Figure 17: WORD2 if PM=Unint., RAW, No L1/L2 Errors (CASE 1A, CASE 1B)

Parse Mode = Uninterpreted and RAW (RAWFULL): WQE WORD2 Data Structure

This case applies if the packet is RAWFULL. For the packet to be RAWFULL, all of the following must be true:

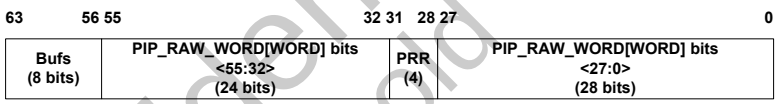
- The packet has a Packet Instruction Header or PCI Instruction Header
- The instruction header's parse mode field is "Uninterpreted"
- The RAW bit is set in the instruction header
- There are no L1/L2 errors

RAWFULL packets use **`PIP_RAW_WORD[WORD]`** to create WQE WORD2. There is no requirement for the contents of **`PIP_RAW_WORD[WORD]`**.

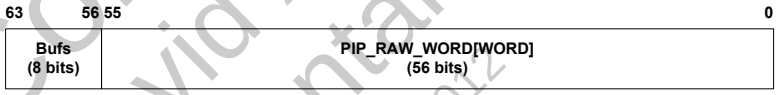
If **`PIP_GBL_CTL[RING_EN]==1`**, and the packet is received on a PCI/PCIe port, PCI ring information is automatically inserted into WORD2 instead of **`PIP_RAW_WORD[WORD]`** bits <31:28>.

The different fields in the data structures are explained in the next table.

Case 1A:
RAWFULL
AND
PCIe
AND
`PIP_GBL_CTL[RING_EN]==1`



Case 1B:
RAWFULL
AND
((NOT PCIe) OR
`PIP_GBL_CTL[RING_EN]=0)`



As of SDK 1.9, there is no software data structure for CASE1 fields, so the following table does not specify a software field name after the hardware field name.

Table 17: WQE WORD2 Fields for RAWFULL (CASE 1A and CASE 1B)

Field	Definition (Fields are in alphabetical order.)
Bufs	Number of Packet Data Buffers: The number of buffers used to store the packet data. A zero value means that a dynamic short packet is stored entirely in the WQE (there is no Packet Data Buffer).
PIP_RAW_WORD	RAW Word: Set to the value of the PIP_RAW_WORD configuration register. Incoming packets may optionally contain a Packet Instruction Header. If the R (RAW) bit is set in the Packet Instruction Header, and the parse mode is uninterpreted, then the packet is *RAWFULL*. RAWFULL packets are not parsed and decoded by PIP/IPD to fill in the WQE WORD2 fields. Instead, the value of the PIP_RAW_WORD register is used to populate WQE WORD2.
PIP_RAW_WORD<55:32>	RAW Word bits <55:32>: If the packet is RAWFULL and arrived on a PCIE port and PIP_GBL_CTL[RING_EN]==1, then bits <31:28> of WQE WORD2 are replaced by PRR. This causes the PIP_RAW_WORD to be split into two parts, with PRR occupying WQE WORD2 bits <31:28>.
PIP_RAW_WORD<27:0>	RAW Word bits <27:0>: The value of PIP_RAW_WORD configuration register, bits <27:0>. See PIP_RAW_WORD<55:32>.
PRR	PCIe Ring RAW: If the packet is RAWFULL and arrived on a PCIE port and PIP_GBL_CTL[RING_EN]==1, then bits <31:28> of WQE WORD2 set to the relative position of the PCIe ring in the PKI input port [0-7]. (Technically, there is only one data structure: if the packet was not received on a PCIe port or PIP_GBL_CTL[RING_EN]==0, then PRR=PIP_RAW_WORD<31:38>. To reduce complexity, this is shown in the figure above as a different data structure with PIP_RAW_WORD<55:0> uninterrupted.)

Figure 18: WORD2 if PM=Unint., NOT RAW, No L1/L2 Errors (CASE 3B)

Parse Mode = Uninterpreted and NOT RAW (NOT RAWFULL):
WQE WORD2 Data Structure

This case applies if:

- The packet has a Packet Instruction Header or PCI Instruction Header, AND
- The instruction header's parse mode field is "Uninterpreted", AND
- The RAW bit is NOT set in the instruction header
- There are no L1/L2 errors

This case also applies if:

- If there is no Packet Instruction Header or PCI Instruction Header AND
- The port's configured parse mode is set to "Uninterpreted" (**`(PIP_PRT_CFGn[MODE])==0`** (no packet inspection)).
- There are no L1/L2 errors

Note: "RAW" cannot be set without an instruction header.

In this case, **`NI`** is set to 1 to indicate this is not an IP packet.

Reserved fields in WQE are not named and are highlighted in gray. They are set to 0.

The different fields in the data structures are explained in the next table.

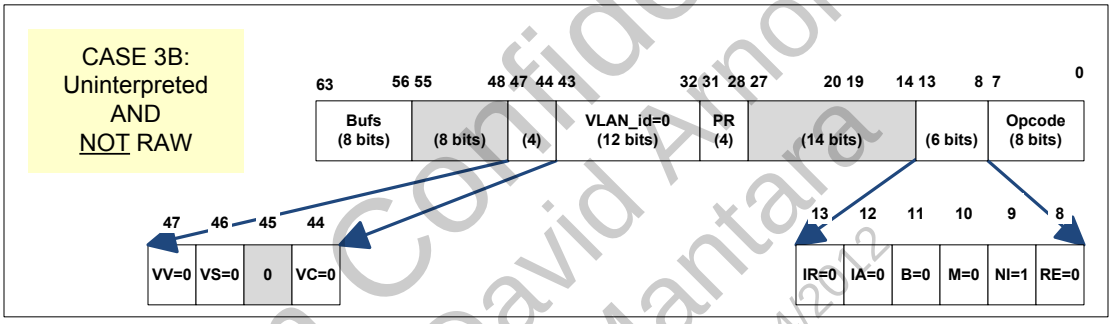


Table 18: WQE WORD2 Fields for Uninterpreted and not RAW (CASE 3B)

Field	Definition *(Fields are in alphabetical order.* *The SDK software field names are shown in parenthesis.)*
`B` `(is_bcast)`	Broadcast: set when the packet's destination MAC address field in the L2 header is the broadcast address (all ones). Note: `B` always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
`Bufs` `(bufs)`	Number of Buffers: The number of buffers used to store the packet data. A zero value means that a dynamic short packet is stored entirely in the WQE (there is no Packet Data Buffer).
`IA` `(is_arp)`	Is ARP: Set when the packet's L2 header `type` field `==0x0806` (an ARP packet). Note: `IA=0` when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.

Field	Definition *(Fields are in alphabetical order. The SDK software field names are shown in parenthesis.)*
IR (is_rarp)	Is RARP: Set when the packet's L2 header type field ==0x0835 (an RARP packet). Note: IA=0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
M (is_mcast)	Multicast: Set when the packet's destination MAC address field in the L2 header is a multicast address (the group bit is set, and at least one of the remaining bits is a zero). Note: M is always zero when NOT in skip-to-L2 mode.
NI (not_IP)	Not IP: Not an IP Packet or an L1/L2 receive error has occurred.
Opcode (err_code)	Error Code: Numeric code indicating specific error which occurred: If there is an error (any of WORD2[RE] or WORD2[IE] or WORD2[LE] is set), then Opcode contains an error code, otherwise Opcode=0. The error codes values depend on which error bit is set (RE, IE, or LE). See those specific errors for details.
PR (pr)	PCIe Ring: The relative position of the PCIe ring in the PKI input port [0-7]. PR is enabled if PIP_GBL_CTL[RING_EN]==1. If the packet was not received on a PCIe port OR PIP_GBL_CTL[RING_EN]==0, then PR=0. Note: Zero is both 1) a legal ring position value and 2) the value if the packet is not received on a PCIe port.
RE (rcv_error)	Receive error (L1/L2 error): For CASE 2, by definition RE==0 because CASE 2 only occurs there is NO L1/L2 error.
S (software)	Software Use: Reserved for software use.
VC (vlan_cfi)	VLAN CFI bit: The VC bit is the VLAN CFI bit (VLAN bit <12>). If VV==0 (NOT VLAN), then VC=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VC is set to the packet's VLAN CFI bit. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN CFI will be used (VLAN0 or VLAN1). If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 CFI is selected, otherwise VLAN1 CFI is selected. Note: VC is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VLAN_id (vlan_id)	VLAN_ID: VLAN_id is the VLAN ID field (VLAN bits <11:0>). If VV==0 (NOT VLAN), then VLAN_id=0. If VV==1 and VS==0 (NOT VLAN STACKED), then VLAN_id is set to the packet's VLAN id. If VV==1 and VS==1 (VLAN STACKED), then PIP_GBL_CTL[VS_WQE] is used to select which VLAN id will be used. If PIP_GBL_CTL[VS_WQE]==0, then VLAN0 id is selected, otherwise VLAN1 id is selected. Note: VLAN_id is always 0 when NOT in skip-to-L2 mode because VV==0. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VS (vlan_stacked)	VLAN STACKED: This bit is only set if VV==1 (VLAN) AND the packet is VLAN STACKED. Note: VS is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.
VV (vlan_valid)	VLAN Valid: This bit is only set if the packet is VLAN or VLAN STACKED. Note: VV is always 0 when NOT in skip-to-L2 mode. The value is unpredictable when there is an L1/L2 error because the packet may be corrupted.

6.5 Registers to Configure WQE WORD2 Content

Work Queue Entry WORD1 and WORD2 field content is controlled by user-configured variables. In addition to the variables shown in this section, see Section 9 – "Error Check Configuration".

The specific FPA pool used for WQE buffers is configurable via the `IPD_WQE_FPA_QUEUE[WQE_QUE]` field.

Table 19: Registers to Configure Work Queue Entry WORD2

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Add PCIe Ring Information: If set to 1, add PCIe ring information to WQE WORD2.	PIP_GBL_CTL	RING_EN	0	0 (H/W Default)
Select Which VLAN to Use: Which VLAN CFI bit to use for VLAN Stacking: 0=use first VLAN (network order) 1=use second VLAN (network order)	PIP_GBL_CTL	VS_WQE	0	0 (H/W Default)
Specify WORD2 for RAWFULL Packets: Contains the WQE WORD2 value for RAWFULL packets. The 8-bit `bufs` field is still set by IPD. Note there is only one configuration register for all ports.	PIP_RAW_WORD	WORD	0	0 (H/W Default)

6.6 Where to Find More Information About Parsing

For readers who need more details, see Section 16 – "Appendix C: Input Packet Parsing", and the *HRM*.

7 Scheduling (WQE WORD1)

The PIP/IPD unit is responsible for setting packet information needed by the scheduler: the packet's Group, QoS, Tag Type, and Tag Value. These are all fields in the WQE which is submitted to the SSO via the `add_work` operation.

The PIP/IPD provides various options in how these fields are set. This section introduces the various options, and presents technical details.

> ***Note that if a Packet Instruction Header is used, and the RAW bit in the instruction header is set, then Group, QoS, Tag Type, and Tag Value are all taken directly from the instruction header: register configuration fields are ignored.***

In addition to register variables which control how these fields are set, PIP/IPD provides port watchers, which look for specific types of packets. Port watchers can be used to set either the group or QoS value of matched packets.

7.1 Work Group Assignment (WQE WORD1 Group Field)

There are four methods for setting the group value:

1. Specify group in the Packet Instruction Header (RAWFULL, RAWSCH)
2. Derive group from the packet's Tag Value (GRPTAG)
3. Set group via a Port Watcher (for matched packets) (See Section 7.5 – "Using Watchers to Set QoS and Group")
4. Take Default value for the port

The per-port GRPTAG feature can be used to direct all the traffic from a flow to one work group ("flow steering"), which can reduce the locking needed by an application. If all locking is per-flow, this feature could be used to implement a completely lockless system. Flow steering is also sometimes used to improve L1 cache hits (core affinity). The problem is that having only one core process a flow, where the Tag Type is `ORDERED` means that the power of parallel processing (multiple cores processing the same flow) is not being used.

This feature can also be used for load balancing. Since the tag is based on a CRC, the bits in it are fairly evenly distributed. Including these bits in the group value results in a random distribution of flows over the groups. The groups are then mapped to the available cores (load balancing). One caveat occurs when the number of groups created does not evenly map to the number of cores used to process the groups. No matter what the `GRPTAGMASK` value is, the result is always a power-of-two number of groups (1, 2, 4, 8, and 16). If the number of available cores is not a power of two (3, 5-7, or 9-15), then the groups will not map evenly to the cores, and the traffic load will be unbalanced between cores.

> ***Note: This feature is not generally useful for load balancing because the incoming traffic load is not balanced.***

The GRPTAG formula is:

```
Group=
```

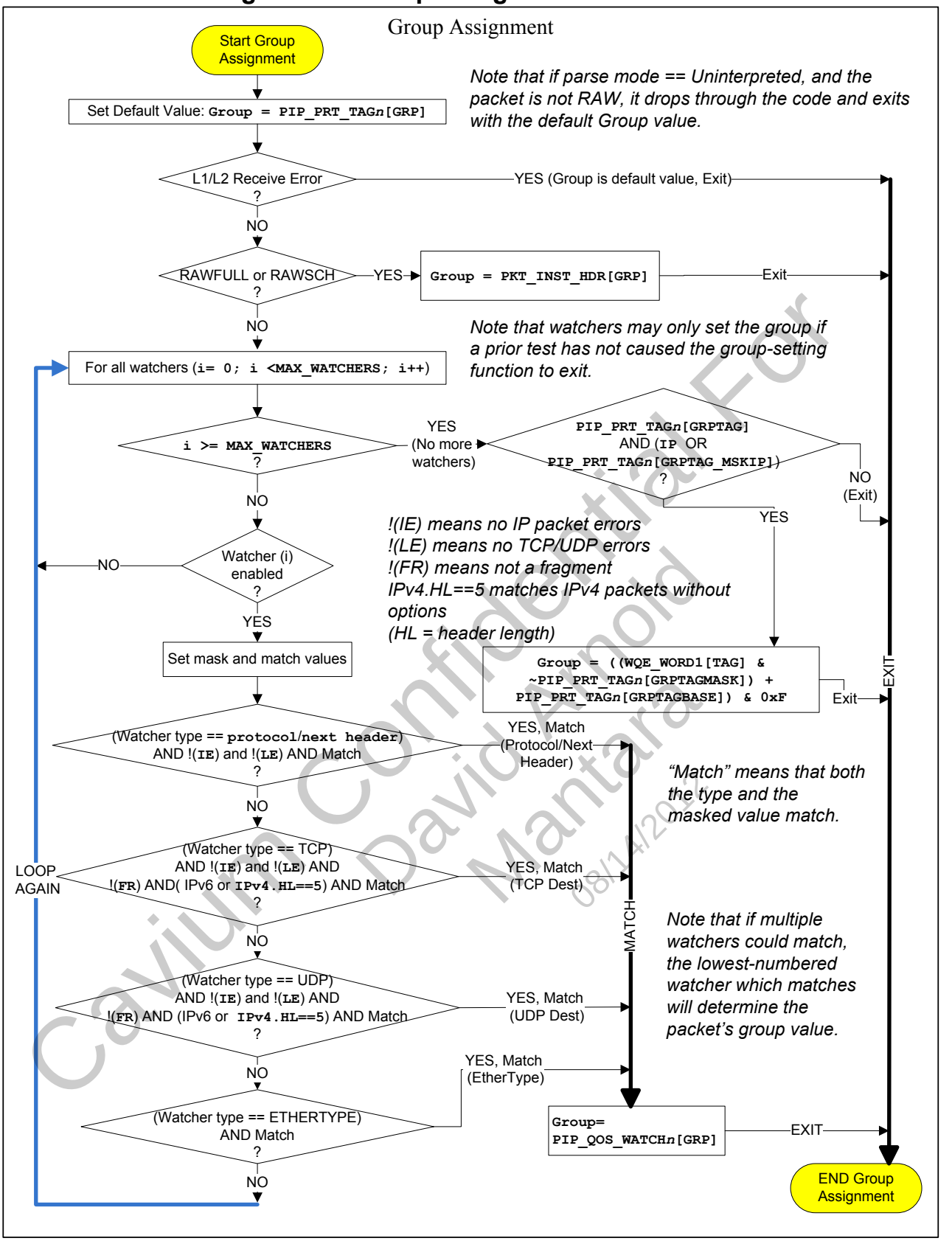
```
((WQE_WORD1[TAG] & ~PIP_PRT_TAGn[GRPTAGMASK]) + PIP_PRT_TAGn[GRPTAGBASE]) & 0xF
```

Note that GRPTAGMASK specifies the bits to be *excluded* (the bits that should not be considered). When the AND operation is done it is with the NOT of GRPTAGMASK: TAG & ~GRPTAGMASK.

GRPTAGBASE is an offset which allows low group numbers to be excluded from the GRPTAG calculation. The lower group numbers can then be used for special traffic mapping. For example, For example, the group number for ARP packets (which are not part of a specific flow) can be set to 0, while flows are directed to group numbers greater than 0.

For example, to process a mixture of IP and non-IP traffic, the IP traffic will use the GRPTAG feature while the non-IP traffic will not. GRPTAGBASE allows you to differentiate between the two. Non-IP can use groups 0 through (GRPTAGBASE-1), while IP uses groups GRPTAGBASE through 15.

Figure 19: Group Assignment Flow Chart



7.1.1 Registers to Configure Group Assignment

The registers fields in the following table are in logical order, not in *HRM* order.

Table 20: Registers to Configure WQE WORD1 Group Assignment

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Default Group for the Port				
Default Group: Set the default work group number for the packet	PIP_PRT_TAG*n* (one per port)	GRP	0	0 (H/W Default)
Calculate Group from Tag Value				
IP GRPTAG: GRPTAG enable for IP packets: If GRPTAG==1, use WQE tag value to create the Work Group value for an IP packet	PIP_PRT_TAG*n* (one per port)	GRPTAG	0	0 (H/W Default)
Non-IP GRPTAG: Additional enable for non-IP packets: If GRPTAG==1 and GRPTAG_MSKIP==1, use the WQE tag value to create the Work Group value even if the packet is not an IP packet	PIP_PRT_TAG*n* (one per port)	GRPTAG_MSKIP	0	0 (H/W Default)
GRPTAG Mask: If GRPTAG==1, specify which bits of WQE Tag value to exclude from the Work Group computation	PIP_PRT_TAG*n* (one per port)	GRPTAGMASK	0	0 (H/W Default)
GRPTAG Base: If GRPTAG==1, specifies the offset to use to compute the WQE Work Group from tag value	PIP_PRT_TAG*n* (one per port)	GRPTAGBASE	0	0 (H/W Default)

7.2 QoS Assignment

The QoS value can be controlled in various ways. It can come from:

1. A default value
2. The Packet Instruction Header (for RAWFULL and RAWSCH packets)
3. A Broadcom HiGig Header priority converted to a QoS
4. A VLAN or VLAN stacked priority converted to a QoS
5. An IP Diffserv priority (which can be configured to take precedence over VLAN priority)
5. A Port Watcher (similar process as for group value setting) (See Section 7.5 – "Using Watchers to Set QoS and Group")

The function `cvmx_pip_config_vlan_qos()` can be used to configure the VLAN-to-QoS mapping Table0 (shown in the figure below). As of SDK 1.9, there is no function to configure VLAN-to-QoS mapping Table1.

The function `cvmx_pip_config_diffserv_qos()` can be used to configure the diffserv-to-QoS mapping table.

Figure 20: Deriving QoS From VLAN Priority

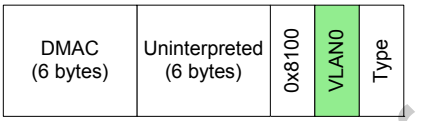
Deriving QoS from VLAN Priority

If the L2 header contains VLAN or VLAN Stacking (two VLAN entries), then the VLAN "priority" field (PCP) can be used to set the packet's QoS value.

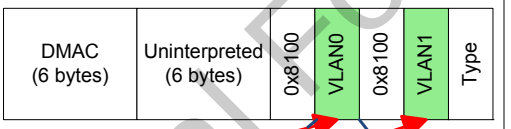
To enable setting QoS using VLAN information for the port:
`PIP_PRT_CFGn[QOS_VLAN]=1`

If VLAN stacking (two VLAN entries in the L2 header), decide which VLAN to use *for all ports* to get the VLAN priority field:
`PIP_GBL_CTL[VS_QOS]`
0=Use VLAN0
1=Use VLAN1

Ethernet II + VLAN

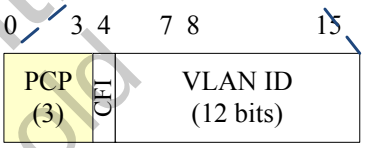


Ethernet II + VLAN Stacked



The VLAN priority is extracted from the selected VLAN entry in the L2 header (VLAN0 if no VLAN stacking).

This priority is used as an index into a VLAN-to-QoS mapping table.



PCP:
Priority 0=lowest, 7=highest

There are two tables used to map VLAN priority to QoS.
Each port can select which table to use via **`PIP_PRT_CFGn[QOS_VSEL]`**:
0=table0
1=table1

Table0

VLAN Priority	QOS
0	
1	
2	
3	
4	
5	
6	
7	

Table1

VLAN Priority	QOS1
0	
1	
2	
3	
4	
5	
6	
7	

The user configures the VLAN priority-to-QoS mapping values in the two VLAN-to_QoS mapping tables, using the **`PIP_QOS_VLANn`** register, where *n* is the VLAN priority.

Table0	Table1
`PIP_QOS_VLAN0[QOS]`	`PIP_QOS_VLAN0[QOS1]`
`PIP_QOS_VLAN1[QOS]`	`PIP_QOS_VLAN1[QOS1]`
`PIP_QOS_VLAN2[QOS]`	`PIP_QOS_VLAN2[QOS1]`
`PIP_QOS_VLAN3[QOS]`	`PIP_QOS_VLAN3[QOS1]`
`PIP_QOS_VLAN4[QOS]`	`PIP_QOS_VLAN4[QOS1]`
`PIP_QOS_VLAN5[QOS]`	`PIP_QOS_VLAN5[QOS1]`
`PIP_QOS_VLAN6[QOS]`	`PIP_QOS_VLAN6[QOS1]`
`PIP_QOS_VLAN7[QOS]`	`PIP_QOS_VLAN7[QOS1]`

Figure 21: QoS Assignment Flowchart, part 1

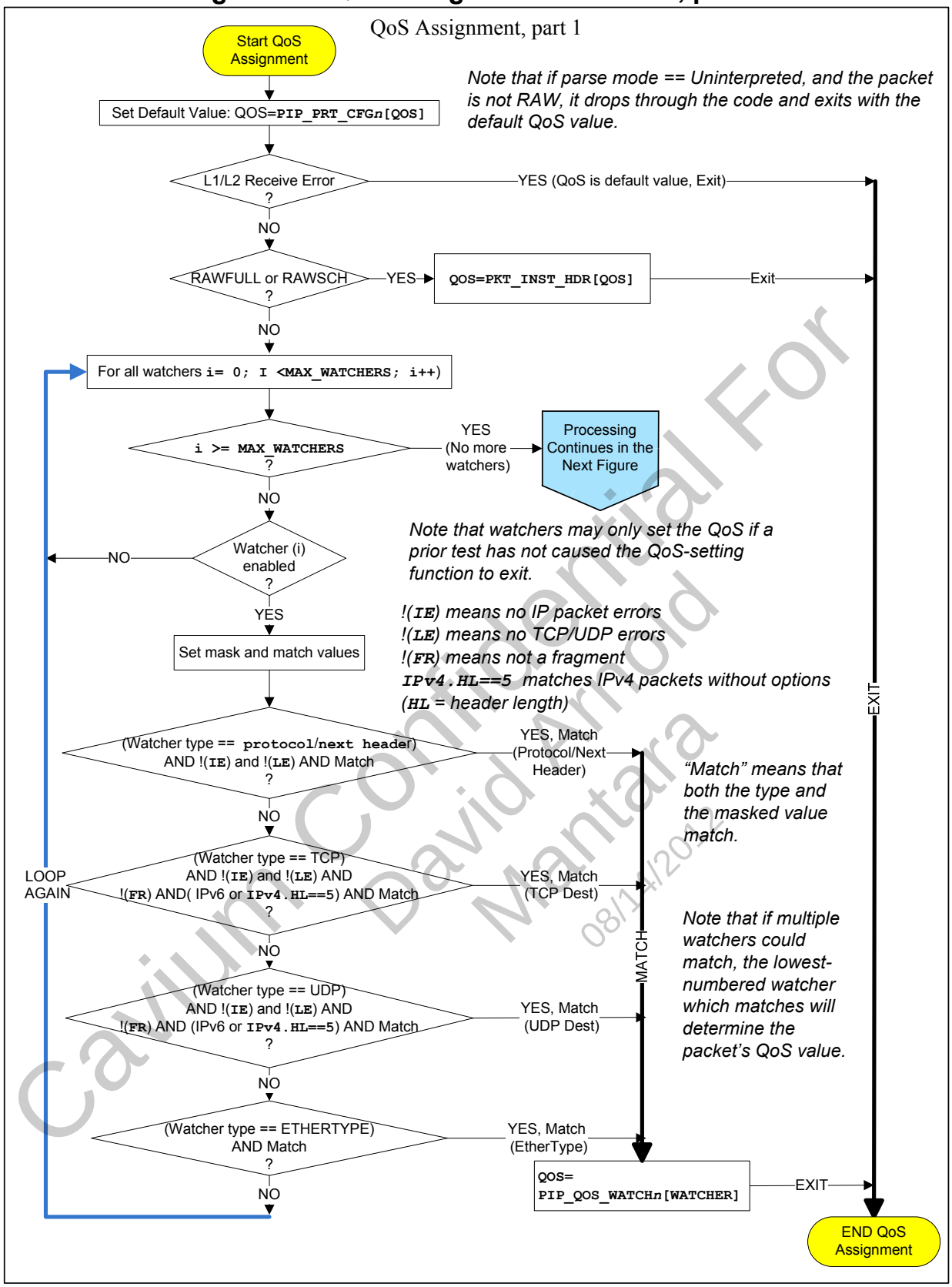
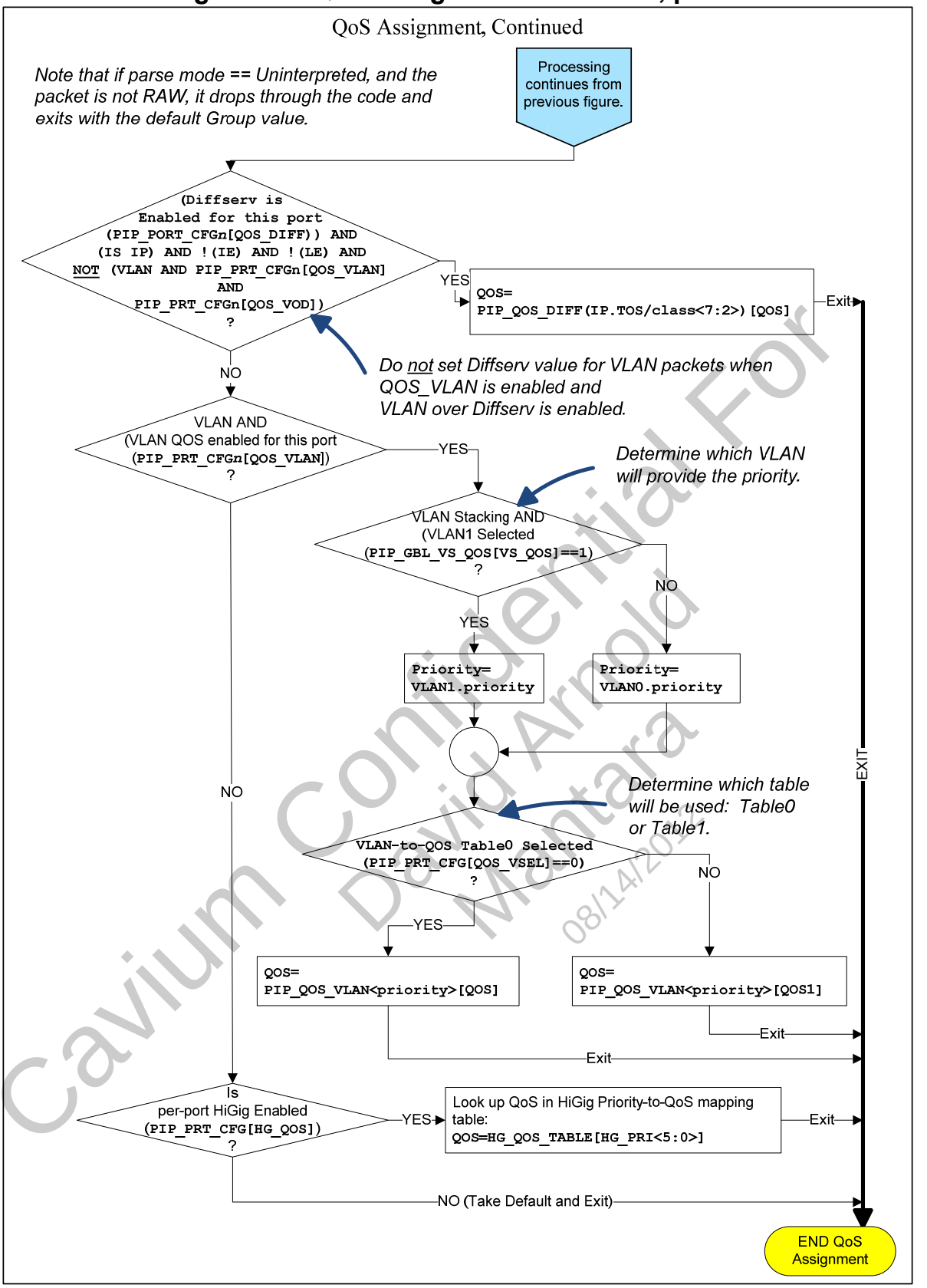


Figure 22: QoS Assignment Flowchart, part 2



7.2.1 Registers to Configure QoS Assignment

In the following table, ignore fields which are not applicable. For example, if you are not using Broadcom HiGig, or VLAN STACKING, then ignore the variables.

The packet's QoS value can also be set by a global watcher. See Section 7.5 – "Using Watchers to Set QoS and Group" for watcher configuration.

The register fields in the following table are in logical order, not *HRM* order.

Table 21: Registers to Configure WQE WORD1 QoS Assignment

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Default QoS for the Port				
Default QoS: One register per port. Default QoS for the port	PIP_PRT_CFG*n* (one per port)	QOS	0	0 (See Note1)
VLAN				
Enable VLAN Priority over QoS: One register per port. If set to 1, If VLAN, then VLAN priority will be used to set QoS	PIP_PRT_CFG*n* (one per port)	QOS_VLAN	0	0 (H/W Default)
Select VLAN-to-QoS MappingTable: One register per port. If VLAN, Select which VLAN-to-QoS mapping table to use for this port 0=PIP_QOS_VLAN*n*[QOS] (Table0) 1=PIP_QOS_VLAN*n*[QOS1] (Table1)	PIP_PRT_CFG*n* (one per port)	QOS_VSEL	0	0 (H/W Default)
Configure VLAN-to-QoS mappping Table0: One entry per VLAN priority (0-7). (See PIP_PRT_CFG*n*[QOS_VSEL])	PIP_QOS_VLAN(0-7) (one per VLAN priority)	QOS	0	0 (H/W Default) (See Note2)
Configure VLAN-to-QoS mappping Table1: One entry per VLAN priority (0-7). (See PIP_PRT_CFG*n*[QOS_VSEL])	PIP_QOS_VLAN(0-7) (one per VLAN priority)	QOS1	0	0 (H/W Default)
VLAN STACKING (VLAN registers also apply if VLAN stacking is used)				
For VLAN STACKING, Select which VLAN field will be used: Global setting (for *all* ports): select which VLAN field in the L2 header will provide the VLAN priority 0=VLAN0 1=VLAN1	PIP_GBL_CTL	VS_QOS	0	0 (H/W Default)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Broadcom HiGig				
Enable Broadcom HiGig: One register per port. Enables HiGig QoS lookup based on HiGig Priority	PIP_PRT_CFG*n* (one per port)	HG_QOS	0	0 (H/W Default) (See Note3)
Select Table Entry to Update: The HiGig Priority (index to the HG_QOS_TABLE). (See also UP_QOS)	PIP_HG_PRI_QOS	PRI	0	0 (H/W Default) (See Note3)
Set QoS Value for Table at Index PRI: Map the priority PRI (used as an index to the HG_QOS_TABLE) to this QoS value. (See also UP_QOS)	PIP_HG_PRI_QOS	QOS	0	0 (H/W Default) (See Note3)
Configure HiGig QoS mapping table: Global register. When set, updates the entry in the HiGig QoS table, where the table index is specified by PRI and the value is specified by QOS. This table maps the HiGig priority to a QoS level.	PIP_HG_PRI_QOS	UP_QOS	0	0 (H/W Default) (See Note3)
Diffserv				
Enables QoS for Diffserv: One register per port. This enables using the diffserv value in the packet header to determine the destination SSO QoS queue.	PIP_PRT_CFG*n* (one per port)	QOS_DIFF	0	0 (H/W Default) (See Note4)
Diffserv Maping Table: One register per Diffserv value. For each Diffserv value (level), specifies the QoS value (maps 64 Diffserv levels to 8 QoS levels)	PIP_QOS_DIFF(0-63) (one per Diffserv level)	QOS	0	0 (H/W Default) (See Note4)
VLAN over Diffserv				
VLAN over Diffserv: One register per port. The VLAN QoS value takes priority over Diffserv in setting the QoS	PIP_PRT_CFG*n* (one per port)	QOS_VOD	0	0 (H/W Default)
Notes				
Note1: When using the helper functions, the SDK configures the QoS value so that each port sends packets to a different SSO queue: `port_config.s.qos = ipd_port & 0x7;`				
Note2: Can be configured via `cvmx_pip_config_vlan_qos()`				
Note3: Can be configured via `cvmx_higig_initialize()`				
Note4: Can be configured via `cvmx_pip_config_diffserv_qos()`				

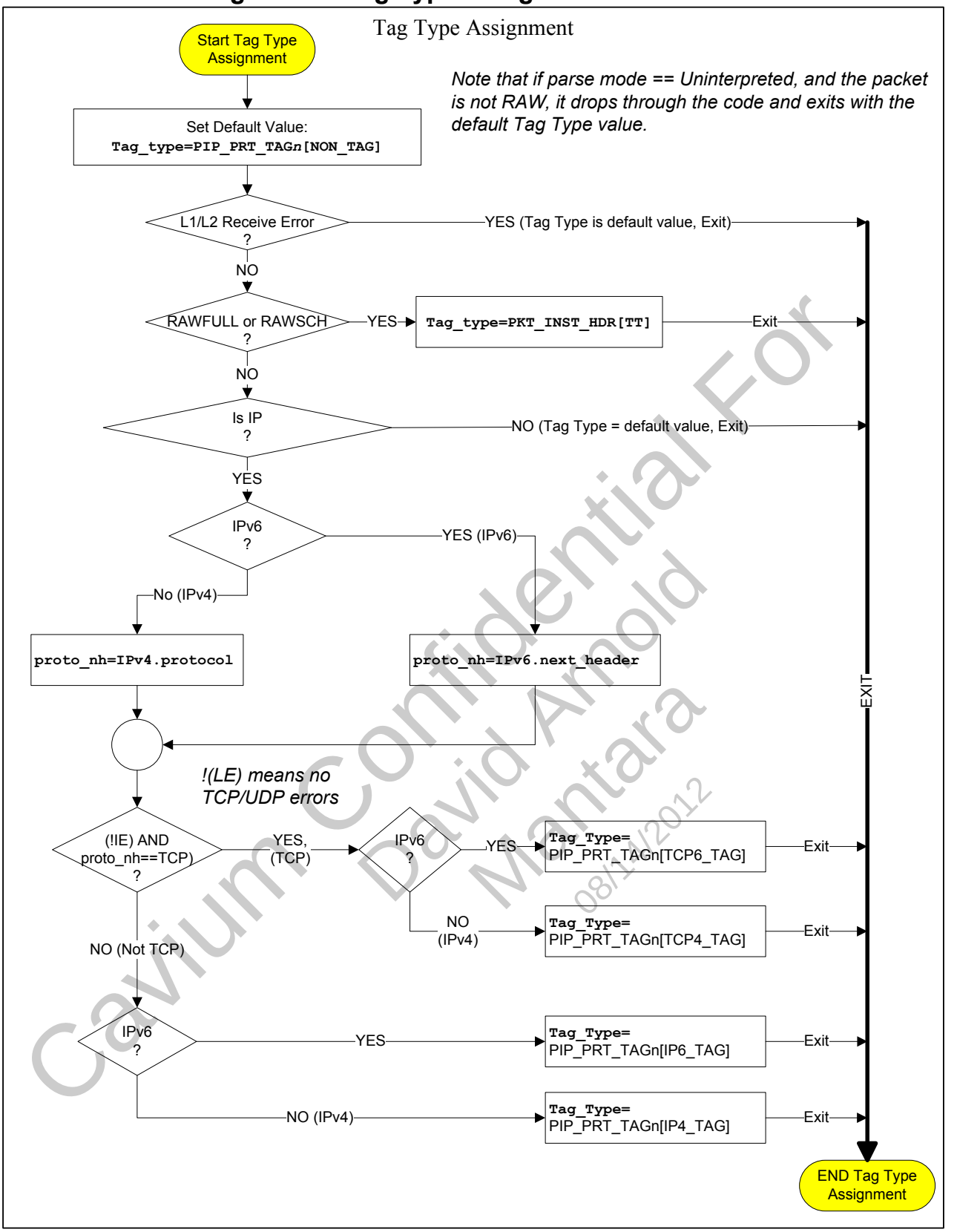
7.3 Tag Type Assignment

7.3.1 WQE WORD1 Tag Type

Tag Type can be any of:

- ATOMIC: Use ATOMIC tag type if there are shared resources which need to be protected with a packet-linked lock: packets with the same tag tuple will be processed one-at-a-time in ingress order. The SSO will maintain the packets in ingress order.
- ORDERED: Use ORDERED tag type if there are no shared resources to protect: packets with the same tag tuple can be processed in parallel. The SSO will maintain the packets in ingress order.
- NULL: Use NULL if neither shared resources nor ingress order need to be protected: packets with the same tag tuple can be processed in parallel. The SSO will *not* maintain the packets in ingress order.

Figure 23: Tag Type Assignment Flowchart



7.3.2 Registers to Configure Tag Type Assignment

Specify Tag Type in the fields in the table below as:

0=ORDERED (CVMX_POW_TAG_TYPE_ORDERED)
1=ATOMIC (CVMX_POW_TAG_TYPE_ATOMIC)
2=NULL (CVMX_POW_TAG_TYPE_NULL)

Table 22: Registers to Configure WQE WORD1 Tag Type Assignment

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Non-IP Tag Type (Default Value)				
Default Tag Type: Set tag type if not IP (See Note1)	PIP_PRT_TAG*n* (one per port)	NON_TAG	0	0 (See Note2)
IP Tag Type - TCP				
IPv6 TCP Tag Type: Set tag type of TCP packet (See Note1)	PIP_PRT_TAG*n* (one per port)	TCP6_TAG	0	0 (See Note2)
IPv4 TCP Tag Type: Set tag type of TCP packet (See Note1)	PIP_PRT_TAG*n* (one per port)	TCP4_TAG	0	0 (See Note2)
IP Tag Type - NOT TCP				
IPv6 !TCP Tag Type: set tag type if NOT TCP (See Note1)	PIP_PRT_TAG*n* (one per port)	IP6_TAG	0	0 (See Note2)
IPv4: !TCP Tag Type set tag type if NOT TCP (See Note1)	PIP_PRT_TAG*n* (one per port)	IP4_TAG	0	0 (See Note2)
Notes				
Note1: Three choices: 0=ORDERED (CVMX_POW_TAG_TYPE_ORDERED) 1=ATOMIC (CVMX_POW_TAG_TYPE_ATOMIC) 2=NULL (CVMX_POW_TAG_TYPE_NULL)				
Note2: Configured via executive-config.h: CVMX_HELPER_INPUT_TAG_TYPE = CVMX_POW_TAG_TYPE_ORDERED The helper function sets the values for all these fields to the same value: ORDERED.				

When using Simple Executive, all fields are set to the same configuration variable which is defined in executive-config.h:
CVMX_HELPER_INPUT_TAG_TYPE:

This variable is set to ORDERED by default:

```
#define CVMX_HELPER_INPUT_TAG_TYPE CVMX_POW_TAG_TYPE_ORDERED
```

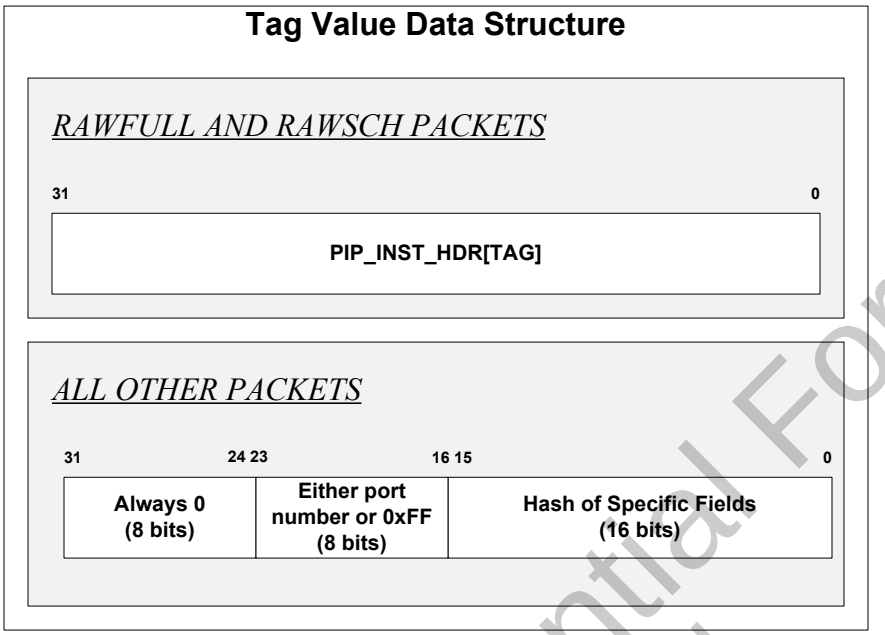
This variable is used in the internal-use function __cvmx_helper_port_setup_ipd().

7.4 Tag Value Assignment

The tag value can be used to create multiple unique virtual work flows. Ideally, the unique work flows correspond to IP header per-flow resources. Then, when a packet-linked lock (ATOMIC tag

type) is used to provide exclusive access to per-flow resources, non-related flows are not blocked waiting for the lock. A tuple tag is recommended for this purpose.

Figure 24: Tag Value Data Structure



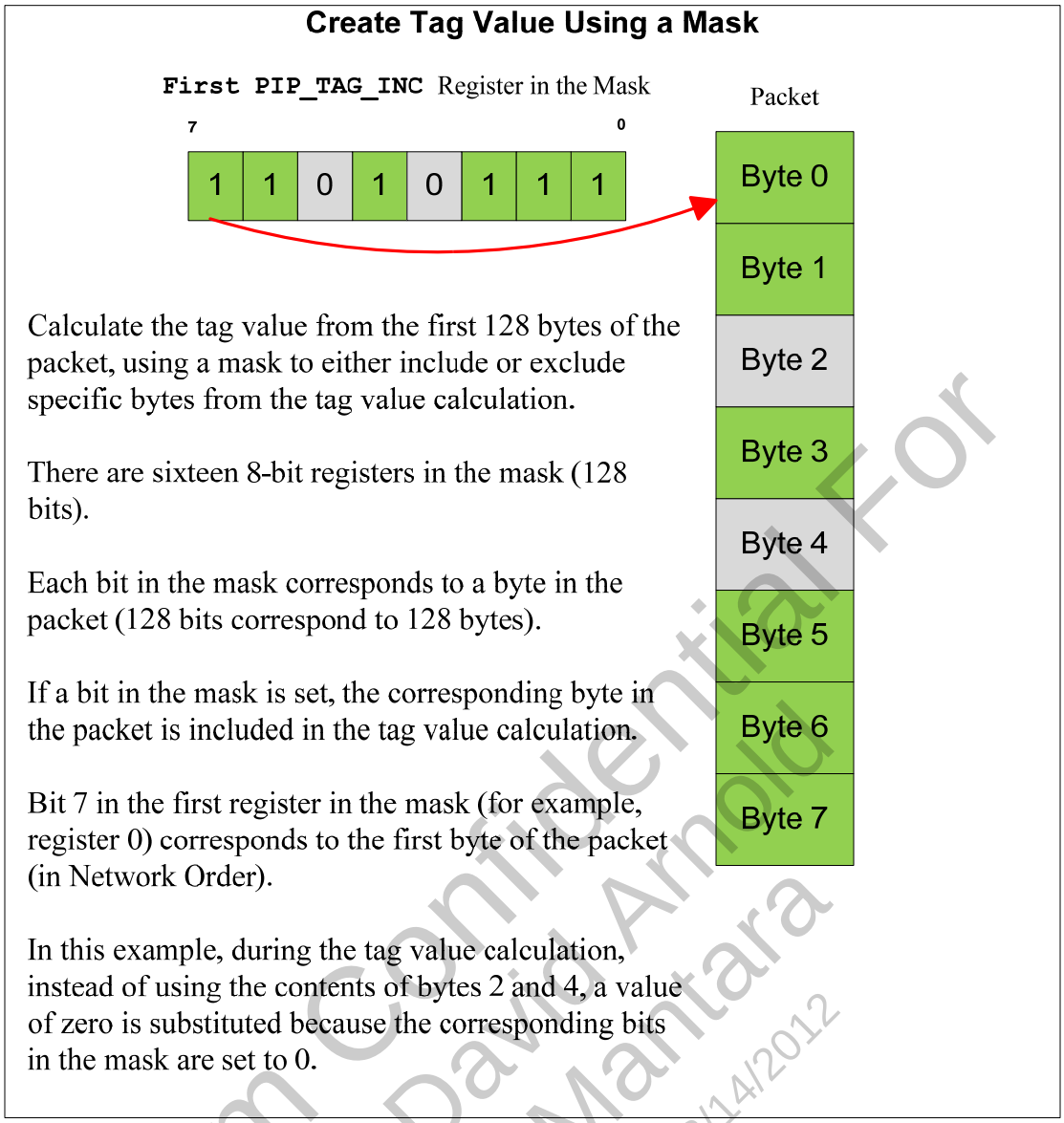
There are four tag mode choices (selected via TAG_MODE):

- Create tag value using the tuple tag algorithm (only useful for IPv4 or IPv6 packets)
- Create tag value using the mask tag algorithm
- Create tag value using tuple tag if IP, else use mask tag
- Create tag value using tuple tag XOR mask tag

The tuple tag is a hash which optionally includes the IP source and destination addresses, the IP protocol (IPv4) or next header (IPv6) value, the TCP/UDP source and destination ports, and the VLAN ID. There are per-port configuration variables which control these options. The tuple tag also includes a secret value. The hash results change with different secret values.

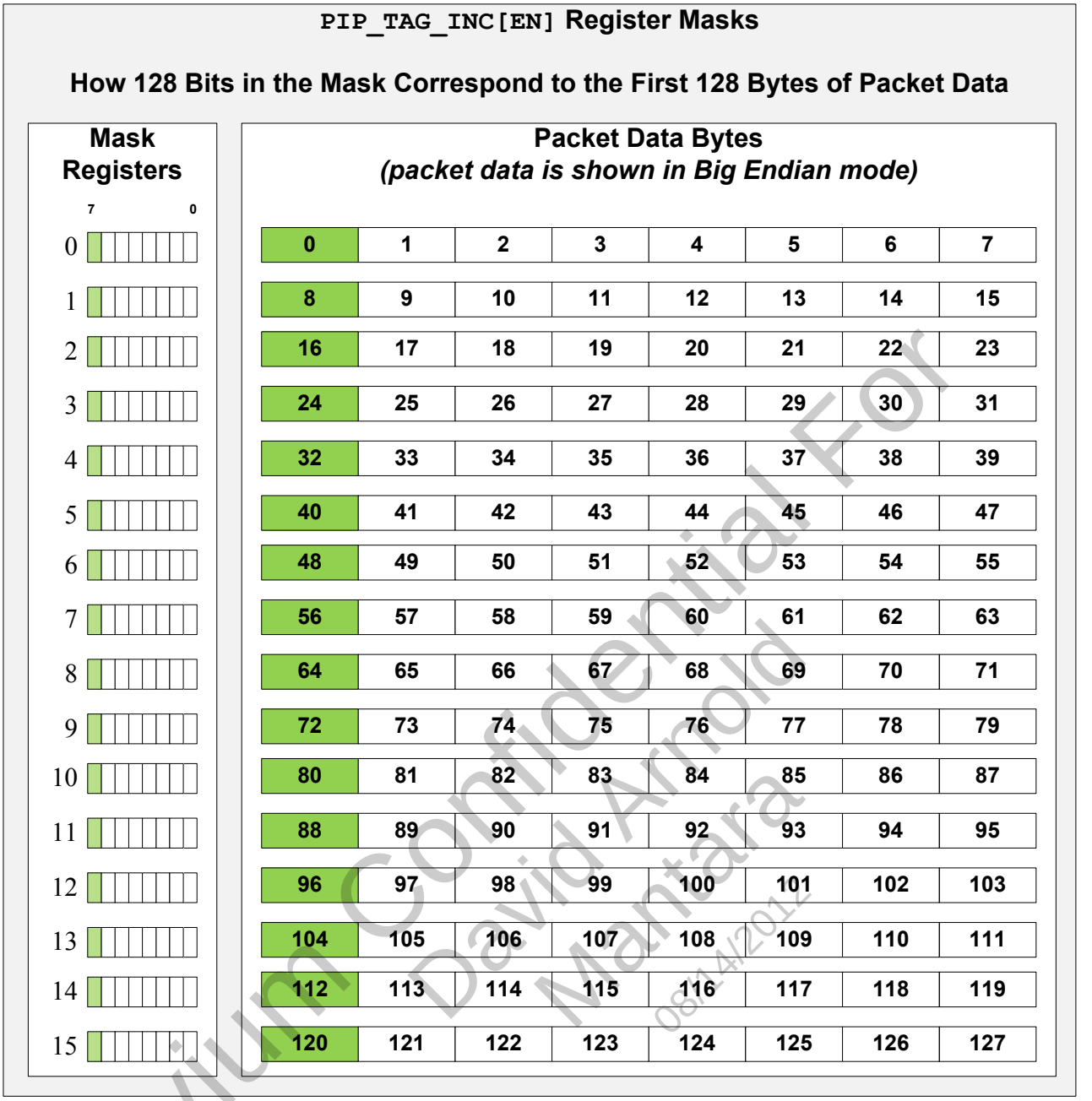
The mask tag is a hash which optionally includes/excludes any of the first 128 bytes of packet data (starting at byte 0). A 128-bit mask is used to select which of the 128 bytes are included: each bit in the mask represents a corresponding byte. The mask consists of 16 8-bit registers (16*8=128 bits), the `PIP_TAG_INC`*r* (PIP tag include) registers. There are four global masks. The specific mask used by a port is selected via the `PIP_PRT_CFG`*n*`[TAG_INC]` field. The following figures show how the bits in the `PIP_TAG_INC`*r* registers correspond to the first 128 bytes of packet data. The `PIP_TAG_INC`*r* registers are used in the tag mask algorithm (`hw_mask_tag()`), and are not used for the tag tuple algorithm (`hw_tuple_tag()`). Byte 0 in the figure below corresponds to the first byte of the packet received.

Figure 25: Using Tag Mask to Include/Exclude Bytes in Mask Tag



The following figure shows an example of how the bits in the `PIP_TAG_INC`*r* registers correspond to bytes of packet data. These registers are used in the tag mask algorithm. The registers are not used for the tag tuple algorithm. In this example, only the first of the four tag masks (registers 0-15) are shown.

Figure 26: Tag Mask Register Bits Correspondence to Packet Data Bytes



Some of these options are easily tunable via Simple Executive configuration variables; other configuration variables require knowledge of the register fields.

When using Simple Executive, relevant fields are set using the following configuration variables, which are defined in `executive-config.h`:

```
// if 1, include PIP/IPD port
CVMX_HELPER_INPUT_TAG_INPUT_PORT

IPv4:
// if 1, include source IP address
CVMX_HELPER_INPUT_TAG_IPV4_SRC_IP
// if 1, include destination IP address
CVMX_HELPER_INPUT_TAG_IPV4_DST_IP
// if 1, include TCP/UDP source port
CVMX_HELPER_INPUT_TAG_IPV4_SRC_PORT
// if 1, include TCP/UDP destination port
CVMX_HELPER_INPUT_TAG_IPV4_DST_PORT
// if 1, include protocol value
CVMX_HELPER_INPUT_TAG_IPV4_PROTOCOL

IPv6:
// if 1, include source IP address
CVMX_HELPER_INPUT_TAG_IPV6_SRC_IP
// if 1, include destination IP address
CVMX_HELPER_INPUT_TAG_IPV6_DST_IP
// if 1, include TCP/UDP source port
CVMX_HELPER_INPUT_TAG_IPV6_SRC_PORT
// if 1, include TCP/UDP destination port
CVMX_HELPER_INPUT_TAG_IPV6_DST_PORT
// if 1, include next_header value
CVMX_HELPER_INPUT_TAG_IPV6_NEXT_HEADER
```

These fields are used in the internal-use function `__cvmx_helper_port_setup_ipd()`, which is called by `cvmx_helper_initialize_packet_io_global()`.

Figure 27: Tag Value Flow Chart

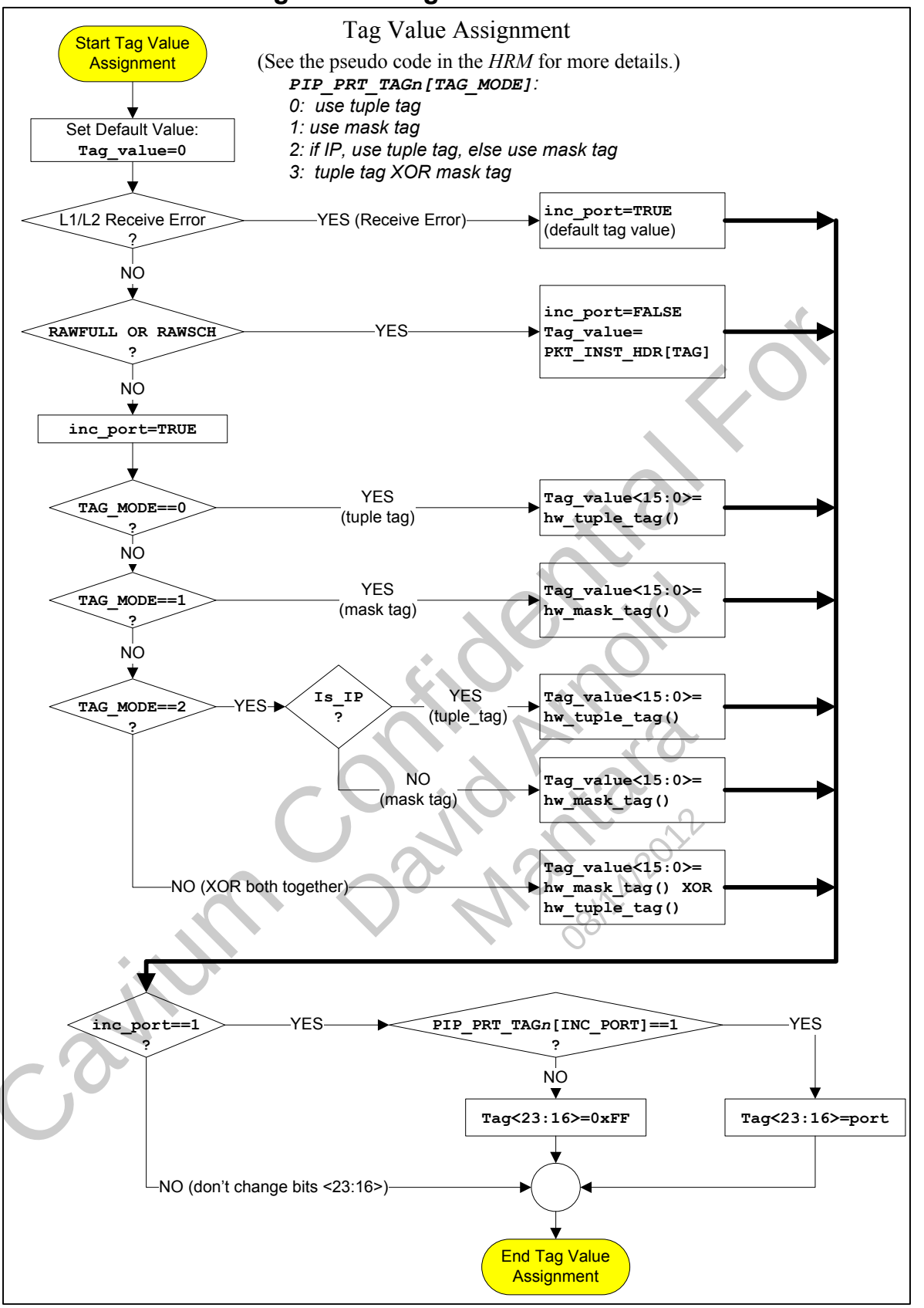


Figure 28: Flowchart for `hw_tuple_tag()` Function

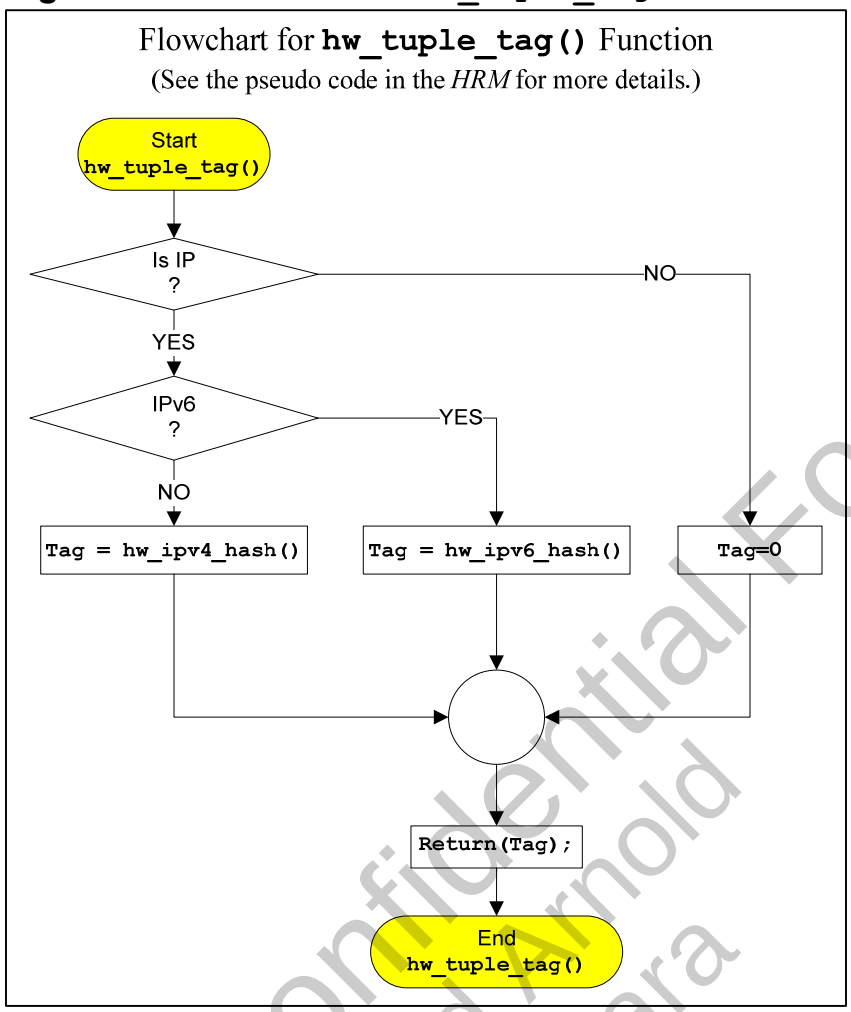


Figure 29: Flowchart for `hw_ipv4_hash()` Function

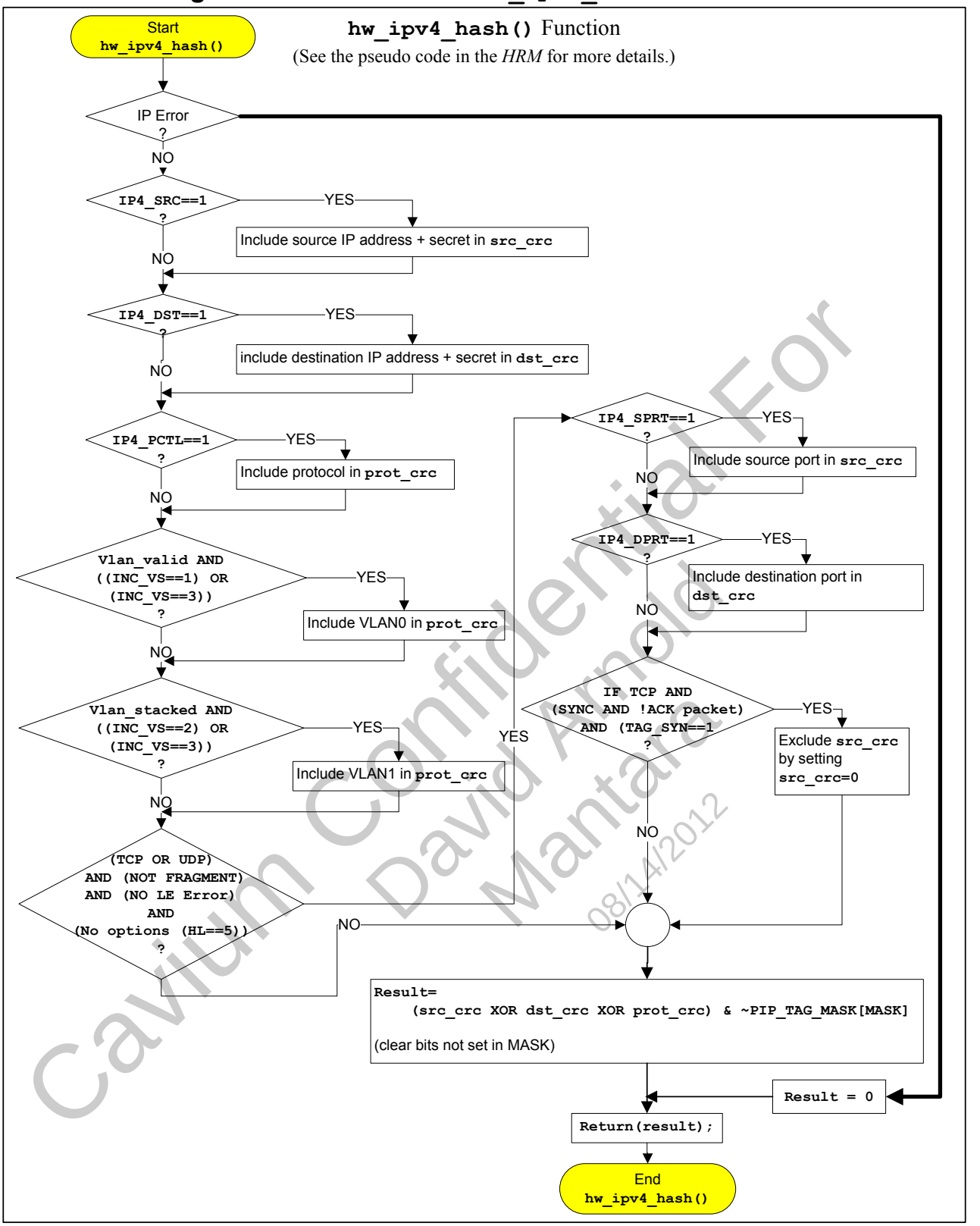


Figure 30: Flowchart for `hw_ipv6_hash()` Function

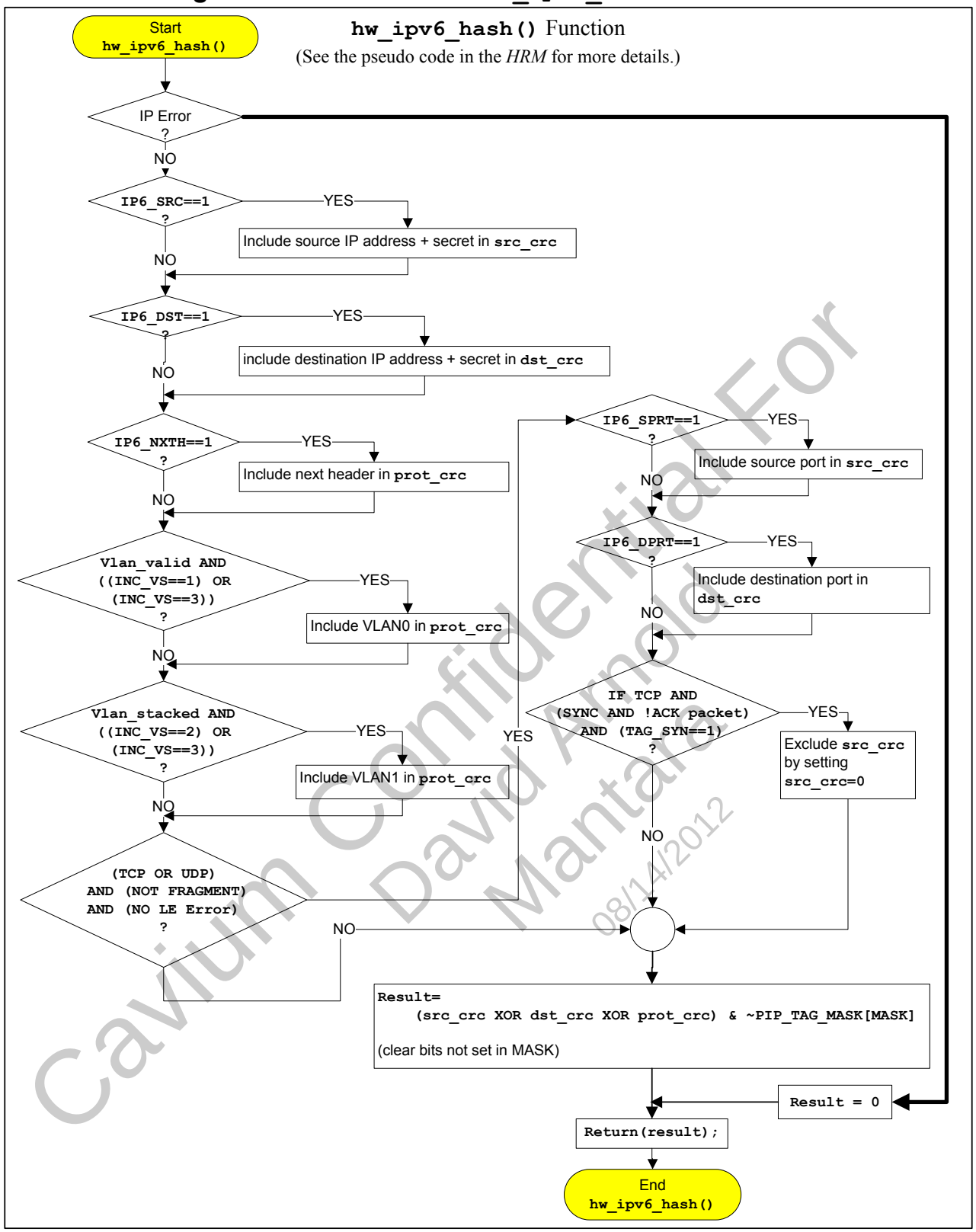
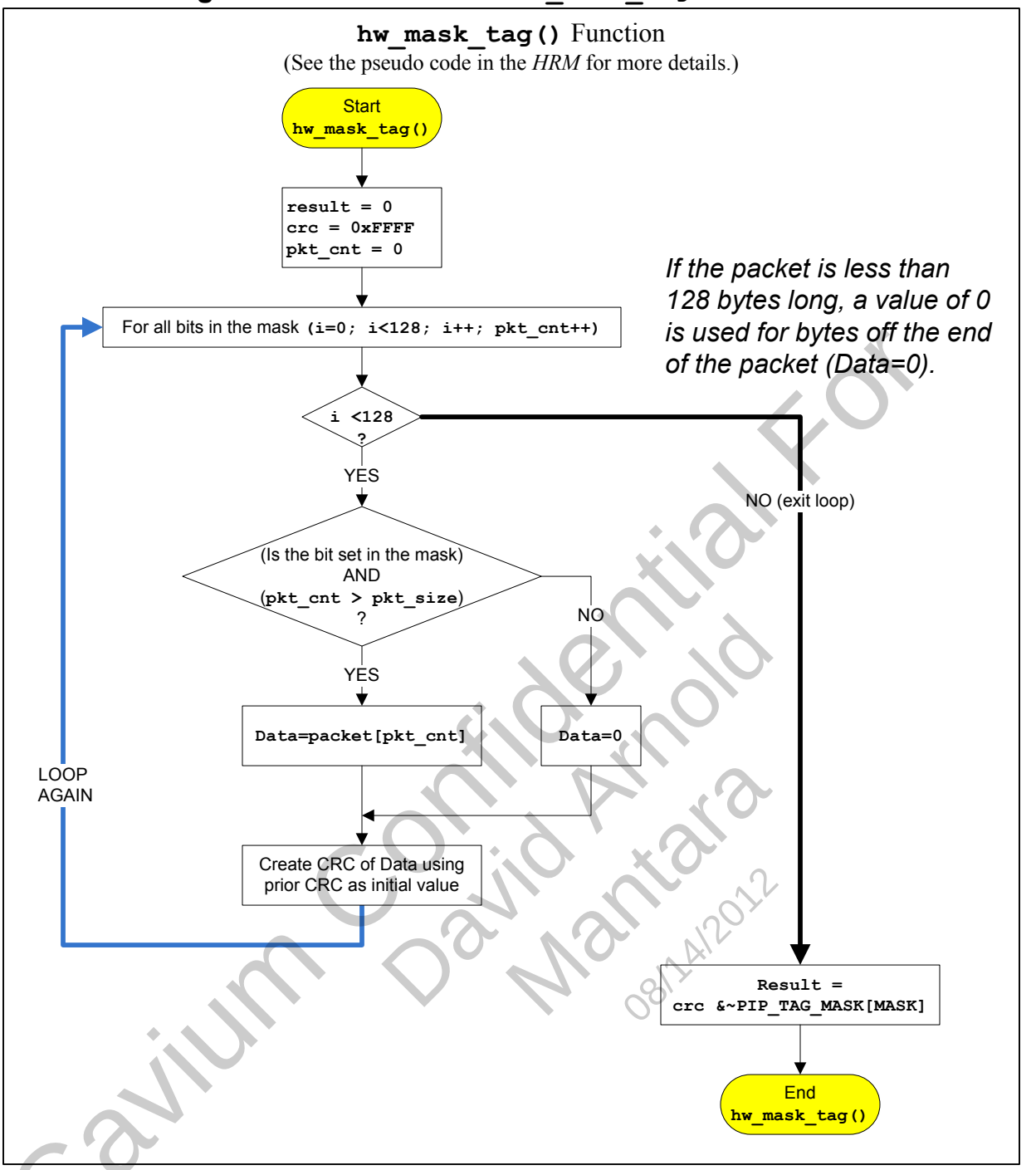


Figure 31: Flowchart for `hw_mask_tag()` Function



7.4.1 Registers to Configure Tag Value Assignment

Table 23: Registers to Configure WQE WORD1 Tag Value Assignment

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Select Tag Mode				
Tag Algorithm: One register per port. Specifies the tag value algorithm to use: 0=create tag value using the tuple tag algorithm 1=create tag value using the mask tag algorithm 2=create tag value using tuple tag if IP, else use mask tag 3=create tag value using tuple tag XOR mask tag	PIP_PRT_TAG*n* (one per port)	TAG_MODE	0	0 (H/W Default) (See Note1)
Select Global Tag MASK				
Mask: Global Register. This mask applies to all tag modes. It is a mask for the lower 16 bits of computed tag. (result & ~MASK)	PIP_TAG_MASK	MASK	0	0 (H/W Default)
Select Whether to Include PIP/IPD Port in Tag Value				
Include PIP/IPD Port: One register per port. Include the PIP/IPD port in tag value	PIP_PRT_TAG*n* (one per port)	INC_PRT	0	1 (See Note3)
Values Used with Mask Tag Option				
Which Register: One register per port. Specify which of the 64 PIP_TAG_INC registers to use when calculating mask tag hash (four 16-entry masks are used to cover 128 bytes). Note that the mask is always applied to the *first* 128 bytes of the packet, without skiping any bytes. 0=use registers 0-15; 1=use registers 16-31; 2=use registers 32-47; 3=use registers 48-63	PIP_PRT_CFG*n* (one per port)	TAG_INC	0	0 (See Note2)
Include Bytes: Each EN field is 8 bits. Each bit represents a byte. The 64 registers can be used to create four different masks used if TAG_MODE is 1 (create mask tag). The PIP_PRT_CFG[TAG_INC] field specifies which of the four masks to use. For example, registers 0-15 are used to create a (8 * 16) = 128 bit mask. Bit <7> corresponds to the MSB and bit <0> corresponds to the LSB of the corresponding 8-byte word.	PIP_TAG_INC(0-63) (Grouped into four 128-bit masks)	EN	0	0 (See Note2)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Values Used with Tuple Tag Option				
Dest Secret: Global register. Secret initial value for *destination* tuple tag CRC calculation. This provides a mechanism for each OCTEON processor to be unique.	PIP_TAG_SECRET	DST	0	0 (H/W Default)
Src Secret: Global register. Secret initial value for *source* tuple tag CRC calculation. This provides a mechanism for each OCTEON processor to be unique.	PIP_TAG_SECRET	SRC	0	0 (H/W Default)
Omit Src CRC: Global register. Do not include src_crc for TCP SYN&!ACK packets (dst_crc is always included): 0=include src_crc 1=do not include src_crc	PIP_GBL_CFG	TAG_SYN	0	0 (H/W Default)
VLAN STACKING - Include VLAN ID: One register per port. Specifies the VLAN ID to be include in the tag value when VLAN stacking: 0=do not include VID 1=include VID (VLAN0) in hash 2=include VID (VLAN1) in hash 3=include VID {VLAN0, VLAN1} in hash	PIP_PRT_TAG*n* (one per port)	INC_VS	0	0 (H/W Default)
VLAN and NOT VLAN STACKING - Include VLAN ID: One register per port. Include VLAN ID in tag value when *not* VLAN stacked: 0=do not include VID in hash 1=include VID in hash	PIP_PRT_TAG*n* (one per port)	INC_VLAN	0	0 (H/W Default)
IPv4				
IPV4 Dst Port: One register per port. Include TCP/UDP dst port in tag value	PIP_PRT_TAG*n* (one per port)	IP4_DPRT	0	0 (See Note4)
IPv4 Src Port: One register per port. Include TCP/UDP src port in tag value	PIP_PRT_TAG*n* (one per port)	IP4_SPRT	0	0 (See Note5)
IPv4 Protocol: One register per port. Include protocol in tag value	PIP_PRT_TAG*n* (one per port)	IP4_PCTL	0	0 (See Note6)
IPv4 Dst Addr: One register per port. Include dst address in tag value	PIP_PRT_TAG*n* (one per port)	IP4_DST	0	0 (See Note7)
IPv4 Src Addr: One register per port. Include src address in tag value	PIP_PRT_TAG*n* (one per port)	IP4_SRC	0	0 (See Note8)
IPv6				
IPV6 Dst Port: One register per port. Include TCP/UDP dst port in tag value	PIP_PRT_TAG*n* (one per port)	IP6_DPRT	0	0 (See Note9)
IPv6 Src Port: One register per port. Include TCP/UDP src port in tag value	PIP_PRT_TAG*n* (one per port)	IP6_SPRT	0	0 (See Note10)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
IPv6 Next_Header: One register per port. Include next_header in tag value	PIP_PRT_TAG*n* (one per port)	IP6_NXTH	0	0 (See Note11)
IPv6 Dst Addr: One register per port. Include dst address in tag value	PIP_PRT_TAG*n* (one per port)	IP6_DST	0	0 (See Note12)
IPv6 Src Addr: One register per port. Incude src address in tag vaue	PIP_PRT_TAG*n* (one per port)	IP6_SRC	0	0 (See Note13)
Notes				
Note1: Available defines are: CVMX_TAG_MODE_TUPLE, CVMX_TAG_MODE_MASK, CVMX_TAG_MODE_IP_OR_MASK, CVMX_TAG_MODE_XOR. The default SDK does not change the H/W default.				
Note2: This field is cleared and set via the functions cvmx_pip_tag_mask_clear() and cvmx_pip_tag_mask_set().				
For the following notes, when using Simple Executive, relevant fields are set using the following configuration variables, which are defined in executive-config.h. The user may change the definition to alter the default configuration.				
Note3: CVMX_HELPER_INPUT_TAG_INPUT_PORT				
Note4: CVMX_HELPER_INPUT_TAG_IPV4_DST_PORT				
Note5: CVMX_HELPER_INPUT_TAG_IPV4_SRC_PORT				
Note6: CVMX_HELPER_INPUT_TAG_IPV4_PROTOCOL				
Note7: CVMX_HELPER_INPUT_TAG_IPV4_DST_IP				
Note8: CVMX_HELPER_INPUT_TAG_IPV4_SRC_IP				
Note9: CVMX_HELPER_INPUT_TAG_IPV6_DST_PORT				
Note10: CVMX_HELPER_INPUT_TAG_IPV6_SRC_PORT				
Note11: CVMX_HELPER_INPUT_TAG_IPV6_NEXT_HEADER				
Note12: CVMX_HELPER_INPUT_TAG_IPV6_DST_IP				
Note13: CVMX_HELPER_INPUT_TAG_IPV6_SRC_IP				

7.5 Using Watchers to Set QoS and Group

Depending on the processor model, there are 4 or 8 global watchers. These watchers can set the packet's QoS value or Group value, or both. The watchers are enabled on a per-port basis (for each port, one bit enables the QoS setting, the other bit enables the Group setting).

If the watcher's QoS bit is enabled for the port and the watcher's configuration matches the packet configuration, then the packet's QoS is set to the watcher's QoS.

If the watcher's Group bit is enabled for the port and the watcher's configuration matches the packet configuration, then the packet's Group is set to the watcher's Group.

Table 24: Registers to Configure Watchers

Description	Register	Field	H/W Default Value	SDK Default Value
Watcher Configuration				
Type of Packet To Match: One register per Watcher. Watcher will match this type of incoming packets: 0=disable across all ports 1=match `protocol` (IPv4) or `next_header` (IPv6) 2=match TCP destination port 3=match UDP destination port 4=match Ethertype field 5-7=reserved	`PIP_QOS_WATCH(0-7)` (one per watcher)	`TYPE`	0	0 (H/W Default)
Value to Match: One register per watcher. Value to watch for	`PIP_QOS_WATCH(0-7)` (one per watcher)	`MATCH`	0	0 (H/W Default)
Match Mask: One register per watcher. Mask a range of values (16 bits: set the bit to mask (`match & ~mask`)	`PIP_QOS_WATCH(0-7)` (one per watcher)	`MASK`	0	0 (H/W Default)
Group Value to Set If Match Group Watcher: One register per watcher. Group number of watcher (set group of matched packet to this value)	`PIP_QOS_WATCH(0-7)` (one per watcher)	`GRP`	0	0 (H/W Default)
QoS Value to Set if Match QoS Watcher: One register per watcher. QoS value of watcher (set QoS of matched packet to this value)	`PIP_QOS_WATCH(0-7)` (one per watcher)	`WATCHER`	0	0 (H/W Default)
Per Port: Enable Watcher to Set Matched Packet's QoS Value				
Enable QoS Watchers 0-3: One register per port. Enable QoS for watchers 0-3. (An enable bit is provided for each watcher. Set bit to 1 to enable the watcher.)	`PIP_PRT_CFGn` (one per port)	`QOS_WAT`	0	0 (H/W Default)
Enable QoS Watchers 4-7: One register per port. Enable QoS for watchers 4-7. (An enable bit is provided for each watcher. Set bit to 1 to enable the watcher.)	`PIP_PRT_CFGn` (one per port)	`QOS_WAT_47`	0	0 (H/W Default)
Per Port: Enable Watcher to Set Matched Packet's Group Value				
Enable Group Watchers 0-3: One register per port. Enable group for watchers 0-3. (An enable bit is provided for each watcher. Set bit to 1 to enable the watcher.)	`PIP_PRT_CFGn` (one per port)	`GRP_WAT`	0	0 (H/W Default)
Enable Group Watchers 4-7: One register per port. Enable group for watchers 4-7. (An enable bit is provided for each watcher. Set bit to 1 to enable the watcher.)	`PIP_PRT_CFGn` (one per port)	`GRP_WAT_47`	0	0 (H/W Default)
Note				
As of SDK 2.0, the SDK does not provide a function to set these values.				

Note: Some OCTEON models do not support an Ethertype watcher. ARP packets must be handled at a high priority, but watchers cannot uniquely classify ARP packets because they are not

IP. To handle ARP packets at a high priority when the Ethertype option is not available on the OCTEON model, set all non-IP packets to a higher QoS.

The following examples show how to use watchers.

Example 1:

Separate IPv4 traffic from IPv6 traffic, and cause IPv4 traffic to go to Group 2 while IPv6 traffic goes to Group 5, configure the watchers as follows:

Watcher 0 will watch for IPv4 traffic:

```
PIP_QOS_WATCH0[TYPE] = 0x4;      // Match EtherType field
PIP_QOS_WATCH0[MATCH] = 0x8000; // IPv4
PIP_QOS_WATCH0[MASK] = 0;        // no masking
PIP_QOS_WATCH0[GRP] = 2;
```

Watcher 1 will watch for IPv6 traffic:

```
PIP_QOS_WATCH1[TYPE] = 0x4;      // Match EtherType field
PIP_QOS_WATCH1[MATCH] = 0x86DD; // IPv6
PIP_QOS_WATCH1[MASK] = 0;        // no masking
PIP_QOS_WATCH1[GRP]  = 5
```

Then, for every port which should be watched, enable the bits corresponding to the watchers to be enabled (watcher 0 and 1 in this example):

```
PIP_PRT_CFGn[GRP_WAT] = 0x3
```

Example 2:

Use watcher 1 to match all values in the range 128-255, and set group to 4:

```
PIP_QOS_WATCH1[TYPE] = 0x2  // Match TCP dest field
PIP_QOS_WATCH1[MATCH] = 128 // 1 0000 0000b
PIP_QOS_WATCH1[MASK] = 127  //   1111 1111b
PIP_QOS_WATCH1[GRP]  = 4
```

(255 is the maximum value for the 16-bit match field.)

Note: When using Simple Executive to read and write CSRs, "`PIP_QOS_WATCH1[GRP] = 4`" translates to:

```
cvmx_pip_qos_watchx_t watcher;
watcher.u64 = cvmx_read_csr(CVMX_PIP_QOS_WATCHX(1));
watcher.s.grp = 4;
cvmx_write_csr(CVMX_PIP_QOS_WATCHX(1), watcher.u64);
```

8 Security

The `WORD2[SE]` (`dec_ipsec`) field is set when the packet is IP and may require IPsec decryption. This bit is set when:

- The packet is IPsec ESP (i.e. the IPv4 `protocol` or the initial IPv6 `next header` equals 50).
- The packet is IPsec AH (i.e. the IPv4 `protocol` or the initial IPv6 `next header` equals 51).
- The packet is TCP (i.e. the IPv4 `protocol` or the initial IPv6 `next header` equals 6) and the packet's TCP destination port matches one of four possible programmed values and `(WORD2[V6] || (IPv4.HL==5))`.

- The packet is UDP (i.e. the IPv4 `protocol` or the initial IPv6 `next header` equals 17) and the packet's UDP destination port matches one of four possible programmed values and `(WORD2[V6] || (IPv4.HL==5))`.

There are four programmable destination ports set via the `PIP_DEC_IPSECn` registers, shared by TCP and UDP. Each programmed port can match TCP and/or UDP.

Table 25: Registers to Configure IP Security

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Dest Port to Match On: UDP or TCP destination port to match on.	PIP_DEC_IPSEC(0-3) (Four destination port values)	DPRT	0	0 (H/W Default)
Dest Port for TCP Packets: This DPRT should be used for TCP packets.	PIP_DEC_IPSEC(0-3) (Four destination port values)	TCP	0	0 (H/W Default)
Dest Port for UDP Packets: This DPRT should be used for UDP packets.	PIP_DEC_IPSEC(0-3) (Four destination port values)	UDP	0	0 (H/W Default)

9 Error Check Configuration

The following registers control error-check configuration. When errors occur, they are reported in the opcode field in the WQE data structure. The meaning of the opcode field depends on which error bit is set in the WQE (`RE`, `LE`, `IE`). There are also registers to enable exception/error interrupts. These interrupts are seldom used because the information is already provided in the packet's WQE. In addition, when the interrupt occurs, there is no way to tell which packet caused the interrupt.

Table 26: Registers To Configure Error Checking

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Packet Length Checks				
Byte Count for max-sized frame check: Two registers (see Note1). Failing packets set the `MAXERR` interrupt and are optionally sent with WQE `WORD2[opcode]==MAXERR` (see `MAXERR_EN` field). The effective `MAXLEN` used by the hardware is `PIP_FRM_LEN_CHK[MAXLEN] + (4 x VV) + (4 x VS)` where (`VV==1` if VLAN or VLAN STACKING) and (`VS==1` if VLAN STACKING)	PIP_FRM_LEN_CHK(0-1) (PIP_FRM_LEN_CHK0 is used for packets on packet interface0, PCIe (except RAW packets), and loopback ports; PIP_FRM_LEN_CHK1 is use for packets on packet interface1 ports and PCIe RAW packets)	MAXLEN	0x600	0x600 (H/W Default) (See Note1)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Byte count for min-sized frame check: Two registers (see Note1). Failing packets set the MINERR interrupt and are optionally sent with WQE WORD2[opcode]=MINERR (see MINERR_EN field).	PIP_FRM_LEN_CHK(0-1) (PIP_FRM_LEN_CHK0 is used for packets on packet interface0, PCIe (except RAW packets), and loopback ports; PIP_FRM_LEN_CHK1 is use for packets on packet interface1 ports and PCIe RAW packets)	MINLEN	0x40	0x40 (H/W Default) (See Note1)
Select largest L2 frame size: The value of the type/length field can be considered either a type or a length. Values under the cutoff are considered to be length. For example, if MAX_L2==0, packets with a type/length value of > 1500 are considered to specify type, not length. 0=1500 / 0x5dc 1=1535 / 0x5ff	PIP_GBL_CFG	MAX_L2	0 (1500)	0 (1500) (H/W Default)
L2 length error check enable: One register per port. Frame was received with length error. This check is typically not enabled for incoming packets on PCIe ports.	PIP_PRT_CFG*n* (one per port)	LENERR_EN	0	0 (H/W Default)
Max frame error check enable: One register per port. Frame was received with length > max_length	PIP_PRT_CFG*n* (one per port)	MAXERR_EN	0	0 (H/W Default) (See Note2)
Min frame error check enable: One register per port. Frame was received with length < min_length. This check is typically not enabled for incoming packets on PCIe ports.	PIP_PRT_CFG*n* (one per port)	MINERR_EN	0	0 (H/W Default) (See Note2)
Disable length check for packets with padding in client data: One register per port. (set to 1 to disable)	PIP_PRT_CFG*n* (one per port)	PAD_LEN	0	0 (H/W Default)
Disable length check for VLAN packets: One register per port. (set to 1 to disable)	PIP_PRT_CFG*n* (one per port)	VLAN_LEN	0	0 (H/W Default)
Other Error Checks (alphabetical order)				
Configure IPv6/UDP checksum: 0=allow optional checksum code 1=do not allow optional checksum code (See the *HRM* for details.)	PIP_GBL_CFG	IP6_UDP	1	1 (H/W Default)
Enable IPv4 header checksum check: Set to 1 to enable. Indicates that an IPv4 packet contained IPv4 header checksum violations. Only applies to packets classified as IPv4.	PIP_GBL_CTL	IP_CHK	1	1 (H/W Default) (See Note3)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Enable TTL (IPv4) / hop (IPv6) check: Set to 1 to enable. Indicates that the IPv4 TTL field or IPv6 HOP field is zero.	PIP_GBL_CTL	IP_HOP	1	1 (H/W Default) (See Note3)
Enable IP malformed check: Set to 1 to enable. Indicates that the packet was malformed. Malformed packets are defined as packets that are not long enough to cover the IP header or not long enough to cover the length in the IP header.	PIP_GBL_CTL	IP_MAL	1	1 (H/W Default) (See Note3)
Enable IPv4 options check: Set to 1 to enable. Indicates the presence of IPv4 options. It is set when the length != 5. This only applies to packets classified as IPv4.	PIP_GBL_CTL	IP4_OPTS	1	1 (H/W Default) (See Note3)
Enable IPv6 early extension headers: Set to 1 to enable. Indicate the presence of IPv6 early extension headers. These bits only apply to packets classified as IPv6. Bit 0 will flag early extensions when next_header is any one of: * hop-by-hop (0) * destination (60) * routing (43) Bit 1 will flag early extentions when next_header is NOT any of: * TCP (6) * UDP (17) * fragmentation (44) * ICMP (58) * IPSEC ESP (50) * IPSEC AH (51) * IPCOMP	PIP_GBL_CTL	IP6_EEXT	1	1 (H/W Default) (See Note3)
Enable L2 malformed check: Set to 1 to enable.	PIP_GBL_CTL	L2_MAL	1	1 (H/W Default) (See Note3)
Enable TCP/UDP checksum check: Set to 1 to enable. Indicates that a packet classified as either TCP or UDP contains an L4 checksum failure.	PIP_GBL_CTL	L4_CHK	1	1 (H/W Default)
Enable TCP/UDP length check: Set to 1 to enable. Indicates that the TCP or UDP length does not match the the IP length.	PIP_GBL_CTL	L4_LEN	1	1 (H/W Default) (See Note3)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Enable TCP/UDP malformed packet check: Set to 1 to enable. Indicates that a TCP or UDP packet is not long enough to cover the TCP or UDP header.	PIP_GBL_CTL	L4_MAL	1	1 (H/W Default) (See Note3)
Enable TCP/UDP illegal port check: Set to 1 to enable. Indicates that a TCP or UDP packet has an illegal port number: either the source or destination port is zero.	PIP_GBL_CTL	L4_PRT	1	1 (H/W Default) (See Note3)
Enable TCP flags check: Set to 1 to enable. Indicates any of the following conditions [URG, ACK, PSH, RST, SYN, FIN] : tcp_flag * 6'b000001: (FIN only) * 6'b000000: (0) * 6'bxxx1x1: (RST+FIN+*) * 6'b1xxx1x: (URG+SYN+*) * 6'bxxx11x: (RST+SYN+*) * 6'bxxxx11: (SYN+FIN+*)	PIP_GBL_CTL	TCP_FLAG	1	1 (H/W Default) (See Note3)
Notes				
Note1: PIP_FRM_LEN_CHK0 is used for packets on packet interface0, PCIe, and PKO loopback ports. PIP_FRM_LEN_CHK1 is used for packet on packet interface1 ports and PCIe RAW packets.				
Note2: The SDK sets this value to 0 in the internal-use function __cvmx_helper_npi_enable() (an internal PCIe block)				
Note3: By disabling the checker, the exception will not be flagged and the packet will be parsed as best it can. Note, by disabling conditions, packets can be parsed incorrectly (.i.e. IP_MAL and L4_MAL could cause bits to be seen in the wrong place. IP_CHK and L4_CHK mean that the packet was corrupted).				

9.1 CRC Check Configuration

Some processors have CRC Check configuration registers in the PIP/IPD register set (for example, CN58XX).

When using CRC, the term *reflect* means to flip the bits (mirror image), so that bit 0 becomes bit 31, and bit 31 becomes bit 0. The sequence 11000100 becomes 00100011.

The term *invert* means to change all zeros to ones and all ones to zeroes. The sequence 11000100 becomes 00111011.

Table 27: Registers Used to Configure CRC Check

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
CRC Calculation: Datapath Reflection Control				
Invert: If set, invert the result.	**Ports 0-15:** PIP_CRC_CTL0 (one bit for all ports) **Ports 16-31:** PIP_CRC_CTL1 (one bit for all ports)	INVRES	1	1 (H/W Default) (See Note1)
Reflect: Reflect the bits in each byte. The byte order does not change: 0=calculate CRC MSB-to-LSB 1=calculate CRC LSB-to-MSB	**Ports 0-15:** PIP_CRC_CTL0 (one bit for all ports) **Ports 16-31:** PIP_CRC_CTL1 (one bit for all ports)	REFLECT	1	1 (H/W Default) (See Note1)
CRC Calculation: Initial Value				
Initial Value: Set initial value (IV) used by the CRC algorithm. The default is FCS32.	**Ports 0-15:** PIP_CRC_IV0 (one bit per port) **Ports 16-31:** PIP_CRC_IV1 (one bit per port)	IV	0x46AF6449	0x46AF6449 (H/W Default) (See Note1)
Notes				
Note1: These registers can be configured via the SDK function cvmx_pip_config_crc().				

10 Packet Storage

This section covers:

1. What part of the packet is stored in Packet Data Buffer(s) and the WQE Buffer
2. Choices for writing Packet Data Buffers to L2/DRAM
3. Packet Storage in Packet Data Buffers, including optional storage of the WQE in the Packet Data buffer
4. Packet Storage in the Work Queue Entry data structure, including dynamic shorts
5. Accessing packet data when some packets are dynamic shorts and some are not
6. Registers used to configure packet storage options

Usually, PIP/IPD writes the entire packet into Packet Data Buffers, and also writes the first 96 bytes (92 bytes if IP and not IPv6 because 4 bytes are used for alignment) of the packet to the Work Queue Entry data structure (WORD4-WORD15).

PIP/IPD does not write the packet to Packet Data Buffers if:

1. All the bytes of the packet will fit into WQE WORD4-WORD15 and the dynamic short option is enabled for the port. This packet is referred to as a *dynamic short* packet. (Dynamic shorts are discussed in more detail in Section 10.4.2 –"Dynamic Short Storage in WQE".)
2. The packet is dropped due to Per-Port Packet Drop or Per-QoS RED. In this case, there is also no WQE. (This is discussed in more detail in Section 12 – "Congestion Control (Backpressure, Packet Drop, RED, WRED)".)
3. The packet is dropped due to receive buffer overflow before reaching PIP/IPD. (A partial packet may be received before the receive buffer overflows, resulting in partial packets. In this case `WORD2[RE]==1` and `WORD2[OPCODE]==1` (partial error).) A WQE exists even if there was a partial receive.

PIP/IPD can support a maximum of 255 buffers for a packet and a maximum packet size of 65535 bytes. The maximum Packet Data Buffer size is 16384 bytes (2048 8-byte words).

10.1 The Part of the Received Data Which is Stored

The packet (packet data) received by PIP/IPD usually starts after the Start Frame Delimiter (SFD) (the preamble is typically excluded), and continues to the CRC. The CRC is optionally not stored (available only for ports (0-31, 36-39): the CRC cannot be removed from packets arriving on PCI/PCIe/DPI ports or sRIO ports.

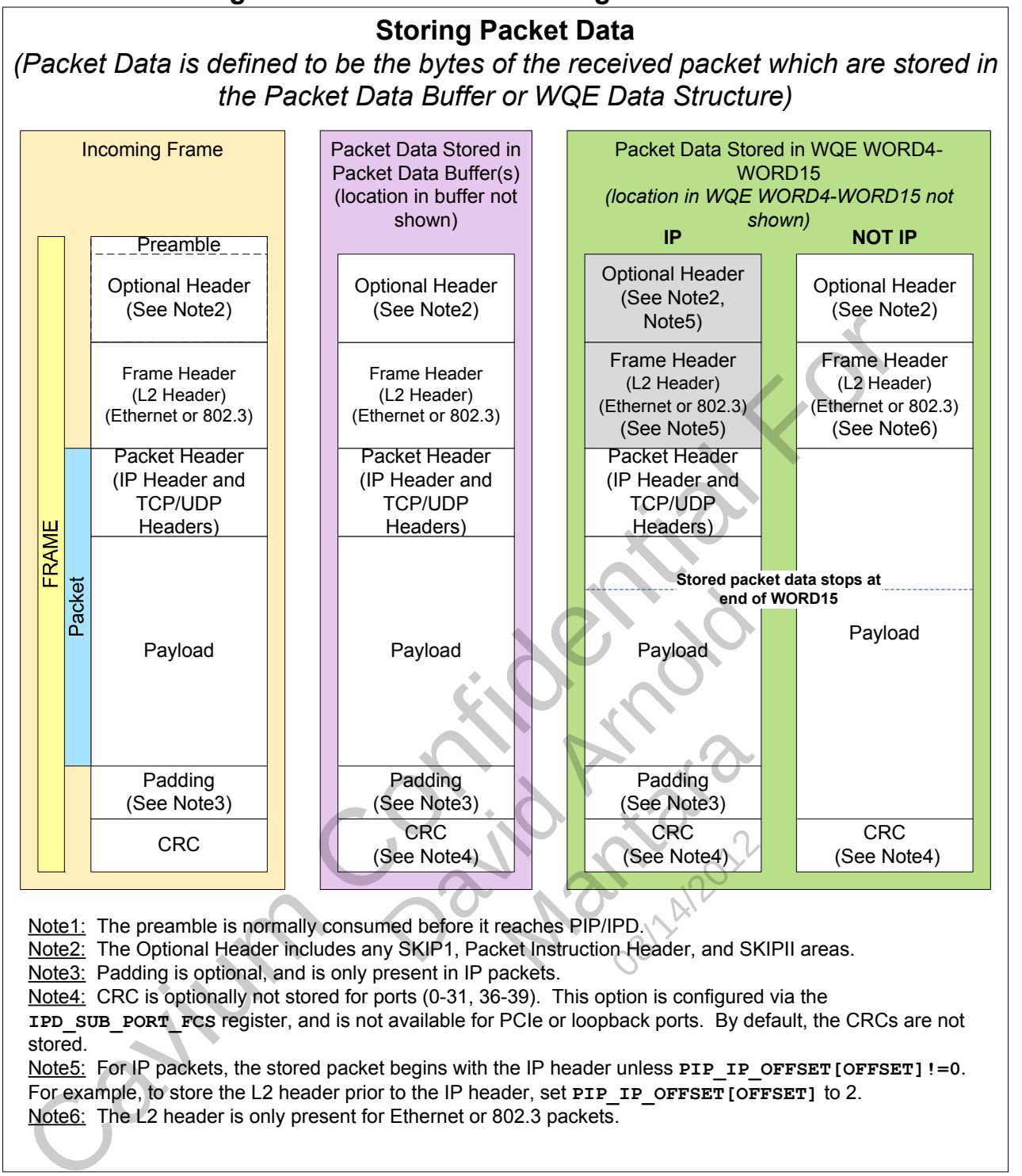
Packet Data Stored in Packet Data Buffers:

- All received bytes after the SFD to the end of the frame, optionally including the CRC

Packet Data Stored in Work Queue Entry WORD4-WORD15 (See Figure 36 – "Format of Packet Data Stored in WQE WORD4-WORD15" for an illustration.). In all cases, the CRC at the end of the frame is optionally not stored.:

- If IP and `PIP_IP_OFFSET[OFFSET]==0`:
  - Stored packet data starts at IP Header and continues until the end of the packet data or the end of WORD15.
- If IP and `PIP_IP_OFFSET[OFFSET]!=0`:
  - `PIP_IP_OFFSET[OFFSET]` specifies the number of 8-byte words to reserve in the WORD4-WORD15 portion of the WQE for packet data which is immediately prior to the IP header.
    - If `PIP_IP_OFFSET[OFFSET]` is large enough to accommodate all of the packet data preceding the IP header, including byte0, then the stored packet data starts at the first byte of packet data and continues until the end of the packet data. If there is any space remaining (header is short), PIP/IPD will fill it with zeroes.
    - Otherwise, the PIP/IPD will backfill any byte before the IP header until `OFFSET x 8` bytes are used. (The alignment pad is reserved.)
- If NOT IP:
  - Stored packet data starts with the first byte after the SFD and continues until the end of the packet data or the end of WORD15.

Figure 32: Overview of Storing Received Data



Note1: The preamble is normally consumed before it reaches PIP/IPD.
Note2: The Optional Header includes any SKIP1, Packet Instruction Header, and SKIPII areas.
Note3: Padding is optional, and is only present in IP packets.
Note4: CRC is optionally not stored for ports (0-31, 36-39). This option is configured via the `IPD_SUB_PORT_FCS` register, and is not available for PCIe or loopback ports. By default, the CRCs are not stored.
Note5: For IP packets, the stored packet begins with the IP header unless `PIP_IP_OFFSET[OFFSET]!=0`. For example, to store the L2 header prior to the IP header, set `PIP_IP_OFFSET[OFFSET]` to 2.
Note6: The L2 header is only present for Ethernet or 802.3 packets.

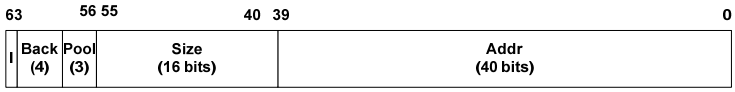
10.2 Packet Storage in Packet Data Buffers

If the packet is not stored as a dynamic short in the WQE, then it is stored in one or more packet data buffers.

The physical address of the packet data in the first packet data buffer is stored in WQE `WORD3[Addr]`. When multiple buffers are needed to store the packet data, each buffer contains the physical address of the next buffer in the buffer chain (`Next_Buf_Ptr`). The `Next_Buf_Ptr` field is the same data structure as WQE WORD3, as shown in the following figure:

Figure 33: Next Buffer Pointer (`Next_Buf_Ptr`) Data Structure

Next Buffer Pointer (`Next_Buf_Ptr`)



Note: When there is only one buffer, the last valid buffer pointer is the one in the WQE; otherwise the last valid buffer pointer is the one in the second-to-last buffer.

Note: The software data structure field name is shown in parenthesis in the list below.

`I (i)`: The Invert bit (used by PKO). This value is set to 0 for an inbound packet (don't free). This bit is used to invert the state of the PKO command **`WORD0[DF]`** flag (don't free). The Invert bit is used by software to free only selected buffers in a buffer chain.
`Back (back)`: The number of cache lines from **`Addr`** (start of packet) to the beginning of the first buffer (usually 0).
`Pool (pool)`: The pool the buffer came from (0 for packet data buffers)
`Size (size)`: The number of bytes from **`Addr`** to **`IPD_PACKET_MBUFF_SIZE[MB_SIZE]`** (not the same as the size of the buffer). This represents the number of bytes of packet data in the buffer, with the possible exception of the last buffer. For the last buffer, the number of bytes of packet data will be smaller than the **`Size`** field in the last valid **`Next_Buf_Ptr`** unless the packet data ends exactly at the end of the region reserved by **`IPD_PACKET_MBUFF_SIZE[MB_SIZE]`**.
`Addr (addr)`: The physical address of the start of the *packet* in the buffer (*not* the same as the start of the *buffer*). **`Addr`** does not need to be cache-line aligned, but the start of the buffer *must* be cache-line aligned.

The start of the first Packet Data Buffer = **`((Addr >> 7) - Back) << 7)`**, where
`back = IPD_1ST_NEXT_PTR_BACK[BACK]`

The start of subsequent Packet Data Buffers **`= (Next_Buf_Ptr – Back)`**, where
`back = IPD_2ND_NEXT_PTR_BACK[BACK]`

`IPD_1ST_NEXT_PTR_BACK[BACK]` and **`IPD_2ND_NEXT_PTR_BACK[BACK]`** are in units of 128-byte words and must be consistent with the mbuf **`SKIP_SZ`** settings.

Note that `Back` is used by hardware units such as the PKO to determine the start of the packet data buffer. (The PKO uses the start address in the buffer free operation.) It is critical to configure these correctly: incorrect configuration causes FPA corruption.

For example, PKO finds the start address of the buffer using the following formula:

```
Buffer Start Address = ((Addr >> 7) - Back) << 7
```

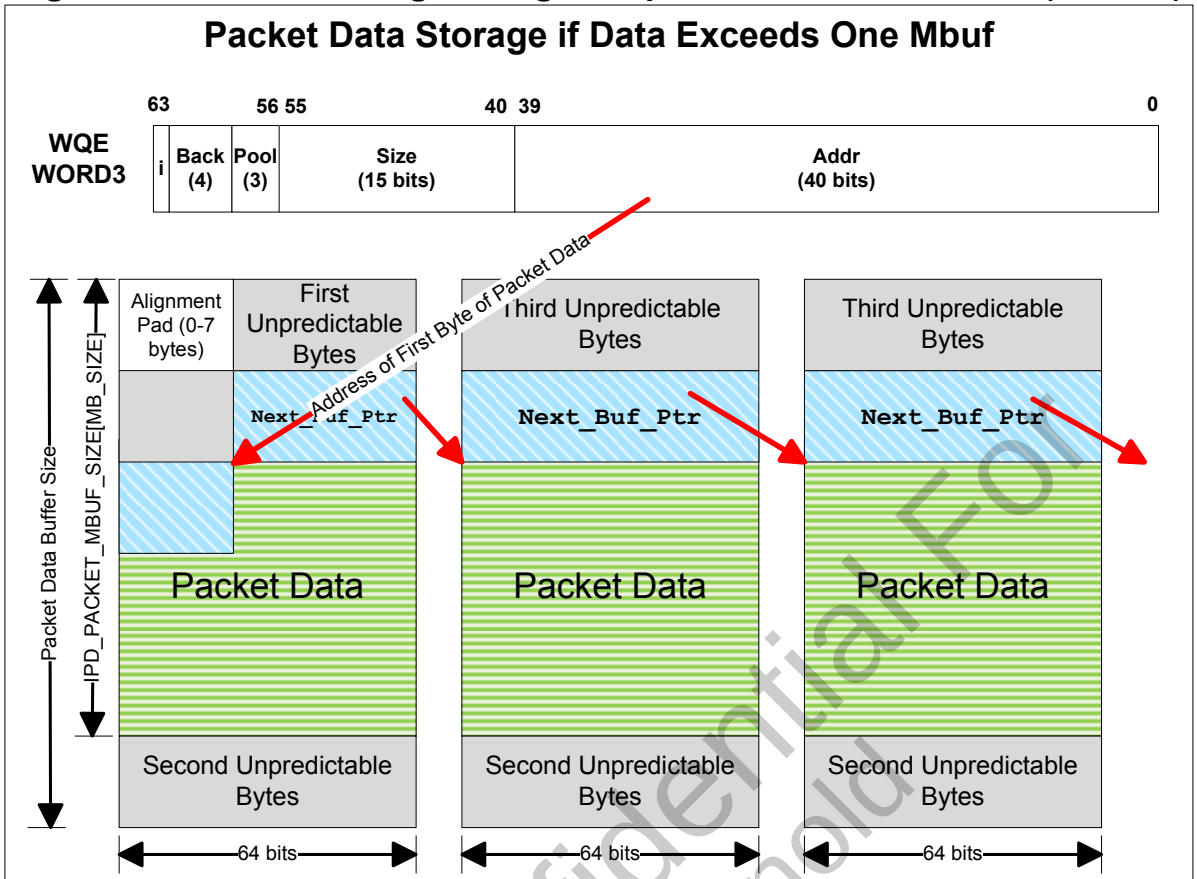
`Back` is set to the value of `IPD_1ST_NEXT_PTR_BACK[BACK]` for the first Packet Data Buffer and `IPD_2ND_NEXT_PTR_BACK[BACK]` for subsequent Packet Data Buffers. `Back` is in units of 128-byte words (cache line size). The value of these registers must be consistent with the corresponding `IPD_1ST_MBUFF_SKIP[SKIP_SZ]` and `IPD_NOT_1ST_MBUFF_SKIP[SKIP_SZ]` variables. By default, the SDK sets these variables using the `MBUFF_SKIP` sizes as shown in the following pseudo code:

```
// The +8 below is to include the Next_Buf_Ptr
first_back = CVMX_HELPER_FIRST_MBUFF_SKIP+8) / 128

// The +8 below is to include the Next_Buf_Ptr
second_back = VMX_HELPER_NOT_FIRST_MBUFF_SKIP+8) / 128
```

When the amount of data exceeds the size of one Packet Data buffer, linked buffers (mbufs) are used, as shown in the following figure.

Figure 34: Packet Storage Using Multiple Packet Data Buffers (MBUFs)



Notes:
Alignment Pad:
* For IP packets, this pad is added by PIP/IPD:
  * If IPv4, Pad = 32 bits
  * If IPv6, Pad = 0
  * If IP, but neither IPv4 or IPv6, Pad = 32 bits
* For non-IP packets, the user can configure the amount of padding. Pad must be between 0-7 bytes.:
  * If RAW, Pad = **PIP_GBL_CFG[RAW_SHF]**
  * If not IP and not RAW, Pad = **PIP_GBL_CFG[NIP_SHF]**

Unpredictable bytes are areas in the mbuf reserved for customer software. The contents is not touched by PIP/IPD (for example, not set to all zeroes), so the content is unpredictable.

First Unpredictable Bytes: The size of the first unpredictable bytes section is configured via **IPD_1ST_MBUFF_SKIP[SKIP_SZ]**. This size is specified in 8-byte words.

Second Unpredictable Bytes: The size of the second unpredictable bytes section is the size of the Packet Data Buffer minus **IPD_PACKET_MBUFF_SIZE[MB_SIZE]**. This size is specified in 8-byte words. Typically this variable is set to the size of the packet data buffer, so that the second unpredictable bytes region does not exist.

Third Unpredicable Bytes: The size of the third unpredictable bytes section is configured via **IPD_NOT_1ST_MBUFF_SKIP[SKIP_SZ]**.

Next_Buf_Ptr: This field is used to point to the start of packet data in the next mbuf in the chain, and provide other information about the buffer. The **Addr** field in the **Next_Buf_Ptr** in the last buffer is invalid.

The `Addr` field in the `Next_Buf_Ptr` in the last buffer is invalid.

Any bytes in the buffer beyond the end of the stored packet data are invalid.

Note that the "unpredictable" areas in the mbuf are not over-written with any data by PIP/IPD, and will contain "random" data. The intended use of these areas is to allow for example growth of packet header length without needing to copy the whole packet payload to a new buffer that includes the new (larger) header.

> ***Note: The PIP/IPD always writes packet data in complete (128 byte) cache blocks, including when it writes the first and last data. This is why the Packet Data Buffer must be 128-byte aligned, and the size of the Packet Data Buffer must be a multiple of cache line size. Otherwise, memory before or after the Packet Data Buffer may be corrupted. The Simple Executive configuration code automatically takes care of this. This is only a problem when not using Simple Executive to configure FPA buffers.***

10.2.1 Storing WQE in Packet Data Buffer instead of WQE Buffer

On some OCTEON models, PIP/IPD can also be configured to not use WQE buffers (`IPD_CTL_STATUS[NO_WPTR]==1`). In this case, the 128-byte WQE data structure is inserted into the area of the first Packet Data Buffer reserved by the register field `IPD_1ST_MBUFF_SKIP[SKIP_SZ]`. To reserve sufficient space for the WQE, set `IPD_1ST_MBUFF_SKIP[SKIP_SZ]` to 16. (This variable's units are in 8-byte words.)

When PIP/IPD adds the work to the SSO queues, it executes the `add_work` function with the WQE pointer set to the location of the WQE in the Packet Data buffer. See the *HRM* for more details.

10.3 Choices for Writing Packet Data Buffer(s) to L2/DRAM

Packet Data Buffers are stored in L2Cache/DRAM based on four configuration choices, as shown in the following table. Option selection is global and affects all ports.

Table 28: Packet Data Buffer Write to L2/DRAM Choices (Global Option)

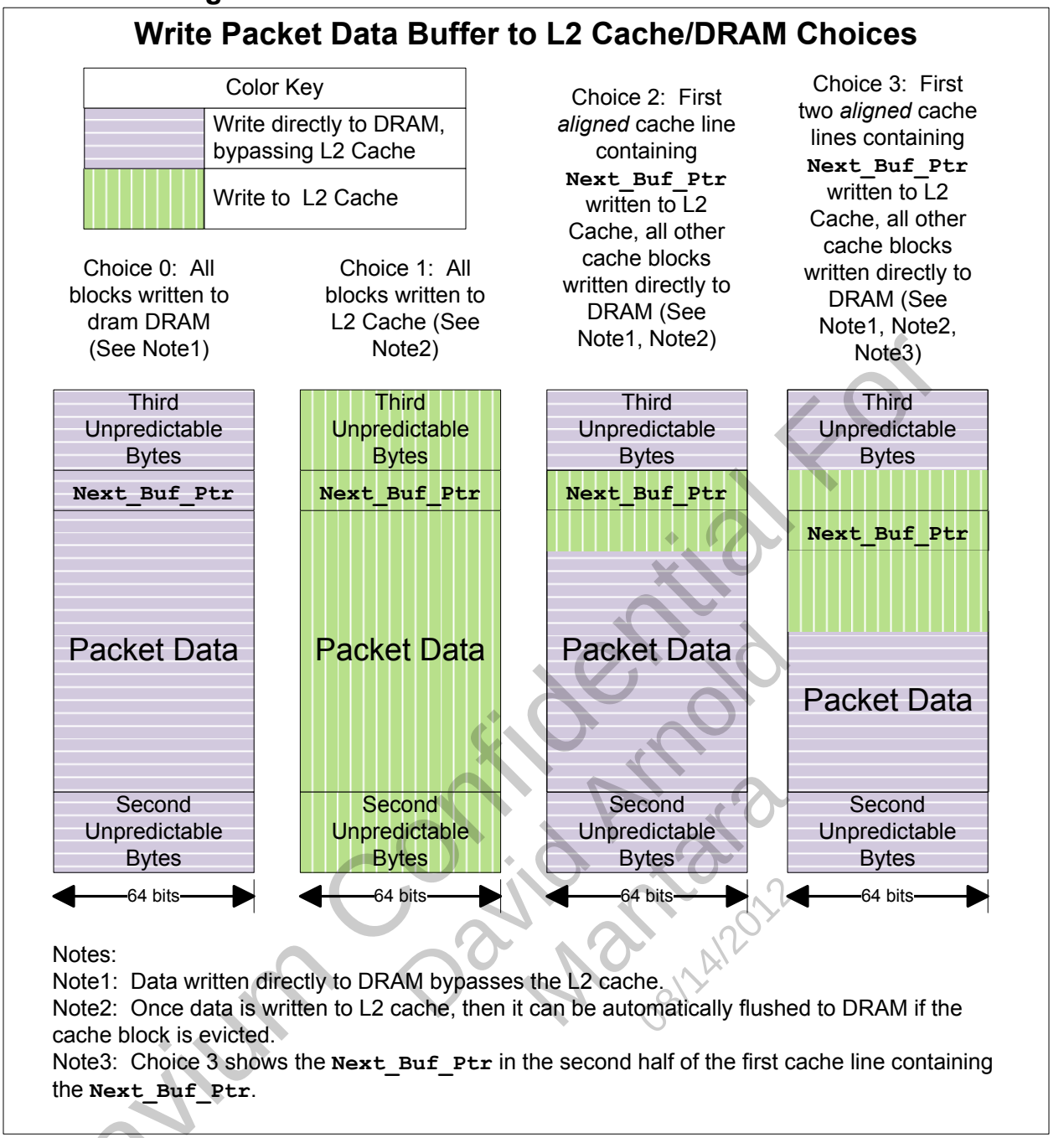
Choice	Description
0	All Packet Data Buffers are written directly to memory (DRAM), bypassing the L2 cache.
1	All Packet Data Buffers are written to L2 cache. (If evicted, cache blocks are written to memory (DRAM)).
2	The *first* aligned cache block holding the `Next_Buf_Ptr` and the packet data is written to the L2 cache. All remaining cache blocks are written directly to memory (DRAM), bypassing the L2 cache. (If evicted, cache blocks are written to memory (DRAM)).
3	The first *two* cache blocks holding the `Next_Buf_Ptr` and the packet data are written to the L2 cache. All remaining cache blocks are written directly to memory (DRAM), bypassing the L2 cache. (If evicted, cache blocks are written to memory (DRAM)).

When using the Simple Executive, these choices are defined as:

```
CVMX_IPD_OPC_MODE_STT = 0LL;       // Write all blocks DRAM, none are
                                   // cached in the L2 cache
CVMX_IPD_OPC_MODE_STF = 1LL;       // Write all blocks into L2 cache
CVMX_IPD_OPC_MODE_STF1_STT = 2LL;  // Write first cache block which
                                   // contains Next_Buf_Ptr to L2 cache,
                                   // others to DRAM
CVMX_IPD_OPC_MODE_STF2_STT = 3LL;  // Write first two cache blocks which
                                   // contain the Next_Buf_Ptr to
                                   // L2 cache, others to DRAM
```

The default Simple Executive configuration is `CVMX_IPD_OPC_MODE_STT` (choice 0: all Packet Data Buffers are written directly to DRAM). In this mode, users typically access the first 96 bytes of packet data via the WQE, and access the Packet Data Buffer(s) as needed.

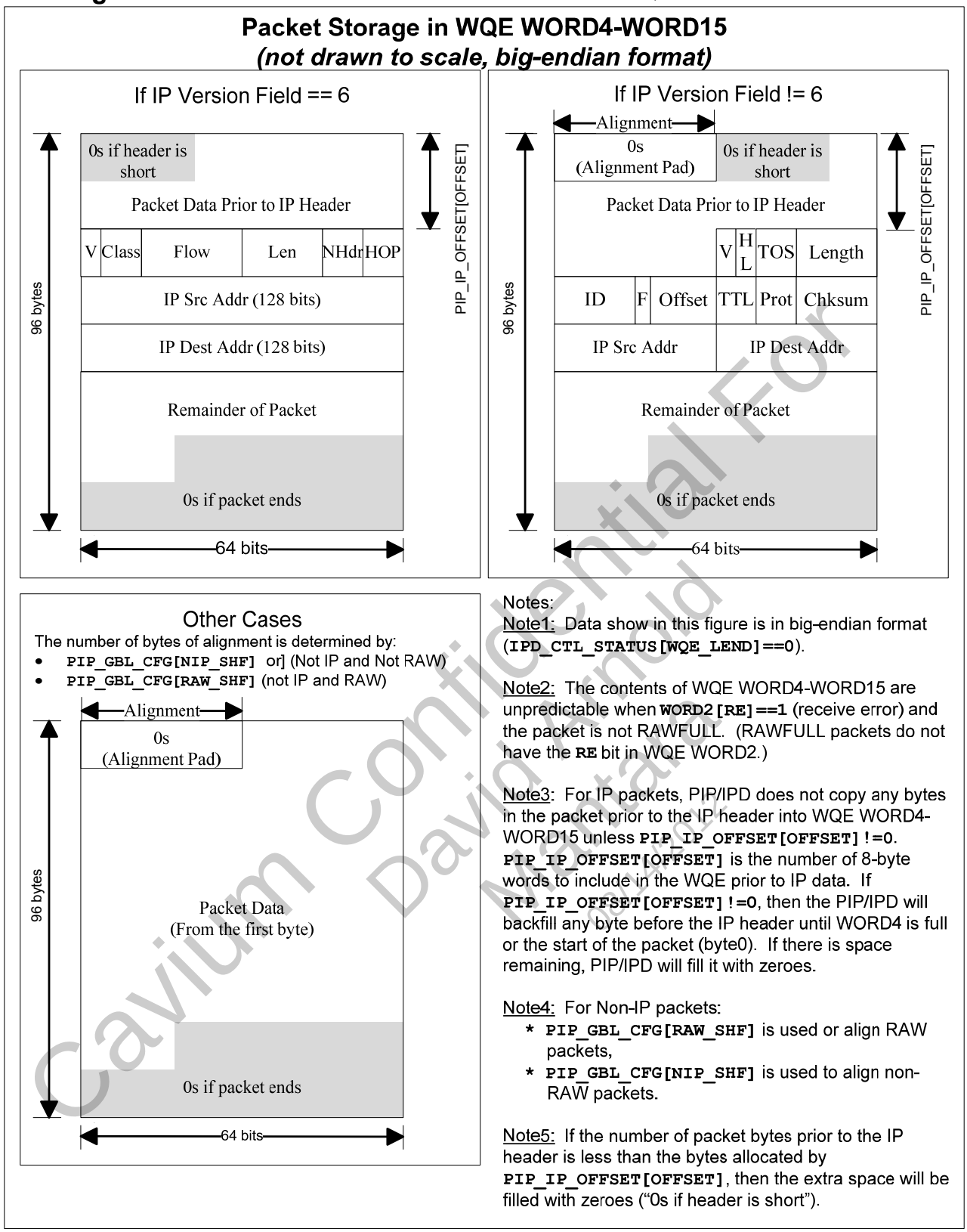
Figure 35: Write Packet Data to L2/DRAM Choices



10.4 Packet Data Storage in WQE WORD4-15

WQE WORD4-15 contains packet data. The format of the packet data in the WQE depends on the type of packet and is shown in the figure below.

Figure 36: Format of Packet Data Stored in WQE WORD4-WORD15



Notes:

Note1: Data show in this figure is in big-endian format (`IPD_CTL_STATUS[WQE_LEND]==0`).

Note2: The contents of WQE WORD4-WORD15 are unpredictable when `WORD2[RE]==1` (receive error) and the packet is not RAWFULL. (RAWFULL packets do not have the `RE` bit in WQE WORD2.)

Note3: For IP packets, PIP/IPD does not copy any bytes in the packet prior to the IP header into WQE WORD4-WORD15 unless `PIP_IP_OFFSET[OFFSET]!=0`. `PIP_IP_OFFSET[OFFSET]` is the number of 8-byte words to include in the WQE prior to IP data. If `PIP_IP_OFFSET[OFFSET]!=0`, then the PIP/IPD will backfill any byte before the IP header until WORD4 is full or the start of the packet (byte0). If there is space remaining, PIP/IPD will fill it with zeroes.

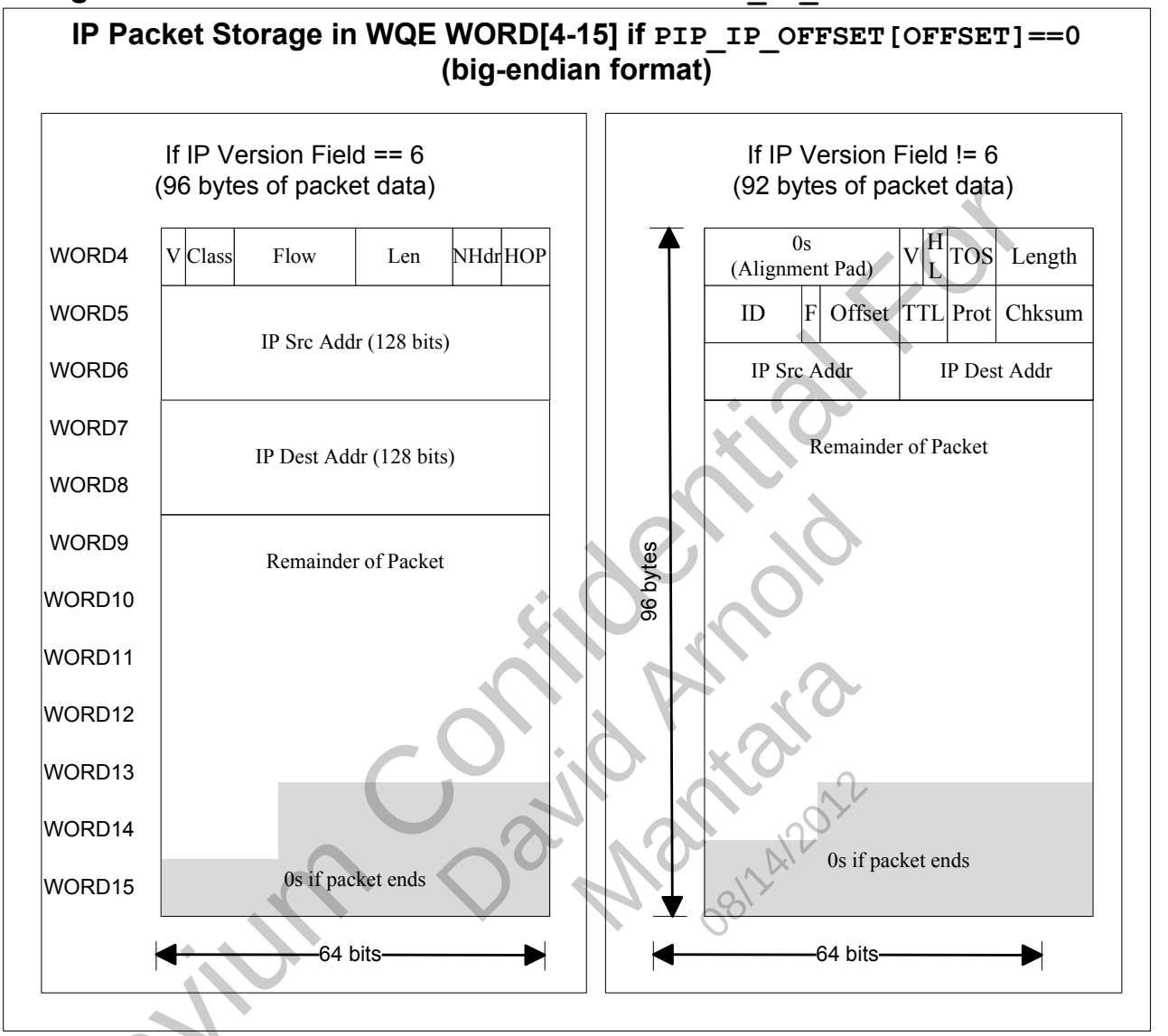
Note4: For Non-IP packets:

* `PIP_GBL_CFG[RAW_SHF]` is used or align RAW packets,
* `PIP_GBL_CFG[NIP_SHF]` is used to align non-RAW packets.

Note5: If the number of packet bytes prior to the IP header is less than the bytes allocated by `PIP_IP_OFFSET[OFFSET]`, then the extra space will be filled with zeroes ("0s if header is short").

The PIP_IP_OFFSET[OFFSET] field is used only for IP packets. PIP/IPD will copy the IP header to the start of WQE WORD4 + alignment + PIP_IP_OFFSET[OFFSET]. Thus, for IP packets, if PIP_IP_OFFSET[OFFSET]==0, the packet will not include the L2 header before the IP header, as shown in the following figure:

Figure 37: Format of Packet Data in WQE if PIP_IP_OFFSET[OFFSET]==0



If PIP_IP_OFFSET[OFFSET] != 0, then PIP/IPD will automatically copy the number of bytes specified by PIP_IP_OFFSET[OFFSET] to the WQE prior to the IP header (backfilling from the IP header toward the start of WQE WORD4), as shown in Figure 36 – "Format of Packet Data Stored in WQE WORD4-WORD15". For example, to copy an Ethernet II header without VLAN (14 bytes) into the WQE prior to the start of the IP header, set PIP_IP_OFFSET[OFFSET]==2 (16 bytes). If there are fewer bytes in the packet than specified by PIP_IP_OFFSET[OFFSET] (the header is "short"), then PIP/IPD will fill these bytes with zeroes (in this example, two bytes are filled with zeroes).

10.4.1 Finding the Start of an IP Packet in the WQE

For IP packets, finding the start of the packet data in the WQE can be tricky. In particular, the alignment bytes, and the `PIP_IP_OFFSET[OFFSET]` value need to be considered. The number of bytes of packet data prior to the IP header may be less than `(PIP_IP_OFFSET[OFFSET] * 8)`. This section describes the math to locate byte0 of the packet data in the WQE, given that `PIP_IP_OFFSET[OFFSET]` is large enough so that byte0 is in the WQE.

If the packet is IP but not IPv6, then a 4-byte alignment pad is added to the start of WQE WORD4.

In the SDK, the `work` data structure contains the WQE. The WQE field `work->packet_data` points to the start of WQE WORD4. This is not the same as the start of the IP packet data because there may be an alignment pad and also zero-filled bytes.

The start of the IP Header is located at:

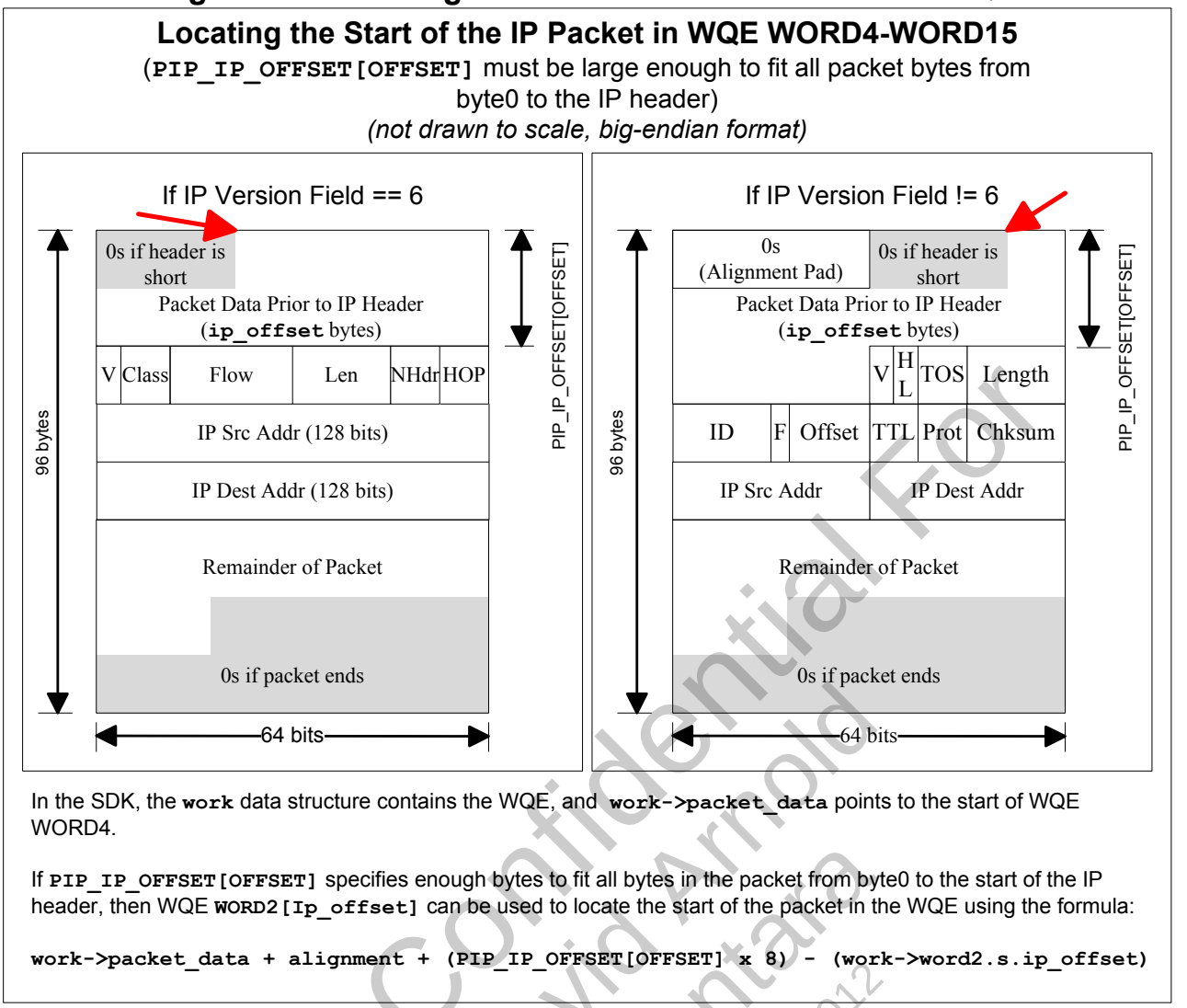
```
// (Start of WQE WORD4) + alignment + (OFFSET in bytes)
work->packet_data + alignment +(PIP_IP_OFFSET[OFFSET] x 8)
```

The following formula assumes that `PIP_IP_OFFSET[OFFSET]` is large enough to include all the packet bytes prior to the IP header. Given the location of the start of the IP header, software can calculate the start of the IP packet by subtracting the number of bytes from byte0 of packet data to the IP header. This is provided in WQE WORD2 field `ip_offset`:

```
// (Start of IP Header) - (distance in bytes from byte0 to the IP header)
work->packet_data + alignment +
(PIP_IP_OFFSET[OFFSET] x 8) - (work->word2.s.ip_offset)
```

See Section 10.5 – "Accessing Packet Data When Some Packets are Dynamic Shorts" for example code.

Figure 38: Locating the Start of an IP Packet in the WQE



In the SDK, the **work** data structure contains the WQE, and **work->packet_data** points to the start of WQE WORD4.

If **PIP_IP_OFFSET[OFFSET]** specifies enough bytes to fit all bytes in the packet from byte0 to the start of the IP header, then WQE **WORD2[Ip_offset]** can be used to locate the start of the packet in the WQE using the formula:

```
work->packet_data + alignment + (PIP_IP_OFFSET[OFFSET] x 8) - (work->word2.s.ip_offset)
```


10.4.2 Dynamic Short Storage in WQE

PIP/IPD analyzes whether the packet can fit entirely into WQE WORD4-WORD15. A packet which can fit into this space is a *dynamic short*. A packet with an L1/L2 receive error can never be a dynamic short. As shown in the figure above, for IP packets, if `PIP_IPD_OFFSET[OFFSET]==0`, packet bytes prior to the IP header are not copied to the WQE. An IP packet cannot be a dynamic short unless `PIP_IPD_OFFSET[OFFSET]` specifies sufficient bytes to copy the entire packet to the WQE.

If PIP/IPD determines a packet is a dynamic short and the dynamic short option is enabled, it will *not* create an additional copy of the packet data to a Packet Data Buffer (which would be redundant). The dynamic short option is enabled if:

- `PIP_PRT_CFGn[DYN_RS]`==1 for the port
- or the `RS` bit is set in the Packet Instruction Header and either:
  - `PIP_GBL_CTRL[IGNRS]==0`
  - or the port is a PCI/PCIe/DPI port (ports 32-35)

Note that PIP/IPD ignores both `PIP_PRT_CFGn[DYN_RS]` and the `RS` bit in the Packet Instruction Header for packets hardware does not classify as dynamic short: it is okay for these fields to be set for all packets.

When the packet data is accessed via WQE WORD4-WORD15 instead of a Packet Data Buffer, the WQE WORD2 `Bufs` field is set to 0, and WQE WORD3 fields `Back`, `Size`, and `Addr` are unpredictable (set to 0 on some OCTEON models, not set on other OCTEON models). The entire packet only exists in the WQE.

If the WQE is in the Packet Data Buffer: Note that in the case of a dynamic short when `IPD_CTL_STATUS[NO_WPTR]==1`, the packet data is written to the WQE data structure, and the WQE data structure is written to the Packet Data Buffer. In this case, no WQE Buffer is allocated. The dynamic short function specifies where the packet data is written; the `NO_WPTR` field specifies where the WQE data structure is written.

If the packet is a dynamic short, and dynamic shorts are enabled, and `IPD_CTL_STATUS[NO_WPTR]==0`, then no Packet Data Buffer is allocated.

10.5 Accessing Packet Data When Some Packets are Dynamic Shorts

When packet data is stored in Packet Data Buffers, software gets the address of the packet data via the address stored in WQE WORD3. The address in WQE WORD3 is unpredictable for dynamic shorts (because they are stored in the WQE instead of in Packet Data Buffers).

To allow the same software to access the packet data regardless of where it is stored, software can create a `buffer_ptr` data structure which is the same data structure as WQE WORD3. Software then initializes the `buffer_ptr` fields differently depending on whether the packet is a dynamic short or not:

- If not a dynamic short: sets the value of `buffer_ptr` to WQE WORD3
- Is dynamic short: software fills in the `pool`, `size`, and `addr` fields in the `buffer_ptr` data structure, setting the value of `addr` to point to the start of packet data in the WQE data structure.

The example code below uses this technique.

The following code is from the `traffic-gen` example:
This code illustrates software creating a separate `WQE[WORD3]` data structure (which includes a pointer to the start of packet data). This data structure can then be used by software without needing to know whether the packet is stored in the Packet Data section of the Packet Data Buffer or in the WQE.

```
/**
 * Given a WQE, return an OCTEON packet pointer for the beginning
 * of the packet data (NOT THE SAME AS A "C" pointer".
 * For packets where data is stored in a packet data buffer, this is trivially
 * the packet pointer in the WQE.  For packets where bufs is zero (dynamic
 * shorts), this is non trival.
 */
static inline cvmx_buf_ptr_t get_packet_buffer_ptr(const cvmx_wqe_t *work)
{
    cvmx_buf_ptr_t buffer_ptr;

    if (cvmx_likely(work->word2.s.bufs == 0)) // dynamic short if bufs==0
     {
         buffer_ptr.u64 = 0;

         buffer_ptr.s.pool = CVMX_FPA_WQE_POOL;
         buffer_ptr.s.size = CVMX_FPA_WQE_POOL_SIZE;

         // work->packet_data points to the start of WQE WORD4
         buffer_ptr.s.addr = cvmx_ptr_to_phys((void*)work->packet_data);

         // WARNING: This code assume that PIP_GBL_CFG[RAW_SHF]=0 and
         //  PIP_GBL_CFG[NIP_SHF]=0. If this was not the case we'd
         //  need to add these offsets depending on if the packet was
         //  in RAW mode or not.
         //  addr += PIP_GBL_CFG[RAW_SHF] for the RAW case.
         //  addr += PIP_GBL_CFG[NIP_SHF]; for the non-IP case

         if (cvmx_likely(!work->word2.s.not_IP)) // likely is an IP packet
         {
             // add the IP alignment
             // alignment==0 for IPv6, otherwise == 4 bytes
             buffer_ptr.s.addr += (work->word2.s.is_v6^1)*4;

             // this code assumes PIP_IP_OFFSET[OFFSET] is large enough so that
             // all the packet bytes from byte0 to the IP header are all stored
             // in WQE WORD4 prior to the IP header.  In this case, ip_offset
             // specifies the number of packet bytes prior to the IP header.
             // PIP_IP_OFFSET is in 8-byte units
             buffer_ptr.s.addr += (PIP_IP_OFFSET*8 - work->word2.s.ip_offset);
         }
     }
     else // not a dynamic short
         buffer_ptr = work->packet_ptr;

    return buffer_ptr;
}
```

10.6 Configuring Packet Storage

Table 29: Registers to Configure Packet Storage

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Packet Storage Choices				
Packet Storage Choices: 0=All packets written to DRAM, bypassing L2 cache 1=All Packet Data Buffers written to L2 cache. 2=The first cache block containing the `Next_Buf_Ptr` is written to L2 cache, all others are written directly to DRAM, bypassing L2 cache. 3=First two cache blocks containing the `Next_Buf_Ptr` and packet data are written to L2 cache, all others are written directly to DRAM, bypassing L2 cache.	`IPD_CTL_STATUS`	`OPC_MODE`	0	See Note1
Omit WQE Buffer: When set to 1, Work Queue Entry buffers are not used. The WQE data is located in the first 128 bytes of the Packet Data Buffer. Space must be reserved using `IPD_1ST_MBUFF_SKIP[SKIP_SZ]`. See the *HRM* register field description for details.	`IPD_CTL_STATUS`	`NO_WPTR`	0	0 (H/W Default)
MBUF Configuration				
First MBUF skip amount. The number of 8-byte words from the start of the first mbuf at which to store the next pointer. Legal values are 0-32. See also the `IPD_1ST_NEXT_PTR_BACK` register. This field can be used to reserve space in the Packet Data Buffer for software use, or for the WQE if the WQE will be stored in the Packet Data Buffer instead of a WQE Buffer.	`IPD_1ST_MBUFF_SKIP`	`SKIP_SZ`	0	See Note2

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
All other MBUF skip amounts. The number of 8-byte words from the start of all mbufs (except the first) at which to store the next pointer. Legal values are 0-32. See also the IPD_2ND_NEXT_PTR_BACK register.	IPD_NOT_1ST_MBUFF_SKIP	SKIP_SZ	0	See Note3
MBUF size. The number of 8-byte words in an MBUF. Legal values are in the range 32-2048. (Pool 0 buffers must be a minimum of 256 bytes.) **If the packet data buffers in the FPA pool are smaller than this size, packet data will be written to adjacent memory, corrupting the system. Packet data buffers may be larger than the mbuf size.**	IPD_PACKET_MBUFF_SIZE	MB_SIZE	0x20 (256 bytes)	See Note4
Back Pointer Configuration				
First Back: The number of 128-byte words to subtract from WQE WORD3[Addr] (the start of the packet in the first mbuf). This is used to locate the start of the mbuf. Legal values are 0-15. The value must be consistent with the IPD_1ST_MBUFF_SKIP[SKIP_SZ] value.	IPD_1ST_NEXT_PTR_BACK	BACK	0	See Note5
Not First Back: The number of 128-byte words to subtract from the next_ptr value (the start of the packet). This is used to locate the start of the mbuf when it is not the first mbuf. Legal values are 0-15. The value must be consistent with the IPD_NOT_1ST_MBUFF_SKIP[SKIP_SZ] value.	IPD_2ND_NEXT_PTR_BACK	BACK	0	See Note6
Length Compliance				
Length compliance bit. When this bit is set to 1, eight bytes are subtracted from the data length field so that it does not include 8 bytes for the Packet Instruction Header.	IPD_CTL_STATUS	LEN_M8	1	1 (See Note7)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Real Short (Dynamic Short) Options				
Ignore RS bit. If set to 1, ignore RS bit in Packet Instruction Header (applies only to ports 0-31).	PIP_GBL_CTL	IGNRS	0	0 (H/W Default)
Dynamic RS calculation. If set to 1, dynamically calculate RS based on packet size.	PIP_PRT_CFG*n* (one per port)	DYN_RS	0	0 (H/W Default)
Padding				
RAW Packet Alignment Pad. The number of bytes to pad a RAW packet (0-7 bytes).	PIP_GBL_CFG	RAW_SHF	0	0 (H/W Default)
Non-IP Packet Alignment Pad. The number of bytes to pad a non-IP packet which is not RAW (0-7 bytes).	PIP_GBL_CFG	NIP_SHF	0	0 (H/W Default)
Pre-IP Data Pad. The number of 8-byte words to include in the WQE prior to IP data. PIP/IPD will backfill packet data bytes starting at the IP header until the beginning of WORD4 or until there is no more packet data (byte0). PIP/IPD will zero-fill any remaining space. IP packets are automatically aligned by PIP/IPD. OFFSET is calculated from the start of the packet and includes the automatic alignment. If OFFSET==0, the IP header starts at WQE WORD4. If OFFSET==1, the IP header starts at WQE WORD5.	PIP_IP_OFFSET	OFFSET	0	0 (H/W Default)
FCS Stripping				
Strip FCS: For ports 0-31: When a bit is set, the Frame Check Sum (also known as CRC) is not stored for packets arriving on the port corresponding to that bit position. This bit should only be set if both the CRC is present and should not be stored. FCS cannot be stripped from PCI/PCIe/DPI or sRIO Messaging ports.	IPD_SUB_PORT_FCS (one bit per port)	PORT_BIT	0xFFFFFFFF	0xFFFFFFFF (H/W Default) (See Note8, Note10, Note11)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Strip FCS: For ports 36-39: When a bit is set, the Frame Check Sum (also known as CRC) is not stored for packets arriving on the port corresponding to that bit position. This bit should only be set if both the CRC is present and should not be stored. FCS cannot be stripped from PCI/PCIe/DPI or sRIO Messaging ports.	IPD_SUB_PORT_FCS (one bit per port)	PORT_BIT2	0xF	0x0 (See Note9, Note11)
Endianness				
Packet Endianness specification. If set to 1, packet is written in little Endian.	IPD_CTL_STATUS	PKT_LEND	0	0 (H/W Default)
Miscellaneous Settings				
Packet buffering off. When set to 1, the IPD does not use its internal buffers to buffer the received packet data. This is not used in normal operation.	IPD_CTL_STATUS	PKT_OFF	0	0 (H/W Default)

Notes

Note1: IPD_CTL_STATUS[OPC_MODE] is initialized to CVMX_IPD_OPC_MODE_STT when cvmx_helper_initialize_packet_io_global() is called.

Note2: IPD_1ST_MBUFF_SKIP[SKIP_SZ] is initialized to (CVMX_HELPER_FIRST_MBUFF_SKIP / 8) when cvmx_helper_initialize_packet_io_global() is called.

Note3: IPD_NOT_1ST_MBUFF_SKIP[SKIP_SZ] is initialized to (CVMX_HELPER_NOT_FIRST_MBUFF_SKIP / 8) when cvmx_helper_initialize_packet_io_global() is called.

Note4: IPD_PACKET_MBUFF_SIZE[MB_SIZE] is initialized to (CMVX_FPA_PACKET_POOL_SIZE / 8) when cvmx_helper_initialize_packet_io_global() is called. IPD_PACKET_MBUFF_SIZE[MB_SIZE] must always be at least 18 64-bit words larger than IPD_1ST_MBUFF_SKIP[SKIP_SZ], and at least 16 64-bit words larger than IPD_NOT_1ST_MBUFF_SKIP[SKIP_SZ].

Note5: IPD_1ST_NEXT_PTR_BACK[BACK] is initialized to (CVMX_HELPER_FIRST_MBUFF_SKIP + 8) / 128 (+8 is for next ptr) when cvmx_helper_initialize_packet_io_global() is called.

Note6: IPD_2ND_NEXT_PTR_BACK[BACK] is initialized to (CVMX_HELPER_NOT_FIRST_MBUFF_SKIP + 8) / 128 (+8 is for next ptr) when cvmx_helper_initialize_packet_io_global() is called.

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Note7: IPD_CTL_STATUS[LEN_M8] is initialized to TRUE (1) if CMVX_ENABLE_LEN_M8_FIX is defined when cvmx_helper_initialize_packet_io_global() is called. CN38XX/CN36XX pass-2 versions have a known issue in which the Size field is too large by 8 (incorrect). On other processors, if IPD_CTL_STATUS[LEN_M8]==0, the Size field is too large by 8 (incorrect). If IPD_CTL_STATUS[LEN_M8]==1, the Size field is correct.				
Note8: Software should not remove the CRC (strip FCS) from ports for which Work Queue Entry's hardware checksum field (HW_Chksum) may be used by software. This is because the CRC bytes are included in the hardware checksum, and software will probably need to reference the CRC value to use the hardware checksum.				
Note9: The default SDK turns off FCS stripping for the loopback ports.				
Note10: There is no FCS stripping on ports 32-35 (PCI/PCIe/DPI) or 43-46 (sRIO Messaging).				
Note11: The FCS strip bit should only be set when both CRC is present and should be removed.				

11 Statistics (Performance, Debugging)

The statistics registers are useful in debugging. There are two types of statistics registers:

- PIP_STAT*x*_PRT*n* per-port registers which contain normal statistics and can be cleared on read. These registers do not count dropped packets.
- PIP_STAT_INB_**n* per-port registers which are used for system debugging and cannot be cleared on read. These registers count all packets including dropped packets and packets with errors.

The function cvmx_pip_get_port_status() will read all of these values and provide them in the cvmx_pip_port_status_t data structure. (See Section 2.4.6 – "The cvmx_pip_port_status_t Data Structure".) The statistics are the same for the different processors.

The register field PIP_STAT_CTL[RDCLR] is used to configure whether the PIP_STAT*x*_PRT*n* registers are cleared after they are read:

If RDCLR==0, then PIP_STAT*x*_PRT*n* registers hold value when read
If RDCLR==1, then PIP_STAT*x*_PRT*n* registers are cleared when read (default value)

Note: To count all packets dropped by the system, sum the (number of packets dropped by the receiver) + (number of packets dropped by IPD).

In the list below, PIP_STAT_INB_**n* register field counters include packets which are dropped by IPD. PIP_STAT*x*_PRT*n* registers do not include dropped packets.

Table 30: Statistics Register Fields (Read Only)

Brief Description	Register	Fields
Packets Dropped Statistics		
Number of inbound packets dropped by IPD due to either Per-Port Packet Drop or Per-QoS RED/WRED	PIP_STAT0_PRT*n* (one per port)	DRP_PKTS
Number of inbound octets dropped by IPD due to either Per-Port Packet Drop or Per-QoS RED/WRED (See Note2)	PIP_STAT0_PRT*n* (one per port)	DRP_OCTS
Traffic Statistics		
Number of packets processed by PIP per port	PIP_STAT2_PRT*n* (one per port)	PKTS
Number of octets processed by PIP per port (both good and bad (with errors)) (See Note 2)	PIP_STAT1_PRT*n* (one per port)	OCTS
RAWFULL and RAWSCH packets without an L1/L2 error processed by PIP per port	PIP_STAT2_PRT*n* (one per port)	RAW
Broadcast and Multicast Statistics		
Number of identified L2 broadcast packets processed by PIP per port. Does not include multicast packets. Only includes packets whose parse mode is skip-to-L2	PIP_STAT3_PRT*n* (one per port)	BCST
Number of identified L2 multicast packets processed by PIP per port. Does not include broadcast packets. Only includes packets whose parse mode is skip-to-L2	PIP_STAT3_PRT*n* (one per port)	MCST
Size Statistics		
Number of 65-to-127 byte packets processed by PIP per port	PIP_STAT4_PRT*n* (one per port)	H65to127
Number of 64-byte packets processed by PIP per port	PIP_STAT4_PRT*n* (one per port)	H64
Number of 256-to-511 byte packets processed by PIP per port	PIP_STAT5_PRT*n* (one per port)	H256to511
Number of 128-to255 byte packets processed by PIP per port	PIP_STAT5_PRT*n* (one per port)	H128to255
Number of 1024-to-1518 byte packets processed by PIP per port	PIP_STAT6_PRT*n* (one per port)	H1024to1518
Number of 512-to-1023 byte packets processed by PIP per port	PIP_STAT6_PRT*n* (one per port)	H512to1023
Number of 1519-to-max byte packets processed by PIP per port	PIP_STAT7_PRT*n* (one per port)	H1519
Error Statistics		
Number of packets processed by PIP with FCS or Align opcode errors per port. (Note: FCS is not checked on PCIe ports (32-35).)	PIP_STAT7_PRT*n* (one per port)	FCS
Number of packets processed by PIP with length < minimum *and* FCS error per port. (Note: FCS is not checked on PCIe ports (32-25).)	PIP_STAT8_PRT*n* (one per port)	FRAG
Number of packets processed by PIP with length < minimum per port	PIP_STAT8_PRT*n* (one per port)	UNDERSZ
Number of packets processed by PIP with length > maximum and FCS error per port. (Note: FCS is not checked on PCIe ports (32-35).)	PIP_STAT9_PRT*n* (one per port)	JABBER
Number of packets processed by PIP with length > maximum	PIP_STAT9_PRT*n* (one per port)	OVERSZ
Inbound Statistics (intended for system debug) (See Note3)		
Number of octets from all packets received by PIP per port (includes packets with errors and packets dropped by IPD) (See Note1)	PIP_STAT_INB_OCTS*n* (one per port)	OCTS
Number of packets with errors received by PIP per port (includes packets with errors and packets dropped by IPD) (See Note1)	PIP_STAT_INB_ERRS*n* (one per port)	ERRS

Brief Description	Register	Fields
Number of packets without errors received by PIP per port (includes packets with errors and packets dropped by IPD) (See Note1)	PIP_STAT_INB_PKTS*n* (one per port)	PKTS
Notes		
Note1: PIP_STAT_INB* register field counters include packets which are dropped by IPD (all packets "received" by PIP/IPD). Otherwise, only "processed" packets are counted (which excludes packets dropped by PIP/IPD or which do not yet have an assigned WQE).		
Note2: Octets (8 bit words) count every byte in each packet: If PIP/IPD receives a single 64-byte packet, the packet statistic would increment by one while the octet statistic would increment by 64.		
Note3: Both sets of registers can accumulate. Only when PIP_STAT_CTL[RD_CLR] is set will the PIP_STAT*x*_PRT*n* registers clear on reads. The PIP_STAT_INB_**n* registers cannot be cleared on read; the values continue to accumulate.		

12 Congestion Control (Backpressure, Packet Drop, RED, WRED)

This section provides:

- A system-level view of congestion causes and prevention
- Overview of the congestion control mechanisms provided by PIP/IPD
- Easy configuration information for PIP/IPD congestion control
- Detailed information on each PIP/IPD congestion control mechanism

12.1 System-Level View of Congestion: Causes and Prevention

Congestion can be caused by either a sudden increase in traffic (normal) or a design/software error (unexpected).

12.1.1 Congestion Management Design Issues:

The design questions pertinent to congestion management:

- What is the expected traffic pattern?
- What should happen during a spike in traffic? (A spike is an unexpected higher than average network load.)
- Can the sender respond to backpressure?
- Can some packets be dropped?
- Should high-priority traffic be protected?

12.1.2 Normal Congestion

A normal cause of temporary congestion is a sudden and temporary increase in traffic (a traffic spike).

For example, if a system is designed to handle 100 packets per second, has a steady traffic of 75 packets per second, and then receives a spike of 500 packets per second, the number of available buffers will drop abruptly, and then slowly recover when the system returns to the steady traffic of 75 packets per second. In this scenario, while the steady traffic continues, the system will be able to process the steady traffic level + 25 buffers per second.

12.1.3 Unexpected Congestion

If unexpected backpressure or packet drop occurs, use the following flow chart as a troubleshooting guide. This chart shows a packet flowing through the system, and congestion issues which can occur, along with user-configurable congestion-control points.

Figure 39: System View of Backpressure/Congestion, part 1

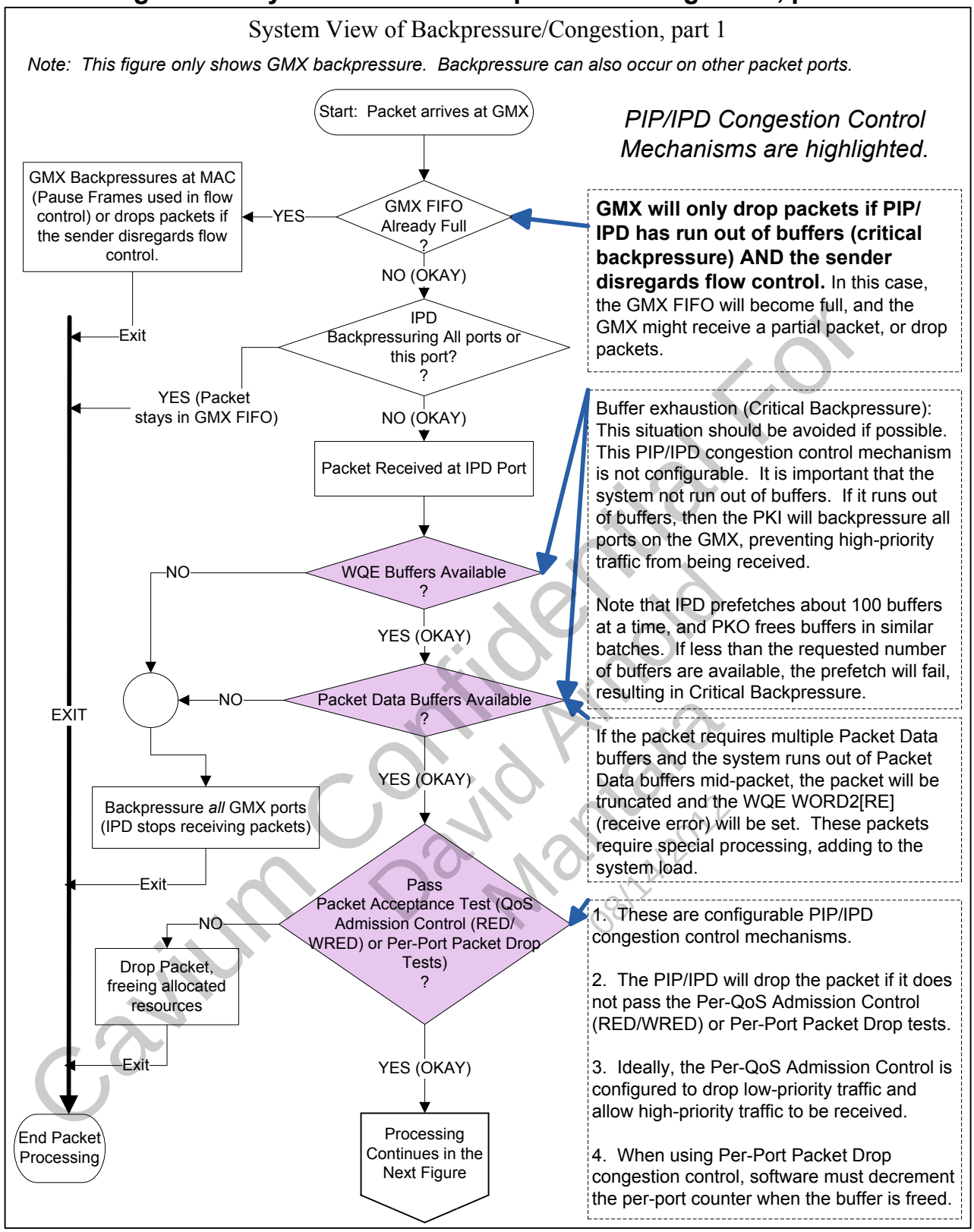
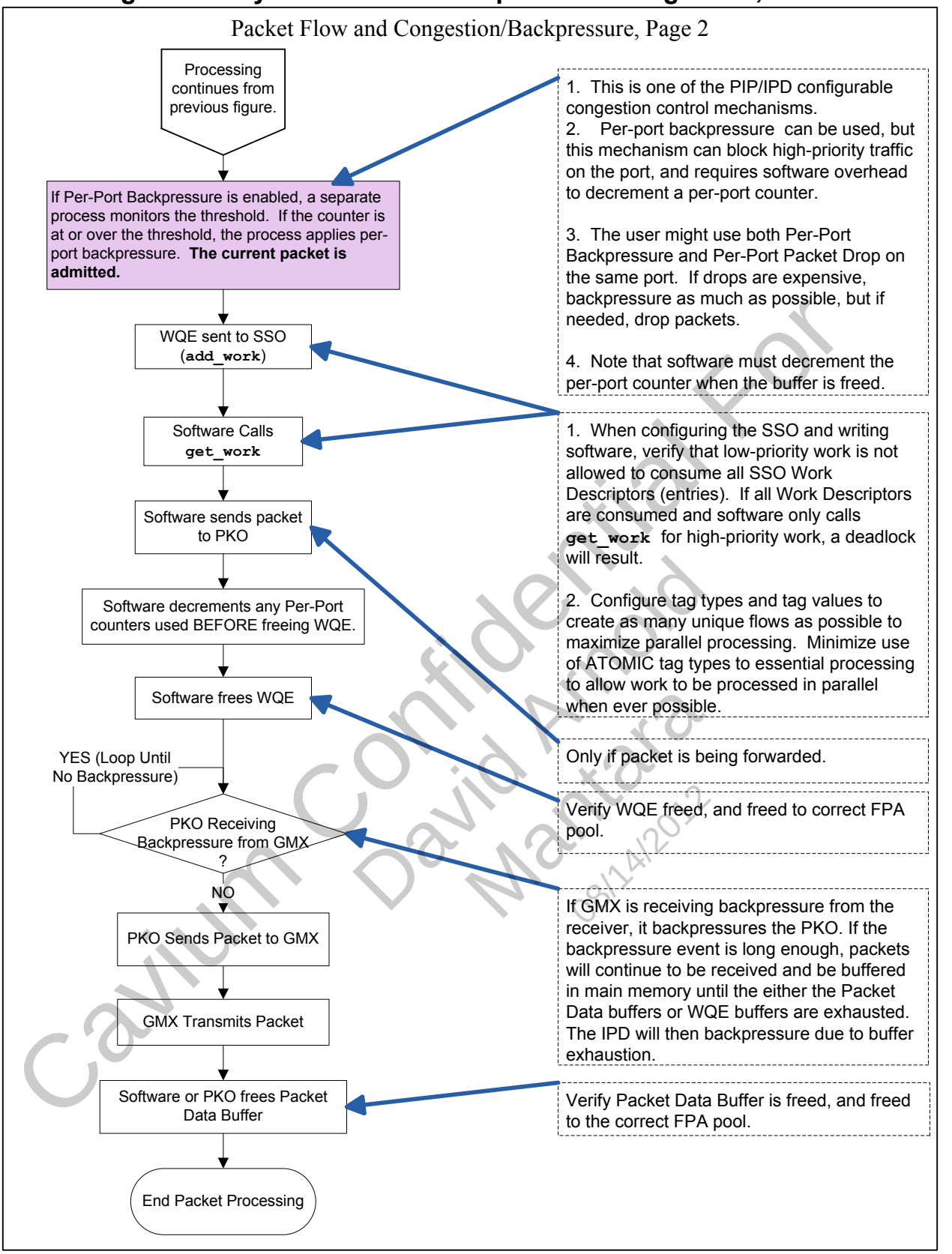


Figure 40: System View of Backpressure/Congestion, Part 2



12.2 Overview of Congestion-Control Mechanisms Provided by PIP/IPD

PIP/IPD provides several congestion-control mechanisms, as shown in the next table:

Table 31: Overview of PIP/IPD Congestion Control Mechanisms

Critical Backpressure
The IPD exerts internal backpressure on the all ports, causing the receivers FIFO to become full (such as the GMX FIFO). Once the receiver's FIFO becomes full, then depending on its configuration it can backpressure/drop packets. This situation should be avoided.
Per-QoS Admission Control (WRED, RED)
The number of available Packet Data buffers is compared to the Per-QoS HIGH and LOW watermarks. If the number of available Packet Data buffers is: • greater than the PASS (high) watermark: all packets are admitted • less than or equal to the DROP (low) watermark: all packets are dropped • equal to or less than PASS and greater than DROP: packets are randomly dropped Each QoS queue can have different watermarks, which is the preferred congestion control method because it allows high-priority traffic to flow while dropping lower priority traffic. This feature may be combined with Per-Port Backpressure.
Per-Port Backpressure
A counter contains the number of in-use buffers for the port. If the counter exceeds the per-port threshold, backpressure the port. (The current packet is accepted.) Software is responsible for decrementing the counter when the buffer is freed. Per-Port Backpressure and Per-Port Packet Drop use similar configuration registers and the same in-use buffer counter. This feature may be combined with Per-QoS WRED/RED or Per-Port RED.
Per-Port Packet Drop
A counter contains the number of in-use buffers for the port. If the counter exceeds the per-port threshold, drop all incoming packets for the port. Software is responsible for decrementing the counter when the buffer is freed. Per-Port Backpressure and Per-Port Packet Drop use similar configuration registers and the same in-use buffer counter.
Per-Port RED
Drop packets randomly on a per-port basis if the number of available buffers drops to a level at or below the threshold set for the QoS queue. This is implemented using the Per-QoS RED Congestion Control mechanism. To implement this, there must be less than 8 ports used in the system (the same as the maximum number of QoS queues). This feature may be combined with Per-Port Backpressure.

Each of these mechanisms is covered in greater detail in the sections below. For additional information, see the *HRM*.

12.3 Critical Backpressure (Buffer Exhaustion)

Critical Backpressure occurs if there are no more available Packet Data buffers or WQE buffers (buffer exhaustion). Critical Backpressure results in IPD backpressuring *all* ports (no packets will be received by IPD). This mechanism is not user-configurable.

If buffer exhaustion occurs, IPD will backpressure *all* ports, stopping the flow of *all* traffic regardless of the traffic priority. Buffer exhaustion should be avoided by using one of the user-configurable congestion control mechanisms.

The following figure illustrates Critical Backpressure due to buffer exhaustion.

Figure 41: Critical Backpressure Situation, Backpressure on All Ports

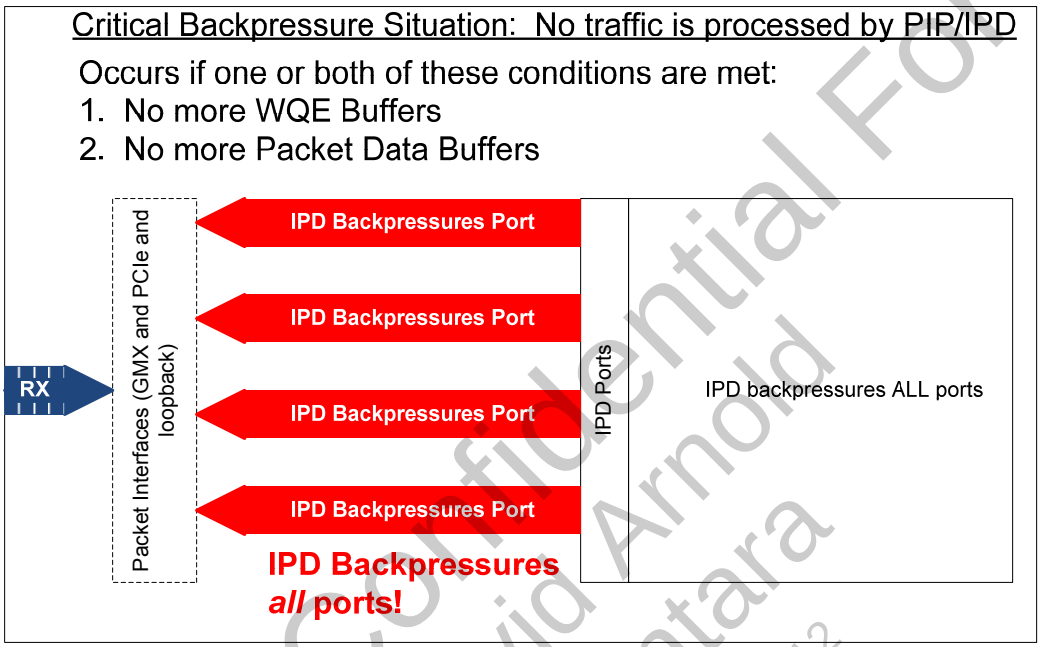


Table 32: Critical Backpressure Overview

Critical Backpressure Overview	
Action	The IPD exerts internal backpressure on the all ports, causing the receivers FIFO to become full (such as the GMX FIFO). Once the receiver's FIFO becomes full, then depending on its configuration it can backpressure/drop packets.
Configuration Options	None. This feature is not configurable and cannot be disabled.
Based on	Not enough Packet Data buffers or Work Queue Entry buffers to receive the entire packet.
Pros	This feature is automatic: no configuration is required.
Cons	If buffer exhaustion occurs, IPD will backpressure *all* ports, stopping the flow of *all* traffic regardless of the traffic priority. Buffer exhaustion should be avoided by using one of the user-configurable congestion control mechanisms.
Possible Configuration Errors	Failing to configure other mechanisms to prevent buffer exhaustion.

Note : Buffer exhaustion can occur when there are still available buffers in the FPA pool in following case: The IPD prefetches a block of WQE and Packet Data buffer pointers from the FPA. If there are not enough buffers available to satisfy the request, the request will fail (all or nothing). The IPD prefetch amount is different for various chips, and is approximately 100 buffers. If less than the prefetch amount of buffers are available, buffer exhaustion will occur even though there are buffers remaining in the pool. We recommend that congestion control mechanisms are configured so if there are ever less than 128 buffers, all input is dropped. This maintains a stable number of available buffers for IPD prefetch.

12.4 PIP/IPD Congestion-Control Configuration

To configure congestion control:

- First decide which traffic classes are to be assigned to the various QoS levels
- Then determine the expected steady-state traffic load for each of those traffic classes, and make sure adequate Packet Data and Work Queue Entry buffers are available for routine traffic
- Then determine the desired action during a traffic spike:
  - Keep high priority traffic flowing while dropping random low-priority traffic: use Per-QoS RED/WRED (recommended)
  - Keep traffic flowing on some ports while blocking and/or dropping all packets regardless of priority on some or all of the ports. Note these mechanisms do not let high-priority traffic through and require software overhead to decrement each port's In-Use Buffer Counter: use Per-Port Backpressure or Per-Port Packet Drop

12.4.1 Basic QoS RED Configuration: `cvmx_helper_setup_red()`

The SDK provides the function `cvmx_helper_setup_red(int pass_thresh, int drop_thresh)`. This function can be called by the user to configure Per-QoS RED. The function will turn off Per-Port Backpressure and Per-Port Packet Drop. All 8 QoS levels will be configured with the same `pass_thresh` and `drop_thresh`, creating a Random Early Drop (RED) solution which is not weighted (not WRED). The number of free packet data buffers will be determined by a periodic snapshot of `IPD_QUE0_FREE_PAGE_CNT`.

This function only turns off Per-Port Backpressure and Per-Port Packet Drop for the packet interfaces, not for PCI/PCIe rings.

See Section 12.5 – "Per-QoS Admission Control (RED and WRED)" for information about Per-QoS RED.

12.4.2 Basic QoS WRED Configuration: `cvmx_helper_setup_red_queue()`

After calling `cvmx_helper_setup_red()`, the user may call `cvmx_helper_setup_red_queue(int queue, int pass_thresh, int drop_thresh)` to adjust the thresholds for the QoS queue to a different value, and thus implement Weighted Random Early Drop (RED).
See Section 12.5 – "Per-QoS Admission Control (RED and WRED)" for information about Per-QoS RED.

12.4.3 Custom Configuration

Custom configuration details are discussed in the sections for each type of PIP/IPD congestion control. There is no SDK function provided as of SDK 2.0 to simplify custom configuration.

12.5 Per-QoS Admission Control (RED and WRED) (PQ-RED)

The Per-QoS Random Early Drop (RED) and Weighted Random Early Drop (WRED) mechanism compares the number of available Packet Data Buffers to each QoS queue's HIGH and LOW watermark values. If the number of available Packet Data buffers is:

- Greater than the PASS (high) watermark: all packets are admitted
- Less than or equal to the DROP (low) watermark: all packets are dropped
- Equal to or less than PASS and greater than DROP: packets are randomly dropped

RED is implemented when all QoS queues have the same PASS and DROP watermarks.

WRED is implemented when the PASS and DROP watermarks are unique for each queue (weighting for the queue's priority), allowing high-priority traffic to be received while lower-priority traffic is randomly dropped.

This feature can be easily configured using the SDK functions `cvmx_helper_setup_red()` and `cvmx_helper_setup_red_queue()`.

This congestion control mechanism is recommended, especially when used with the WRED option, a snapshot of the number of available Packet Data Buffers in the FPA. This combination it is the best choice to keep high-priority traffic to flowing on all ports during times of congestion.

Note: The actual number of available Packet Data buffers may exceed the snapshot value because the snapshot value does not include any buffers prefetched by IPD.

Note: The QoS value calculated for this congestion control mechanism may be different than the value used for the WQE when the packet has one of the following:

- An L2/L1 receive error (`WORD2[RE]==1`)
- An IP error (`WORD2[IE]==1`)
- A TCP/UDP error (`WORD2[LE]==1`)

The difference in QoS value occurs because the Per-QoS RED-WRED QoS calculation occurs before the entire packet is received.

Figure 42: Per-QoS Weighted Random Early Drop (WRED)

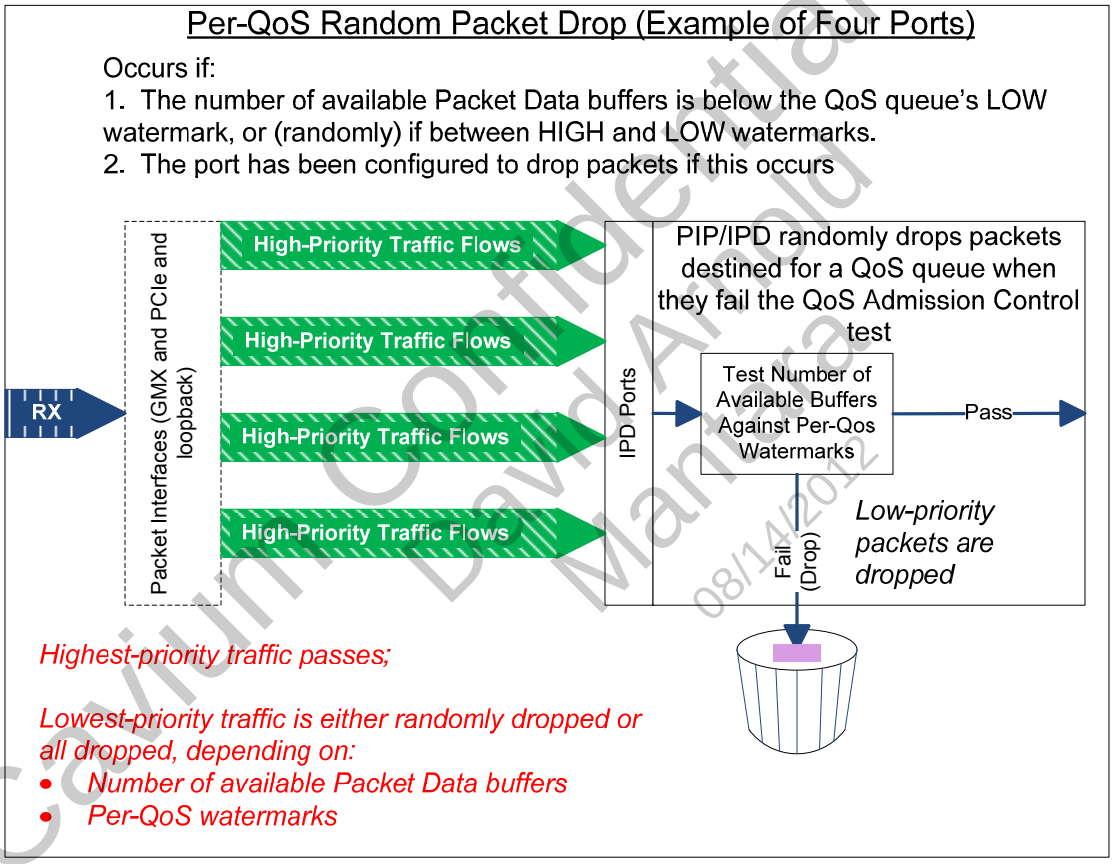


Table 33: Overview of Per-QoS RED and WRED

Per-QoS Congestion Control (WRED or RED) Overview	
Action	Implement either Random Early Drop (RED) or Weighted Random Early Drop (WRED): • RED is implemented when all QoS queues have the same PASS and DROP watermarks. • WRED is implemented when the PASS and DROP watermarks are unique for each queue (weighting for the queue's priority), allowing high-priority traffic to be received while lower-priority traffic is randomly dropped. This mechanism compares the number of available Packet Data buffers to the PASS and DROP watermarks for the target QoS queue. If the number of available Packet Data buffers is: • greater than the PASS (high) watermark: all packets are admitted • less than or equal to the DROP (low) watermark: all packets are dropped • equal to or less than PASS and greater than DROP: packets are randomly dropped The number of available Packet Data buffers can be either a snapshot value (recommended) or a calculated moving average.
Configuration Options	Per-QoS enable Per-QoS PASS and DROP watermarks Snapshot or moving average WRED or RED The function `cvmx_helper_setup_red()` will configure RED based on a snapshot value. The function `cvmx_helper_setup_red_queue()` can be used to configure WRED based on a snapshot value after calling `cvmx_helper_setup_red()`.
Based on	Number of available Packet Data Buffers (either a snapshot or a moving average).
Pros	This is the preferred congestion control method, when implemented with snapshot and WRED, because it allows high-priority traffic to flow while dropping lower priority traffic.
Pros/Cons	If packet drop is not acceptable and the sender will respond to backpressure, use Per-Port Backpressure instead.
Possible Configuration Errors	Setting the HIGH watermark to a value *above* the initial buffer count. Otherwise, the HIGH watermark will never be crossed and the drop mechanism will not be activated.

Figure 43: Per-QoS Admission Control (RED/WRED) Options

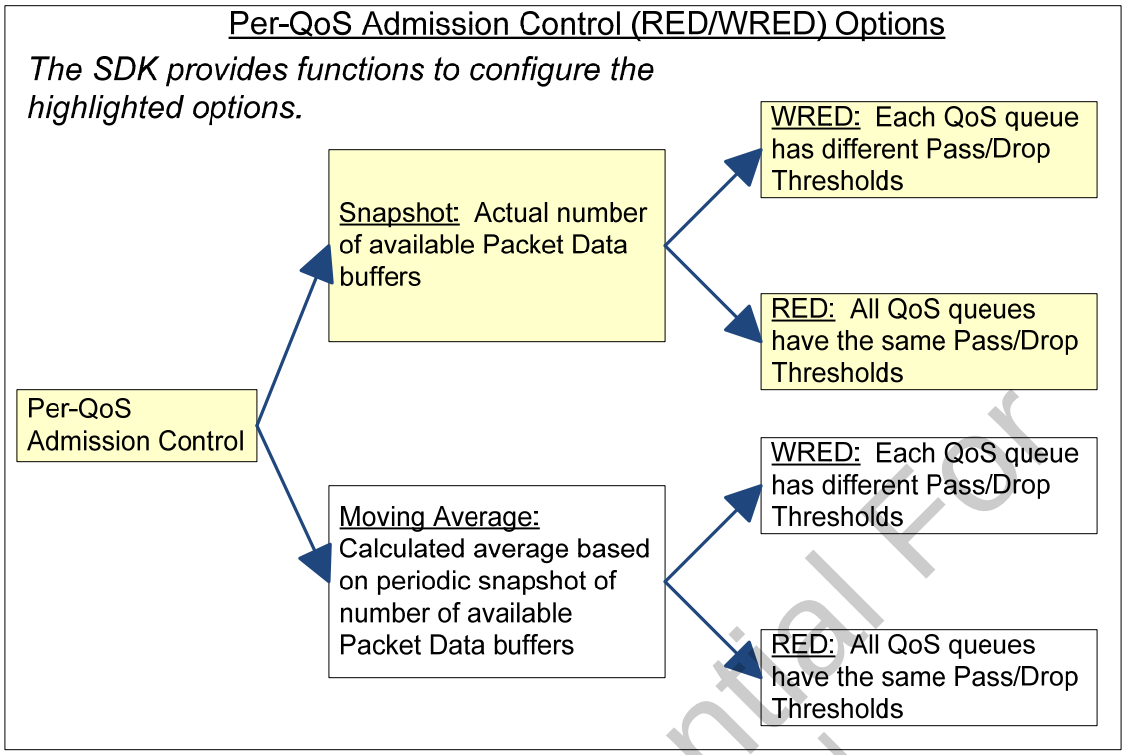
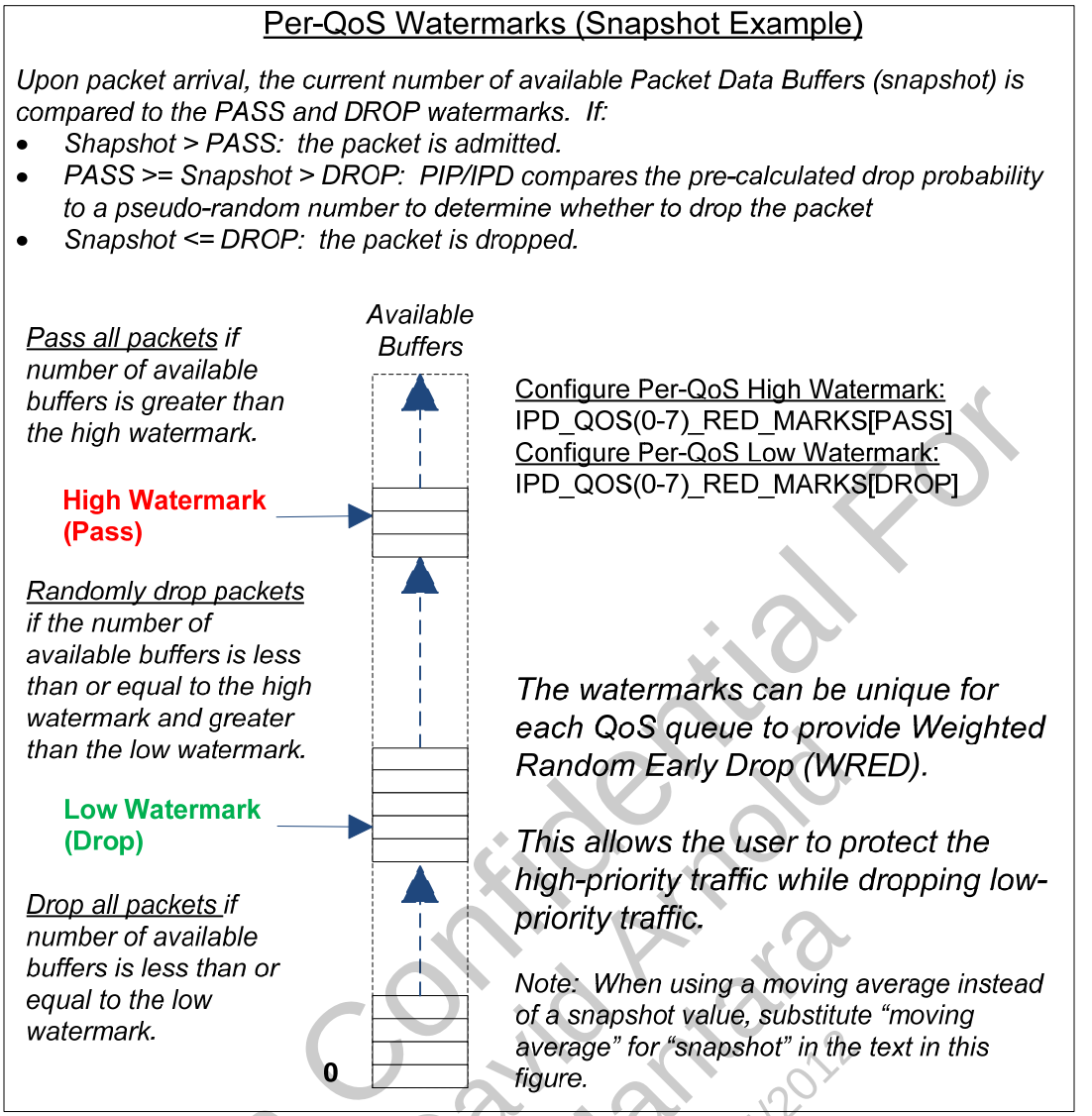


Figure 44: Per-QoS RED – Using Snapshot Value

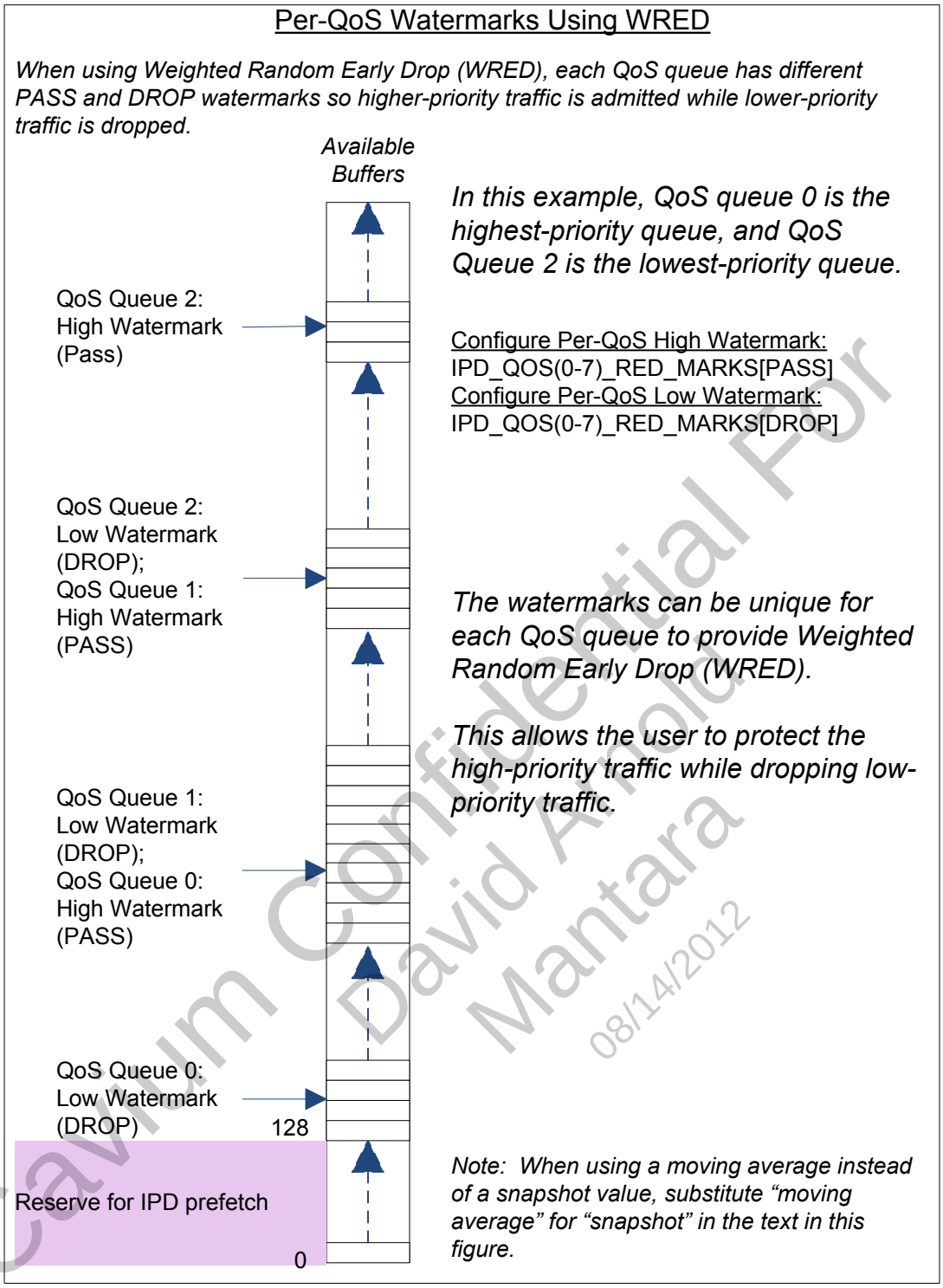


12.5.1 The Simplest Case: Snapshot Value (Recommended)

In the simplest case (recommended), when a packet is received, PIP/IPD uses the actual number of available Packet Data Buffers (the value of `IPD_QUE0_FREE_PAGE_CNT[Q0_PCNT]`) (snapshot value) and compares it to two per-QoS queue watermarks: PASS (high watermark) and DROP (low watermark).

There is a per-port enable bit. If the bit is not set, then the port does not participate in the Per-QoS RED.

Figure 45: Configuring WRED: Different Watermarks for Each QoS Queue



The registers used to configure this feature are shown in the following table.

Table 34: Registers to Configure Per-QoS RED/WRED – Snapshot

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Global Snapshot of Number of Available Packet Data Buffers				
Number of Available Packet Data Buffers: This register is constantly updated by the FPA with the number of buffers left in FPA pool 0 (the Packet Data buffer pool).	IPD_QUE0_FREE_PAGE_CNT	Q0_PCNT	read only	read only
Global Variables				
Allow RAW packet Drop: Set to 1 to allow IPD to drop RAW packets based on Per-Port Packet Drop or Per-QoS RED algorithm.	PIP_PRT_CFG*n* (one per port)	RAWDRP	0	0 (H/W Default)
Pass-Drop Probability Calculation Delay: The number of core-clock cycles to wait before calculating the new packet drop probability for each QoS level. The interval is ([PRB_DLY + 68] x 8) cycles.	IPD_RED_PORT_ENABLE	PRB_DLY	0	0 (H/W Default) See Note1, Note 2
Per-QoS Variables				
Port Enable for ports (0-35): Any bit that is set enables the port's ability to have packets dropped by Per-QoS RED.	IPD_RED_PORT_ENABLE (for ports 0-35)	PRT_ENB	0	0 (H/W Default)
Port Enable for ports (36-39) (loopback ports): Any bit that is set enables the port's ability to have packets dropped by Per-QoS RED.	IPD_RED_PORT_ENABLE2 (for loopback ports)	PRT_ENB	0	0 (H/W Default)
Select snapshot or moving average: Per-QoS variable. Set to 1 to use snapshot value instead of moving average (recommended).	IPD_RED_QUE(0-7)_PARAM (one per QoS queue)	USE_PCNT	0	0 (H/W Default) See Note3
Pass Watermark (Threshold): Per-QoS variable: Packets will be passed if the queue size is greater than this value.	IPD_QOS(0-7)_RED_MARKS (one per QoS queue)	PASS	0	0 (H/W Default)
Drop Watermark (Threshold): Per-QoS variable: Packets will be dropped if the queue size is equal to or less than this value.	IPD_QOS(0-7)_RED_MARKS (one per QoS queue)	DROP	0	0 (H/W Default)
Probability Constant: This value is used in calculating the probability of a packet being passed or dropped by the RED engine. Set this to: (255ul<<24) / (PASS - DROP).	IPD_RED_QUE(0-7)_PARAM (one per QoS queue)	PRB_CON	0	0 (H/W Default) See Note4, Note5
Interrupts				
Interrupt: Global interrupt: Packet dropped due to Per-QOS RED	PIP_INT_REG	PKTDRP	0	0 (H/W Default)
Enable interrupt: Global interrupt enable: Packet dropped due to Per-QOS RED	PIP_INT_EN	PKTDRP	0	0 (H/W Default)

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Notes				
Note1: Use the SDK functions `cvmx_helper_setup_red()` and `cvmx_helper_setup_red_queue()` to configure this mechanism. First call `cvmx_helper_setup_red()`, then call `cvmx_helper_setup_red_queue()` to adjust queue priorities to weighted RED.				
Note2: We recommend a value of 0 for `PRB_DLY` so the calculation occurs as frequently as possible.				
Note3: We recommend `USE_PCNT` be set to 1 to use the snapshot value.				
Note4: (Although the hardware could be used to calculate the `PRB_CON` value from `DROP` and `PASS`, this has not been implemented.)				
Note5: "`CON`" stands for "CONSTANT"				

Note the following variables are ignored when using a snapshot value: `IPD_RED_PORT_ENABLE[AVG_DLY]`, `IPD_RED_QUE`*q*`_PARAM[NEW_CON, AVG_CON]`.

12.5.2 More Complex: Moving Average

A more complex option involves calculating a moving average based on periodic snapshots. The moving average can be weighted either toward the new snapshot value or the prior moving average, depending on the degree of responsiveness needed. As of SDK 2.0, the SDK does not provide a function to configure this option.

Caveats: When using the moving average, buffer exhaustion can occur if traffic is accepted but there are not enough buffers to accommodate it. Traffic can also be dropped unnecessarily when the moving average is lower than the actual value. This situation can occur because:

- The moving average is not as precise as the snapshot value.
- IPD prefetches about 100 packets at a time, causing the number of buffers to fluctuate even in steady-state traffic.

When using the moving average option, the variables `IPD_RED_PORT_ENABLE[AVG_DLY]`, `IPD_RED_QUE`*q*`_PARAM[NEW_CON, AVG_CON]` are used.

- `AVG_DLY` – controls how often the moving average is recalculated
- `NEW_CON` – a high value weights the moving average toward the most recent snapshot value
- `AVG_CON` – a high value weights the moving average toward the current average value

`NEW_CON + AVG_CON` must == `256`.

See the *HRM* for more detail.

Note the registers in Table 34 – "Registers to Configure Per-QoS RED/WRED – Snapshot" are also used to configure the Per-QoS RED/WRED using the moving average. In addition to those registers, the following registers are required.

Table 35: Registers to Configure Per-QoS RED/WRED – Moving Average

Brief Description	Register	Fields	H/W Default Value	SDK Default Value
Global Variables				
Moving Average Calculation Delay: The number of core-clock cycles to wait before calculating the new packet drop probability for each QoS level.	IPD_RED_PORT_ENABLE	AVG_DLY	0	0 (H/W Default)
Per-QoS Variables				
Select snapshot or moving average: Per-QoS variable. Set to 1 to use snapshot value instead of moving average (recommended). The interval is ([AVG_DLY + 10] x 8) cycles.	IPD_RED_QUE(0-7)_PARAM (one per QoS queue)	USE_PCNT	0	0 (H/W Default)
New Snapshot Weight: When calculating the moving average, a higher value will place more weight on the new snapshot value than the current moving average value. NEW_CON + AVG_CON must = 256.	IPD_RED_QUE(0-7)_PARAM (one per QoS queue)	NEW_CON	0	0 (H/W Default)
Moving Average Weight: When calculating the moving average, a higher value will place more weight on the current moving average value than the snapshot value. NEW_CON + AVG_CON must = 256.	IPD_RED_QUE(0-7)_PARAM (one per QoS queue)	AVG_CON	0	0 (H/W Default)
Notes				
Note: The registers used for the Per-QoS RED/WRED also apply. These registers are used in addition to those, with the exception of the USE_PCNT field which is 1 to use the snapshot and 0 to use the moving average.				
Note: See the *HRM* for details.				

12.6 Per-Port Congestion Control (Backpressure, Packet Drop) (PP-B, PP-PD)

The Per-Port Backpressure (PP-B) and Per-Port Packet Drop (PP-PD) are user-configurable congestion control mechanisms. Both mechanisms share many configuration registers.

Each port has a per-port in-use buffer counter, which usually is configured to count the number of Packet Data buffers currently used by the port. Each port also has a per-port in-use buffer limit (threshold) which is the limit on how many buffers it can use. Software is responsible for decrementing the in-use buffer counter when the buffer is freed, adding software overhead.

Per-Port Backpressure will cause IPD to backpressure the port when the per-port in-use buffer limit is reached.

Per-Port Packet Drop will cause IPD to drop all incoming packets on the port when the per-port in-use buffer limit is reached.

Each port's counter can be configured to count Packet Data Buffers (recommended), Work Queue Entry Buffers, or both.

There is both a global-enable and a per-port enable. Both need to be set for the mechanism to function on a specific port.

Both mechanisms are turned off by default when using the `cvmx_helper_setup_red()` function to configure the Per-QoS RED/WRED mechanism.

The threshold value must be configured carefully or the mechanism will not function as desired.

> ***Warning: If the threshold is set above the actual number of free buffers, the threshold will never be crossed (the actual count will always be below the threshold). For example, if the threshold was set to 2048 packets but the initial buffer count is only 1024 Packet Data Buffers, buffer exhaustion would occur before the threshold was reached. In this case, Per-Port Backpressure and/or Per-Port Packet Drop will never be turned on.***
>
> ***Setting the per-port thresholds becomes even more complicated when you have multiple ports. The sum of all the port's thresholds must be below the initial buffer count.***

Note: The per-port in-use buffer counters will wrap around if not decremented by software.

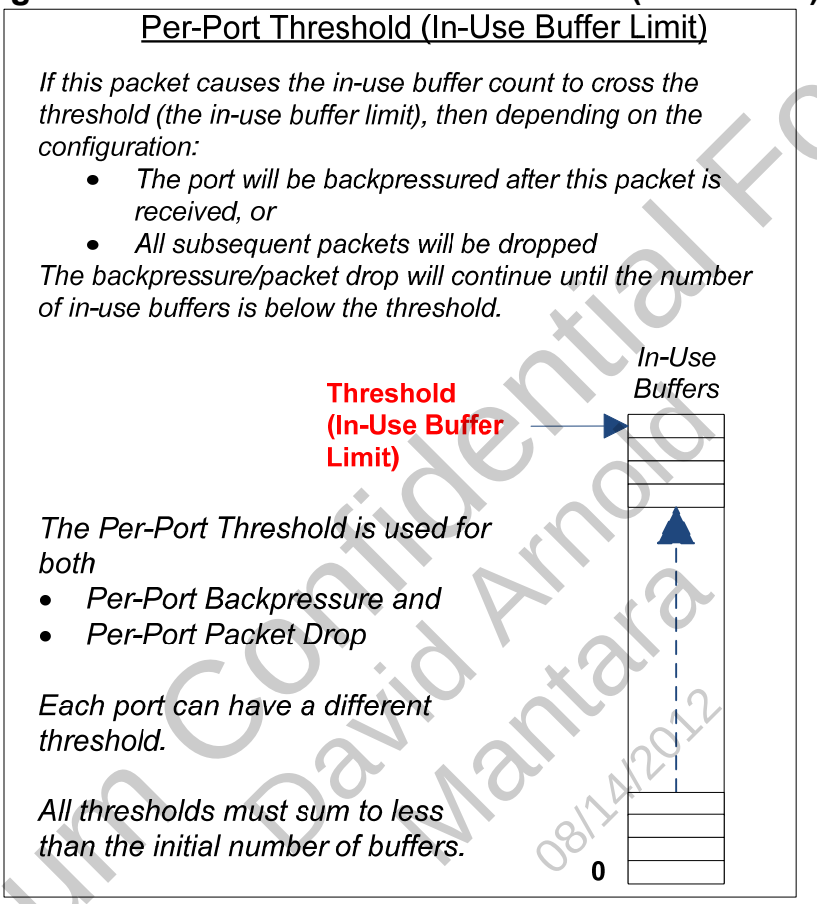
When using per port congestion control, decrement the per-port in-use buffer counter using the following code fragment BEFORE freeing the WQE (this code accesses fields in the WQE):

```
cvmx_ipd_sub_port_bp_page_cnt_t    page_cnt;

page_cnt.u64 = 0; // initialize all fields in the register to 0
page_cnt.s.page_cnt = -work->word2.s.bufs; // 2s complement
page_cnt.s.port = work->ipprt; // specify which port's counter

cvmx_write_csr(CVMX_IPD_SUB_PORT_BP_PAGE_CNT, page_cnt.u64);
```

Figure 46: Per-Port In-Use Buffer Limit (Threshold)



12.6.1 Per-Port Backpressure (PP-B)

Per-Port Backpressure is applied if the port's in-use buffer counter exceeds the port's in-use buffer limit. Per-Port Backpressure can be combined with Per-QoS RED/WRED or Per-Port RED: if the sender does not respond to backpressure, packets can be dropped to prevent buffer exhaustion. (See Section 17 – "Appendix D: A Note about Configuring GMX Backpressure" for a brief introduction to configuring Pause Frames for GMX. This is a hardware-level view.)

Figure 47: Congestion Control: Per-Port Backpressure

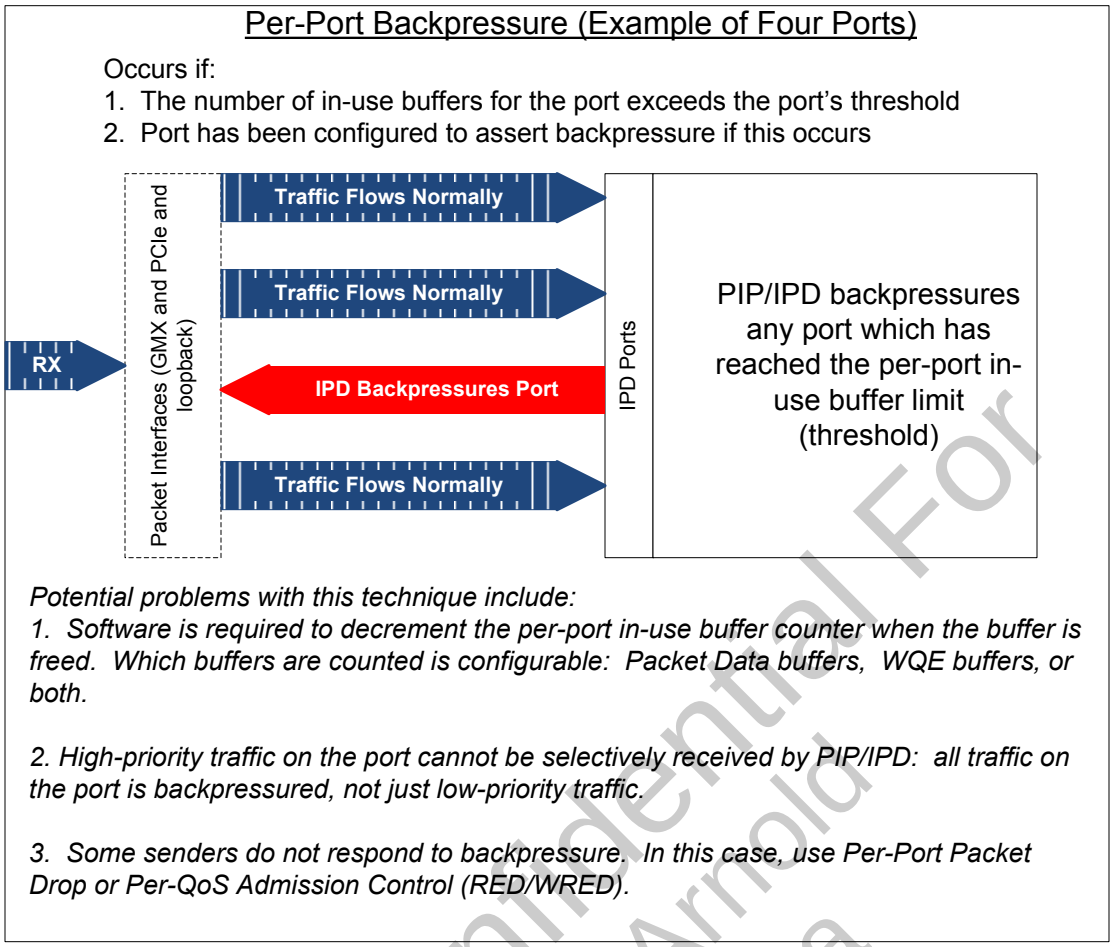


Table 36: Per-Port Backpressure Overview

Per-Port Backpressure Overview	
Action	Backpressure the port if the number of buffers in-use by the port (in-use buffer counter) exceeds the configurable per-port in-use buffer limit (threshold). Backpressure continues until the number of in-use buffers drops below the threshold.
Configuration Options	Each port's in-use buffer limit (threshold). The in-use buffer counter is also configurable: it can count the number of Packet Data Buffers (recommended), the number of Work Queue Entry Buffers (used if all packets are short enough to fit into a WQE), or both.
Based on	Each port's in-use buffer limit and in-use buffer counter.
Pros	If some ports are by definition higher priority than other ports, then this mechanism might be the right choice. It is possible to implement a lossless flow control system. Because many registers are used in common, it is easy to configure some ports for Per-Port Backpressure and others for Per-Port Packet Drop.
Cons	Software is responsible for decrementing the counter when the buffer is freed, adding software overhead. This method blocks all traffic on the port, including high-priority traffic. This mechanism only works if the sender responds to the backpressure. When configuring this option, verify that the sum of all in-use buffer limits can use does not exceed the total number of available buffers. The fixed limit may mean backpressure is applied on a port because the threshold is reached, but there are still available buffers.
Possible Configuration Errors	The sum of all in-use buffer limits must not exceed the total number of available buffers. Failing to decrement the per-port in-use buffer counter.

Table 37: Registers to Configure Per-Port Backpressure

Register	Details
Global Enable: Global per-port backpressure enable	
IPD_CTL_STATUS[PBP_EN] (See Note 1, Note2)	Global configuration register. Global backpressure enable for per-port backpressure and per-port packet drop functionality. (Reset value == 0).
Per-Port Enable: Individual per-port backpressure enable	
IPD_PORT*n*_BP_PAGE_CNT[BP_ENB] (one per port)	There is one register per port. Both the global and per-port enable must be on for the per-port packet drop functionality to be enabled. If this field is set to 1, the port's per-port backpressure is enabled. (Reset value == 0)

Register	Details
Threshold Configuration: Configure per-port in-use buffer limit	
RGMII/SPI-4/SGMII/XAUI (ports 0-31): IPD_PORT[0-31]_BP_PAGE_CNT[PAGE_CNT] **PCI/PCIe (ports 32-35):** IPD_PORT[32-35]_BP_PAGE_CNT[PAGE_CNT] **Loopback (ports 36-39):** IPD_PORT[36-39]_BP_PAGE_CNT2[PAGE_CNT] **sRIO (ports 40-43):** IPD_PORT[40-43]_BP_PAGE_CNT3[PAGE_CNT] (See Note 1)	There is one register per port. Maximum number of buffers (WQE and/or Packet Data Buffers, depending on configuration) which the port may use. The register IPD_PORT_BP_COUNTERS[2]_PAIR*n*[CNT_VAL] count the actual number of buffers in use by the port. When this number is exceeded, backpressure is applied to the port. Note that PAGE_CNT is in units of 256 buffers, so 1=256 buffers, 2=512 buffers. **Note that the threshold must be configured to be lower than the number of buffers in the FPA pool to prevent buffer exhaustion.** (Reset value == 0)
Configure Counter, Part 1: Configure per-port in-use buffer counter configuration for WQE buffers.	
IPD_CTL_STATUS[ADDPKT] (See Note 1)	Global configuration register. If set to 1, count the number of packets which have arrived on the port (the number of WQE buffers sent by the port to the SSO). (Reset value == 0)
Configure Counter, Part 2: Configure per-port in-use buffer counter configuration for Packet Data buffers.	
IPD_CTL_STATUS[NADDBUF]	Global configuration register. If set to 1, do NOT count the Packet Data buffers allocated by IPD for the port. (Reset value == 0)
Counter: Per-port in-use buffer count value	
RGMII/SPI-4/SGMII/XAUI (ports 0-31): IPD_PORT_BP_COUNTERS_PAIR[0-31][CNT_VAL] **PCI/PCIe (ports 32-35):** IPD_PORT_BP_COUNTERS_PAIR[32-35][CNT_VAL] **Loopback (ports 36-39):** IPD_PORT_BP_COUNTERS2_PAIR[36-39][CNT_VAL] **sRIO (ports 40-43):** IPD_PORT_BP_COUNTERS3_PAIR[40-43][CNT_VAL] (See Note 1)	There is one register per port. Automatic count of buffers in-use by the port. Depending on ADDPKT and NADDBUF configuration, will count one of: 1) All in-use Packet Data buffers 2) All in-use WQE buffers 3) Both in-use Packet Data buffers and WQE buffers (Reset value == 0)
Counter Decrement Selector: Specify Which Port's Counter to Decrement	
IPD_SUB_PORT_BP_PAGE_CNT[PORT] (See Note 1, Note 3)	There is one in-use buffer counter per port. This field specifies which counter to decrement. (Software decrements the in-use buffer count when the buffers are freed). (Reset value == 0)

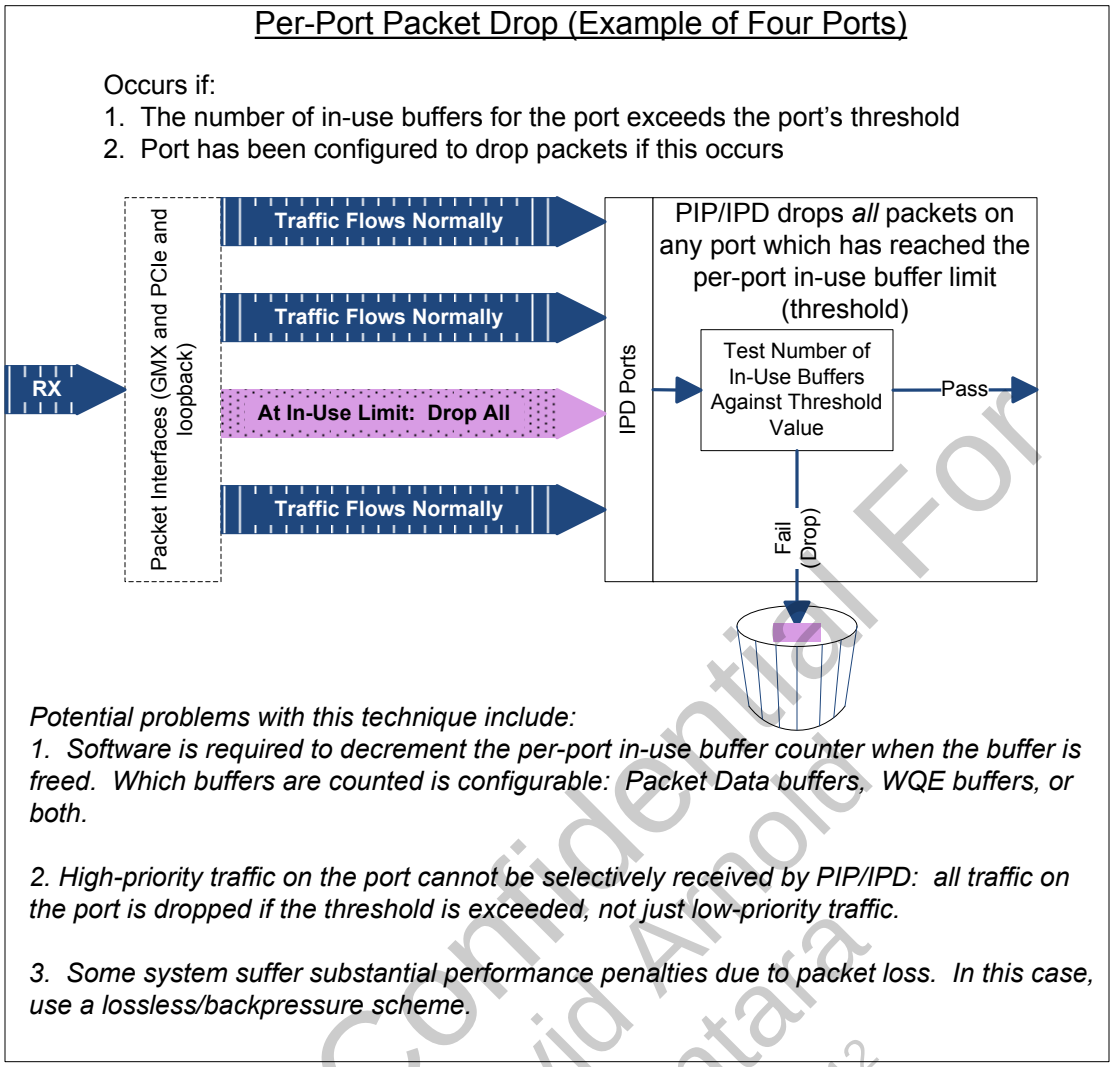
Register	Details
Counter Decrement: Decrement per-port in-use buffer count value	
IPD_SUB_PORT_BP_PAGE_CNT[PAGE_CNT] (See Note 1, Note 3)	There is one register per port. Software decrements the in-use buffer count when the buffers are freed via this register. Software writes the twos-complement value to be added to IPD_PORT*n*_BP_COUNTERS[2]_PAIR*n*[CNT_VAL]. The port is specified via IPD_SUB_PORT_BP_PAGE_CNT[PORT]. (Reset value == 0)
Notes	
Note1: Register(s) also used by the per-port drop mechanism.	
Note2: Key Issue Regarding IPD_CTL_STATUS[PBP_EN]: There is a known issue related to this bit for all CN3XXX and CN5XXX. This bit cannnot be programmed arbitrarily and actually must be transitioned from zero to one on a specific cycle. It should never be set to zero after being set once. Our SDK goes to great lengths to make sure IPD_CTL_STATUS[PBP_EN] is set a the right time to avoid the issue The cvmx-helper function does extensive cleanup on packet shutdown. Uboot sets this bit during early boot on some processors. To get reliable per-port backpressure, you must use the cvmx-helper functions for initialization and shutdown.	
Note 3: To subtract X from the counter, create a twos-complement of X (negative X). Store the twos-complement value in IPD_SUB_PORT_BP_PAGE_CNT[PAGE_CNT], which will add the twos-complement value to IPD_PORT_BP_COUNTERS_PAIRn[CNT_VAL], decrementing the counter. (A twos-complement number is created by taking the one's complement of the number (invert every bit in the binary number), then add 1 to the result.)	

Note: GMX_INF_MODE[EN] must be set to 1 for each packet interface that requires port backpressure prior to setting PBB_EN to 1. Once enabled, the sending of per-port backpressure cannot be disabled by changing the value of IPD_CTL_STATUS[PBP_EN]. See the *HRM* for more details.

12.6.2 Per-Port Packet Drop (PP-PD)

Per-Port Packet Drop is applied if the port's in-use buffer count exceeds the port's in-use buffer limit (threshold). This mechanism will drop all packets on the port until the in-use buffer count is below the threshold. Although the HRM calls this "RED", there is no randomness to the drop, so it is misnamed.

Figure 48: Congestion Control: Per-Port Packet Drop



Potential problems with this technique include:

1. Software is required to decrement the per-port in-use buffer counter when the buffer is freed. Which buffers are counted is configurable: Packet Data buffers, WQE buffers, or both.

2. High-priority traffic on the port cannot be selectively received by PIP/IPD: all traffic on the port is dropped if the threshold is exceeded, not just low-priority traffic.

3. Some system suffer substantial performance penalties due to packet loss. In this case, use a lossless/backpressure scheme.

Table 38: Per-Port Packet Drop Overview

Per-Port Packet Drop Overview	
Action	Drop *all* incoming packets on the port if the number of buffers in-use by the port (in-use buffer counter) exceeds the configurable per-port in-use buffer limit (threshold). Packet drop continues until the number of in-use buffers drops below the threshold.
Configuration Options	Each port's in-use buffer limit (threshold). The in-use buffer counter is also configurable: it can count the number of Packet Data Buffers (recommended), the number of Work Queue Entry Buffers (used if all packets are short enough to fit into a WQE), or both.
Based on	Each port's in-use buffer limit and in-use buffer counter.
Pros	If some ports are by definition higher priority than other ports, then this mechanism might be the right choice. Because many registers are used in common, it is easy to configure some ports for Per-Port Backpressure and others for Per-Port Packet Drop.
Cons	Software is responsible for decrementing the counter when the buffer is freed, adding software overhead. This method drops *all* traffic on the port, including high-priority traffic. When configuring this option, verify that the sum of all in-use buffer limits can use does not exceed the total number of available buffers. The fixed limit may mean backpressure is applied on a port because the threshold is reached, but there are still available buffers.
Possible Configuration Errors	The sum of all in-use buffer limits must not exceed the total number of available buffers. Failing to decrement the per-port in-use buffer counter.

Table 39: Registers to Configure Per-Port Packet Drop

Register	Details
Global Enable: Global per-port backpressure enable	
IPD_CTL_STATUS[PBP_EN] (See Note 1, **Note 2**)	Global configuration register. Global backpressure enable for per-port backpressure and per-port packet drop functionality. (Reset value == 0)
Per-Port Enable: Individual per-port packet drop enable	
IPD_BP_PRT_RED_END[PRT_ENB*n*] (one bit per port)	One configuration register, one bit per port. Both the global and per-port enable must be on for the per-port packet drop functionality to be enabled. Each port corresponds to a bit in this field. When the port's bit is set to 1, the per-port packet drop check for the port is enabled. (Reset value == 0)
Allow Raw Packet Drop: Optionally allow raw packets to be dropped by this mechanism	
PIP_PRT_CFGn[RAWDRP] (one per port)	Global configuration register. If set to 0, then RAWFULL and RAWSCH packets are never dropped by per-port packet drop. (Reset value == 0)
Threshold Configuration: Configure per-port in-use buffer counter	
RGMII/SPI-4/SGMII/XAUI (ports 0-31): IPD_PORT[0-31]_BP_PAGE_CNT[PAGE_CNT] **PCI/PCIe (ports 32-35):** IPD_PORT[32-35]_BP_PAGE_CNT[PAGE_CNT] **Loopback (ports 36-39):** IPD_PORT[36-39]_BP_PAGE_CNT**2**[PAGE_CNT] **sRIO (ports 40-43):** IPD_PORT[40-43]_BP_PAGE_CNT**3**[PAGE_CNT] (See Note 1)	There is one register per port. Maximum number of buffers (WQE and/or Packet Data Buffers, depending on configuration) which the port may use. The register IPD_PORT_BP_COUNTERS[2]_PAIR*n*[CNT_VAL] count the actual number of buffers in use by the port. When this number is exceeded, packets arriving on this port are dropped. Note that PAGE_CNT is in units of 256 buffers, so 1=256 buffers, 2=512 buffers. **Note that the threshold must be configured to be lower than the number of buffers in the FPA pool to prevent buffer exhaustion.** (Reset value == 0)
Configure Counter, Part 1: Configure per-port in-use buffer counter configuration for WQE buffers.	
IPD_CTL_STATUS[ADDPKT] (See Note 1)	Global configuration register. If set to 1, count the number of packets which have arrived on the port (the number of WQE buffers sent by the port to the SSO). (Reset value == 0)
Configure Counter, Part 2: Configure per-port in-use buffer counter configuration for Packet Data buffers.	
IPD_CTL_STATUS[NADDBUF]	Global configuration register. If set to 1, do NOT count the Packet Data buffers allocated by IPD for the port. (Reset value == 0)

Register	Details
Counter: Per-port in-use buffer count value	
RGMII/SPI-4/SGMII/XAUI (ports 0-31): IPD_PORT_BP_COUNTERS_PAIR[0-31][CNT_VAL] **PCI/PCIe (ports 32-35):** IPD_PORT_BP_COUNTERS_PAIR[32-35][CNT_VAL] **Loopback (ports 36-39):** IPD_PORT_BP_COUNTERS2_PAIR[36-39][CNT_VAL] **sRIO (ports 40-43):** IPD_PORT_BP_COUNTERS3_PAIR[40-43][CNT_VAL] (See Note 1)	There is one register per port. Maximum number of buffers (WQE and/or Packet Data Buffers, depending on configuration) which the port may use. The register IPD_PORT_BP_COUNTERS[2]_PAIR*n*[CNT_VAL] count the actual number of buffers in use by the port. When this number is exceeded, backpressure is applied to the port. Note that PAGE_CNT is in units of 256 buffers, so 1=256 buffers, 2=512 buffers. **Note that the threshold must be configured to be lower than the number of buffers in the FPA pool to prevent buffer exhaustion.** (Reset value == 0)
Counter Decrement Selector: Specify Which Port's Counter to Decrement	
IPD_SUB_PORT_BP_PAGE_CNT[PORT] (See Note 1, Note 3)	There is one in-use buffer counter per port. This field specifies which counter to decrement. (Software decrements the in-use buffer count when the buffers are freed). (Reset value == 0)
Counter decrement: Decrement per-port in-use buffer count value	
IPD_SUB_PORT_BP_PAGE_CNT[PAGE_CNT] (See Note 1, Note 3)	There is one register per port. Software decrements the in-use buffer count when the buffers are freed via this register. Software writes the twos-complement value to be added to IPD_PORT*n*_BP_COUNTERS[2]_PAIR*n*[CNT_VAL]. The port is specified via IPD_SUB_PORT_BP_PAGE_CNT[PORT]. (Reset value == 0)
Notes	
Note1: Register(s) also used by the per-port backpressure mechanism.	
Note2: Key Issue Regarding IPD_CTL_STATUS[PBP_EN]: There is a known issue related to this bit for all CN3XXX and CN5XXX. This bit cannnot be programmed arbitrarily and actually must be transitioned from zero to one on a specific cycle. It should never be set to zero after being set once. Our SDK goes to great lengths to make sure IPD_CTL_STATUS[PBP_EN] is set a the right time to avoid the issue The cvmx-helper function does extensive cleanup on packet shutdown. Uboot sets this bit during early boot on some processors. To get reliable per-port backpressure, you must use the cvmx-helper functions for initialization and shutdown.	

Register	Details
Note 3: To subtract X from the counter, create a twos-complement of X (negative X). Store the twos-complement value in `IPD_SUB_PORT_BP_PAGE_CNT[PAGE_CNT]`, which will add the twos-complement value to `IPD_PORT_BP_COUNTERS_PAIRn[CNT_VAL]`, decrementing the counter. (A twos-complement number is created by taking the one's complement of the number (invert every bit in the binary number), then add 1 to the result.)	

12.7 Per-Port RED

Per-Port RED allows packets to be randomly dropped on a per-port basis rather than per-QoS. There is no unique mechanism to support this, but if there are less than 8 ports used in the system (the same as the maximum QoS queues), then by setting the default QoS for the port to the port number, each port's traffic will go to a different QoS. By setting each QoS to the *same* threshold value, all ports will have an equal chance to receive packets. Because the per-QoS mechanism uses random drop, per-port RED will have been implemented.

This feature can also be combined with Per-Port Backpressure.

13 Per QoS/Port Buffer Tracking

This feature is listed in the *HRM* as "Per-Port and QoS Threshold Interrupts". In this feature, the PIP/IPD hardware maintains a counter per port/QoS that PIP/IPD increments when it sends a packet to the SSO. There is one counter for each port/QoS combination of port number (up to 16) and the SSO QoS level (0-7). PIP/IPD maintains 128 counters (16 ports times 8 QoS levels), corresponding interrupt bits and enables, and corresponding watermarks (one for each QoS level and port combination).

This feature is only used in special cases and is not documented here.

14 Appendix A: PIP/IPD Registers and Register Fields

The registers and fields contained in this chapter are specifically for CN54/55/56/57XX. Many of these registers and register fields are identical on the other processors.

This chapter does not include register fields used for BIST (power on memory test), reset, enable, Per QoS/Port Buffer Tracking registers, or registers or fields reserved for internal use.

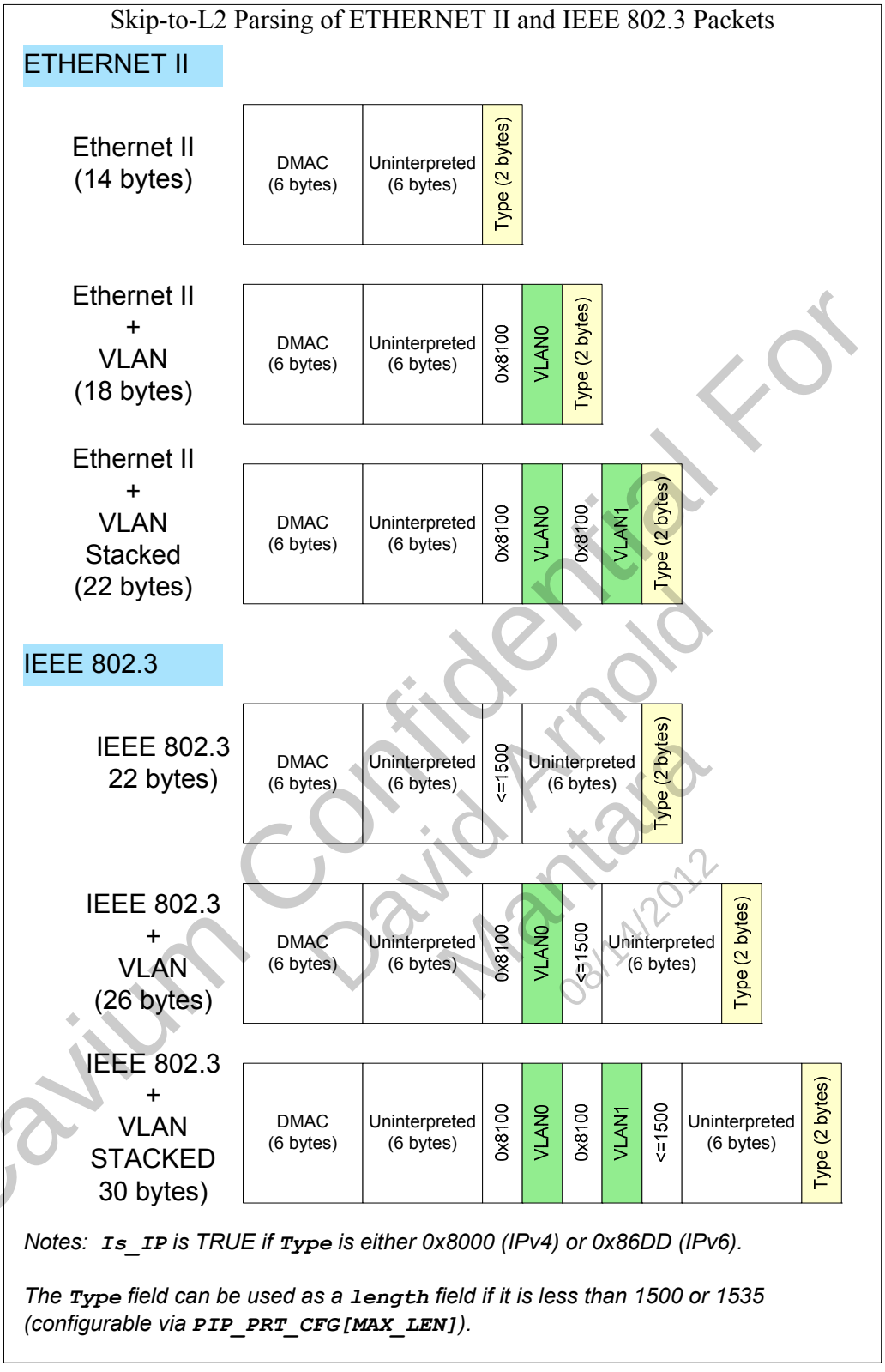
Registers are divided into different tables by purpose. The following links will help locate the tables in the chapter:

15 Appendix B: Industry-Standard Reference Information

These industry-standard data structures are provided here for quick reference.

15.1 L2 Header Formats

Figure 49: L2 Header Formats



Notes: ***`Is_IP`*** *is TRUE if* ***`Type`*** *is either 0x8000 (IPv4) or 0x86DD (IPv6).*

The ***`Type`*** *field can be used as a* ***`length`*** *field if it is less than 1500 or 1535 (configurable via* ***`PIP_PRT_CFG[MAX_LEN]`****).*

15.1.1 L2 Header `Type` Field Values (EtherType)

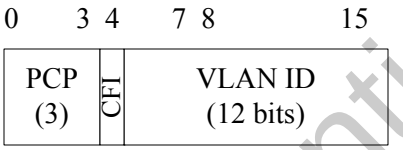
Table 40: L2 Header `Type` Field Values (EtherType)

Type	Value
ARP	0x0806
IPv4	0x0800
IPv6	0x86DD
MAC_CTRL	0x8808
RARP	0x0835
VLAN_ID	0x8100

15.1.2 L2 Header VLAN, VLAN 1 Field Details

Figure 50: L2 Header and VLAN, VLAN1 Field Details – CFI, VLAN ID

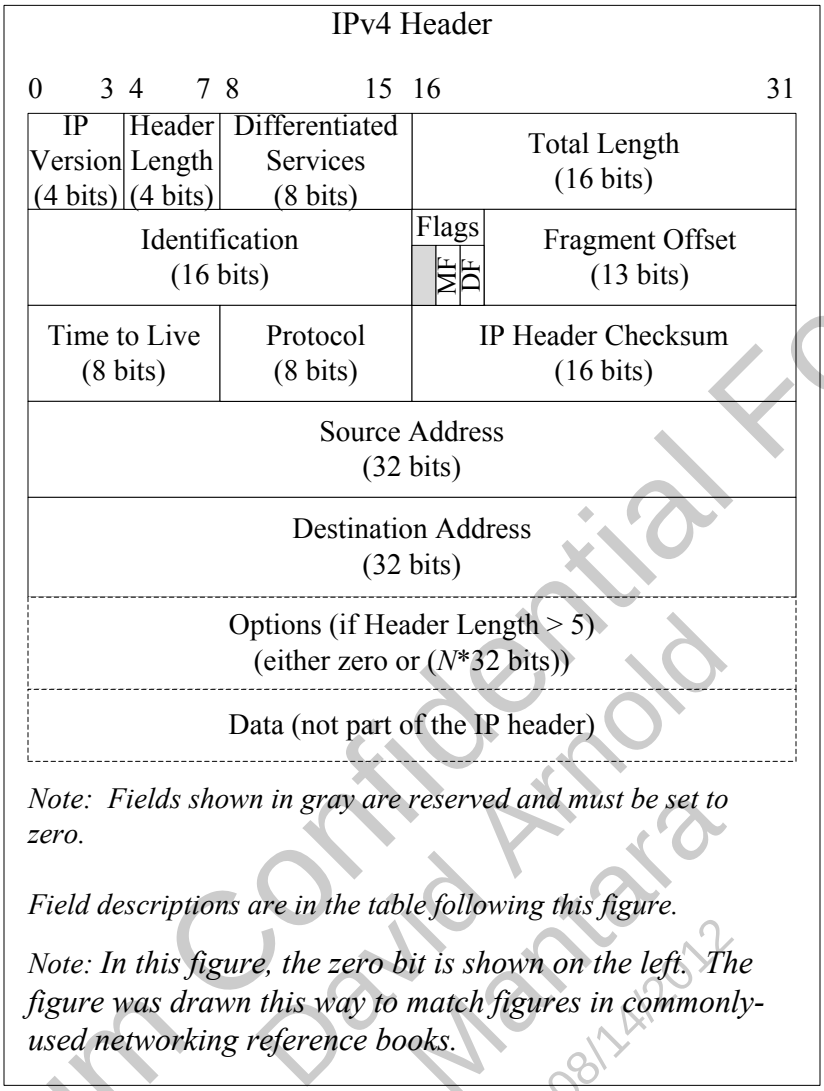
L2 Header VLAN and VLAN1 Field Formats



PPC: Priority Point Code. This is the IEEE 802.1p priority, a number from 0-7 (0=lowest, 7=highest). This number indicates the packet's priority.
CFI: Canonical Format Indicator: If 1, MAC address is in non-canonical format. If 1, MAC address is in canonical format.
VLAN ID: Identifies the VLAN the frame belongs to. A value of 0 means the frame doesn't belong to any VLAN.

15.2 L3: IPv4 Header

Figure 51: IPv4 Header



Note: Fields shown in gray are reserved and must be set to zero.

Field descriptions are in the table following this figure.

Note: In this figure, the zero bit is shown on the left. The figure was drawn this way to match figures in commonly-used networking reference books.

Table 41: IPv4 Header Fields

Field	Description
Version	Version Number: For IPv4, the version number is four.
Header Length	Header Length: The number of 32-bit words in the header. If header length is 5 (20 bytes), there are no options. The minimum value is 5.
Differentiated Services	Diffserve Value: (Originally Type of Service (TOS)), now is Differentiated Services (diffserv). This is used to specify a packet-handling preference, such as low delay or high reliability.
Total Length	Total Datagram Length: Size of header plus data, in bytes. The minimum value is 20 bytes (header, and zero data).

Field	Description
Identification	Identification: Optional, and primarily used to uniquely identify each fragment of an original datagram.
Flags	Flags: DF (Don't Fragment): If the DF bit is set and the datagram must be fragmented to be routed, then it is dropped. MF (More Fragments): If the packet is not fragmented, the MF bit is set to zero. If the packet is fragmented, all fragments except the last fragment have this bit set to one. The last fragment has the bit set to zero. Note: The third flag is reserved and must be set to zero.
Fragment Offset	Fragment Offset: The offset of the fragment relative to the beginning of the original unfragmented datagram. This value is specified in eight-byte blocks. The first fragment will have an offset of zero.
Time to Live	Time to Live: Used to prevent a datagram from going in circles on the Internet forever. Also considered to be a "hop count": each switch or router which handles the packet decrements the TTL field by 1. When TTL equals zero, the packet is discarded.
Protocol	Protocol: The format of the data portion, such as TCP or UDP.
Header Checksum	Header Checksum: The checksum of the IP header. Note that the data is part of the IP header, and therefore is not included in the checksum.
Source Address	Source Address: The IPv4 address of the sender. Note this may not be the true address, if network address translation is used.
Destination Address	Destination Address: The IPv4 address the receiver. Note this may not be a true address if network address translation is used.
Options	Options: These are rarely used. They must be padded out to make 32-bit words.
Data	Data: Not part of the header, and not included in the IP header checksum. The format of the contents is specified in the Protocol field.

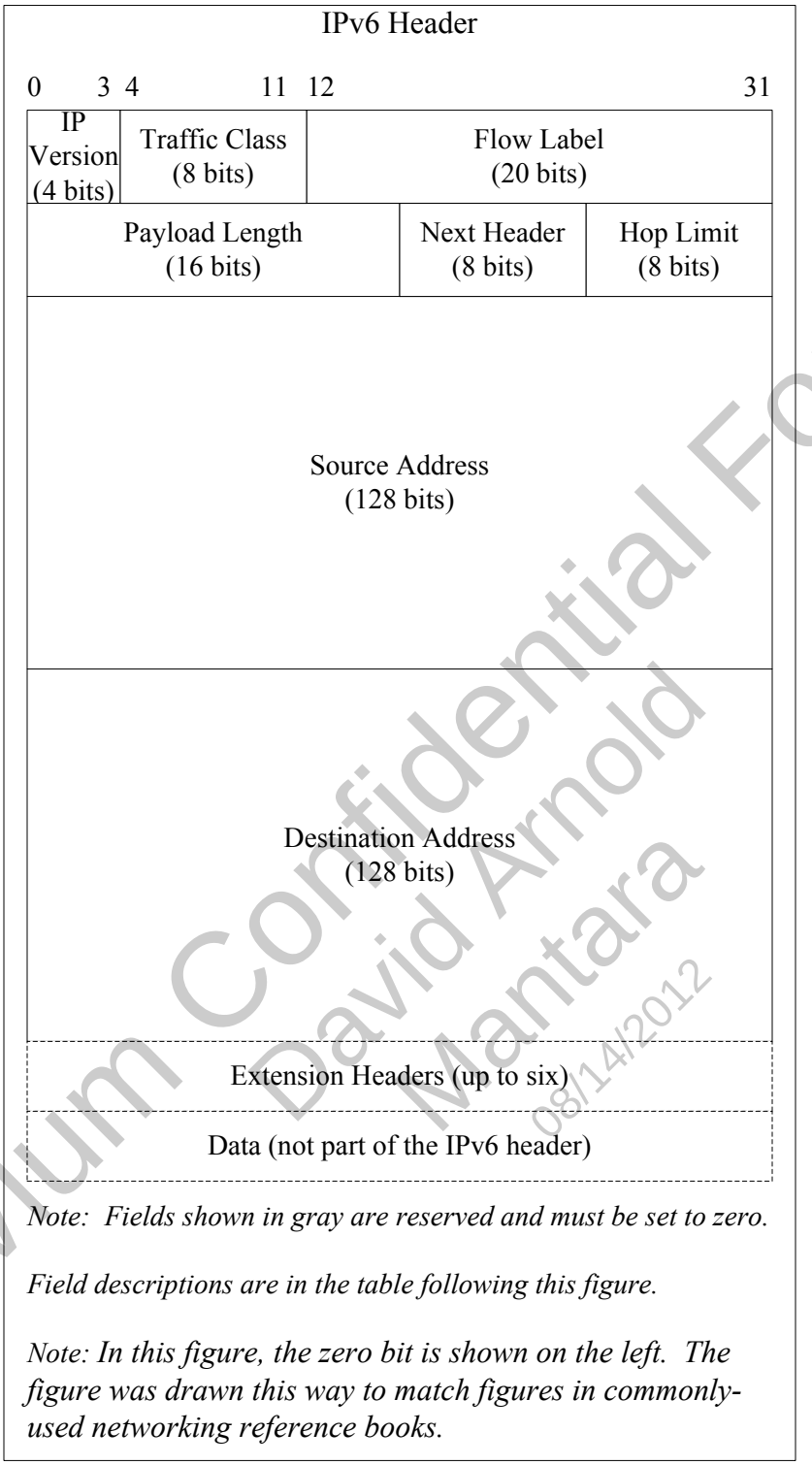
15.2.1 IPv4 `Protocol` Field Values

Table 42: IPv4 Protocols

IPv4 Protocols	
HOP_BY_HOP	0
TCP	6
UDP	17
ROUTING	43
FRAG	44
IPSEC_ESP	50
IPSEC_AH	51
ICMP	58
DESTINATION	60
IPCOMP	108
OTHER	255

15.3 L3: IPv6 Header

Figure 52: IPv6 Header



Note: Fields shown in gray are reserved and must be set to zero.

Field descriptions are in the table following this figure.

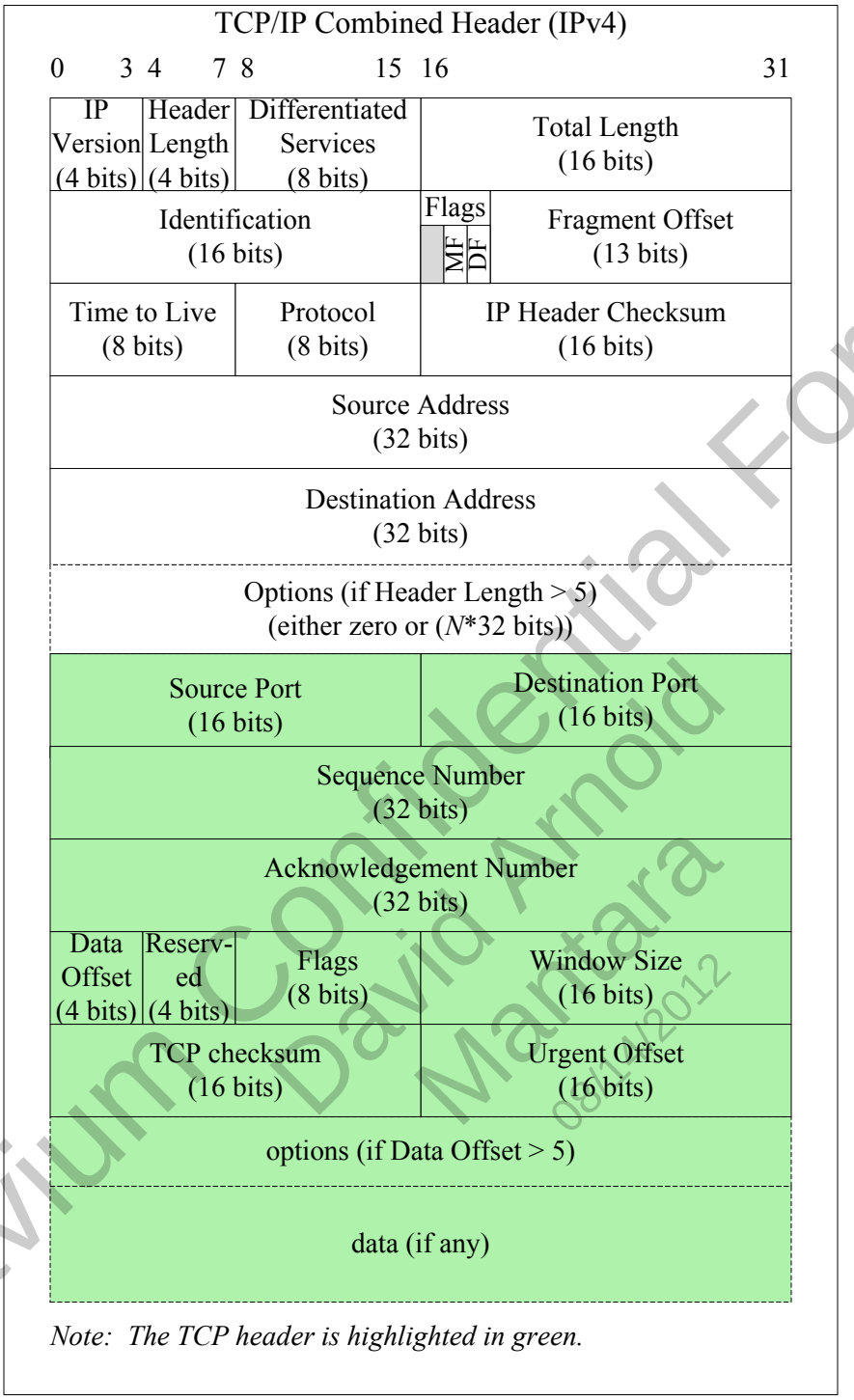
Note: In this figure, the zero bit is shown on the left. The figure was drawn this way to match figures in commonly-used networking reference books.

Table 43: IPv6 Header Fields

Field	Description
Version	Version Number: For IPv6, the version number is six.
Traffic Class	Traffic Class: The number of 32-bit words in the header. If header length is 5 (20 bytes), there are no options. The minimum value is 5.
Flow Label	Flow Label: (Originally Type of Service (TOS)), now is Differentiated Services (diffserv). This is used to specify a packet-handling preference, such as low delay or high reliability.
Payload Length	Payload Length: Size of header plus data, in bytes. The minimum value is 20 bytes (header, and zero data).
Next Header	Next Header: Optional, and primarily used to uniquely identify each fragment of an original datagram.
Hop Limit	Hop Limit:
Source Address	Source Address: The IPv6 address of the sender. Note this may not be the true address, if network address translation is used.
Destination Address	Destination Address: The IPv6 address the receiver. Note this may not be a true address if network address translation is used.
Extension Headers	Extension Headers: These are rarely used. They must be padded out to make 32-bit words.
Data	Data: Not part of the header, and not included in the IP header checksum. The format of the contents is specified in the Protocol field.

15.4 L4: TCP Header

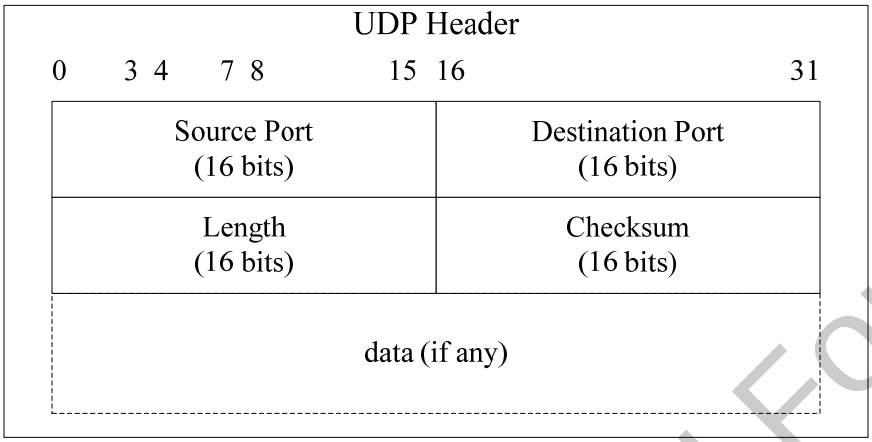
Figure 53: IPv4 Header with TCP/IP



Note: The TCP header is highlighted in green.

15.5 L4: UDP Header

Figure 54: UDP Header



16 Appendix C: Input Packet Parsing Details

The *HRM* includes parsing pseudo code. The following figures show the different options and a parsing flow chart. The "Cases" in these figures match the cases shown in the WQE WORD2 figures. This figure is identical to the figure shown earlier in this chapter.

Figure 55: Input Packet Parsing Cases

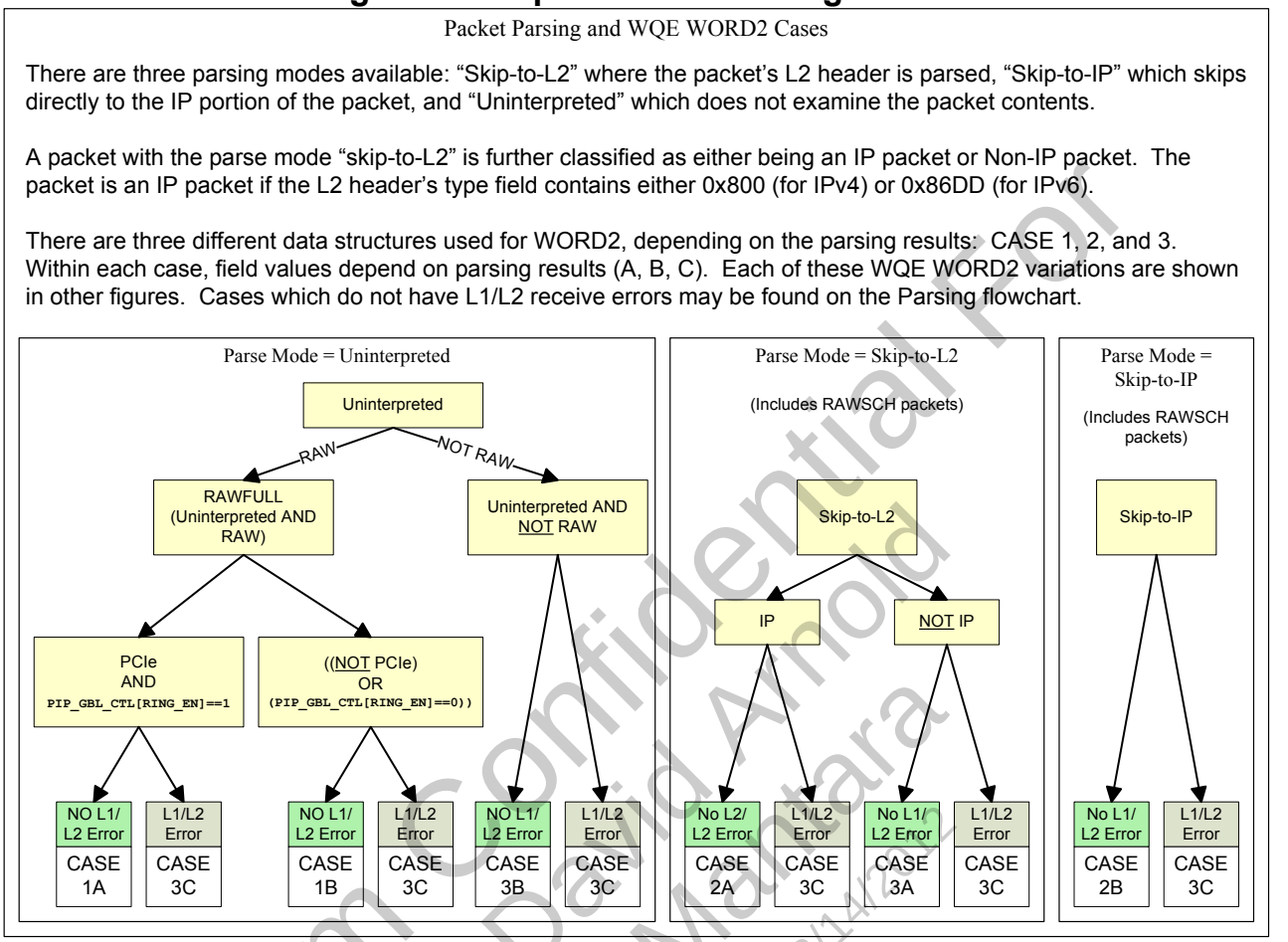


Figure 56: Input Packet Parsing Flowchart, Part 1

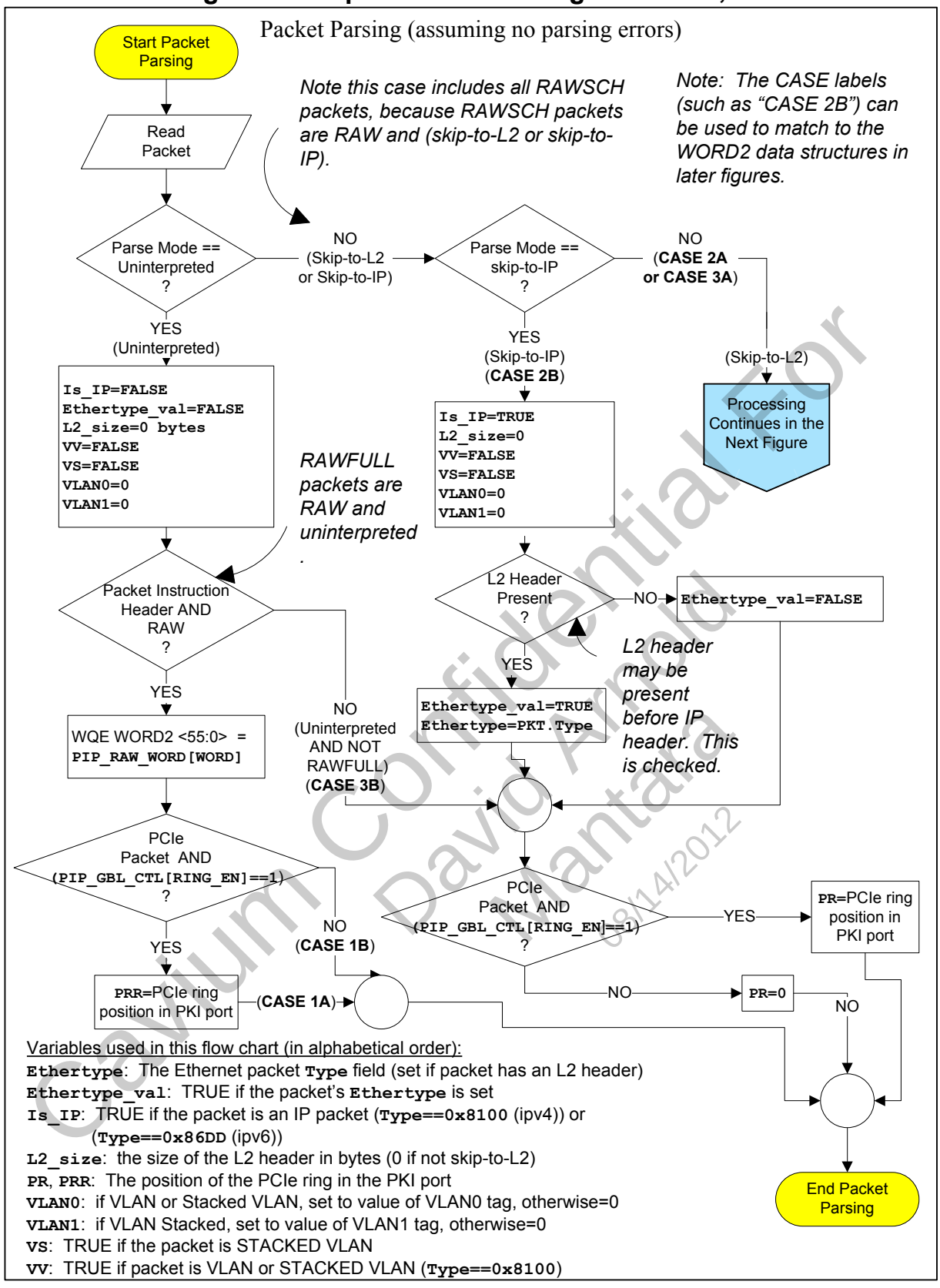
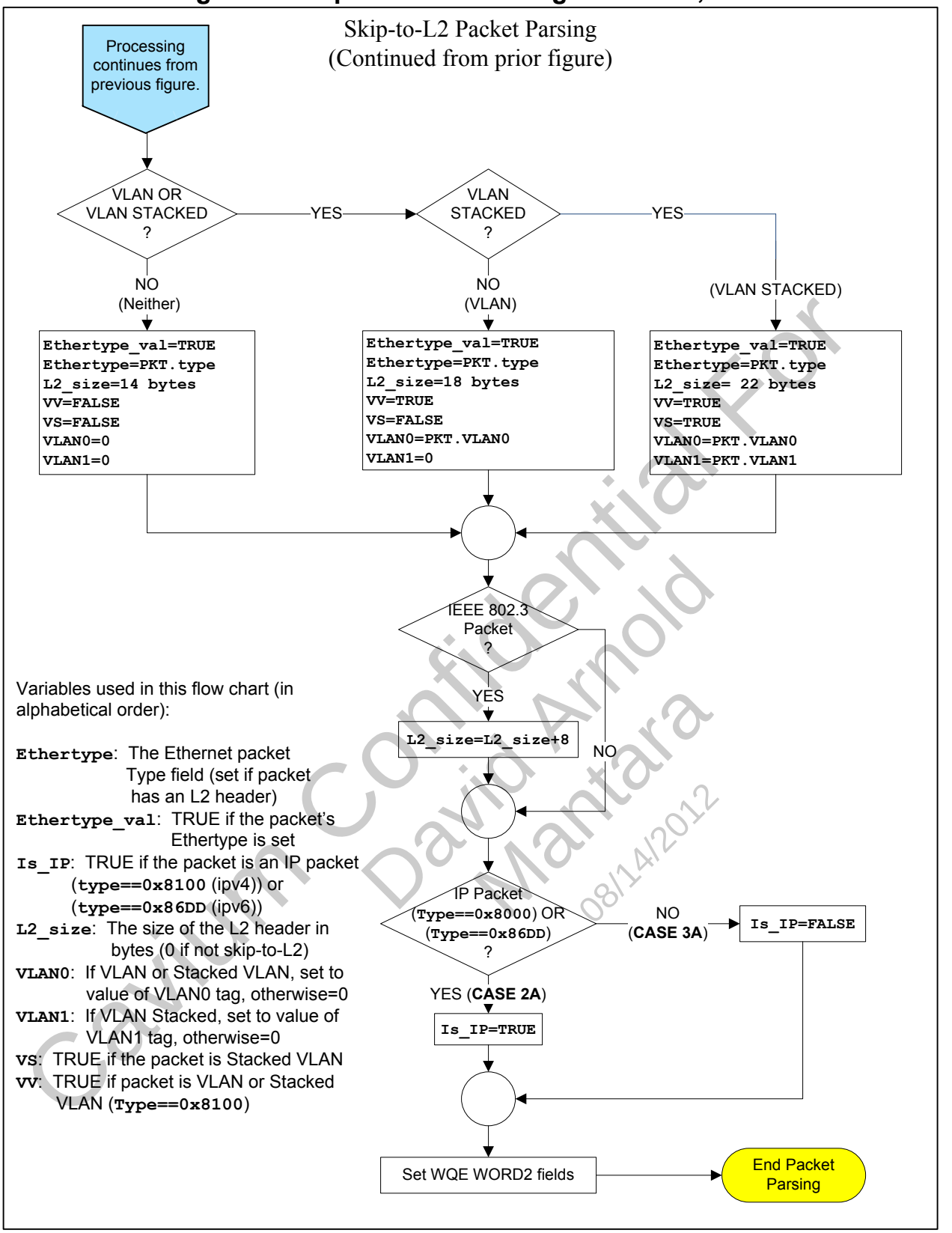


Figure 57: Input Packet Parsing Flowchart, Part 2



17 Appendix D: A Note about Configuring GMX Backpressure

This section is included in this chapter for reference only. It is not intended to be a complete discussion of how backpressure is implemented at the MAC layer. See the *HRM* for more information.

The GMX registers that control the pause frames are introduced in the table below. (These are CN54/55/56/57 registers.)

Table 44: Overview of GMX Registers Used to Configure Backpressure

Register	Description
GMX*n*_TX_OVR_BP[EN]	Global enable. Set to 1 to turn backpressure on.
GMX*n*_TX_OVR_BP[BP]	Per-Port enable. Set bit corresponding to port to turn backpressure on for the port. The global EN bit must be 1 for per-port backpressure to work.
GMXn_TX*n*_PAUSE_PKT_TIME[TIME]	The pause_time field placed in out bound 802.3 pause packets
GMX*n*_TX*n*_PAUSE_PKT_INTERVAL[INTERVAL]	How often the pause packet is sent
GMX*n*_TX*n*_PAUSE_ZERO[SEND]	If this variable is set to 1, the bus can be used more efficiently. CN54/55/56/57 autogenerates a pause packet with a pause time of 0 when flow-control deasserts, which can allow the remote transmitter to restart more quickly.
GMX*n*_TX_STAT9[CTL]	Number of Control packets generated by hardware
GMX*n*_TX_PAUSE_PKT_DMAC[DMAC]	The DMAC field placed is outbound pause packets
GMX*n*_TX_PAUSE_PKT_TYPE[TYPE]	The TYPE field placed is outbound pause packets
Notes	
Note: *n* can be either 0 or 1.	
Note: To find these registers in the HRM, use the search string of the form "GMX0/1_TX0_PAUSE_PKT" (to find them in a block of text) or "GMX0_TX0_PAUSE_PKT" (to find them in the registers description section).	

Choosing proper values of GMX_TX_PAUSE_PKT_TIME[TIME] and GMX_TX_PAUSE_PKT_INTERVAL[INTERVAL] can be challenging for the system designer. It is suggested that TIME be much greater than INTERVAL and GMX_TX_PAUSE_ZERO[SEND] be set. This allows a periodic refresh of the PAUSE count and then when the backpressure condition is lifted, a PAUSE packet with TIME==0 will be sent indicating that the OCTEON processor is ready for additional data.

If the system chooses to not set GMX_TX_PAUSE_ZERO[SEND], then it is suggested that TIME and INTERVAL are programmed such that they satisfy the following rule:

```
INTERVAL <= (TIME - (largest_pkt_size + IFG + pause_pkt_size))
```

where:

- largest_pkt_size is that largest packet that the system can send (normally 1518B)
- IFG is the Inter Frame Gap
- pause_pkt_size is the size of the PAUSE packet (normally 64B)

18 Appendix E: Example Code (linux-filter)

In the linux-filter example, the Ethernet driver configures PIP/IPD. (In the code example, the term POW is used. POW is another term for SSO.) Linux filter modifies the group for the IPD port, and gets all incoming packets from that port.

Figure 58: Linux-Filter

`linux-filter`: Forwarding a Packet to the Control Plane

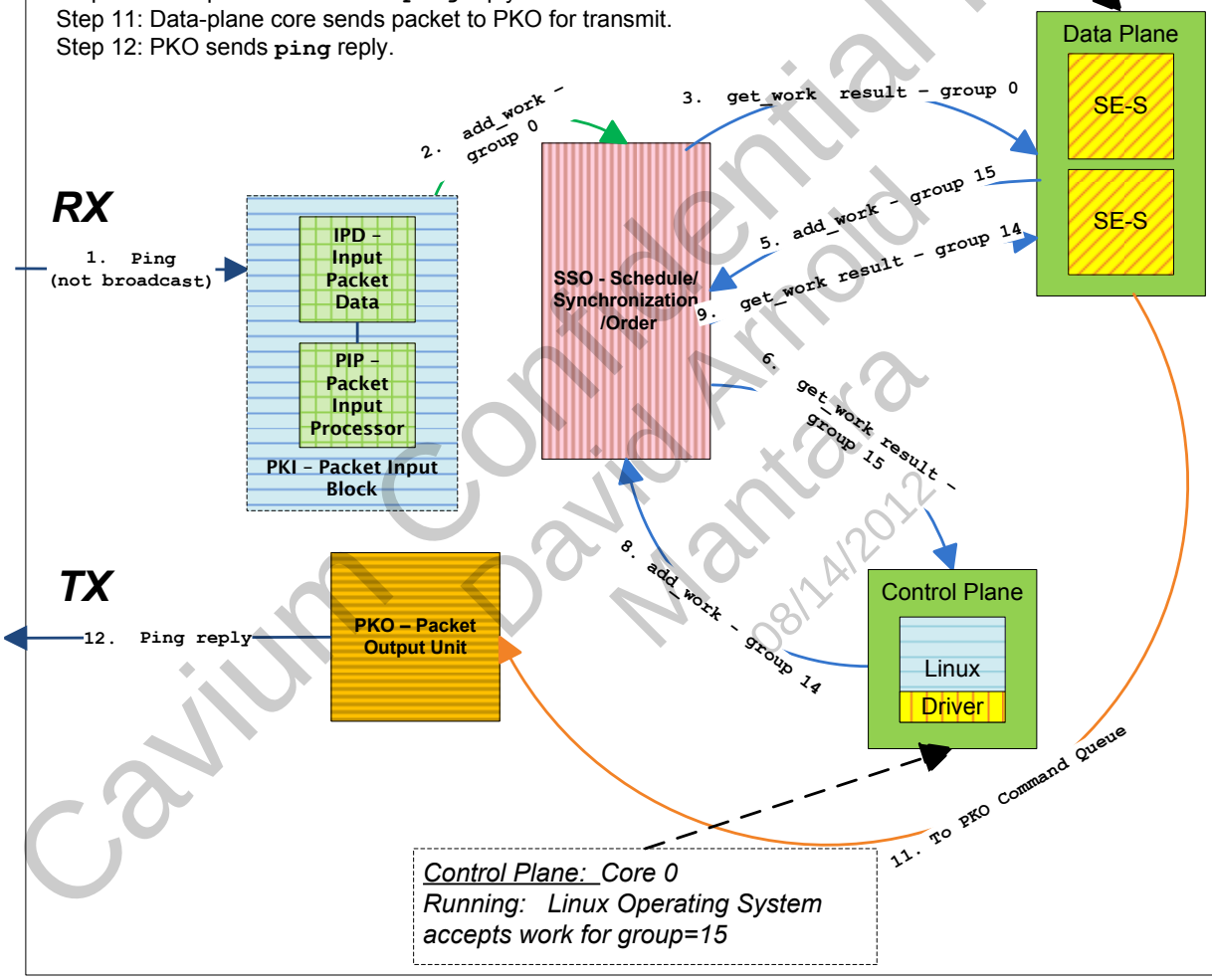
The hardware units and data plane cores perform a large amount of packet processing without requiring any action from the cores running the control plane. Processing is shown in steps 1-12, below.

OCTEON HOST

Step 1: **`ping`** packet is received by PIP/IPD.
Step 2: PIP/IPD sets group to 0 sends the packet to the SSO.
Step 3: Data-plane core calls **`get_work()`** and receives the packet.
Step 4: Data-plane core tests the packet: Test if packet ((broadcast) && (IP)) = FALSE. Send packet to Linux.
Step 5: Data-plane core changes the group to 15 and sends the packet to the SSO
Step 6: Control-plane core calls **`get_work()`** and receives the packet.
Step 7: Control-plane core processes the **`ping`** request and replies, using Group 14
Step 8: Control-plane core sends the packet to the SSO.
Step 9: Data-plane calls **`get_work()`** and receives the packet.
Step 10: Data-plane receives the **`ping`** reply
Step 11: Data-plane core sends packet to PKO for transmit.
Step 12: PKO sends **`ping`** reply.

Data Plane: Cores 1 and 2 Running **`linux-filter`** *as a SE-S application: accepts work for groups 0 and 14*

Note that either data-plane core can handle the packet processing.



Control Plane: Core 0 Running: Linux Operating System accepts work for group=15

The following code is from the `linux-filter` example:

Set up the definitions:

```
/* This POW group ID is used for packets destined to the Linux kernel.  This
Group ID must match the kernel Ethernet driver's pow_receive_group parameter */
#define TO_LINUX_GROUP          15 // send work to linux kernel

/* This POW group ID is used by the Linux kernel for egress packets.  This
group ID must match the Ethernet driver's pow_send_group parameter */
#define FROM_LINUX_GROUP        14 // get work from linux kernel

/* This POW group ID is used for ingress packets that must be intercepted by
cores running the linux-filter Simple Executive application.  Packets from the
intercept_port are assigned to this POW group instead of the normal ethernet
pow_receive_group */
#define FROM_INPUT_PORT_GROUP   0   // get all work from input port (group 0)

/* Packets ingressed on the intercept_port are intercepted by linux-filter and
processed.  Packets received from the kernel Ethernet virtual POW0 device are
sent out this port. */
CVMX_SHARED int intercept_port = 0;  // this value is board-dependent
```

At `main()`, the SE core responsible for SE-specific initialization waits for the IPD initialization (performed by a core running Linux) to be complete:

```
/* Have one core do the hardware initialization */
    if (cvmx_coremask_first_core(sysinfo->core_mask))
    {
        printf("\n\nLoad the Linux ethernet driver with:\n"
               "\t $ modprobe cavium-ethernet pow_send_group=%d
pow_receive_group=%d\n",
               FROM_LINUX_GROUP, TO_LINUX_GROUP);

        printf("Waiting for ethernet module to complete
initialization...\n\n\n");
        cvmx_ipd_ctl_status_t ipd_reg;
        do
        {
            ipd_reg.u64 = cvmx_read_csr(CVMX_IPD_CTL_STATUS);
        } while (!ipd_reg.s.ipd_en);
<code omitted>
```

Make sure all incoming traffic on IPD port 0 (`intercept_port`) sets Group value to zero (the value of the `FROM_INPUT_PORT_GROUP`):

```
/* Change the group for only the port we're interested in */
        cvmx_pip_port_tag_cfg_t tag_config;
        // load data structure with current values
        tag_config.u64 = cvmx_read_csr(CVMX_PIP_PRT_TAGX(intercept_port));
        if (tag_config.s.grp == TO_LINUX_GROUP)  // not the desired value
        {
            tag_config.s.grp = FROM_INPUT_PORT_GROUP;
            // change group for this port to 0 (FROM_INPUT_PORT_GROUP)
            cvmx_write_csr(CVMX_PIP_PRT_TAGX(intercept_port), tag_config.u64);
        }
```

Get ready to receive group 0 (FROM_INPUT_PORT_GROUP) and group 14:
(FROM_LINUX_GROUP) work:

```
/* Accept any packet except for the ones destined to the Linux group */
    cvmx_pow_set_group_mask(cvmx_get_core_num(),
                            (1<<FROM_INPUT_PORT_GROUP)|(1<<FROM_LINUX_GROUP));
<code omitted>
```

In the following code segment, note the error check, and buffer free:

```
  while (1)
  {
#ifdef __linux__
        /* Under Linux there better thing to do than halt the CPU waiting for
            work to show up. Here we use NO_WAIT so we can continue processing
            instead of stalling for work */
        cvmx_wqe_t *work = cvmx_pow_work_request_sync(CVMX_POW_NO_WAIT);
        if (work == NULL)
        {
            /* Yield to other processes since we don't have anything to do */
            usleep(0);
            continue;
        }
#else
        /* In standalone CVMX, we have nothing to do if there isn't work, so
            use the WAIT flag to reduce power usage */
        cvmx_wqe_t *work = cvmx_pow_work_request_sync(CVMX_POW_WAIT);
        if (work == NULL)
            continue;
#endif

        /* Check for errored packets, and drop.  If sender does not respond to
            backpressure or backpressure is not sent, packets may be truncated
            if the GMX fifo overflows. */
        if (work->word2.s.rcv_error) <<< receive error
        {
            /* Work has error, so drop */
            cvmx_helper_free_packet_data(work);  << free Packet Data buffer
            cvmx_fpa_free(work, CVMX_FPA_WQE_POOL, 0); << free WQE buffer
            continue;
        }

        /* See if we should filter this packet */
        if (is_filtered_packet(work))
        {
            printf("Received %u byte packet. Filtered.\n", work->len);
            cvmx_helper_free_packet_data(work); << free Packet Data buffer
            cvmx_fpa_free(work, CVMX_FPA_WQE_POOL, 0); << free WQE buffer
        }
        else if (work->grp == FROM_LINUX_GROUP)
        {
<code omitted>
```

19 Appendix F: Input Port Configuration

This section contains port configuration information for additional processors.

For more information about CN54XX, CN55XX, CN56XX, and CN57XX, see Section 3 – "IPD Input Ports".

Links to the different figures in this section:

Figure 59: Input Ports: CN3005

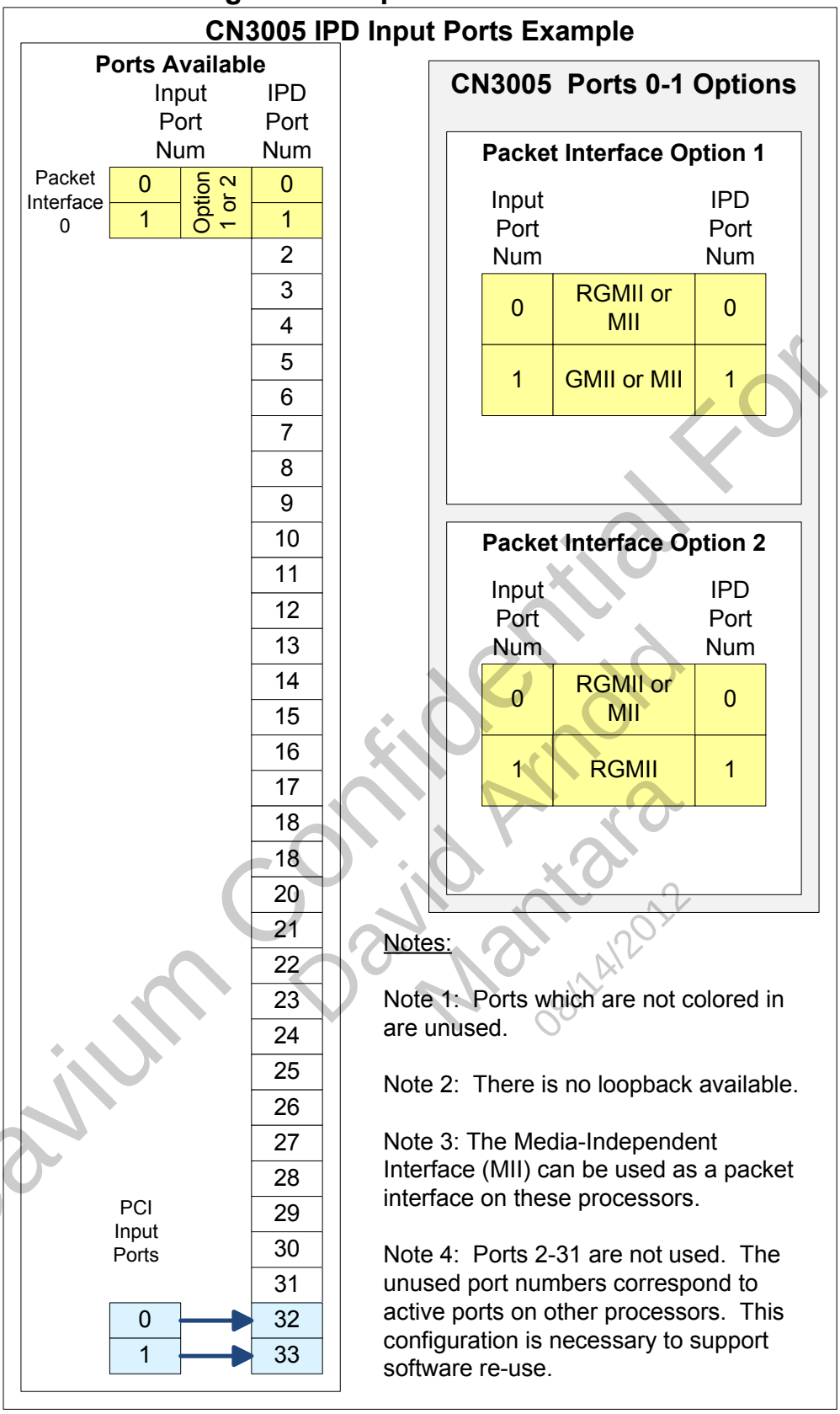


Figure 60: Input Ports: CN3010

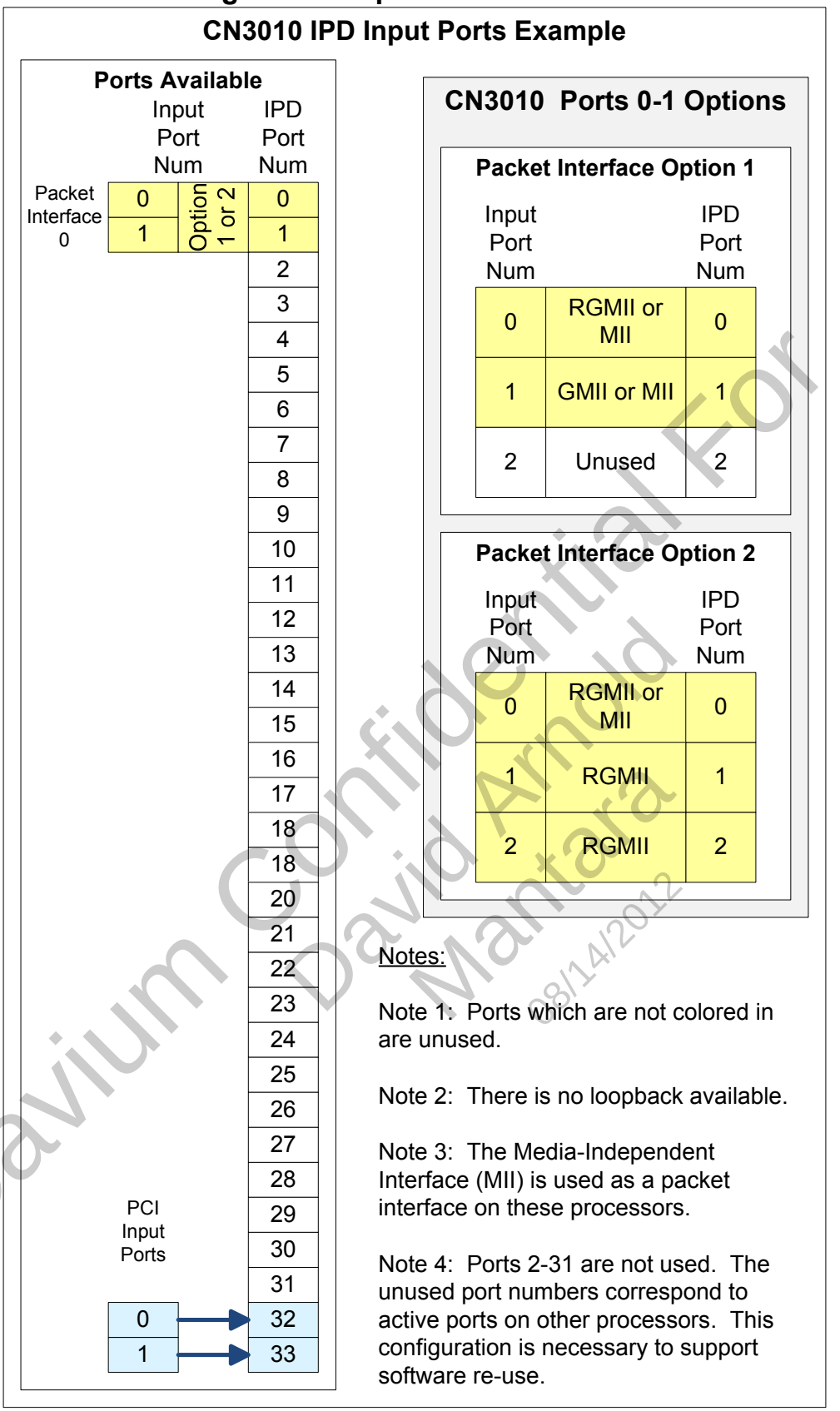


Figure 61: Input Ports: CN3020

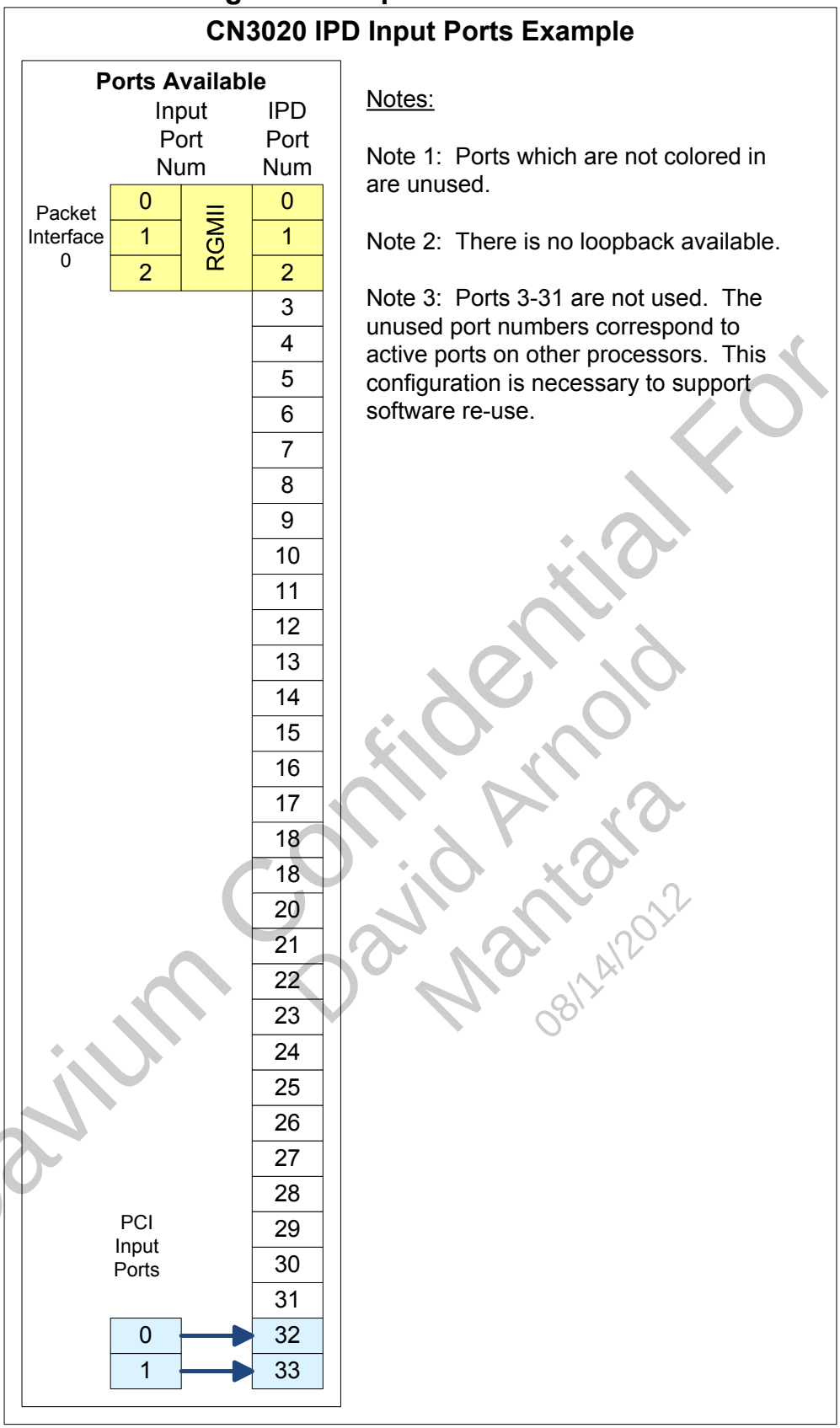


Figure 62: Input Ports: CN31XX

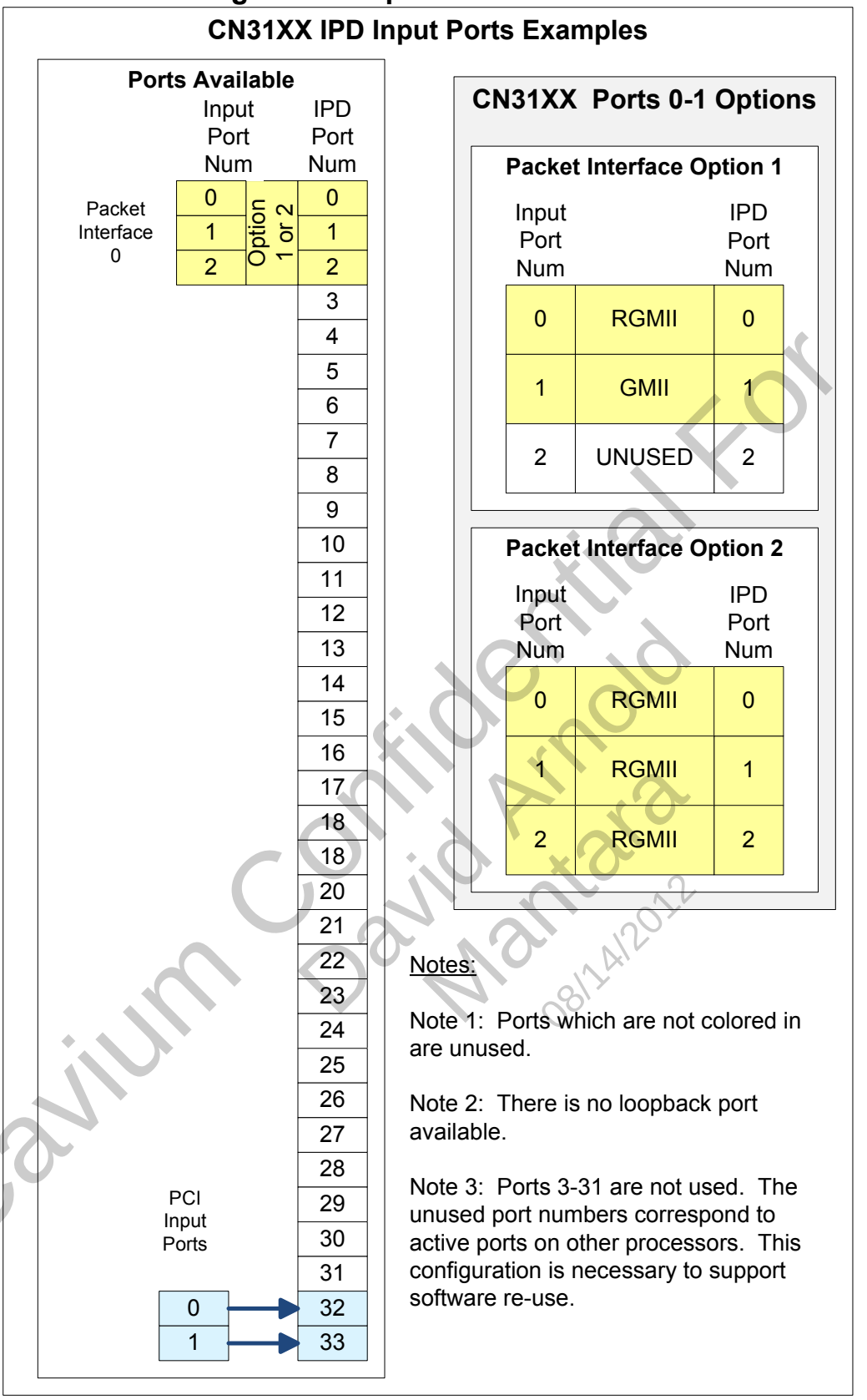
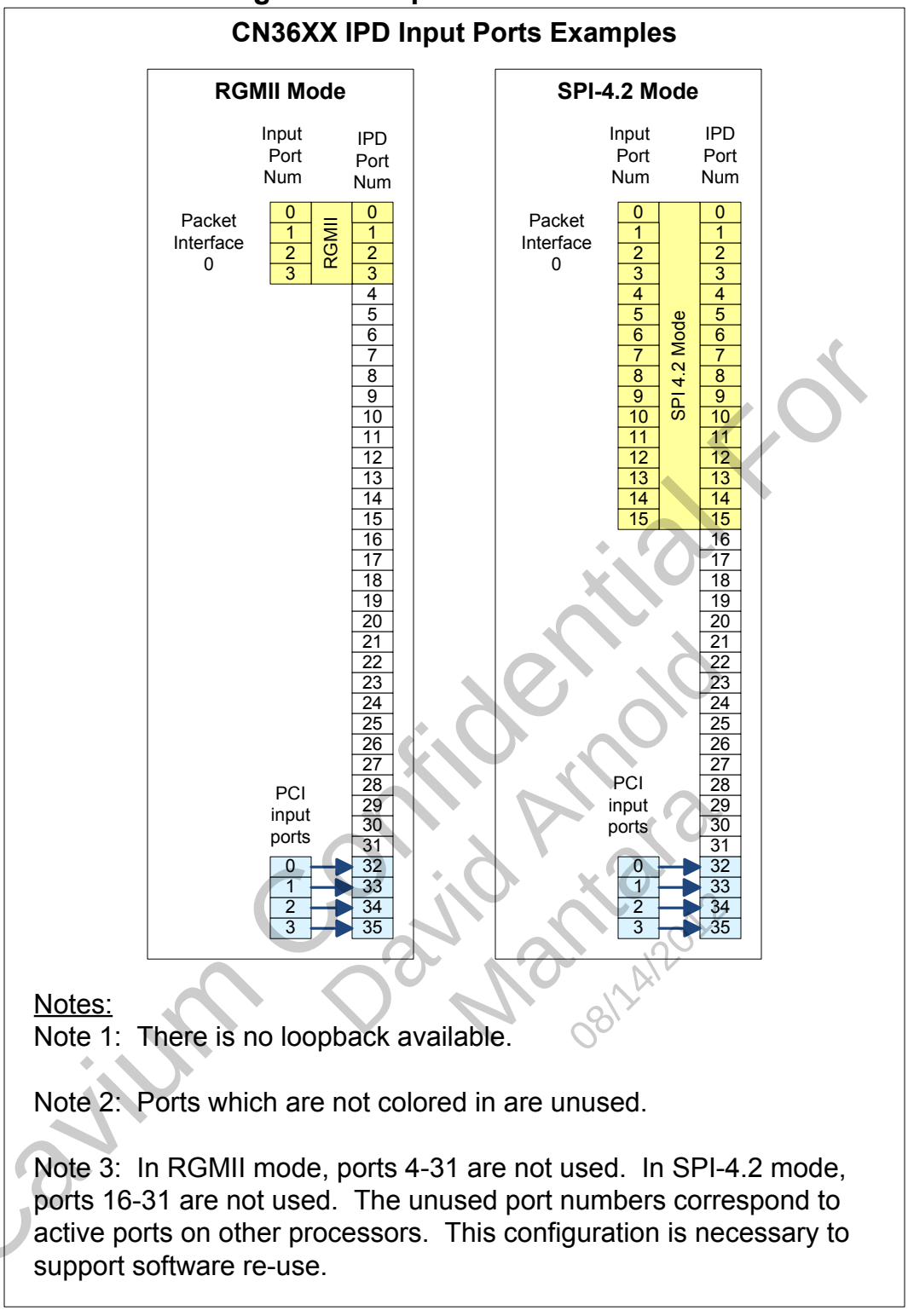


Figure 63: Input Ports: CN36XX



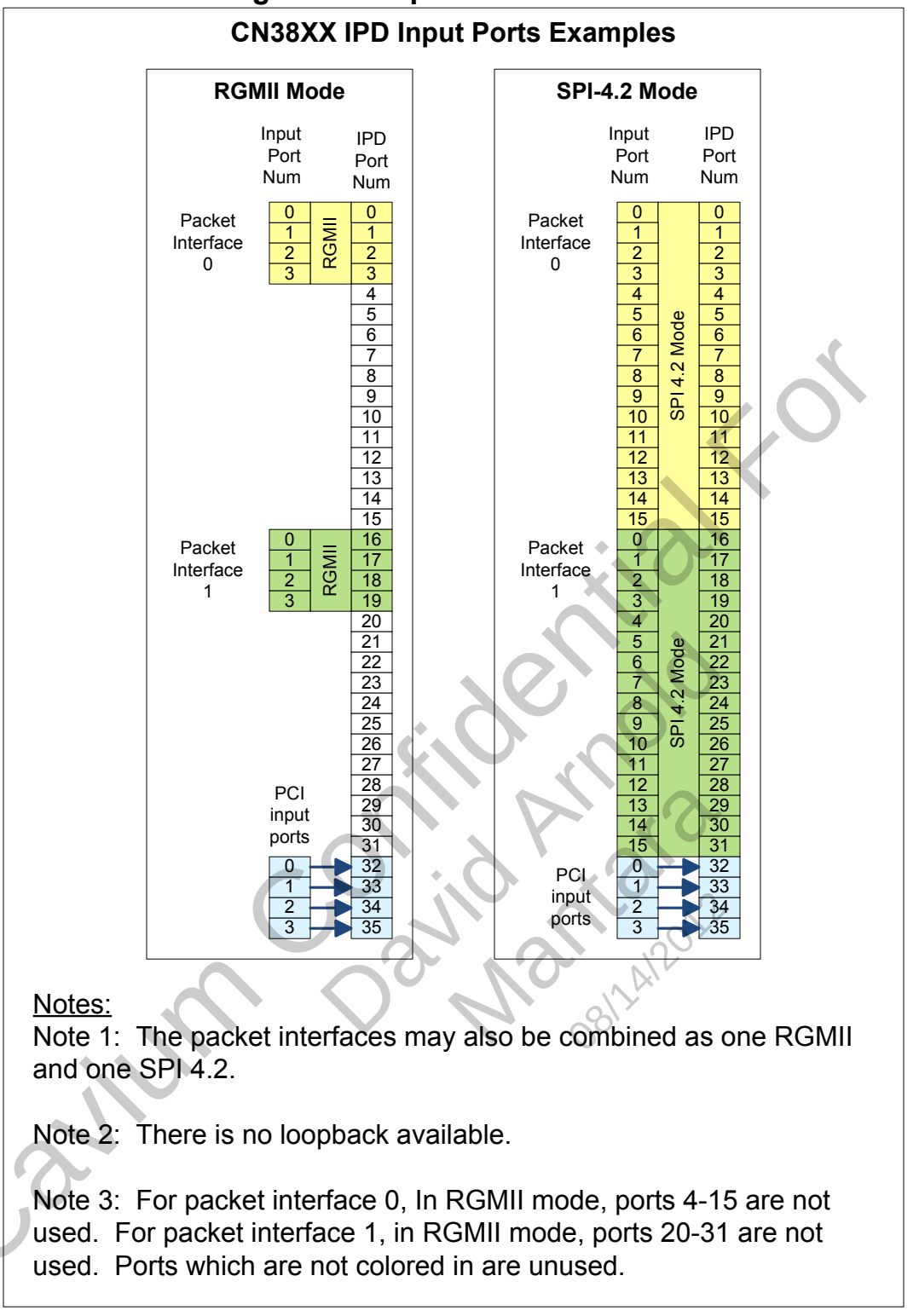
Notes:

Note 1: There is no loopback available.

Note 2: Ports which are not colored in are unused.

Note 3: In RGMII mode, ports 4-31 are not used. In SPI-4.2 mode, ports 16-31 are not used. The unused port numbers correspond to active ports on other processors. This configuration is necessary to support software re-use.

Figure 64: Input Ports: CN38XX



Notes:

Note 1: The packet interfaces may also be combined as one RGMII and one SPI 4.2.

Note 2: There is no loopback available.

Note 3: For packet interface 0, In RGMII mode, ports 4-15 are not used. For packet interface 1, in RGMII mode, ports 20-31 are not used. Ports which are not colored in are unused.

Figure 65: Input Ports: CN50XX

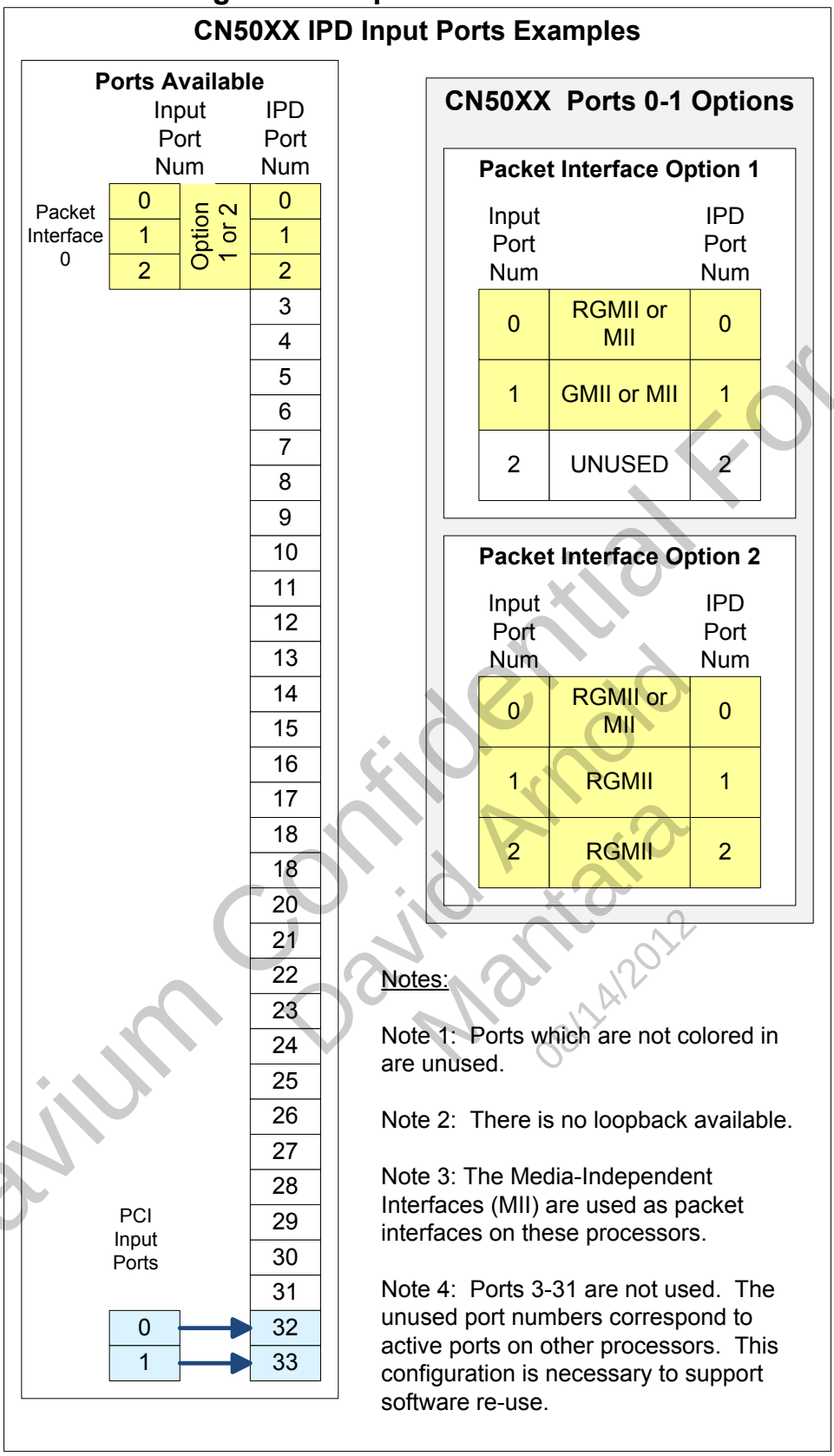
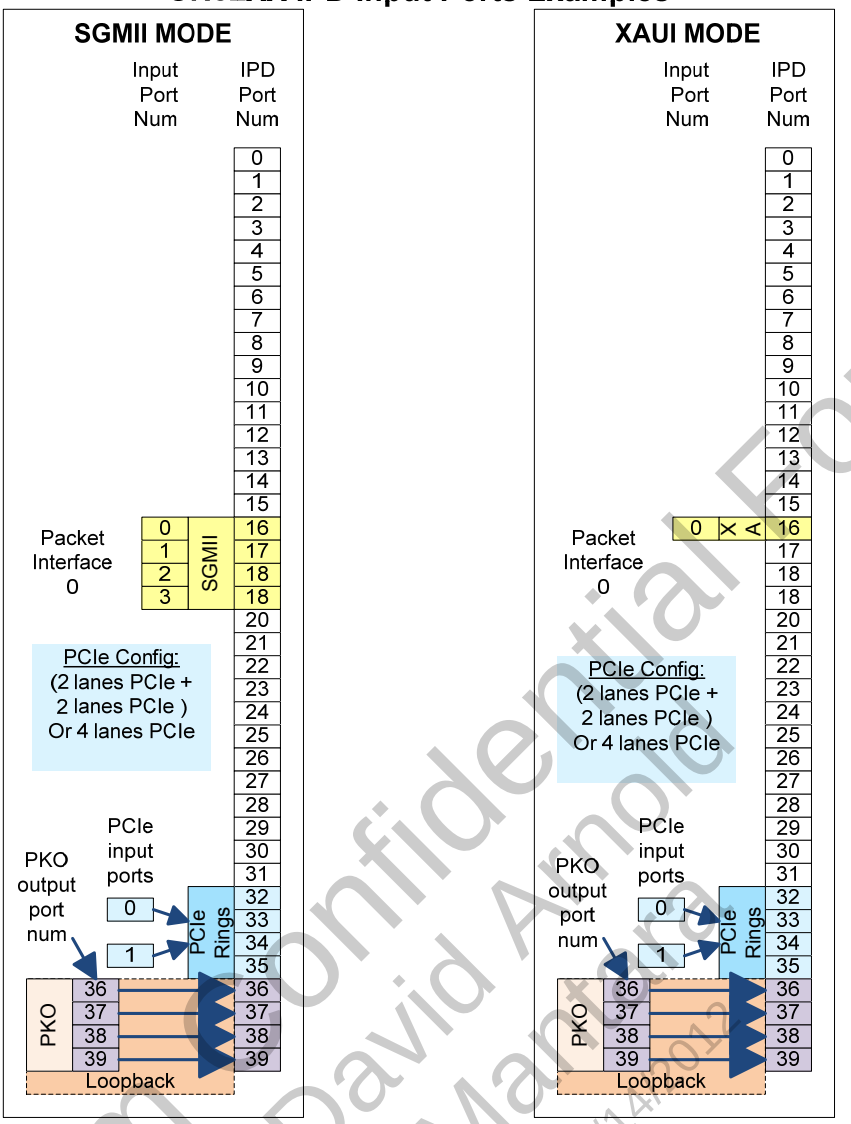


Figure 66: Input Ports: CN52XX

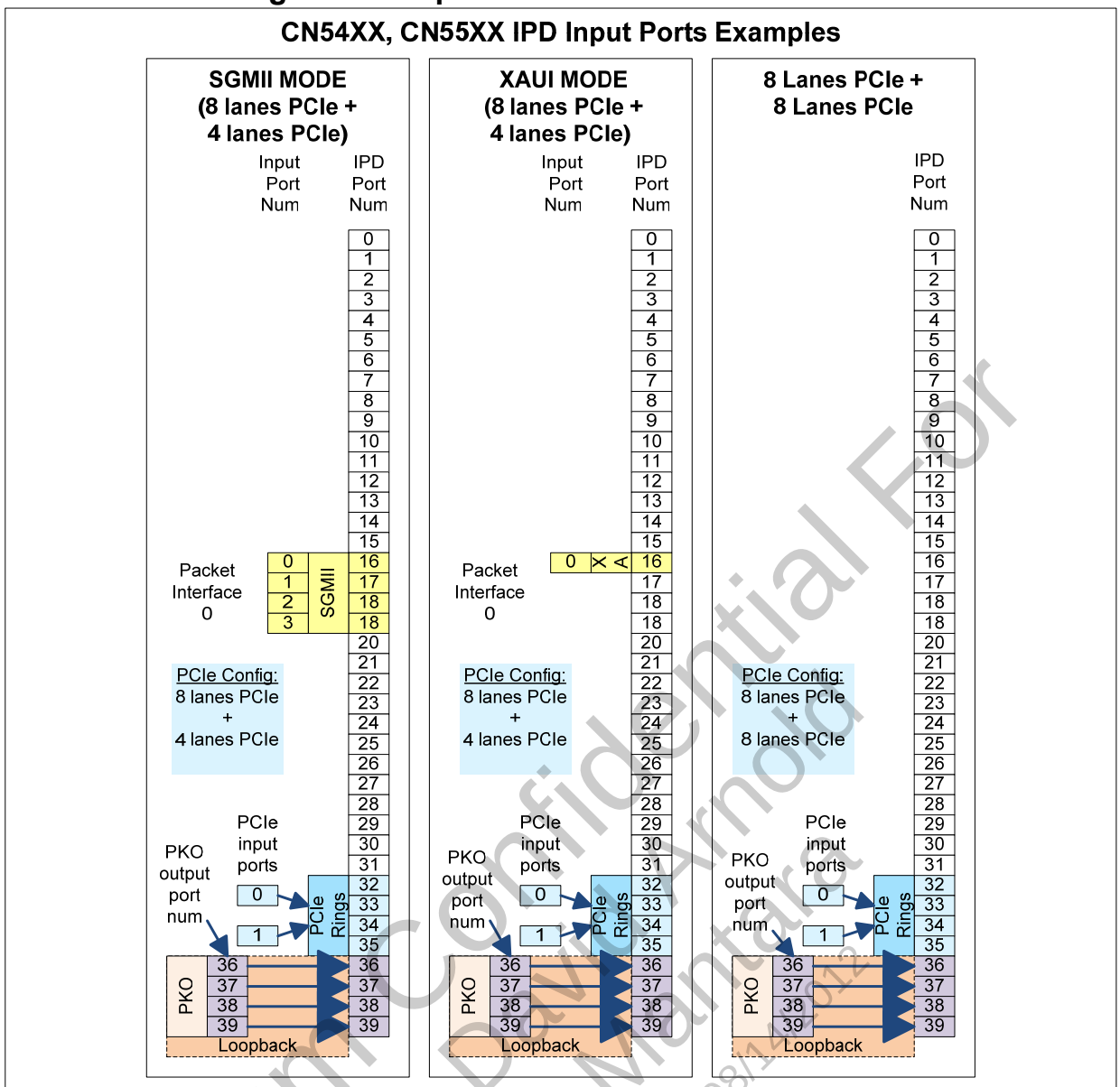


Notes:
Note 1: "XA" = XAUI
Note 2: Ports which are not colored in are unused.
Note 3: The Media-Independent Interfaces (MII) are not used as packet interfaces on these processors.
Note 4: Ports 0-15 are not used. The unused port numbers correspond to active ports on other processors. This configuration is necessary to support software re-use.

Note 5: Input is provided by 2 Quad-Lane Input Modules (QLMs).
QLM0 can be connected to PCIe controller 0 and (optionally) PCIe controller 1, creating either one 4-lane PCIe interface or two 2-lane PCIe interfaces.

Note 6: If QLM1's packet interface type may be configured to be SGMII, XAUI (XA), or PICMC. QLM mode is configured in hardware.

Figure 67: Input Ports: CN54XX and CN55XX



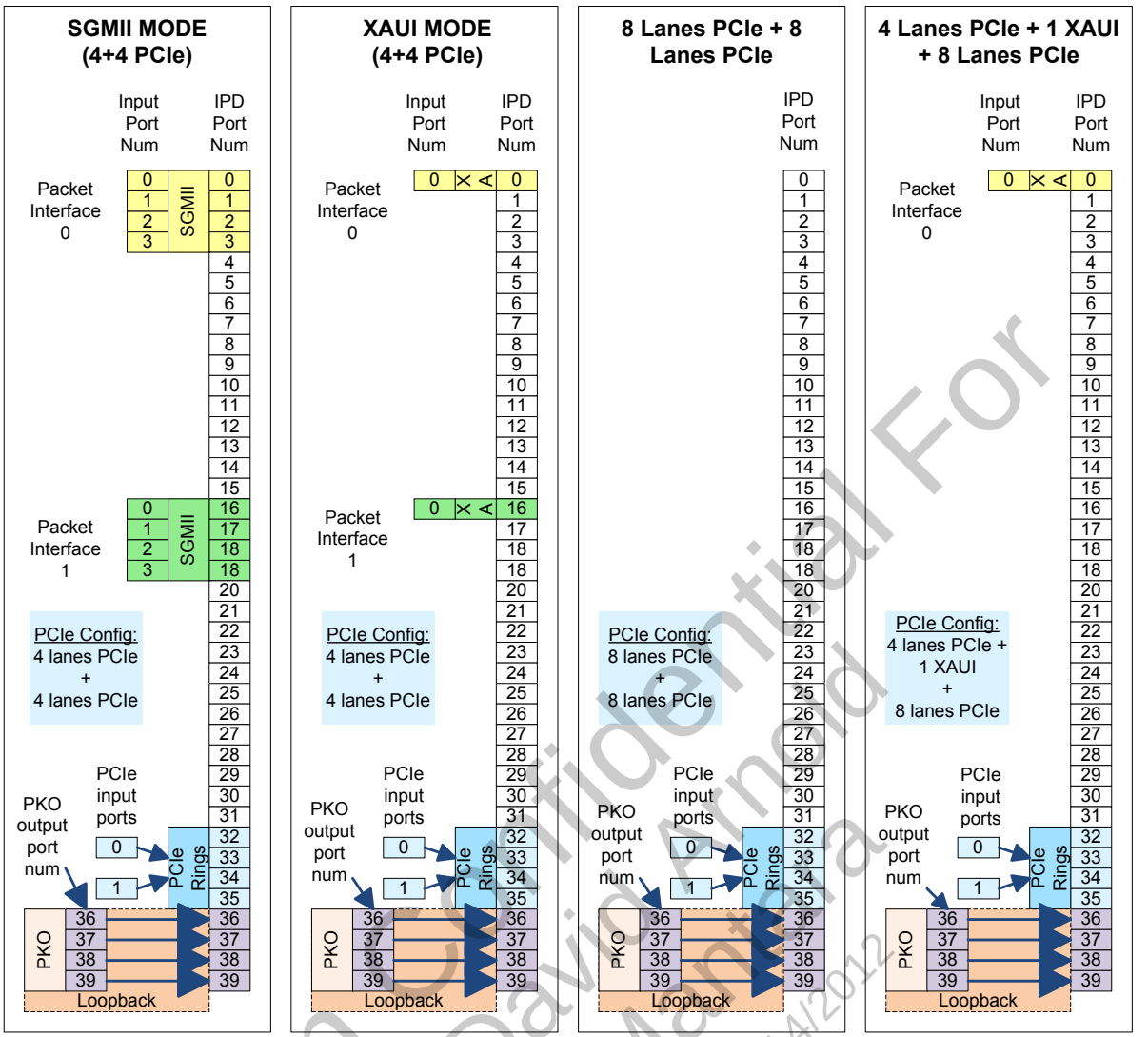
Note 1: Input is provided by 4 Quad-Lane Modules (QLMs), each able to provide 4 lanes of SerDes.
QLM0 and QLM1: dedicated to PCIe, connected to PCIe controller 0 (8 lanes of SerDes)
QLM2: dedicated to PCIe, connected to PCIe controller 1 (4 lanes of SerDes)
QLM3: optionally either connected to PCIe controller 1, or to Packet Interface 1
If QLM3 is configured as PCIe, it is combined with QLM2 to provide 8 lanes of SerDes on PCIe controller 1

Note 2: If QLM3 is configured as a packet interface, its packet interface type may be configured to be SGMII, XAUI (shown as "XA" in the figurer), or PICMG. QLM mode is configured by hardware.

Note 3: IPD ports which shown in white are unused. This information is provided to emphasize the gap in IPD port numbers.

Note 4: The Media-Independent Interface (MII) does not supply packets to IPD. MII packets traverse memory ring buffers outside of the IPD/PIP/SSO path.

Figure 68: Input Ports: CN56XX and CN57XX



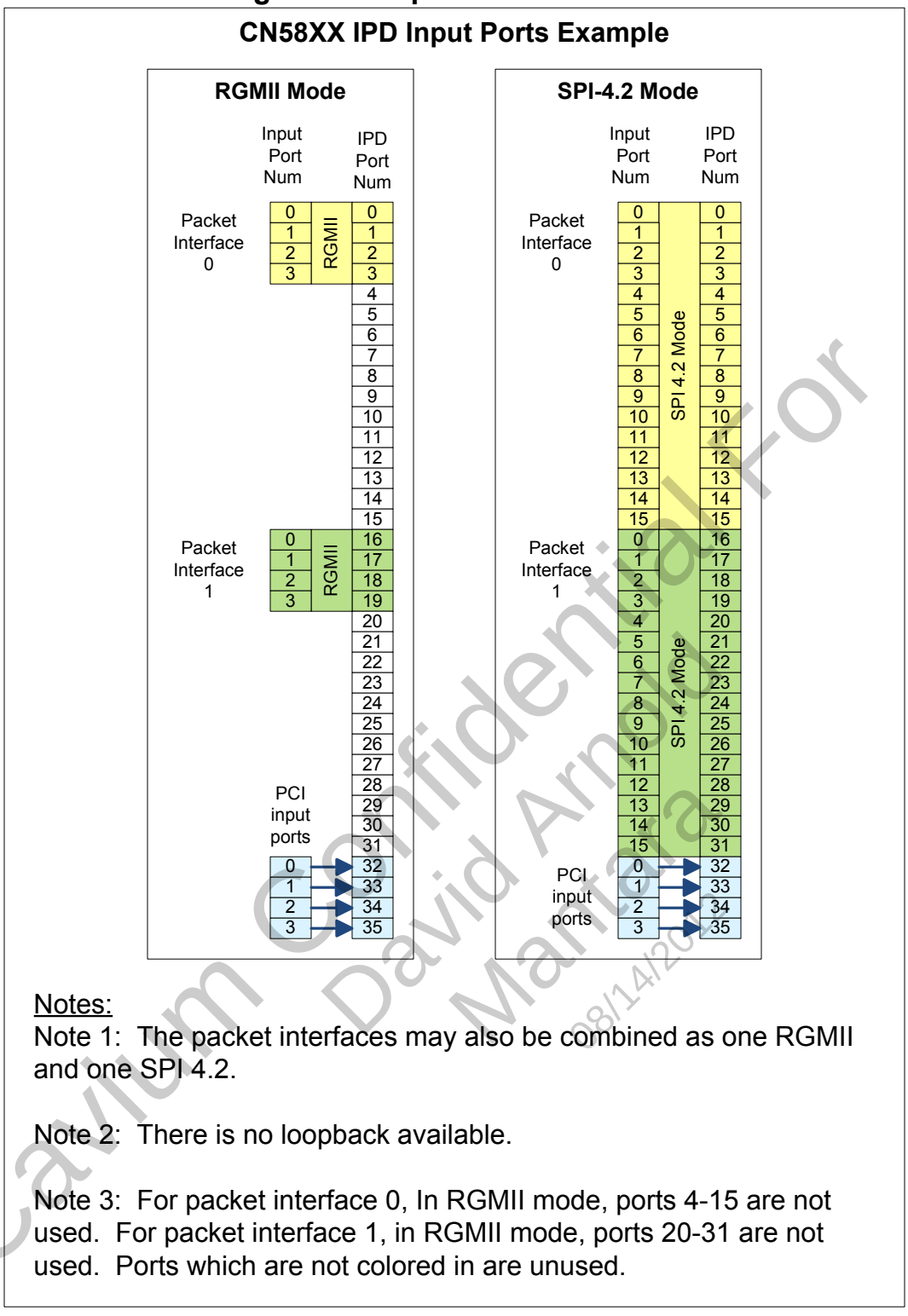
Note1: Input is provided by 4 Quad-Lane Input Modules (QLMs), each able to provide 4 lanes of SerDes.
QLM0: dedicated to PCIe, connected to PCIe controller 0.
QLM1: optionally either connected to PCIe controller 0 or to Packet Interface 0
QLM2: dedicated to PCIe, connected to PCIe controller 1.
QLM3: optionally either connected to PCIe controller 1 or to Packet Interface 1
If QLM1 is configured as PCIe, it is combined with QLM0 to provide 8 lanes of SerDes on PCIe controller 0.
If QLM3 is configured as PCIe, it is combined with QLM2 to provide 8 lanes of SerDes on PCIe controller 1.

Note2: If the QLM is configured as a packet interface, its packet interface type may be configured to be SGMII, XAUI (shown as "XA" in the figure), or PICMG. QLM mode by configured in hardware.

Note3: IPD ports which shown in white are unused. This information is provided to emphasize the gap in IPD port numbers.

Note4: The Media-Independent Interface (MII) does not supply packets to IPD. MII packets traverse memory ring buffers outside of the IPD/PIP/SSO path.

Figure 69: Input Ports: CN58XX



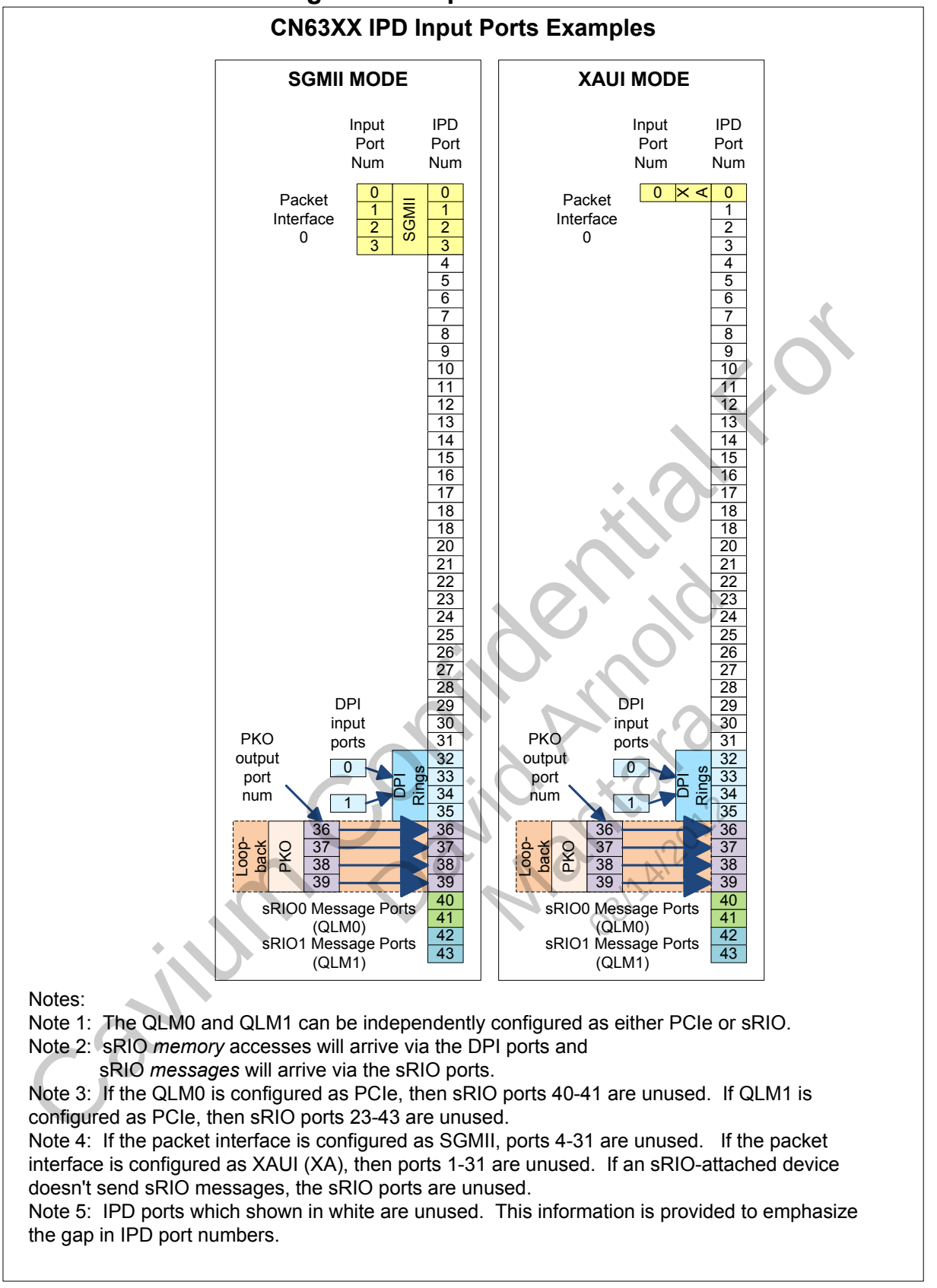
Notes:

Note 1: The packet interfaces may also be combined as one RGMII and one SPI 4.2.

Note 2: There is no loopback available.

Note 3: For packet interface 0, In RGMII mode, ports 4-15 are not used. For packet interface 1, in RGMII mode, ports 20-31 are not used. Ports which are not colored in are unused.

Figure 70: Input Ports: CN63XX



Notes:
Note 1: The QLM0 and QLM1 can be independently configured as either PCIe or sRIO.
Note 2: sRIO *memory* accesses will arrive via the DPI ports and sRIO *messages* will arrive via the sRIO ports.
Note 3: If the QLM0 is configured as PCIe, then sRIO ports 40-41 are unused. If QLM1 is configured as PCIe, then sRIO ports 23-43 are unused.
Note 4: If the packet interface is configured as SGMII, ports 4-31 are unused. If the packet interface is configured as XAUI (XA), then ports 1-31 are unused. If an sRIO-attached device doesn't send sRIO messages, the sRIO ports are unused.
Note 5: IPD ports which shown in white are unused. This information is provided to emphasize the gap in IPD port numbers.

19.1 Fast Links for Input Port Figures

The following links were provided at the beginning of this section, and are repeated here to provide a fast way to access the different figures.

Links to the different figures in this section: