Free Pool Allocator (FPA)

TABLE OF CONTENTS

TABLE OF CONTENTS 1
LIST OF TABLES 4
LIST OF FIGURES 4
1 Introduction 6
2 Overview of FPA 7
2.1 Functional Overview 7
2.2 Hardware Blocks Which Use FPA-Managed Buffers 7
2.3 Operations 10
2.4 FPA Registers 10
2.5 Using the FPA 11
2.5.1 Configure the FPA Unit 11
2.5.2 Initialize the FPA Unit 11
2.5.3 Enable the FPA Unit 11
2.5.4 Populate the FPA Pools 11
2.5.4.1 Allocating Memory for FPA-Managed Buffers 11
3 General Pool Configuration and Population Information 12
3.1 Default Simple Executive Pool and Scratchpad Area Configuration 12
3.2 Configuration Overview 13
3.2.1 Rules 13
3.2.2 Is There a Limit on the Number of Buffers? 14
3.2.3 Can a Pool Contain Different-Sized Buffers? 14
3.2.4 Can More Buffers be Added to a Buffer Pool at Any Time? 14
3.2.5 Configuring Hardware Units Which Automatically Use the Buffers 15
3.2.6 Allocating Buffers from Linux and the Affect on Buffer Size 15
4 Packet Data Buffers 17
4.1 Packet Data Buffer Size 18
4.2 Packet Data Buffer Count 19
4.2.1 Calculate the Maximum Number of Packet Data buffers Needed 19
4.2.1.1 What if the Formula Yields a Negative Number? 21
4.2.2 Packet Data Buffer Count and PIP/IPD Congestion Control 21
4.2.3 What if the System Runs Out of Available Packet Data Buffers? 21
4.2.4 Linux and Packet Data Buffer Count 21
5 WQE Buffers 23
5.1 WQE Buffer Size 23
5.2 WQE Buffer Count 23

Cavium Confidential For David Arnold Mantara 09/06/2012

5.3 Other Uses for WQE Buffers 24
6 PKO Command Buffers 25
6.1 PKO Command Buffer Size 26
6.2 PKO Command Buffer Count 27
6.3 More Precise PKO Command Buffer Size and Count Calculations 27
7 Simple Executive API 28
7.1 Limits and other Definitions 28
7.2 Data Structures 29
7.2.1 The cvmx_fpa_pool_info_t (pool_info) Data Structure 29
7.3 Easy-to-Use Executive FPA API Functions 29
7.3.1 Pool Information Functions 30
7.3.1.1 Example Code: cvmx_fpa_get_block_size(), cvmx_fpa_get_name(), cvmx_fpa_get_base() 31
7.3.1.2 Example Code: cvmx_fpa_is_member() 32
7.3.2 Easy-to-Use Initialize, Allocate, and Free Functions 32
7.3.2.1 Example Code: cvmx_helper_initialize_fpa() 33
7.3.2.2 Example Code: cvmx_fpa_alloc() 35
7.3.2.3 Example Code: cvmx_fpa_async_alloc() 36
7.3.2.4 Example Code: cvmx_fpa_free() 36
7.3.2.5 Example Code: cvmx_helper_free_packet_data() 36
7.4 Advanced Functions 38
7.4.1.1 cvmx_fpa_enable() 39
7.4.1.2 Example Code: Calling cvmx_fpa_setup_pool() 39
7.4.1.3 The cvmx_fpa_free_nosync() Function 40
7.4.1.4 Example Code: cvmx_fpa_shutdown_pool() 40
8 Basic Code Review Checklist 42
9 Internal Details 44
9.1 Buffer Organization 44
9.2 In-Unit Buffer Address Cache (Address Cache) 45
9.3 Watermarks for the In-Unit Buffer Address Cache 49
10 Debugging 53
10.1 Interrupts and Detected Error Conditions 54
10.1.1 Permission Error (PERR) 54
10.1.2 Page Count Off (Incorrect) Error (COFF) 54
10.1.3 Underflow (UND) 54
10.1.4 Single and Double Bit Memory Errors (SBE, DBE) 55
10.2 Debugging and Status Information 55
10.3 Common Mistakes 55
10.3.1 Buffer Alignment: Bad Alignment at Start of Buffer 56
10.3.2 Buffer Alignment: Bad Alignment at End of Buffer 56
10.3.3 Buffer Size and Don't Write Back (DWB) 57
10.3.4 Buffer Freed to the Wrong Pool 59
10.4 Buffer Freed More than Once 60
11 Performance Tuning 61
11.1 Enough Buffers 61

Cavium Confidential For David Arnold Mantara 09/06/2012

11.2 Prefetch Buffers 61
11.3 Initializing the Per-Pool Address Cache Allotment or Watermarks 61
11.4 Don't Write Back (DWB) 63
11.5 Pool Number 63
11.6 Performance Tuning Checklist 63
12 Advanced Code Review Checklist 64
13 Beyond the SDK – When not Using the Provided API 65
13.1 Design Considerations 65
13.2 Enable the FPA and Populating the Pools 65
13.3 Synchronous Buffer Allocation 65
13.4 Asynchronous Buffer Allocation 66
13.5 Freeing a buffer 66
14 FPA Registers 66
15 Configuring Units Which Allocate/Free FPA Buffers 70

Cavium Confidential For David Arnold Mantara 09/06/2012

LIST OF TABLES

Table 1: Units Allocating/Freeing Buffer Addresses, or Accessing Buffers.................................... 9
Table 2: Default Simple Executive Pool Configuration 12
Table 3: Default Simple Executive Scratchpad Configuration.. 13
Table 4: Packet Data Buffers Overview 17
Table 5: Packet Data Buffer Requirements 18
Table 6: Work Queue Entry Buffers Overview 23
Table 7: WQE Buffer Requirements 23
Table 8: PKO Command Buffers Overview.. 26
Table 9: PKO Command Buffers Requirements 26
Table 10: Pool Information Functions... 30
Table 11: Easy-to-Use Functions... 32
Table 12: Advanced FPA Functions.. 38
Table 13: Basic Code Review Checklist 42
Table 14: Defaults for Configurable Per-Pool Buffer Cache and Watermarks 45
Table 15: Fixed Per-Pool Watermarks and Size (If No Configuration Registers) 46
Table 16: Watermark Facts.. 50
Table 17: Status and Debugging Registers.. 55
Table 18: Advanced Initialization Registers.. 62
Table 19: Performance Tuning Checklist.. 63
Table 20: Advanced Code Review Checklist 64
Table 21: FPA Registers used in Buffer Allocate and Free Operations 66
Table 22: FPA Register Summary.. 67
Table 23: FPA Key Register Field Summary... 68
Table 24: DFA Unit.. 70
Table 25: IPD Unit... 70
Table 26: PCI/PCIe DMA Engine 70
Table 27: PKO Unit.. 71
Table 28: RAID Unit 71
Table 29: TIMER Unit... 72
Table 30: ZIP Unit... 72

Cavium Confidential For David Arnold Mantara 09/06/2012

LIST OF FIGURES

Figure 1: Units Allocating or Freeing FPA-Managed Buffers 8
Figure 2: A Free Buffer used as a Page of Free Buffer Addresses... 44
Figure 3: FPA In-Unit Buffer Address Cache – Simplified Internal View 47
Figure 4: FPA Buffer Pool: Simplified Internal View.. 48
Figure 5: FPA Read Watermark: Simplified Internal View 51
Figure 6: FPA Write Watermark – Simplified Internal View 52
Figure 7: Overwriting Memory: Buffer Not Cache Line Size Aligned.. 56
Figure 8: Overwriting Memory: Buffer not a Multiple of Cache Line Size.................................. 57
Figure 9: DWB and Why Buffers Need to be a Multiple of Cache Line Size 58
Figure 10: Buffer Freed to the Wrong Pool.. 59
Figure 11: Buffer Accidentally Freed More Than Once 60

Figure 12: Flowchart - FPA Unit Start-Up 62

Cavium Confidential For David Arnold Mantara 09/06/2012

1 Introduction

This chapter provides details about the Free Pool Allocator (FPA) unit. The FPA manages pools of pre-allocated memory buffers, for example Packet Data buffers or WQE buffers. Software running on the cores and also some hardware units may allocate and free buffers from/to the FPA pools.

Everyone who is writing or debugging code running on OCTEON processors should read this chapter, although some material toward the end is targeted to readers who are adding to the Simple Executive API or writing a custom API.

Before reading this chapter, please read the *OCTEON Programmer's Guide, Volume 1* and the *Configuration* and *Advanced Topics* chapters before reading this chapter. The *Configuration* chapter provides basic information on how to configure the FPA pools, but additional information is provided here.

This chapter provides:

- Configuration information
- Details on easy-to-use API data structures and functions
- Basic code review checklist
- Details on advanced API functions
- Debugging information
- Performance tuning information

Note that a downloadable simple example (`fpa_simplified`) of using the FPA API is provided at the Cavium Networks Technical Support Site in the same directory where an electronic copy of this chapter may be found. The downloadable file is a `tar` file. Untar it into the `examples` directory of the SDK. To build and run the example, follow the directions in the `README.txt` file provided with the example.

Note that the `fpa_simplified` example code only shows the FPA API, and is not useful for doing any actual work. The `passthrough` example code supplied with the SDK provides a realistic example of using the FPA to do packet processing.

Other example code can be found in the `examples` directory, in the SDK.

Cavium Confidential For David Arnold Mantara 09/06/2012

2 Overview of FPA

2.1 Functional Overview

As described in the *Packet Flow* chapter, the FPA supplies hardware units with Packet Data buffers, Work Queue Entry (WQE) buffers, and PKO Command buffers (the three pools essential for packet processing). All applications performing packet I/O will require these three types of buffers. The FPA may also supply buffers for TIM, DFA, RAID, PCI DMA Engines, and ZIP hardware units as well as user-defined buffers for use by the cores. (Note that the code supplied in the SDK configures PCI DMA Engines, RAID, and ZIP units to use the PKO Command buffers. Each of these engines frees the PKO Command buffer back to the FPA after it has been used.)

Hardware acceleration includes some buffer allocation and free by hardware units. For example, PIP/IPD and PKO both make use of FPA Pool 0 in order to support high-speed packet I/O without burdening the cores with a lot of buffer management overhead. PIP/IPD automatically allocates available Packet Data buffers from FPA pool 0 to contain new ingress packets, and the PKO can optionally free Packet Data buffers back to FPA pool 0 after it egresses the content of the Packet Data buffer.

On startup, the application enables the FPA, and then populates the FPA pools by allocating memory for each type of buffer (such as Packet Data buffer or Work Queue Entry buffer) and freeing the buffers to the FPA to create buffer pools for subsequent use. The FPA manages these pre-allocated buffer pools.

Once the FPA and associated hardware units are correctly initialized, use of FPA-managed buffers is efficient, easy and convenient.

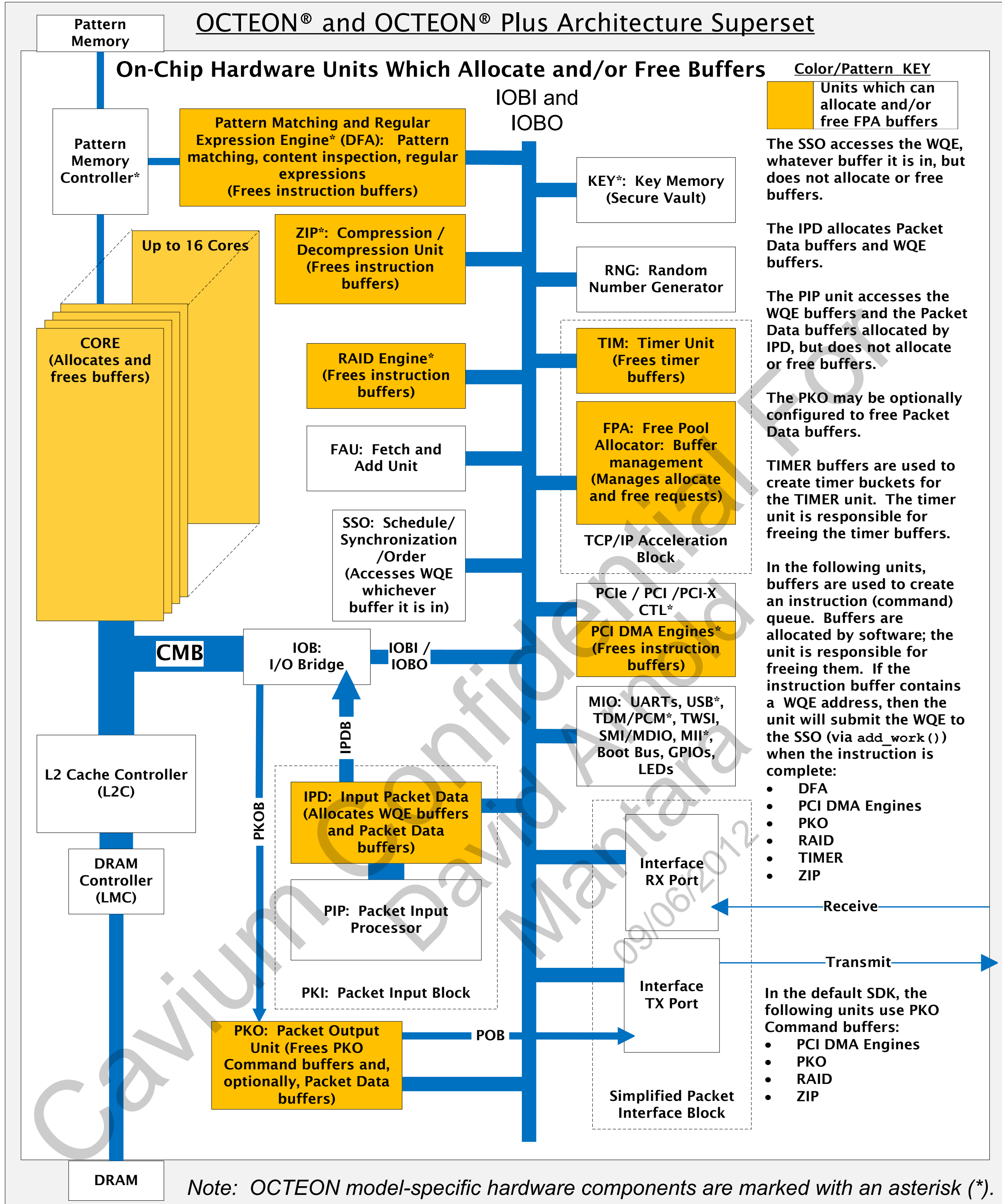
2.2 Hardware Blocks Which Use FPA-Managed Buffers

In the figure below the blocks highlighted in yellow are hardware units, including the cores, which allocate, use, and/or free FPA-managed buffers.

FPA-managed buffers are used by almost every unit in the OCTEON processor:

Cavium Confidential For David Arnold Mantara 09/06/2012

Figure 1: Units Allocating or Freeing FPA-Managed Buffers



Cavium Confidential For David Mantara 09/06/2012

Table 1: Units Allocating/Freeing Buffer Addresses, or Accessing Buffers

Unit	Populate Pools	Manage Allocate and Free Requests	Allocate Buffer	Free Buffer	Access Buffer	Submit WQE to SSO
Cores (via software)	Yes	No	Yes	Yes	Yes - all buffers	Yes
DFA	No	No	No	Yes - Note1	Yes - See Note3	Yes - Note2
FPA	No	Yes	No	No	Yes - all buffers	No
IPD	No	No	Yes - Packet Data Buffers and WQE buffers	No	Yes - Packet Data buffers and WQE buffers	Yes
PCI DMA Engines	No	No	No	Yes - Note1	Yes - See Note3	Yes - Note2
PKO	No	No	No	Yes - PKO Command buffers and optionally Packet Data buffers	Yes - PKO Command buffers and Packet Data buffers	No
RAID Engine	No	No	No	Yes - Note1	Yes - See Note3	Yes - Note2
SSO	No	No	No	No	Yes - WQE Buffers	N/A
TIM	No	No	No	Yes - Note1	Yes - See Note3	Yes - Note2
ZIP	No	No	No	Yes - Note1	Yes - See Note3	Yes - Note2
Notes						
Note1: Frees the Instruction buffer when all instructions in it have been processed						
Note2: Optionally submits the WQE buffer address to the SSO to be scheduled to a core.						
Note3: This unit accesses the instruction buffer						

Cavium Confidential For David Arnold Mantara 09/06/2012

This is why correct FPA configuration and careful use of FPA-managed buffers are essential to a healthy system. If something goes wrong with the FPA-managed buffers, the entire system may be affected, including errors which can be very distant from the original problem, and therefore very tricky to debug.

Despite the large number of units using the FPA-managed buffers, only a few buffer pools are needed:

- Packet Data buffers
- WQE buffers
- PKO Command buffers (in the default SDK, these are used by the PCI DMA Engines, PKO, RAID, and ZIP units)
- TIMER buffers
- DFA buffers

Hardware units which automatically allocate, use, and free buffers from the FPA must be correctly initialized. Incorrect hardware unit initialization can result in:

- over-writing buffers (For example, if the *mbuf* size provided to the IPD is incorrectly set to larger than the amount of available space in the Packet Data buffer, then the IPD will overwrite adjacent memory at the end of the buffer. The term *mbuf* stands for memory buffer. See the *Glossary* for more information.)
- allocating buffers from the wrong pool
- freeing buffers to the wrong pool

For these reasons, care must be taken when initializing the hardware units.

More information on hardware unit initialization is provided in Section 3 – "General Pool Configuration and Population Information".

2.3 Operations

The FPA supports three operations:

- **`buffer_allocate` (synchronous)**: The core waits until the address of an available buffer is returned (or NULL if no buffers are available)
- **`buffer_allocate` (asynchronous)**: The core does not wait for the buffer address to be returned. At a later time, the core retrieves the address from the specified scratchpad area.
- **`buffer_free` (synchronous)**: This operation returns the specified buffer address to the specified pool.

Either the cores or some of the hardware units can perform these operations.

These operations are discussed in detail in the *Configuration* and *Advanced Topics* chapters.

2.4 FPA Registers

The FPA registers may be used to customize the FPA configuration and retrieve status information. For instance, the register `FPA_QUEn_AVAILABLE` has a field `QUE_SIZ` which contains the number of available buffers in the specified pool ("*n*" is a number [0-7] which specifies the pool, as in `FPA_QUE2_AVAILABLE`.) (The register/field combination is written: `FPA_QUEn_AVAILABLE[QUE_SIZ]`.)

When customizing the SDK by accessing registers, use the API provides functions to read and write the registers safely and conveniently:

- `cvmx_read_csr()`
- `cvmx_write_csr()`

Race conditions which may otherwise occur are discussed in the *Configuration* and *Advanced Topics* chapters.

> ***Note: Do not modify these registers after the FPA is enabled. See Figure 12 – "Flowchart - FPA ".***

Cavium Confidential For David Arnold Mantara 09/06/2012

2.5 Using the FPA

Before the FPA can be used, it must be:

1. Configured
2. Initialized
3. Enabled
4. Pools must be populated with buffers

These concepts were introduced in the *Configuration* chapter. This section provides FPA-specific information. Functions mentioned in this section are discussed in more detail in Section 7 – "Simple Executive API".

2.5.1 Configure the FPA Unit

Simple Executive configuration is done at build time. See Section 3 – "General Pool Configuration and Population Information" for FPA-specific configuration information. See the *Configuration* chapter for more information.

2.5.2 Initialize the FPA Unit

Unit initialization is done at runtime. The Simple Executive API will manage any basic initialization items. Advanced initialization items are covered in the Advanced Section of this chapter, which begins with Section 11.3 – "Initializing the Per-Pool Address Cache Allotment or Watermarks".

2.5.3 Enable the FPA Unit

Unit enable is done at runtime. The FPA hardware unit is enabled via either the `fpa_enable()` function or `cvmx_helper_initialize_fpa()` (which calls the `fpa_enable()` function. The FPA must be enabled before the pools are populated, or runtime errors will occur (such as an incorrect available buffer count).

2.5.4 Populate the FPA Pools

Pool population is done at runtime. For easy pool population, see Section 7.3.2.1 – "Example Code: `cvmx_helper_initialize_fpa()`", for advanced pool population, see Section 7.4.1.2 – "Example Code: Calling `cvmx_fpa_setup_pool()`".

See Section 3.2.4 – "Can More Buffers be Added to a Buffer Pool at Any Time?" for more information.

2.5.4.1 Allocating Memory for FPA-Managed Buffers

If the Cavium Networks Ethernet driver is used with the application, then the Ethernet driver is responsible for allocating memory for the Packet Data buffers, WQE buffers, and PKO Command buffers.

Both SE-S and SE-UM applications must allocate memory for FPA-managed buffers using `cvmx_bootmem_alloc()`. This function will return memory which is suitable for DMA operations.

Cavium Confidential For David Arnold Mantara 09/06/2012

The Linux function `malloc()` cannot be used: `malloc()` does not return memory suitable for DMA: the virtual address is not mapped to physical memory. The DMAs would go to some unknown address and corrupt memory.

3 General Pool Configuration and Population Information

The *Configuration* chapter provides information on how to configure FPA pools and scratchpad areas. This section provides guidelines for making configuration and population (buffer count) choices for the three FPA pools essential for packet processing:

- Packet Data buffer pool
- Work Queue Entry buffer pool (Note that some OCTEON models can store the WQE data structure in the Packet Data buffer. If this feature is used, the WQE buffers are not needed for packet processing.)
- PKO Command buffer pool

The default SDK uses PKO Command buffers for the following hardware unit's instruction queues. This reduces the number of FPA pools needed:

- PCI DMA Engines
- PKO
- RAID
- ZIP

3.1 Default Simple Executive Pool and Scratchpad Area Configuration

The following table shows the default Simple Executive pool and scratchpad area configuration. In the default Simple Executive, PCI DMA Engines, PKO, RAID, and ZIP all use the PKO Command buffers pool. Separate pools can be configured for these uses if required.

Table 2: Default Simple Executive Pool Configuration

DFA Buffers Pool (`ifdef CVMX_ENABLE_DFA_FUNCTIONS`)	
Name	`CVMX_FPA_DFA_POOL`
Descriptive String	"DFA command buffers"
Pool Number (Default value)	4
Buffer Size (Default value)	2 * cache line size (256 bytes)
Protected / Permanent	Yes
Packet Data Buffers Pool (`ifdef CVMX_ENABLE_PKO_FUNCTIONS`)	
Name	`CVMX_FPA_PACKET_POOL`
Descriptive String	"Packet buffers"
Pool Number (Default value)	0 (cannot be changed)
Buffer Size (Default value)	16 * cache line size (2048 bytes)
Protected / Permanent	Yes
PKO Command Buffers Pool (`ifdef CVMX_ENABLE_PKO_FUNCTIONS`)	
Name	`CVMX_FPA_OUTPUT_BUFFER_POOL`
Descriptive String	"PKO queue command buffers"
Pool Number (Default value)	2

Cavium Confidential For David Arnold Mantara 09/06/2012

Buffer Size (Default value)	8 * cache line size (1024 bytes)
Protected / Permanent	Yes
Timer Buffers Pool (`ifdef CVMX_ENABLE_TIMER_FUNCTIONS`)	
Name	`CVMX_FPA_TIMER_POOL`
Descriptive String	"TIM command buffers"
Pool Number (Default value)	3
Buffer Size (Default value)	8 * cache line size (1024 bytes)
Protected / Permanent	Yes
Work Queue Entry Buffers Pool (`ifdef CVMX_ENABLE_HELPER_FUNCTIONS`)	
Name	`CVMX_FPA_WQE_POOL`
Descriptive String	"Work queue entries"
Pool Number (Default value)	1
Buffer Size (Default value)	1 * cache line size (128 bytes)
Protected / Permanent	Yes

Table 3: Default Simple Executive Scratchpad Configuration

Scratchpad 1 - generic - used for `cvmx_fpa_async_alloc()` (`ifdef CVMX_ENABLE_PKO_FUNCTIONS`)	
Name	`CVMX_SCR_SCRATCH`
Descriptive String	"Generic scratch iobdma area"
Size (Default value)	8 bytes
Protected / Permanent	No

Cavium Confidential For David Arnold Mantara 09/06/2012

3.2 Configuration Overview

3.2.1 Rules

The following rules must be followed when creating buffer pools. The Simple Executive APIs ensure these requirements and recommendations are met:

- All buffers must be aligned on a 128-byte boundary (the cache line size) (see Section 10.3.1 – "Buffer Alignment: Bad Alignment at Start of Buffer").
- All buffers must be at least 128 bytes (one cache line size) because they are also used to maintain the internal FPA pool structure (see Section 9.1 – "Buffer Organization" for details).
- Packet Data buffers must be a minimum size of 256 bytes (2 * cache line size) and a maximum buffers size of 16 Kbytes (128 * cache line size). (Note that this is a PIP/IPD hardware requirement, but the hardware does not check or enforce this requirement.)
- Packet Data buffers must also be a multiple of cache line size (see Section 10.3.2 – "Buffer Alignment: Bad Alignment at End of Buffer"). All other buffers are strongly recommended to be a multiple of cache line size (see Section 10.3.3 – "Buffer Size and Don't Write Back (DWB").

- Packet Data buffers must always be in FPA pool 0. This is a hardware requirement: the IPD and PKO units automatically allocate and free buffers from/to pool0.

3.2.2 Is There a Limit on the Number of Buffers?

The size of the buffer pool is not limited by the hardware; it is only limited by the amount of physical memory available. (When using Linux, the configuration system imposes a maximum limit of 8192 Packet Data buffers, but there is no limit imposed by the OCTEON processor. See Section 4.2.4 – "Linux and Packet Data Buffer Count" for more information.)

3.2.3 Can a Pool Contain Different-Sized Buffers?

Although more than one size of buffer can be in the same pool, this configuration is incredibly difficult to manage and creates an impossibly complex debugging environment. For example, if the IPD requests a Packet Data Buffer and receives a buffer which is smaller than the expected mbuf size (`IPD_PACKET_MBUFF_SIZE[MB_SIZE]`), when it writes the packet data to the buffer, it may overwrite adjacent memory. This error can also occur when buffers are freed to the wrong pool (for example, a smaller buffer is mistakenly freed to the Packet Data buffer pool). See Section 10.3.4 – "Buffer Freed to the Wrong Pool" for an illustration of this problem.

3.2.4 Can More Buffers be Added to a Buffer Pool at Any Time?

The pools are typically populated at runtime right after the FPA has been initialized and enabled.

> ***Note: Future processors may provide optional FPA hardware range checking on buffer addresses provided to the FPA via the `buffer_free` operation. If this feature is used, then buffer pools should only be populated once so that the buffer addresses all fall within the same range.***

The question often arises whether buffers may be added at any time, for instance if the buffer pool is low in buffers after the system has been running for a while:

- If the Cavium Networks Ethernet driver is *not* being used, then buffers can be added at any time by any core, given some caveats.
- If the Cavium Networks Ethernet driver is being used:
  - Packet Data buffers: More buffers cannot be added to the Packet Data buffer pool because the Linux kernel initializes Packet Data buffers to contain the *`sk_buff`* data structure, which allows the driver to receive packets without copying them (for faster performance).[1] The Linux kernel uses the `sk_buff` data structure for network buffers and queues, and relies on information which it put into the

Cavium Confidential For David Arnold Mantara 03/06/2012

[1] Note that, for performance reasons `USE_SKBUFS_IN_HW` is automatically set to TRUE when possible. If this variable is TRUE, then 32-bit SE-UM applications cannot access Packet Data buffers, so if the `menuconfig` variable `CONFIG_CAVIUM_RESERVE32 is TRUE` (1), then `USE_SKBUFS_IN_HW` is automatically set to FALSE (0), otherwise `USE_SKBUFS_IN_HW` is automatically set to TRUE (1). (For SDK 1.9 and SDK 2.0, this variable is defined in `ethernet-defines.h`, in the Linux kernel directory: `$OCTEON_ROOT/linux/kernel_2.6/drivers/net`. For SDK 1.9, the Cavium Ethernet driver code is in the `cavium-ethernet` sub-directory. For SDK 2.0, the driver code is in the `octeon` sub-directory.

sk_buff. If a core which is not running Linux allocates buffers and adds them to the pool, they will not contain the information needed by the Linux kernel.

- WQE buffers: More buffers should not be added because the ratio of WQE buffers to Packet Data buffers is carefully configured to prevent exhaustion of Packet Data buffers (no more buffers available in the pool). If the PIP/IPD congestion control mechanisms depend on the number of WQE buffers available, and this number is increased, then the Packet Data buffer pool could become exhausted.
- PKO Command buffers: Although as of SDK 2.0 there is no restriction on adding more PKO Command buffers, future SDKs may add performance improvements affecting the content of the buffers. Thus, applications should not depend on adding more PKO Command buffers.

Buffers may *not* be added at any time by any core if:

- The Cavium Networks Ethernet driver is in use (to Packet Data buffers, WQE buffers, or PKO Command buffers (see details, above)).
- The function `fpa_is_member()` is used (determines if a buffer address is within the address range of the original chunk of memory allocated for the pool).
- The function `fpa_get_base()` is used (returns the start address of the original chunk of memory allocated for the pool).
- Custom software is used to track whether a buffer belongs to a pool or provide the starting address of the memory allocated for the pool.

Note that the underlying hardware does not care about any relationship or lack of it between all the addresses in the pool. It is the application which may care, and this is what imposes the constraint on adding more buffers to a pool at any time.

Note that it is best to allocate the maximum required amount of memory when the application initially populates the FPA pools because there is more chance the request will be filled from a contiguous block of memory, especially when allocating large blocks of memory.
Also note that some applications rely on the fact that all buffers that reside in a particular FPA pool belong to a single contiguous memory region for additional functionality, such as verification that the received buffer is valid by checking that its address is within the original allocated memory range (`fpa_is_member()`). For SDK 1.X, the Cavium Networks Ethernet driver performs this check on each received Packet Data buffer to check whether it is valid.

3.2.5 Configuring Hardware Units Which Automatically Use the Buffers

When configuring the FPA, the interaction between the FPA and other hardware units must be considered. For example, if a hardware unit is configured to automatically free buffers to pool 5, but the buffers actually came from pool 4, then at runtime pool 4 will run out of buffers. More information on hardware units which automatically free and allocate buffers is provided in Section 15 – "Configuring Units Which Allocate/Free FPA Buffers".

3.2.6 Allocating Buffers from Linux and the Affect on Buffer Size

When running the Cavium Networks Ethernet driver (for example, `linux-filter`), then the Ethernet driver allocates the memory for FPA-managed buffers and creates the buffer pools.

Cavium Confidential For David Arnold Marvara 09/06/2012

For kernel 2.6 and previous kernel versions, slab allocation is done. In this type of allocation, the size of the requested memory is rounded up to the nearest available power of 2.

Linux can add overhead to the requested size. For example, in SDK 2.0, if the requested buffer size is 2048, Linux adds 256 bytes to that request for a `sk_buff` data structure, increasing the size to 2304 bytes (in SDK 1.9, Linux adds 8 bytes). In this case, the slab memory allocator will allocate the next power of two size buffer: 4096. One work-around is to request a buffer size which is a multiple of 128 bytes, but less than 2048 (such as 1536 bytes), to allow room for kernel additions.

Depending on the configuration, Linux may increase the size for other purposes. If the user starts with a large buffer size, the added space can tip the buffer size into the 32,768-byte or even 65,536-byte buffer size.

To check on the allocated buffer size, after booting Linux on the target board, cat `/proc/slabinfo` to see which size was allocated for the buffers. If necessary, adjust the size of the original request until the amount of memory consumed is acceptable.

> ***Performance Tip: The standard MMU/TLB page size for Linux is 4096 bytes. If buffers are larger than 4096 bytes, performance may be improved by changing the MMU/TLB page size to be large enough so that large buffers do not consume multiple pages. OCTEON hardware supports MMU/TLB page sizes up to 256 MBytes.***

Cavium Confidential For David Arnold Mantara 09/06/2012

4 Packet Data Buffers

Packet Data buffers are used to receive packet data. If one Packet Data buffer is not large enough to hold all the packet data, then Packet Data buffers will be chained together automatically by the PIP/IPD.

Table 4: Packet Data Buffers Overview

Packet Data Buffers	
Default Pool Name	`CVMX_FPA_PACKET_POOL`
Default Pool String	"Packet Data Buffers"
Unit Allocating Buffer	IPD (always Pool 0)
What controls the allocation and use?	In the Simple Executive, the function `cvmx_helper_global_setup_ipd()` sets the value of `IPD_PACKET_MBUFF_SIZE[MB_SIZE]`. The IPD always allocates Packet Data Buffers from pool 0.
Recommended Buffer Size	Up to 2048 bytes (sixteen cache lines) (MTU of 1500 bytes)
Recommended Number of Buffers	Depends on the application and the expected system load.
Unit Freeing Buffer	PKO or core
SDK Function to Configure Timers to Use the Correct Pool	The PKO will optionally free the buffer to the specified pool (usually pool 0). The core may also optionally free the buffer.

Cavium Confidential For David Arnold Mantara 09/06/2012

Table 5: Packet Data Buffer Requirements

Buffer Size Multiple
Size is Multiple of Cache-Line Size: Buffer size is *required* to be a multiple of the cache line size (128 bytes). The IPD always writes packet data in complete 128-byte blocks, including the last data in the packet. Since the IPD will write 128 bytes even if that would exceed the end of the buffer, it is important that the buffer is large enough and that the end of the buffer is correctly aligned to the 128-byte boundary. This size restriction also provides support for "Don't Write Back" (DWB) functionality (see the *Advanced Topics* chapter for details.) Note that hardware does not enforce this requirement: if the buffer is incorrectly configured, difficult-to-debug problems will occur.
Minimum Buffer Size
PIP/IPD configuration rules impose minimum buffer size of 256 bytes (2 * cache line size) and a maximum buffers size of 16 Kbytes (128 * cache line size). The default configured size is 2048 bytes (16 * cache line size). Note that hardware does not enforce this requirement: if the buffer is incorrectly configured, difficult-to-debug problems will occur.
Pool Number
This pool is required to be FPA Pool 0, because the IPD always gets Packet Data buffers from pool 0 (not configurable). Note that hardware does not enforce this requirement: if the buffer is incorrectly configured, difficult-to-debug problems will occur.
Buffer Count

Assuming packet data will fit into one Packet Data buffer:
`min_buffers = [max_burst_size *`
`(1 - (num_cores * core_processing_rate / max_burst_rate))] + prefetch`
(`prefetch` is the number of buffers prefetched by IPD (about 130 buffers)). Congestion control mechanisms can be used to ensure that low priority traffic does not consume all of the available buffer space (see the *PIP/IPD* chapter).

4.1 Packet Data Buffer Size

For optimum performance, most packets should fit inside a single packet data buffer. The required size of the packet data buffer doesn't only depend on the size of the packet, it also depends on configurable options and other factors. See the "Packet Storage" section in the *PIP/IPD* chapter for more information.

For any packet, the incoming packet data is not required to fit into a single Packet Data buffer. If the data cannot fit into one Packet Data buffer, multiple Packet Data buffers will be allocated and linked together automatically by the PIP/IPD (the format of the scatter list is chained buffers). For best performance, most traffic should fit into a single buffer; chained buffers incur higher overhead, and also require more FPA bandwidth. Therefore, it is recommended that the Packet Data buffer size be set to large enough to accommodate the packet size of most of the traffic.

For example, on an Ethernet most frames will be a maximum of 1500 bytes of payload, unless jumbo frames are in use. When the MTU is 1500, the buffer size should be set to 2048 bytes, allowing each frame to be stored in a single buffer. When jumbo frames are in use, a 2048 byte

Cavium Confidential For David Arnold Mantara 09/20/2012

buffer size is still often a good balance between memory consumption and system performance, depending on how frequently jumbo frames are seen.

See also Section 3.2.6 – "Allocating Buffers from Linux and the Affect on Buffer Size".

4.2 Packet Data Buffer Count

The required number of Packet Data buffers is application dependent. This section provides a formula for calculating the number of packet data buffers needed.

In the following formula, one Packet Data buffer per ingress packet is assumed. This formula will need to be adjusted if this is not the ratio used by the application. The *PIP/IPD* chapter explains packet storage and variations on packet storage. Here are a few items to be aware of (see the *PIP/IPD* chapter for details):

- The entire Packet Data buffer is not available to store data due to mbuf overhead and configuration options.
- If the packet data will not fit into one Packet Data buffer, it is automatically stored in multiple Packet Data buffers by PIP/IPD.
- Dynamic shorts can optionally be configured, allowing short packets (less than approx 90 bytes) to be stored completely stored in the WQE buffer: no Packet Data buffer is used in this case.

Note that users sometimes increase the number of Packet Data buffers unnecessarily when the underlying problem is per-core throughput (the `core_processing_rate` in the formula below, which is measured in Million packets per second (Mpps)). When tuning a system, examining the per-core throughput, identifying, and fixing any bottlenecks in the code can be helpful.

System tuning is a complex process which can require a deep knowledge and understanding of the hardware architecture and application requirements. The following chapters provide information which can help with system tuning:

- Information on how to measure the number of cycles consumed by a section of code is provided in the *Essential Topics* chapter.
- Packet processing math is provided in the *Software Overview* chapter.
- Congestion control information provided in the *PIP/IPD* chapter. This section provides a system-level view of common causes for congestion and proposed solutions.

4.2.1 Calculate the Maximum Number of Packet Data buffers Needed

The following calculation can help determine the minimum number of Packet Data buffers needed in order to support a given traffic profile without unpredictable loss due to congestion control mechanisms. The formula calculates the number of Packet Data buffers needed for a given maximum burst rate and burst size. This example assumes a single QoS level for all ingress packets, and that packets can be egressed "immediately" once initial core processing is complete.

Cavium Confidential For David Arnold Mantara 09/06/2012

The maximum number of Packet Data buffers needed depends on:

- The ingress burst rate
- The time needed for a core to process a packet
- The time for the core or PKO to free the Packet Data buffer
- The number of cores processing packets
- Whether the average packet will fit into only one Packet Data buffer

Note that the FPA pool 0 should be configured with a slightly greater amount (the `prefetch` amount shown in the formula below) of roughly 130 buffers to allow for variations in system dynamic performance, buffers being pre-allocated by PIP/IPD, PKO transmit backlog (delay in freeing the buffer), and other variables.

The following formula assumes that the average packet will fit into only one Packet Data buffer:

```
min_buffers = [max_burst_size *
(1 - (num_cores * core_processing_rate / max_burst_rate))] + prefetch
```

Where `min_buffers` is the number of Packet Data buffers needed to hold the backlog.

(The `core_processing_rate/max_burst_rate` = the amount of the burst a single core can handle. This number multiplied by the number of cores is the amount of the burst the set of cores can handle. (1 – (the number of packets all the cores can handle)) is the fraction of the whole which is *not* being handled "real time" by the cores (a backlog of work to do). This fraction will be stored in the Packet Data buffers until the cores process the backlog.)

(Packets are received at `max_burst_rate` for a short period; the duration is defined by the ratio of the `max_burst_size` (number of packets received at `max_burst_rate`) to the actual `max_burst_rate`.)

<u>Example:</u>
This example is for a TCP/IP Toolkit IP forwarding example program:

- `core_processing_rate = 2Mpps` (2 million packets per second) per core (the throughput is approximately 2Mpps per core @ 800MHz)
- `max_burst_rate = 16 Mpps` (ie roughly a 10 G line)
- `max_burst_size = 4Mp` (4 million packets)
- Each packet will fit into only one Packet Data buffer, so multiple Packet Data buffers per packet are not needed

So if:

```
core_processing_rate = 2Mpps (pps=packets per second)
max_burst_rate = 16Mpps
max_burst_size = 4Mp (p=packets)
```

then:

```
1 core:  min_buffers = 4Mp * (1 - 1 core  * 2Mpps / 16Mpps) = 3.5M + prefetch
2 cores: min_buffers = 4Mp * (1 - 2 cores * 2Mpps / 16Mpps) = 3.0M + prefetch
4 cores: min_buffers = 4Mp * (1 - 4 cores * 2Mpps / 16Mpps) = 2.0M + prefetch
```

Cavium Confidential For David Arnold Mantara 09/06/2012

```
8 cores: min_buffers = 4Mp * (1 - 8 cores * 2Mpps / 16Mpps) = 0.0M + prefetch
```

Note that since 8 cores can process 16Mpps, a burst of 16Mpps will not build a back-log, so the formula reports `0.0M + prefetch` buffers are needed.

4.2.1.1 What if the Formula Yields a Negative Number?

If the formula yields a negative value for the number of buffers (excluding the prefetch amount), this indicates that the cores are able to process the incoming traffic at a rate faster than the `max_burst_rate` (they are keeping up with the traffic). In this scenario, the number of buffers needed is much smaller and is dependent on the maximum processing time needed for an individual packet (for example, an error condition can result in a longer processing time).

The following formula gives a worst case minimum buffer requirement to be used when the formula above yields a negative number (excluding the prefetch amount):

```
min_buffers = max_burst_rate / core_processing_rate
```

Using the numbers from the example, that is:

```
min_buffers = 16Mpps / 2Mpps = 8 buffers
```

4.2.2 Packet Data Buffer Count and PIP/IPD Congestion Control

Many PIP/IPD congestion control mechanisms are based on the number of available Packet Data buffers so if the WQE buffers are exhausted first, the congestion control mechanism will not detect the buffer exhaustion and take correct action. If these mechanisms are used, the number of Packet Data buffers should be in the correct ratio with the number of WQE buffers to prevent WQE buffer exhaustion. For example, if there are 2048 WQE buffers, and 8192 Packet Data buffers, with only one Packet Data buffer needed per packet, then the WQE buffers will be exhausted while there are still 6144 Packet Data buffers available (wasted, because they cannot be used).

If the number of Packet Data buffers was 2048, then the PIP/IPD congestion control mechanisms can prevent both Packet Data buffer and the WQE buffer exhaustion (assuming dynamic shorts are not configured), and no Packet Data buffers are unused.

Cavium Confidential For David Arnold Martara 09/06/2012

4.2.3 What if the System Runs Out of Available Packet Data Buffers?

If PIP/IPD runs out of Packet Data buffers, it will stop receiving packets (critical buffer exhaustion). When more buffers are available, it will resume receiving packets. PIP/IPD congestion control mechanisms should be used to prevent critical buffer exhaustion. See the *PIP/IPD* chapter for details. Note that PIP/IPD congestion control mechanisms focus on the number of Packet Data buffers, not WQE buffers. Be sure to configure sufficient WQE buffers (there should be more WQE buffers than Packet Data buffers).

4.2.4 Linux and Packet Data Buffer Count

When the Cavium Networks Ethernet driver is used, the number of Packet Data buffers is configured via the Linux `make menuconfig` command. The Cavium SDK Linux kernel configuration imposes an arbitrary maximum of 8,192 Packet Data buffers. This limit is imposed in order to maintain compatibility with the many different configuration permutations and to avoid

excessive use of the kseg0/DRAM0 memory block which can occur in some SDK 1.X based configurations. It is expected that this number should be sufficient for standard Linux applications.

If more buffers are required for a standard Linux application, or a hybrid system (Linux cores + SE-S cores is being designed), a detailed review is required in order to understand why so many buffers are required, and to select the appropriate system design/kernel configuration. See the packet processing math figure in the *Software Overview* chapter. Also see the *Essential Topics* chapter for information on how to measure the number of cycles consumed by a section of code.

Cavium Confidential For David Arnold Mantara 09/06/2012

5 WQE Buffers

The WQE buffers are used to store the WQE data structure. (The WQE data structure may optionally be stored at the start of the Packet Data buffer instead of in a WQE buffer for some OCTEON models.)

Table 6: Work Queue Entry Buffers Overview

WQE Buffers	
Default Pool Name	`CVMX_FPA_WQE_POOL`
Default Pool String	"Work queue entries"
Unit Allocating Buffer	IPD or the core (via software)
What controls the allocation and use?	In the Simple Executive, the function `cvmx_helper_global_setup_ipd()` configures the IPD to use the correct pool.
Recommended Buffer Size	128 bytes (one cache line)
Recommended Number of Buffers	Depends on the system design.
Unit Freeing Buffer	core (via software)
How does the system know which pool the buffer returns to?	The pool number is in the WQE data structure. The core will free the buffer.

Table 7: WQE Buffer Requirements

Buffer Size Multiple
Buffer size is strongly recommended to be a multiple of the cache line size (128 bytes).
Minimum Buffer Size
128 bytes (1 * cache line size)
Pool Number
The pool number can be any unused pool number from [1-7]. The pool number by default is "1".
Buffer Count
This pool must never run out of buffers (see "buffer exhaustion" and "critical backpressure" in the *PIP/IPD* chapter). If WQE buffer exhaustion occurs, then the packet interfaces will not be able to receive new packets, stopping all traffic regardless of priority.

Cavium Confidential For David Arnold Mantara 09/06/2012

5.1 WQE Buffer Size

The WQE is usually configured to be 128 bytes (the minimum size). Users may increase the size to allow for application-specific data storage after the hardware-defined WQE fields (two 64-bit words are required as a minimum by the SSO).

5.2 WQE Buffer Count

Tuning the number of WQE buffers is a complex process. The number needed depends on the system architecture, expected load, and the congestion control mechanism being used. The

discussion below presents different options. See Section 4.2 – "Packet Data Buffer Count" for more discussion. The number of Packet Data buffers relative to WQE buffers is essential.

Factors which can drive the WQE buffer count include:

1. Determine whether PIP/IPD generates WQEs separately from Packet Data buffers, or stores the WQE in the Packet Data buffer. (On some OCTEON models, PIP/IPD may be configured to not use WQE buffers. See the *PIP/IPD* chapter for details.)
2. If PIP/IPD generates WQEs separate from the Packet Data buffers, then allocate at least 1 WQE for every packet ingressed and awaiting processing. See the *PIP/IPD* chapter for more information on packet storage.
3. Account for WQE buffers used by other accelerators and/or core software. (See Section 5.3 – "Other Uses for WQE Buffers").

5.3 Other Uses for WQE Buffers

The WQE buffers are not only used for ingress packets. Here are some things to consider during system tuning:

1. Are other accelerators such as TIMER, ZIP, HFA, RAID used? (All require WQEs to provide results back to the cores.)
2. Are there other custom software uses for WQE buffers in the application? (In addition to the automatic allocation by the IPD, cores may allocate these buffers and send them to other cores via the SSO (using the `add_work` and `get_work` operations).)
3. Are PKO 3-word commands used? (These commands are used to instruct the PKO to submit a WQE after packet transmission).
4. Analyze carefully the use of PCI/PCIe packet I/O and high performance DMA transfer. (WQEs can be used to notify the core of operation completion.)

Cavium Confidential For David Arnold Mantara 09/06/2012

6 PKO Command Buffers

The PKO Command buffers are used to create command (instruction) queues. Software allocates a PKO Command buffer, and adds commands to it. At the end of each command buffer, there is a pointer to the next command buffer. When the first PKO Command buffer is full, software allocates another one and links it into the command queue. This creates a PKO command/instruction queue. The hardware unit (such as the PKO) is given the address of the first buffer in the chain. When new commands are put into the command queue, the hardware unit is notified, so it can track the number of outstanding commands to process. When the hardware unit has processed all the commands in one PKO Command buffer, it frees the buffer back to the FPA pool.

PKO Command buffers are used by other hardware units in addition to the PKO: they are optionally used by the PCI DMA engines, RAID, and/or ZIP units (this is the default configuration in the SDK). They may also be used by the DFA unit and TIMER unit. All of these units use buffers to create a list of commands/instructions. All of these units automatically free the buffers back to the FPA pool when the commands in them have been processed. The `cvmx_cmd_queue*()` functions provide a single API that allows applications to queue commands to any of the hardware units listed above.

The PKO Command buffer pool can become depleted if:

- One or more PKO ports are backlogged (packets are submitted to the PKO for egress at a rate faster than the port's physical egress rate)
- Other hardware units are using the PKO Command buffer pool, and one of these hardware units has a backlog of commands.

Applications must therefore check whether the allocation command failed and handle the error.

If the application needs to prevent PKO Output Queues from using up an unfair share of PKO command buffers, then the application code must monitor the length of each PKO Output Queue and limit the maximum commands appropriately. The FAU hardware can help monitor the number of packet sent to the PKO Output Queue versus the number the PKO has sent. See the FAU section in the *Essential Topics* chapter for an overview of how this is done.

There is no need for the application to monitor the PKO Output Queue length to prevent doorbell overflow: the doorbell register accommodates a maximum of 2^{20} words (a maximum of 2^{19} commands), so the `cvmk_pko_doorbell*()` functions already monitor the queue size.

Cavium Confidential For David Arnold Mantara 09/06/2012

Table 8: PKO Command Buffers Overview

PKO Command Buffers	
Default Pool Name	`CVMX_FPA_OUTPUT_BUFFER_POOL`
Default Pool String	"PKO queue command buffers"
Unit Allocating Buffer	core (via software)
What controls the allocation and use?	In the Simple Executive, the function `cvmx_fpa_alloc(pool)` will return a buffer pointer; the function `cvmx_fpa_async_alloc(scratch_address, pool)` will write the buffer address to the scratchpad.
Recommended Buffer Size	1024 bytes (eight cache lines)
Recommended Number of Buffers	4 buffers per each user is recommended, assuming there is no specific requirement to support a defined number of backlogged commands. One user is 1 PKO Output Queue, RAID, ZIP, or 1 PCI DMA Engine.
Unit Freeing Buffer	Unit using the buffer, such as the PKO.
How does the system know which pool the buffer returns to?	PKO: `cvmx_pko_initialize_global()` PCI DMA Engines: `cvmx_dma_engine_initialize()` RAID: `cvmx_raid_initialize()` ZIP: `cvmx_zip_initialize()` configure the unit to know the pool and to enable DWB or set the DWB count (depending on the unit being configured)

Table 9: PKO Command Buffers Requirements

Buffer Size Multiple
Buffer size is strongly recommended to be a multiple of the cache line size (128 bytes).
Minimum Buffer Size
128 bytes (1 * cache line size)
Maximum Buffer Size
The max size is 511 cache lines, dictated by the PKO register field `PKO_REG_CMD_BUF[SIZE]`. This is the number of 8-byte command words in each PKO Command buffer. The largest number this register can hold is 8,191. 511 cache lines provide 8,176 8-byte command words.
Pool Number
The pool number can be any unused pool number from [1-7]. The pool number by default is "2".
Buffer Count
It is okay for this pool to run out of buffers: software must check for this condition when allocating buffers.

Cavium Confidential For David Arnold Montara 09/06/2012

6.1 PKO Command Buffer Size

The recommended size is 1024 bytes (8 cache lines). The minimum size allowed by the hardware unit is 128 bytes. See Section 6.3 – "More Precise PKO Command Buffer Size and Count Calculations" for more details.

6.2 PKO Command Buffer Count

Note that it is okay for this pool to run out of buffers. Software is responsible for handling error conditions for the buffer allocation commands (`cvmx_fpa_alloc()` returns NULL if there are not enough available buffers to satisfy the request).

The default SDK configuration is for the following units to use the PKO Command buffers: PCI DMA Engines, PKO, RAID, and ZIP.

The minimum is 2 PKO Command buffers for every user of them, plus 1 per core that may be generating commands. An optimum value is 4. So, if PKO has 256 output ports, ZIP and RAID are being used, and there are 8 PCI DMA Engines, the minimum is:

```
(4 PKO Command buffers) * ((DMA * 8 engines) + RAID + (PKO * 256 ports) + ZIP)
= (4 * 266) = 1064 PKO Command buffers
```

(The number of PKO output ports and PCI DMA Engines varies with the OCTEON processor model.)

See Section 6.3 – "More Precise PKO Command Buffer Size and Count Calculations" for more details.

6.3 More Precise PKO Command Buffer Size and Count Calculations

This section details the math involved if more precise calculations are needed.

<u>Calculate Number of Packets Accommodated by Each PKO Command Buffer:</u>

1. Assume PKO Command buffer size is 1024 bytes.
2. This provides space for 128 8-byte command words.
3. Reserve one command word to point to the next PKO Command buffer in the linked list, leaving 127 command words for use by PKO commands.
4. Determine how many command words are needed for each PKO command. The maximum PKO command size is 3 command words (many are only 2 command words).
5. Calculate the number of PKO commands which are accommodated by each PKO Command buffer: 42 command words / (command words per PKO command). For example, if there are 3 PKO command words needed per PKO command, each buffer will accommodate: truncate (127 command words / 3 words per command = 42 3-word PKO commands, plus 1 command word left over).
6. If there are 4 PKO Command buffers per PKO Output Queue, then the maximum number of packets which can be held in the queue at one time is ((4 buffers * 42 PKO commands) + 1 PKO command) = 169 PKO commands, enough PKO commands for 169 packets.

In the calculation shown above, commands for 169 queued packets will fit into each PKO Output Queue.

Cavium Confidential For David Arnold Maniara 09/06/2012

7 Simple Executive API

Before the FPA API functions can be used Simple Executive must be configured to support the FPA functions.

> ***Note that all cores may allocate and free buffers, but only one core should initialize and enable the FPA. Usually all of the FPA pools are populated by the same core during the application initialization routine.***

Example code showing use of each function, is shown here, and is also in the fpa_simplified example provided at the Cavium Networks Technical Support Site in the same directory where an electronic copy of this chapter may be found.

The following useful non-FPA functions are shown in the example code contained in this section:

- cvmx_dprintf()
- cvmx_bootmem_alloc()
- cvmx_sysinfo_get()
- cvmx_coremask_first_core()
- cvmx_helper_get_version()
- cvmx_coremask_barrier_sync()

7.1 Limits and other Definitions

The maximum number of pools (eight) is a hardware limit. The other limits shown below are strongly recommended limits defined by the Simple Executive software.

Note that CACHE_LINE_SIZE = 128 bytes. (The API uses this as a minimum size. When not using the API the user could erroneously create a buffer which is smaller and try to use it. The FPA hardware would not object, but the system would malfunction. See Section 10.3 – "Common Mistakes".)

Key limits are defined in cvmx-fpa.h:

```
#define CVMX_FPA_NUM_POOLS        8 // maximum of 8 pools: a
                                    // hardware limit
#define CVMX_FPA_MIN_BLOCK_SIZE 128 // (bytes) an API-provided limit
#define CVMX_FPA_ALIGNMENT      128 // (bytes) an API-provided limit
```

CVMX_FPA_MIN_BLOCK_SIZE: The minimum block size is set to 128 bytes to match CACHE_LINE_SIZE.

CMVX_FPA_ALIGNMENT: The CVMX_FPA_ALIGNMENT define is used as an argument to cvmx_bootmem_alloc(). This will cause the memory allocated for FPA-managed buffers to be aligned on CACHE_LINE_SIZE, which is ideal. See – Section 10.3 – "Common Mistakes" for a discussion on why these two defines are important.

Cavium Confidential For David Arnold Mantara 03/06/2012

Note that the buffer size must be an integer multiple of CACHE_LINE_SIZE. This will allow the aligned memory to be divided up into buffers which are also aligned on CACHE_LINE_SIZE. If the buffer size is not a unit of CACHE_LINE_SIZE, when the memory is divided into buffers, the buffers will not be properly aligned.

7.2 Data Structures

The following data structure is created and maintained by software: it is not part of the FPA hardware.

7.2.1 The cvmx_fpa_pool_info_t (pool_info) Data Structure

The cvmx_fpa_pool_info_t data structure (the pool_info data structure) is added by the Simple Executive, and is defined in cvmx-fpa.h. The fields in this data structure are filled in when the pool is populated by either cvmx_helper_initialize_fpa() or cvmx_fpa_setup_pool().

```
/**
 * Data structure describing the current state of a FPA pool.
 */
typedef struct
{
    const char *name;  // The Name of the pool specified by
                       // cvmx_helper_initialize_fpa() or
                       // cvmx_fpa_setup_pool()
                       // when creating the pool - used for debugging
    uint64_t   size;   // Size of the buffers in the pool (bytes)
    void       *base;  // The base memory address of whole block
    uint64_t   starting_element_count;    // The number of elements in
                                          // the pool at creation
} cvmx_fpa_pool_info_t;
```

Note that the pool_info data structure includes pool names which are specified in the cvmx-resources.config file. These names are only used to help programmers remember which pool is used for which purpose.

Cavium Confidential For David Arnold Mamara 09/06/2012

7.3 Easy-to-Use Executive FPA API Functions

The easy-to-use API functions include the pool information functions, the helper function used to set up the typical FPA pool, and allocate and free functions.

7.3.1 Pool Information Functions

The following functions access the `pool_info` data structure.

Note: If the Cavium Networks Ethernet driver is in use, the FPA functions `cvmx_fpa_is_member()`, `cvmx_fpa_get_base()`,`cvmx_fpa_get_block_size()`, and `cvmx_fpa_get_name()` will not work because the driver does not call the API function (`cvmx_fpa_setup_pool()`) which fills in the `cvmx_fpa_pool_info` data structure. All of these functions depend on that data structure (see the *FPA* chapter for more information).

The functions `cvmx_fpa_is_member()`, `cvmx_fpa_get_base()` will not work if the more than one chunk of memory is divided into the buffers used to populate the pool. See the *Essential Topics* chapter, in the "Pool Population" section.

Example code showing the use of these functions follows the table.

Table 10: Pool Information Functions

Pool Information Retrieval Functions
`const char *cvmx_fpa_get_name(uint64_t pool)`
Accesses the `pool_info` data structure to get the name of the specified pool. The name is set up in `cvmx_helper_initialize_fpa()` or `cvmx_helper_initialize_fpa_pool()`. The argument is: `pool`: specify which pool Returns the name of the pool (a string). Note this function will not work if the Cavium Networks Ethernet driver is used to populate the pool.
`void *cvmx_fpa_get_base(uint64_t pool)`
Accesses the `pool_info` data structure to get the starting physical address of the specified pool's chunk of system memory. The argument is: `pool`: specify which pool Returns the start of the physical address of the chunk of memory allocated for the pool. Note this function will not work if the Cavium Networks Ethernet driver is used to populate the pool, or if more than one chunk of memory is divided into the buffers used initialize the pool.
`int cvmx_fpa_is_member(uint64_t pool, void *ptr)`
Accesses the `pool_info` data structure to determine if the buffer belongs in the specified pool by checking whether the buffer's virtual address is within the block of memory allocated for the pool. The arguments are: `pool`: specify which pool `ptr`: a pointer containing the buffer's virtual address Non-zero if the buffer belongs in the pool; otherwise returns zero. Note this function will not work if the Cavium Networks Ethernet driver is used to populate the pool, or if more than one chunk of memory is divided into the buffers used initialize the pool.

Cavium Confidential For David Arnold Mantara 09/06/2012

Pool Information Retrieval Functions
`uint64_t cvmx_fpa_get_block_size(uint64_t pool);`
Gets the configured (#define) size for the pool to get the buffer size for the specified pool. For example, will print the value of `CVMX_FPA_POOL_0_SIZE`. The argument is: `pool`: specify which pool If the pool has been configured into Simple Executive, returns the configured buffer size (block size) for the pool (the value of the #define buffer size for the pool). Otherwise, returns zero. Note this function will not work if the Cavium Networks Ethernet driver is used to populate the pool.

7.3.1.1 Example Code: `cvmx_fpa_get_block_size()`, `cvmx_fpa_get_name()`, `cvmx_fpa_get_base()`

The following code is from the `fpa_simplified` example:

```
/**
 * Prints the information for each pool
 *
 * pool_num:     Pool number (0-7) (no check for illegal pool number)
 * returns void
 */
void application_print_pool_data(uint64_t pool_num)
{
    const char *pool_name;
    void *pool_base;
    uint64_t pool_buffer_size;

    pool_buffer_size = cvmx_fpa_get_block_size(pool_num);
    if (pool_buffer_size > 0) /* pool in use */
    {
      printf("Pool %lu\n", pool_num);
      pool_name = cvmx_fpa_get_name(pool_num);
      printf("    name = %s\n", (char *)pool_name);

      pool_base = cvmx_fpa_get_base(pool_num);

      printf("    base = %p\n", pool_base);
      printf("    block_size = %lu\n", pool_buffer_size);

      // there is currently no API function to get the original total
      // number of buffers (starting element count)
      printf("    Initial buffer count = %lu\n",
             cvmx_fpa_pool_info[pool_num].starting_element_count);
    }
    else
    {
      printf("Pool %lu is not configured in the system.\n", pool_num);
    }
    printf("\n");
}
```

Cavium Confidential For David Arnold Mantara 09/06/2012

7.3.1.2 Example Code: `cvmx_fpa_is_member()`

The following code is from the `fpa_simplified` example:

```
// this function should return "yes"
printf("\nTesting whether buffer %p belongs in my FIRST pool\n",
                  my_buffer);
printf("Expecting the answer to be 'yes'\n");
result = cvmx_fpa_is_member(CVMX_MY_FIRST_POOL, my_buffer);
if (result == 1)
{
    printf("Yes, buffer %p is a member of my pool\n", my_buffer);
}
else
{
    printf("No, buffer %p is NOT a member of my pool.\n", my_buffer);
}
```

7.3.2 Easy-to-Use Initialize, Allocate, and Free Functions

The functions in the next table are easy-to-use. Example code showing the use of these functions follows the table.

Table 11: Easy-to-Use Functions

Easy-to-Use Functions
`int cvmx_helper_initialize_fpa(int packet_buffers, int work_queue_entries, int pko_buffers, int tim_buffers, int dfa_buffers);`
This function: 1) calls `cvmx_fpa_enable()` 2) populates the five commonly used pools using the buffer sizes set in `cvmx-config.h`. Note: if the number of buffers passed to the function is zero, the function will not create a matching pool. Calls `cvmx_bootmem_alloc()` to allocate memory for each pool. Memory will be aligned on `CVMX_CACHE_LINE_SIZE`. Calls `cvmx_fpa_setup_pool()` for each pool. The arguments are the number of buffers for each of 5 pools. To not populate any pool, set the number of buffers for that pool to 0: `packet_buffers`: number of packet data buffers `work_queue_entries`: number of WQE buffers `pko_buffers`: number of PKO command buffers `tim_buffers`: number of TIM buffers `dfa_buffers`: number of DFA buffers Returns 0 on success. Returns non-zero if out of memory.
`void *cvmx_fpa_alloc(uint64_t pool)`
This function gets a buffer from the pool, synchronously. On success, returns a pointer to the buffer's virtual address, otherwise returns NULL.

Cavium Confidential For David Arnold Mantara 09/06/2012

`void cvmx_fpa_async_alloc(uint64_t scr_addr, uint64_t pool)`
Gets a buffer from the specified pool asynchronously; and puts the buffer's physical address into the core's scratchpad memory. Before reading the address from the scratchpad, issue a `SYNCIOBDMA` instruction to force the operation to complete. The arguments are: `scr_addr`: the scratchpad offset to write the buffer address `pool`: specify which pool to get the buffer from Returns void. Side Effect: writes buffer's address to the `scr_addr`, or writes all zeroes to `scr_addr` if the pool is empty.
`void cvmx_fpa_free(void *ptr, uint64_t pool, uint64_t num_cache_lines)`
Frees the buffer back to the specified pool. Issues the `syncws` instruction to flush the data from the write buffer before the buffer free operation. This function is the most convenient and safest way to free a buffer. The `num_cache_lines` argument specifies the number of cache lines to DWB (Don't Write Back - see the *Essential Topics* chapter). Note that this function will not verify that the buffer belongs in the pool it is being freed to!! The arguments are: `ptr`: A pointer containing the virtual address of the buffer `pool`: specify which pool to free the buffer to `num_cache_lines`: The number of 128-byte cache lines to Don't Write Back (DWB) Returns void. Side Effects: 1) flushes the data from the write buffer, 2) frees the buffer back to the specified FPA pool.
`static inline void cvmx_helper_free_packet_data(cvmx_wqe_t *work)`
Free all of the Packet Data buffers for the Work Queue Entry. If multiple Packet Data buffers are needed to hold all of the packet data, then these buffers are connected into a chain. This routine will free all Packet Data buffers in the chain. Note that the Work Queue Entry buffer is NOT freed. This function hides a lot of complexity: it handles the case of a dynamic short (the packet data fits into the WQE and there is no Packet Data buffer), the case where the WQE is in the Packet Data buffer, and the math to locate the start of the buffer. The argument is: work: A pointer to the Work Queue Entry buffer associated with the packet.

7.3.2.1 Example Code: `cvmx_helper_initialize_fpa()`

There are five commonly used pools, one for each type of buffer:

1. Packet Data Buffers
2. Work Queue Entry Buffers
3. PKO Command Buffers
4. Timer Buffers
5. DFA Buffers

If only one or more of the five commonly used pools are needed, call `cvmx_helper_initialize_fpa()` to populate the pools. Note that this function uses default values set in `cvmx-resources.config`. The default values can be changed by editing a local copy of `cvmx-resources.config`. See the *Configuration* chapter for more information.

Cavium Confidential For David Arnold Mantara 09/06/2012

To use the helper function `cvmx_helper_initialize_fpa()`, include `cvmx-helper.h`.

Note: If this function is called, do not call **`cvmx_fpa_enable()`.**

Note that two of the pools are not populated in this example: the argument to the function is "0" for these pools.

The following code is from the `fpa_simplified` example:

```
/**
 * Populate the Packet Data buffer, WQE buffer, and PKO Command
 * buffer pools
 *
 * num_packet_buffers is the
 *           number of outstanding packets to support
 * returns Zero on success
 */
static int application_init_simple_exec(int num_packet_buffers)
{
    uint64_t pool_num; // used in for loop at the end
    int result = 0;

    // Setup the 3 of the 5 commonly used pools
    // make the number of WQE buffers = number of Packet Data Buffers
    // make the number of PKO Command Buffers depend on the number
    // of PKO output queues
    result = cvmx_helper_initialize_fpa(num_packet_buffers,
                              num_packet_buffers,
                             (CVMX_PKO_MAX_OUTPUT_QUEUES * 4), 0, 0);

    // print data for each pool
    for (pool_num = 0; pool_num < CVMX_FPA_NUM_POOLS; pool_num++)
    {
      application_print_pool_data(pool_num);
    }
    return result;
}
```

All system initialization should be done by one core only, usually the first core in the core mask for the application. For example, in the `fpa_simplified` example `main()` function: check if the code is running the first core in the core mask, if so then do the initialization. The following code also shows use of the `cvmx_sysinfo_get()`, `cvmx_coremask_first_core()`, `cvmx_helper_get_version()`, and `cvmx_coremask_barrier_sync()` functions.

The following code is from the `fpa_simplified` example:

```
sysinfo = cvmx_sysinfo_get();
coremask_example = sysinfo->core_mask;

// use the first core to initialize the simple executive, enable
// the FPA, and populate the FPA pools
```

Cavium Confidential For David Arnold Mantara 09/06/2012

```
if (cvmx_coremask_first_core(coremask_example))
{
    printf("SDK Version: %s\n", cvmx_helper_get_version());
    printf("FIRST CORE:  INITIALIZING THE SIMPLE EXEC\n\n");

    result = application_init_simple_exec(number_of_packet_buffers);
    if (result != 0)
    {
        printf("Simple Executive initialization failed.\n");
        printf("TEST FAILED\n");
        // set this flag to cause all the cores to exit
        g_error_flag=1;
    }
}
else
{
     printf("I am NOT the first core: it is NOT MY JOB to initialize"
            " the exec.\n");
}

printf("Board type = %d\n", sysinfo->board_type);

// wait for all cores to get to this point
printf("Wait for all cores to get to the same point\n");
cvmx_coremask_barrier_sync(coremask_example);

// if I don't check for initialization failure, then all the
// cores will hang in the barrier sync:  the first core has
// exited so they cannot ALL reach the barrier sync
if (g_error_flag)
{
    printf("g_error_flag is set:  initialization failed -  exiting\n");
    return g_error_flag;
}

// run the application on each core
printf("All cores ready:  have each core run the application\n");

application_play(coremask_example);
```

7.3.2.2 Example Code: `cvmx_fpa_alloc()`

The following code is from the `fpa_simplified` example:

```
// Allocate a buffer from my pool
printf("\nAllocating a buffer synchronously\n");

my_buffer = cvmx_fpa_alloc(CVMX_FPA_OUTPUT_BUFFER_POOL);
printf("Buffer returned from synchronous allocation = %p\n",
       my_buffer);
```

Cavium Confidential For David Arnold Mantara 09/06/2012

7.3.2.3 Example Code: `cvmx_fpa_async_alloc()`

The following code is from the `fpa_simplified` example:

```
// Asynchronously allocate a buffer
printf("\nGet buffer using the asynchronous allocate function\n");
cvmx_fpa_async_alloc(CVMX_SCR_SCRATCH, CVMX_FPA_OUTPUT_BUFFER_POOL);

// force the async operation to complete
CVMX_SYNCIOBDMA;

async_buffer = (void *)cvmx_scratch_read64(CVMX_SCR_SCRATCH);
printf("Buffer returned from asynchronous allocate = %p\n",
       async_buffer);


if (cvmx_unlikely(!async_buffer))
{
    // no available buffers
    printf("Expect one ERROR message that pool is out of buffers\n");
    printf("ERROR:  Out of buffers in CVMX_FPA_OUTPUT_BUFFER_POOL (pool
%d)\n",
            CVMX_FPA_OUTPUT_BUFFER_POOL);
}
```

7.3.2.4 Example Code: `cvmx_fpa_free()`

The following code is from the `fpa_simplified` example:

```
// free the buffer
printf("\nFreeing the buffer I just got (%p) back to my pool\n",my_buffer);

if (my_buffer != NULL)
{
  cvmx_fpa_free(my_buffer, CVMX_MY_FIRST_POOL, 0);
}
```

7.3.2.5 Example Code: `cvmx_helper_free_packet_data()`

The following code from the `passthrough` example shows the use of helper functions to simplify coding. In this example, software:

- detects an error in the packet
- discards the packet by freeing all of the packet's Packet Data buffers
- frees the WQE buffer.

The routine `cvmx_helper_free_packet_data()` will free the entire Packet Data buffer chain (more than one Packet Data buffer is linked together if the size of the data exceeds the size available in one Packet Data buffer). (More information on the specific error condition can be found in the *PIP/IPD* chapter.)

Note: The Packet Data buffer must be freed before the WQE because the WQE contains the address of the Packet Data buffer.

Cavium Confidential For David Arnold Mantara 09/06/2012

The following code is from the fpa_simplified example:

```
/* 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. We ignore the CVMX_PIP_OVER_ERR error so we
** can support jumbo frames */
if (cvmx_unlikely(work->word2.snoip.rcv_error) &&
                      (work->word2.snoip.err_code != CVMX_PIP_OVER_ERR))
{
    // Work has error, so free the packet data buffers
    // The WQE buffer (work) contains the Packet Data buffer address
    cvmx_helper_free_packet_data(work);

    /* Free the work queue entry */
    cvmx_fpa_free(work, CVMX_FPA_WQE_POOL, 0);
    continue;
}
```

Cavium Confidential For David Arnold Mantara 09/06/2012

7.4 Advanced Functions

Table 12: Advanced FPA Functions

Advanced Functions Used for Special Customization
`void cvmx_fpa_enable(void)`
This function should only be used if `cvmx_helper_initizlize_fpa()` is not called. This function must be called before the pools are populated, otherwise errors will occur. Returns void.
`int cvmx_fpa_setup_pool(uint64_t pool, const char *name, void *buffer, uint64_t block_size, uint64_t num_blocks);`
Steps: 1) Before calling this function, call `cvmx_fpa_enable()` (only once for all pools) 2) then call `cvmx_bootmem_alloc()` with the amount of total memory you need for the pool, and the alignment value of `CVMX_CACHE_LINE_SIZE`; 3) call `cvmx_fpa_setup_pool()` The arguments are: `pool`: the pool number (0-7) `name`: string describing the pool `buffer`: chunk of aligned memory (large enough to be divided into `block_size * num_blocks` buffers) `block_size`: size of buffers; `num_blocks`: number of buffers the memory should be divided into This routine will assume that the chunk of memory is large enough to hold (`block_size * num_blocks`). This routine will set up the `pool_info` data structure, then call `cvmx_fpa_free()` to load the buffers into the pool. Returns 0 on success. Otherwise, returns a negative number which represents the error (bad pool number; block size is less than `CVMX_FPA_MIN_BLOCK_SIZE`; buffer pointer is NULL; buffer is not aligned properly).
`void cvmx_fpa_free_nosync(void *ptr, uint64_t pool, uint64_t num_cache_lines)`
Free the buffer back to the pool without saving the buffer contents (does not issue the `SYNCWS` instruction). ***Note that improper use of this function can result in data corruption.*** The arguments are: `ptr`: a pointer containing the virtual address of the buffer to be freed `pool`: which pool to free the buffer to `num_cache_lines`: the number of 128-byte cache lines to Don't Write Back (DWB) No return value. Side Effect: frees the buffer back to the specified FPA pool.

Cavium Confidential For David Arnold Mantara 09/06/2010

`unit64_t cvmx_fpa_shutdown_pool(uint64_t pool)`
Shutdown a memory pool, and perform health checks on the pool during the shutdown. Report pool health check result back to user via return code. The argument is: `pool`: specify which pool to shutdown Returns zero on success, otherwise returns non-zero. A positive return value is the number of missing buffers. A negative return value specifies there were too many buffers or the buffer pointers are corrupted. Note: buffers are prefetched by the IPD or other hardware units. This function can return a positive number, meaning some buffers have not returned to the pool. If some are missing, it may be because they were prefetched. See the passthrough example for sample code which checks whether missing buffers were prefetched. Note: It is not necessary to call this function to gracefully shutdown the application. To restart the application, the chip must be reset. Since there is no graceful shutdown/restart, there is no requirement to use this function. Because the IPD and cores may have prefetched buffers, this function might add confusion in reporting buffers "missing" when there is no actual error.

7.4.1.1 `cvmx_fpa_enable()`

It is rare for an application to need to call this function, so no example code is included. The `cvmx_helper_initialize_fpa()` code calls `fpa_enable()` at the appropriate time. For an example, see the code in the `executive` directory.

> ***This function should only be called once, before populating the pools. Do NOT call `cvmx_fpa_enable()` if `cvmx_helper_initialize_fpa()` will be called because this helper function calls `cvmx_fpa_enable()`.***

Cavium Confidential For David Arnold Mantara 03/05/2012

7.4.1.2 Example Code: Calling `cvmx_fpa_setup_pool()`

The `cvmx_fpa_setup_pool()` is an advanced function which should only be used for configuring additional pools beyond the customary 5 pools. Before populating the pools, configure the Simple Executive to support the needed pools, and enable the FPA. The FPA can be enabled by calling `cvmx_helper_initialize_fpa()` *before* populating the additional pools, as shown in the example code below. This example also shows the use of `cvmx_boomem_alloc()` and `cvmx_dprintf()`.

For each pool, first allocate the memory using `cvmx_bootmem_alloc`(), then call `cvmx_fpa_setup_pool()`. The following code is from the `fpa_simplified` example:

```
// Setup the 5 commonly used pools
// make the number of WQE buffers = number of Packet Data Buffers
// make the number of PKO Command Buffers depend on the number
// of PKO output queues
```

```
status = cvmx_helper_initialize_fpa(num_packet_buffers, num_packet_buffers,
                                    (CVMX_PKO_MAX_OUTPUT_QUEUES * 4), 0, 0);

result |= status;

// or else you can create your own, allocating a chunk of memory
// aligned on cache line size, then creating the pool
memory = cvmx_bootmem_alloc(pool_buffer_size_1 *
                            pool_number_of_buffers_1, CVMX_CACHE_LINE_SIZE);


if (memory == NULL)
{
    cvmx_dprintf("ERROR: Out of memory initializing pool number %lu(%s)\n",
                 pool_num_1, pool_name_1);
    status = 1;
}
else
{
    status = cvmx_fpa_setup_pool(pool_num_1, pool_name_1, memory,
                                 pool_buffer_size_1, pool_number_of_buffers_1);
}
result |= status;
```

7.4.1.3 The `cvmx_fpa_free_nosync()` Function

The function `cvmx_fpa_free_nosync()` requires an understanding of what the `syncws` instruction does and why it is important. This function does not issue the `syncws` instruction. See the *Configuration* chapter for a discussion on the different synchronization instructions available, why and when they should be used. Note that using this function improperly can result in data corruption.

This function can be used to free multiple buffers, then issue one `syncws` instruction.

7.4.1.4 Example Code: `cvmx_fpa_shutdown_pool()`

The following code (from the `passthough` example) calls `cvmx_fpa_shutdown_pool()`:

```
/**
 * Clean up and properly shutdown the simple exec libraries.
 *
 * @return Zero on success.  Non zero means some resources are
 *         unaccounted for.  In this case error messages will have
 *         been displayed during shutdown.
 */
static int application_shutdown_simple_exec(void)
{
    int result = 0;
    int status;
    int pool;

    cvmx_pko_shutdown();

    for (pool=0; pool<CVMX_FPA_NUM_POOLS; pool++)
    {
```

Cavium Confidential For David Arnold Mantara 09/06/2012

```
            // check whether pool is in use
            if (cvmx_fpa_get_block_size(pool) > 0)
            {
                status = cvmx_fpa_shutdown_pool(pool);

                // Special check to allow PIP to lose packets due to
                // hardware prefetch
                if ((pool == CVMX_FPA_PACKET_POOL) && (status > 0) &&
                    (status < CVMX_PIP_NUM_INPUT_PORTS))
                {
                    status = 0;
                }

                result |= status;
            } // end if this pool is in use
        } // end for all pools

        return result;
    } // end application_shutdown_simple_exec()
```

Cavium Confidential For David Arnold Mantara 09/06/2012

8 Basic Code Review Checklist

This basic checklist is for all users, including those using the FPA API.

Table 13: Basic Code Review Checklist

Basic Code Review Checklist
Simple Executive Initialization:
□ Verify that all needed functionality is enabled, pools are correctly configured, and that there are enough scratchpad areas.
Basic Hardware Unit Configuration:
□ Verify that all configuration register writes have completed before enabling the FPA. Use the `cvmx_write_csr()` function to write configuration registers to avoid potential race conditions (see the *Configuration* chapter for details).
□ Verify that no FPA configuration registers are modified after the FPA is enabled. (Note that the FPA configuration register fields are only used to adjust the In-Unit Buffer Address Cache Size and the Watermarks.)
Configuration of Other Hardware Units:
□ Verify that all the hardware units which use the FPA-managed buffers were correctly configured.
Hardware Unit Enable:
□ Verify `cvmx_fpa_enable()` is called only once, and only by the initializing core.
□ Verify `cvmx_fpa_enable()` is called before the pools are populated.
□ Verify that the FPA is enabled *after* the FPA registers are configured.
Hardware Unit Initialization:
□ Verify the amount of memory allocated is sufficient for the `(number of buffers) * (buffer size)`.
Buffer Size:
□ Verify the PIP/IPD configuration variable `IPD_PACKET_MBUFF_SIZE[MB_SIZE]` is correctly set. If the value of this variable is larger than the size of the Packet Data buffer, the IPD will write beyond the end of the buffer, corrupting memory.
Buffer Alignment:
□ Verify the buffers allocated for each pool are aligned on 128 bytes at the beginning (When using `cvmx_bootmem_alloc()`, specify `CVMX_CACHE_LINE_SIZE` as the second argument).
□ Verify the buffer end alignment is correct: allocate buffers in multiples of `CVMX_CACHE_LINE_SIZE`.
Buffer Count:
□ Verify sufficient Packet Data buffers (pool 0) so that the Packet Data buffer pool will not run out of buffers. Use the PIP/IPD congestion control mechanisms to limit the maximum number of Packet Data buffers used.
□ Verify sufficient Work Queue Entry buffers so the WQE buffer pool will not run out of buffers. Use the PIP/IPD congestion control mechanisms to limit the maximum number of queued packets (limit use of WQE buffers) used by PIP/IPD.

Cavium Confidential For David Arnold Manfara 09/06/2012

Buffer is Freed Correctly:
□ Verify the buffer is not freed more than once.
□ Verify the buffer is freed eventually (no memory leaks).
□ Verify `cvmx_fpa_free()` was not given a NULL or illegal pointer as an argument.
□ Verify the buffer was not freed to the wrong pool. (For example, freeing a 128-byte buffer to pool 0 (the Packet Data buffer pool) can cause the IPD to have a smaller buffer than expected (if `IPD_PACKET_MBUFF_SIZE[MB_SIZE]` is configured for a buffer larger than 128 bytes, when IPD writes the packet data to the buffer, it may overwrite adjacent memory).

Cavium Confidential For David Arnold Mantara 09/06/2012

9 Internal Details

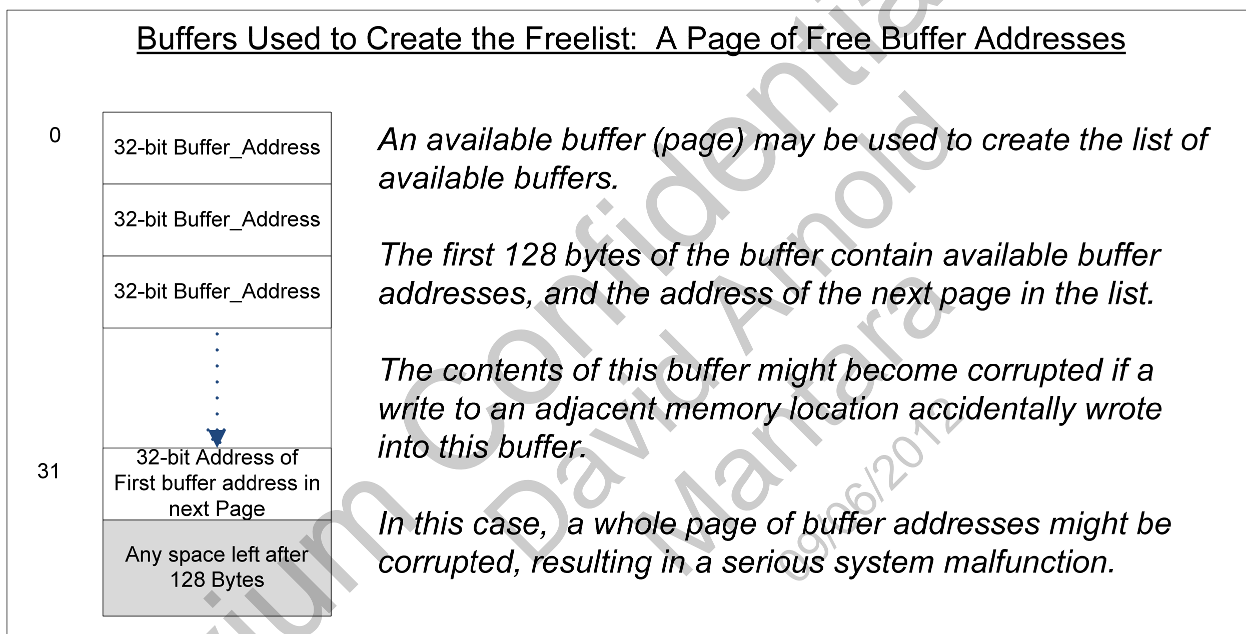
The following information is helpful for advanced initialization, debugging, and performance tuning. It is not necessary to read this section to use basic FPA functions. This section describes the internal buffer organization, the In-Unit Buffer Address Cache, and Watermarks.

9.1 Buffer Organization

The FPA uses some of the free buffers to create an internal data structure: a free list of the available buffers. In the *Hardware Reference Manual*, each buffer is referred to as a *page*.

Each page contains an array of 32 physical addresses. The 7 least significant bits are not stored because the address must always be 128-byte aligned, so 32 bits is enough to store the physical address of each buffer. Of the 32 addresses, 31 contain buffer addresses, and one is used to point to the next page. (Note that 32 addresses * 4 bytes per address = 128 bytes. This is one reason why buffers must be at least 128 bytes long.)

Figure 2: A Free Buffer used as a Page of Free Buffer Addresses



Note that overwriting memory adjacent to a buffer might cause a whole page of buffer addresses to be overwritten, resulting in a serious malfunction.

Cavium Confidential For David Arnold Marnara 09/06/2012

Each page is identified by its page index (the page index is created and managed by the FPA). The first page into the pool (the last page out of the pool) is index number "0". This information is used by the FPA to check for memory corruption. When the FPA reports an error, it may include the page number in the report. See Section 10.1 – "Interrupts and Detected Error Conditions".

The FPA register `FPA_QUEn_PAGE_INDEX` contains the current page index for the pool (the highest index available). This value can be used to help in debugging and tuning whether the number of pages in the pool matches expectations.

Because the internal structure is a linked list of pages created by the free buffers, there is no limit to the number of buffers in a pool. (See Figure 4 – "FPA Buffer Pool: Simplified Internal View".)

The buffers pointed to by a page are all part of the same block of DRAM allocated by the pool, but are otherwise unrelated: they are not necessarily contiguous.

> ***WARNING: Writing beyond the end of the buffer will corrupt memory. It may corrupt a page of addresses, which would cause a serious problem.***

9.2 In-Unit Buffer Address Cache (Address Cache)

There is an In-Unit Buffer Address Cache (Address Cache) of buffer addresses in the FPA memory, which is divided between the pools (per-pool allotment). On some OCTEON models, the per-pool allotment is fixed; on other OCTEON models it is configurable.

Processors which allow configuration of this unit have the following registers: `FPA_FPFx_SIZE` and `FPA_FPFx_MARKS`. On these OCTEON models, the per-pool allotment and the per-pool watermarks are configurable. (Watermarks are discussed in the next section.) See the following table for the default values for CN38XX, CN54XX, CN55XX, CN56XX, CN57XX, CN58XX, CN63XX, and CN68XX):

Table 14: Defaults for Configurable Per-Pool Buffer Cache and Watermarks

Pool	Buffer Cache Size	Per-Pool Allotment (H/W Default)	Write Watermark (H/W Default)	Read Watermark (H/W Default)
Pool 0	2048 32-bit entries	256 (0x100) entries	196 (0xC4)	64 (0x40)
All others (See Note1)	2048 32-bit entries	256 (0x100) entries	196 (0xC4)	64 (0x40)
Notes				
Note1: These values are for CN38XX, CN54XX, CN55XX, CN56XX, CN57XX, CN58XX, and CN63XX. For other OCTEON models, check the HRM for the default FPA register values.				

Cavium Confidential For David Arnold Mantaro 09/06/2012

If the FPA_FPFx_SIZE and FPA_FPFx_MARKS registers are not present (for example, the CN31XX, CN50XX, and CN52XX), then the size of the buffer cache is fixed, as shown in the following table:

Table 15: Fixed Per-Pool Watermarks and Size (If No Config. Registers)

Pool	Buffer Cache Size	Per-Pool Allotment (Fixed)	Write Watermark (Fixed)	Read Watermark (Fixed)
All pools	512 32-bit entries	64 entries	56	16
Notes				
Note1: These values are for CN31XX, CN50XX, and CN52XX.				

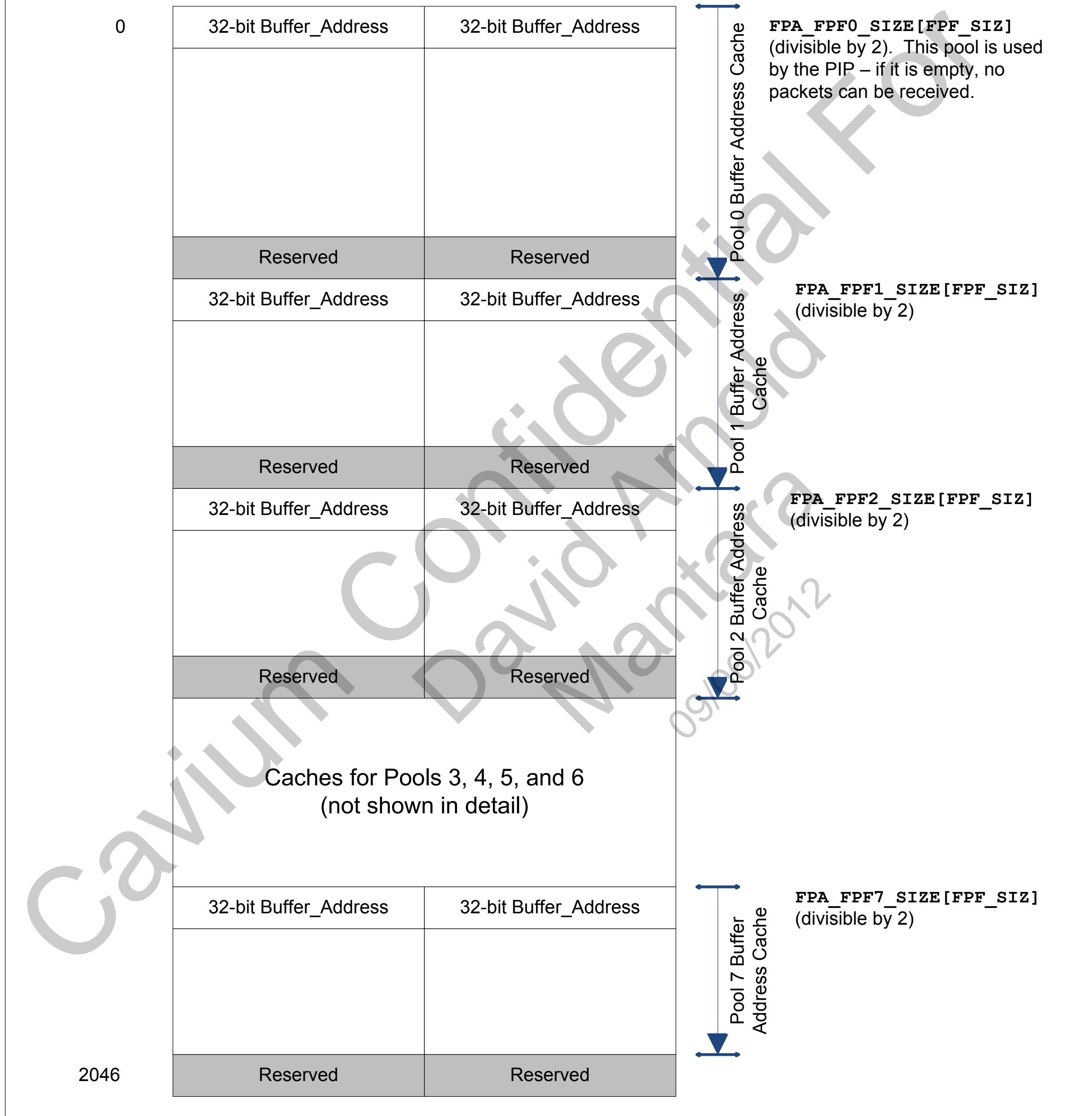
If the specific model of OCTEON being used does not have configurable Address Cache allotments and watermarks, the other information contained in this section is still relevant. Understanding the Address Cache internals will provide insight into the impact of memory corruption on the pool. This information can help prevent/locate coding errors.

Cavium Confidential For David Arnold Mantara 09/06/2012

Figure 3: FPA In-Unit Buffer Address Cache – Simplified Internal View

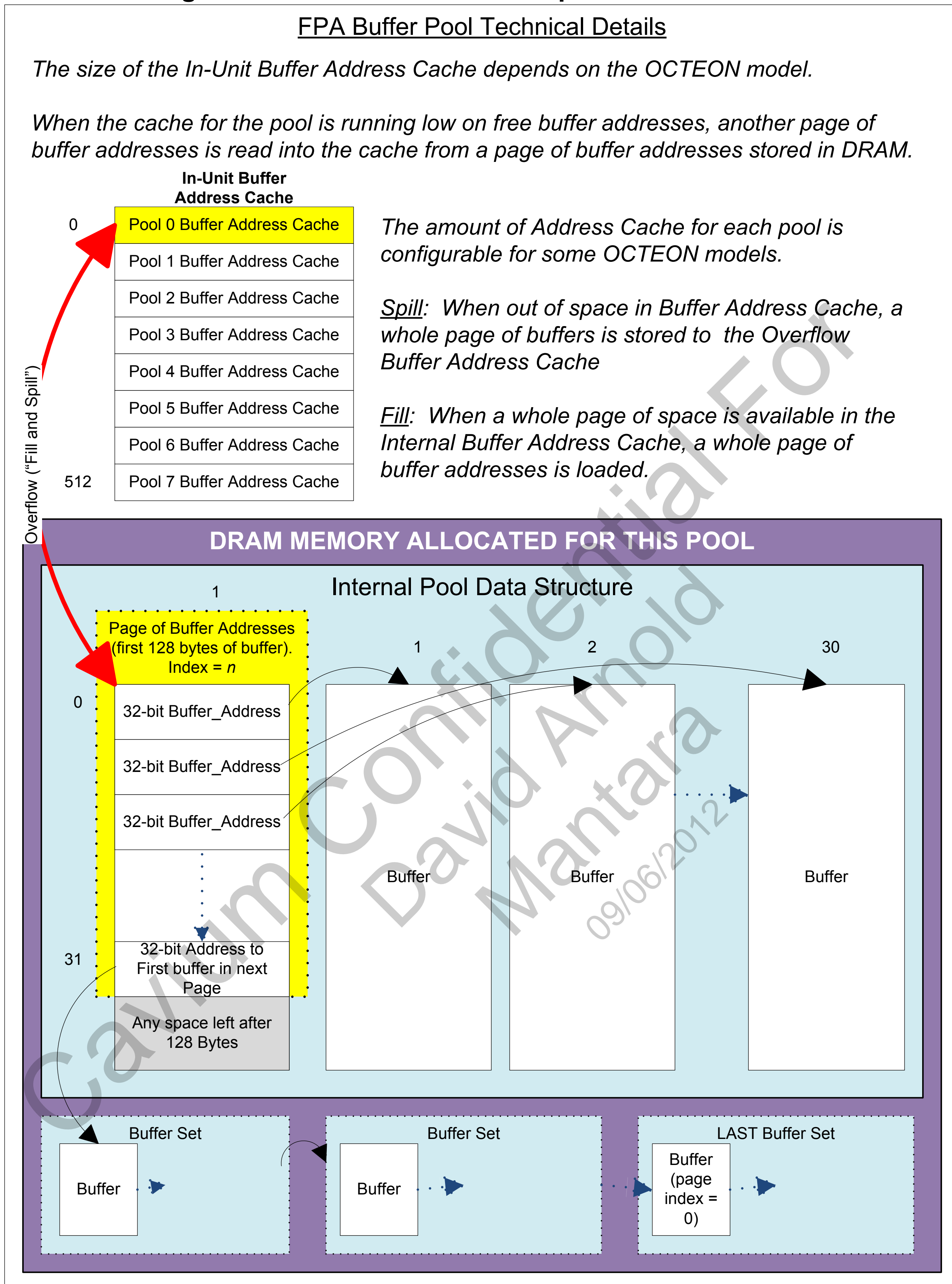
FPA In-Unit Buffer Address Cache With Configurable Per-Pool Allotments

- For some processors, such as the CN58XX, the FPA's In-Unit Buffer Address Cache holds 2048 32-bit entries, and the amount of cache (per-pool allotment) reserved for each pool is configurable. This example shows the CN58XX Address Cache configuration (2048 entries).
- Each buffer address is 32 bits long.
- For each pool, the amount of buffer pointers in the Address Cache is configurable via the **`FPA_FPFx_SIZE[FPF_SIZ]`** register field.
- The total of all the sizes for all the pools cannot exceed the Address Cache size.
- For each pool, two of the 32-bit entries are reserved.
- Higher priority queues can have larger per-pool allotments.



Cavium Confidential For David Arnold Mantara 09/05/2012

Figure 4: FPA Buffer Pool: Simplified Internal View



Cavium Confidential For David Arnold Mantara 09/06/2012

On OCTEON models with configurable Address Cache thresholds, configuring the Address Cache properly will improve performance:

- No Wasted Resources: Address Cache space is not wasted on unused pools
- Faster:
  - Buffers addresses are available in the Address Cache when a `buffer_allocate` request is made instead of needing to bring in another page of buffers from L2/DRAM
  - Freed buffer addresses are not stored to DRAM unnecessarily

Address Cache Configuration Reminders:

- Sum of Sizes: The sum of all per-pool Buffer Address Cache allotments (sizes) cannot exceed the Address Cache size. (The Address Cache size varies with OCTEON model (for example 512 or 2048).)
- Assign Sizes in Order: The sizes must be assigned in order: pool [0, 1, 2, 3, 4, 5, 6, 7] without gaps (not pool [0, 1, 4], omitting [2, 3, 5, 6, 7]). Initialize the size of any unused pool to 0.
- Set Size for Unused Pools to 0: The per-pool allotment for unused pools should be set to 0. When the chip is reset, the per-pool allotment non-zero default values are set for all pools: configuration software should change this. This will allow the entries to be re-allocated to an in-use pool.
- Adjust Size by Need: The greatest per-pool allotment should be used for the highest-priority pool (one which needs the greatest number of addresses fastest).
- Size Divisible by Two: The per-pool allotment per pool *must* be divisible by two.
- Account for Overhead: For every in-use pool, two of the per-pool Buffer Address Cache entries are reserved for internal use, as shown in Figure 3 – "FPA In-Unit Buffer Address Cache – Simplified Internal View".

WARNING: Remember to set the `FPA_FPFx_SIZE[FPF_SIZ]` field to 0 for unused pools. Otherwise, they will waste space in the Address Cache. Also, if the per-pool allotments for unused pools are not set to zero, then an error can occur if the user forgets to include the unused pool's allotment in the sum of all per-pool allotments, which can cause the total per-pool allotments to exceed the Address Cache size.

9.3 Watermarks for the In-Unit Buffer Address Cache

For each pool, the In-Unit Buffer Address Cache has a read and a write watermark

When the cache for the pool is running low on free buffer addresses (hits the read watermark), another page of buffer addresses is read into the cache from a page of buffer addresses stored in DRAM.

When the cache for the pool is overflowing with too many free buffer addresses (hits the write watermark), it stores a page of buffer addresses out to DRAM.

Cavium Confidential For David Arnold Mantara 02/06/2012

Table 16: Watermark Facts

Item	Requrements and Recommendations
Read Watermark: Recommended	25% of the pool's Address Cache allotment (`25% * FPA_FPFx_SIZE[FPF_SIZE]).`
Read Watermark: Minimum	`>= 16`
Read Watermark: Maximum	`FPA_FPFx_SIZE[FPF_SIZE] - 34`
Write Watermark: Recommended	75% of the pools' Address Cache allotment (`75% * FPA_FPFx_SIZE[FPF_SIZE]`).
Write Watermark: Maximum	`<= (FPA_FPFx_SIZE[FPF_SIZE] - 2).`
Difference Between Two Watermarks	`>= 34` (See Note1) (`FPA_FPFx_MARKS[FPF_WR] - FPA_FPFx_MARKS[FPF_RD] >= 34`).
Notes	
Note1: If the two watermarks are not at least 34 entries apart, then the system will bring in addresses only to write them out again immediately.	

Cavium Confidential For David Arnold Mantara 09/06/2012

Figure 5: FPA Read Watermark: Simplified Internal View

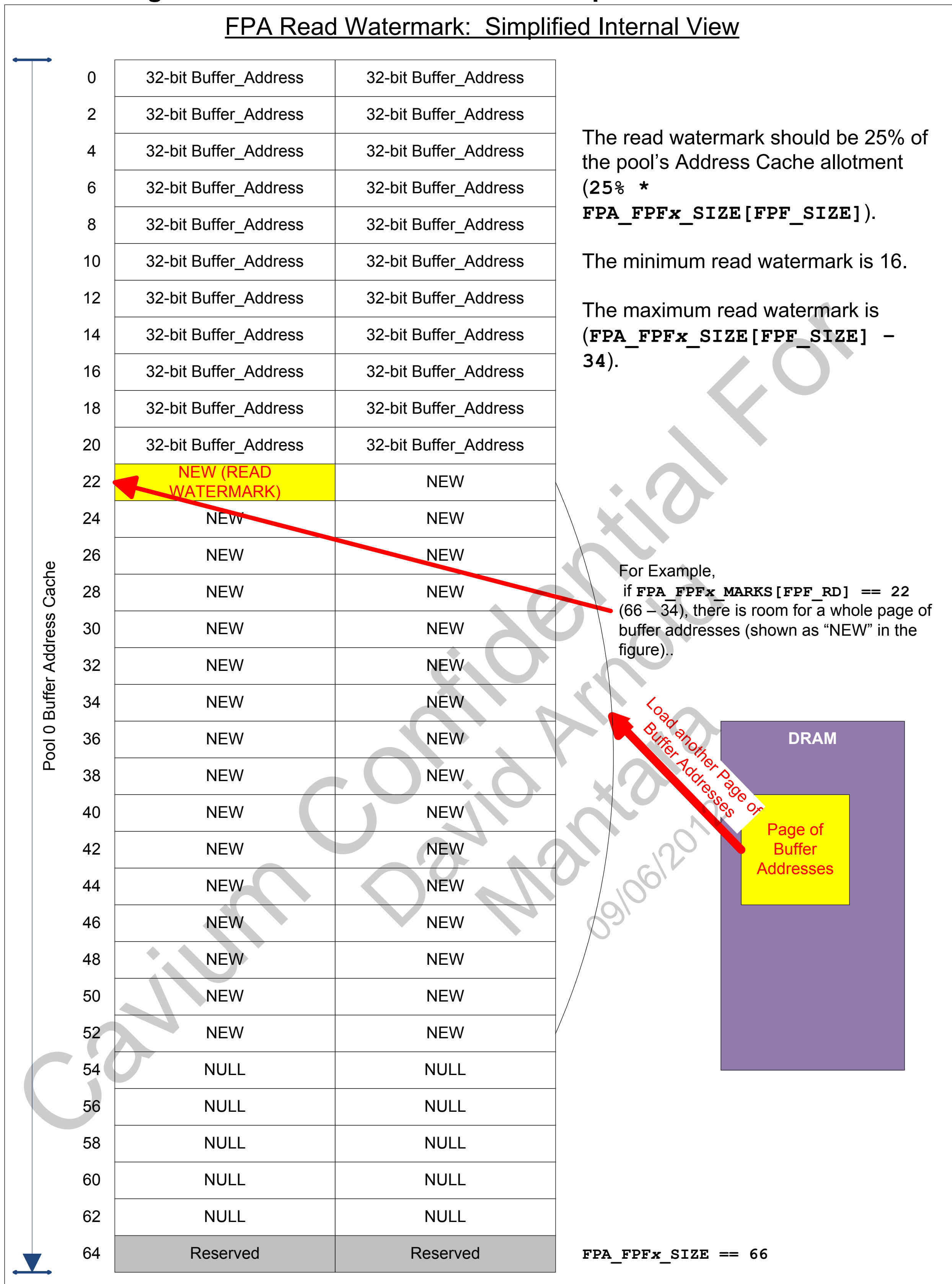
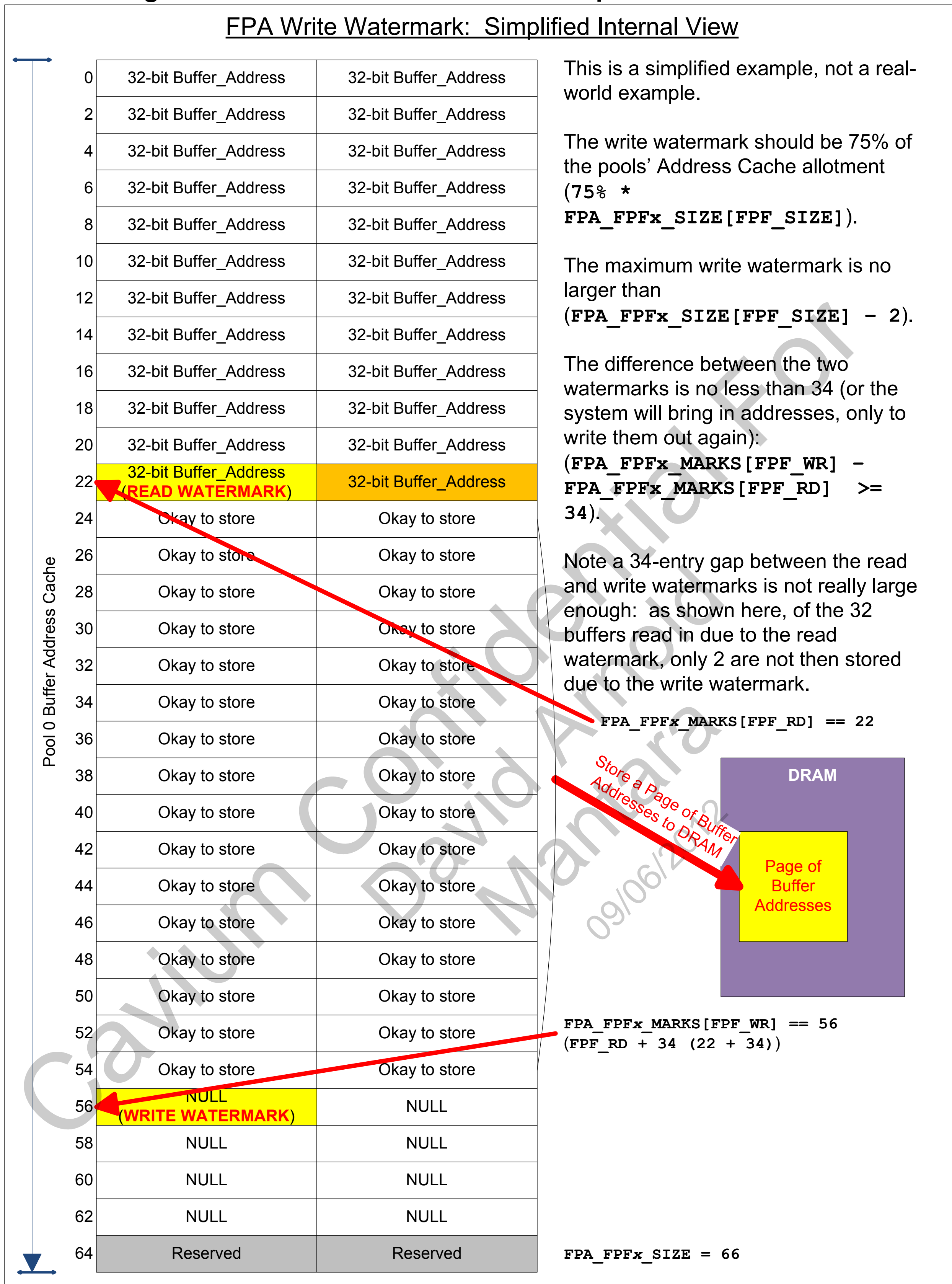


Figure 6: FPA Write Watermark – Simplified Internal View



10 Debugging

OCTEON application programmers must be knowledgeable about OCTEON configuration capabilities, application requirements, and how different elements can interact to change valid limit scenarios.

The hardware accelerators do not perform any form of error checking on configured values.

The Simple Executive library can be built with additional built-in code that performs various sanity checks on parameter values. To enable these checks for logical errors, run `env-setup` with the `--checks` option (`source env-setup –checks <OCTEON_MODEL>`) and test with all expected combinations of traffic flows and events. The `--checks` option enables various consistency and programming checks. To see the effect of enabling these checks, type `env` on the command line:

```
host$:  env | grep GLOBAL_ADD
OCTEON_CPPFLAGS_GLOBAL_ADD= -DUSE_RUNTIME_MODEL_CHECKS=1
-DCVMX_ENABLE_PARAMETER_CHECKING=1 -DCVMX_ENABLE_CSR_ADDRESS_CHECKING=1
-DCVMX_ENABLE_POW_CHECKS=1
```

The most common errors are:

- Freeing the same buffer more than once
- Freeing an invalid pointer.

This section discusses these and other common errors. See Section 9 – "Internal Details", and also the *Configuration* and *Advanced Topics* chapters, where race conditions are discussed. Another place to look for common errors is the *Congestion Control* section of the *PIP/IPD* chapter.

If the FPA detects memory corruption, it will send an interrupt to the Central Interrupt Unit (CIU). From there the interrupt will go to the cores. Memory errors in the buffer pool may be caused by overwriting memory, by freeing a pointer more than once, or by passing an incorrect value to `cvmx_fpa_free()`. The FPA detects some, but not all, error conditions, and sends an interrupt to the CIU.

Read the FPA registers to get debugging information such as the number of buffers free, or the number of free pages of buffers (discussed below), to see if the pool has more or less buffers than expected. For example, use `FPA_QUEn_AVAILABLE` to find out how many free buffers are currently in the pool.

One way to check for problems in the pools is to use `cvmx_fpa_shutdown_pool()`, which checks each pool, verifies that each buffer belongs in the pool, and that all buffers have been returned to the pool, before it disables the FPA.

Note: Packet Data buffers and WQE buffers are prefetched by the PIP/IPD. The `cvmx_fpa_shutdown_pool()` function can return a positive number, meaning some buffers have not returned to the pool. If some are missing, it may be because they were prefetched. See Section 7.4.1.4 – "Example Code: `cvmx_fpa_shutdown_pool()`" for example code showing a check for buffers missing due to prefetch.

Cavium Confidential For David Arnold Marnara 09/06/2012

10.1 Interrupts and Detected Error Conditions

The FPA will detect some types of memory corruption in the pools and send an interrupt to the CIU and from there to the cores. It will *not* detect all errors. The types of errors which are detected are listed below.

The interrupts are used to detect memory corruption, which can be caused by improperly designed or coded software. For code inspection suggestions, see Section 12 – "Advanced Code Review Checklist".

All the interrupts may be enabled: it does not matter if interrupts are enabled for unused pools. No spurious interrupts will occur for the unused pools.

If interrupts are *not* enabled, the interrupt status may be retrieved by reading the FPA_INT_SUM register.

> ***Note: There is no interrupt for "Pool is out of buffers". You must populate the pools so that they do not run out of buffers. If the pool is out of buffers, an address value of zero will be returned. The function `cvmx_fpa_alloc()` will return NULL.***

10.1.1 Permission Error (PERR)

PERR: The page of buffer addresses in memory is marked as owned by the FPA. It also has its page index, and an *owner* (which pool owns it) stored inside (this information is not shown in the figures). This information helps the FPA check for memory corruption. The PERR (Permission error) interrupt means that, when the FPA tried to read in a new page of addresses, either the FPA permission bit was not set, the page belongs to the wrong pool, or the page index did not match the expected value. This might happen if the page's memory had become overwritten.

If a PERR interrupt occurs, the FPA_QUE_ACT register (fields ACT_QUE and ACT_INDX) will be set to the actual values of the pool number and page index number read from memory. The FPA_QUE_EXP register (fields EXP_QUE and EXP_INDX) will be set to the values the FPA expected to find on the read. To get this debugging information, read these registers from software.

10.1.2 Page Count Off (Incorrect) Error (COFF)

COFF: The FPA marks the last page of buffer addresses (the first one in) with a "stack end tag" (no more pages available). If the "Count Off" error is set then we have reached the pool's "stack end tag" but the FPA_QUEn_PAGE_INDEX[PG_NUM] (the index of the current page) is not 0 (the PG_NUM count is incorrect). This might happen if a page's memory had become overwritten.

10.1.3 Underflow (UND)

UND: The last page of addresses can become corrupted. If this happens, then the "stack end tag" might be incorrect, causing the FPA to not detect the end of the pool correctly. In that case, then FPA_QUEn_PAGE_INDEX[PG_NUM] (the index of the current page) would become negative. If that happens, the UND error is asserted.

Cavium Confidential For David Arnold Mantara 09/06/2012

10.1.4 Single and Double Bit Memory Errors (SBE, DBE)

FED1_DBE, FED1_SBE, FED0_DBE, and FED1_SBE: These errors indicate OCTEON internal hardware conditions which should not normally occur: an internal memory error has been detected. Software cannot cause these failures. A Single Bit Error will be corrected by the hardware: no software action is needed, other than noting the occurrence of this unusual problem. Double Bit errors cannot be corrected by the hardware. Software will need to manage Double Bit errors, usually by resetting the system or raising an alarm. Typical causes of this type of error are power supply noise or droop (a temporary drop in power level).

10.2 Debugging and Status Information

Debugging and status information can be obtained by reading the following registers:

Table 17: Status and Debugging Registers

Brief Description	Register	Fields
Debugging and Statistics Registers		
The number of free buffers available in the pool.	FPA_QUEx_PAGES_AVAILABLE	QUE_SIZ
The index of the current page (is also the number of pages of buffer addresses in the pool).	FPA_QUEx_PAGE_INDEX	PG_NUM
Expected page index read from memory (latched on PERR). The page index is an FPA-internal number.	FPA_QUE_EXP	EXP_INDX
Expected FPA Pool number (queue) read from memory (latched on PERR)	FPA_QUE_EXP	EXP_QUE
Actual page number index (latched on PERR). The page index is an FPA-internal number.	FPA_QUE_ACT	ACT_INDX
Actual FPA pool number (queue) read from L2C (latched on PERR)	FPA_QUE_ACT	ACT_QUE

> ***Note: For Packet Data buffers, the best register to read to retrieve the number of available buffers is the IPD register `IPD_QUE0_FREE_PAGE_CNT [Q0_PCNT]`, which is on the I/O bus. The value of the FPA register field `FPA_QUE0_AVAILABLE[QUE_SIZ]` will be less accurate because it is on the much slower RSL bus, so the snapshot value will be older than the value of `IPD_QUE0_FREE_PAGE_CNT[Q0_PCNT]`.***

Cavium Confidential For David Arnold Mantara 09/06/2012

10.3 Common Mistakes

This section provides information on common mistakes which can cause difficult to debug errors. For other common mistakes, see Section 8 – "Basic Code Review Checklist" and Section 12 – "Advanced Code Review Checklist".

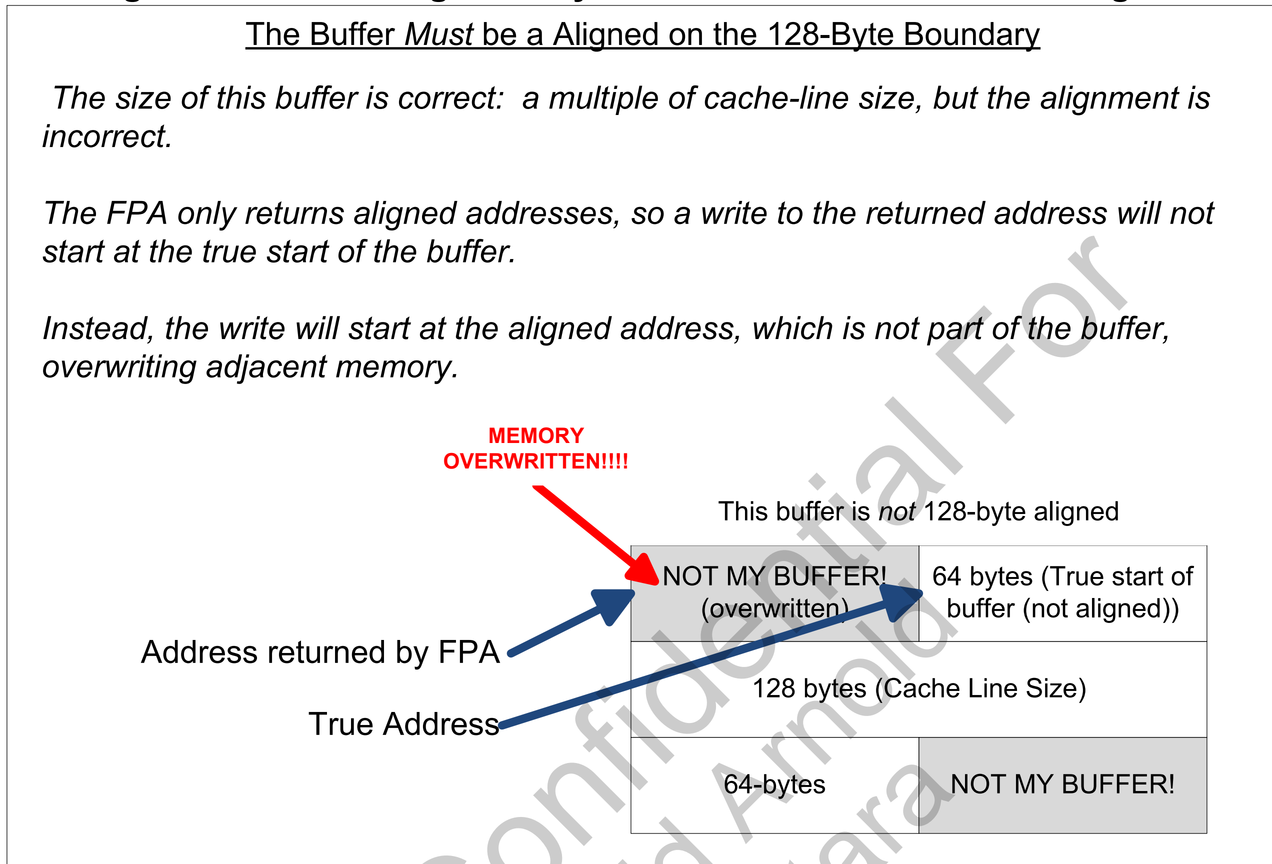
In addition to the examples shown here, software can also cause difficult to debug errors by freeing either NULL or invalid pointers.

The FPA hardware does not have knowledge of which buffer addresses are valid, which pool they belong in, or the expected buffer size.

10.3.1 Buffer Alignment: Bad Alignment at Start of Buffer

The start of the buffer needs to be aligned on the cache line size boundary. If it is not, adjacent memory can be overwritten as shown below.

Figure 7: Overwriting Memory: Buffer Not Cache Line Size Aligned



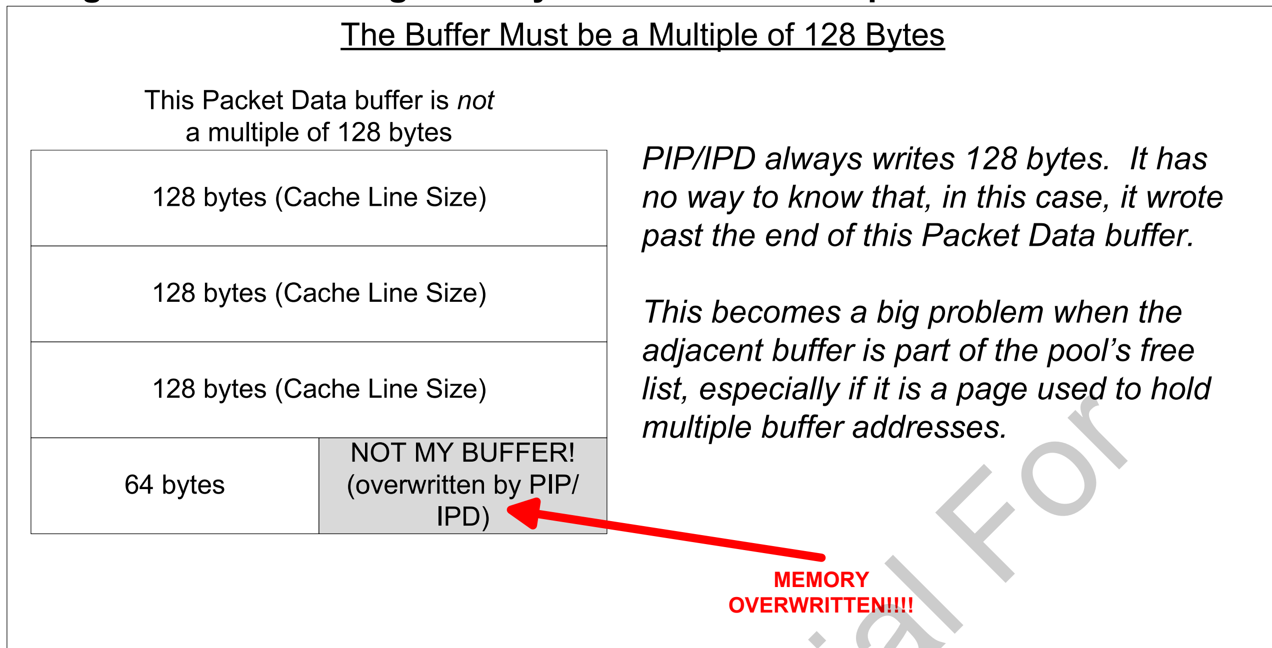
WARNING: It is critical to verify that the memory allocated for each buffer is aligned on a 128-byte boundary at the start. Otherwise, the FPA (which assumes buffer memory is 128-byte aligned) will not return the address of the true start of the buffer. When a write to the buffer occurs, adjacent memory will be corrupted.

10.3.2 Buffer Alignment: Bad Alignment at End of Buffer

Some buffers, especially Packet Data buffers, must be whole units of cache line size. If the end of the buffer is not correctly aligned, adjacent memory can be overwritten as shown in the following figure.

Cavium Confidential For David Arnold Mantara 09/06/2012

Figure 8: Overwriting Memory: Buffer not a Multiple of Cache Line Size



WARNING: For PIP/IPD, It is critical to verify that the Packet Data buffer size is a multiple of the cache line size (128 bytes). The PIP/IPD always writes packet data in complete 128-byte blocks, including the last data in the packet. Since PIP/IPD will write 128 bytes even if that exceeds the end of the buffer, the adjacent memory will become corrupted.

Memory corruption can occur if, as shown above, the write occurs before the beginning or after the end of the buffer.

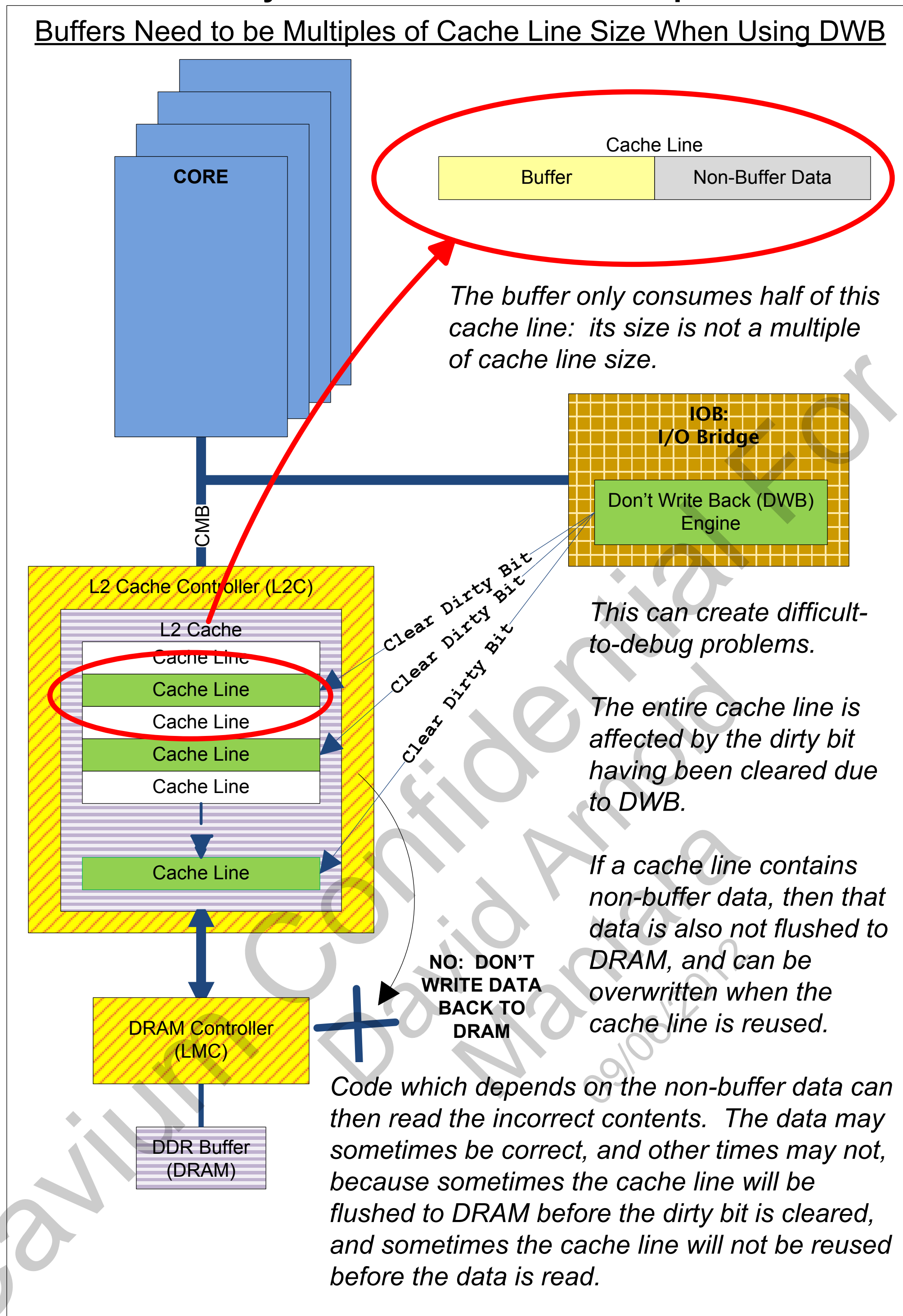
The details shown in Section 9 – "Internal Details" provide insight into the impact of coding errors on the system. For instance, overwriting a buffer can cause a pool to become corrupted if the adjacent memory contains a page of buffer addresses in use by the same or a different pool. Corrupting the page of buffer addresses corrupts the pool's internal data structure, severely damaging the pool. The resultant system error can be difficult to track to the original cause of the problem.

10.3.3 Buffer Size and Don't Write Back (DWB)

If the buffer is not a multiple of cache line size and Don't Write Back (DWB) is used, then data stored in the same cache line might not be stored to DRAM. This problem can be difficult to debug because sometimes the flush of the cache line to DRAM occurs before the dirty bit is cleared. See the *Advanced Topics* chapter for more information.

Cavium Confidential For David Arnold Mantara 09/06/2012

Figure 9: DWB and Why Buffers Need to be a Multiple of Cache Line Size



Cavium Confidential For David Arnold Maniatis

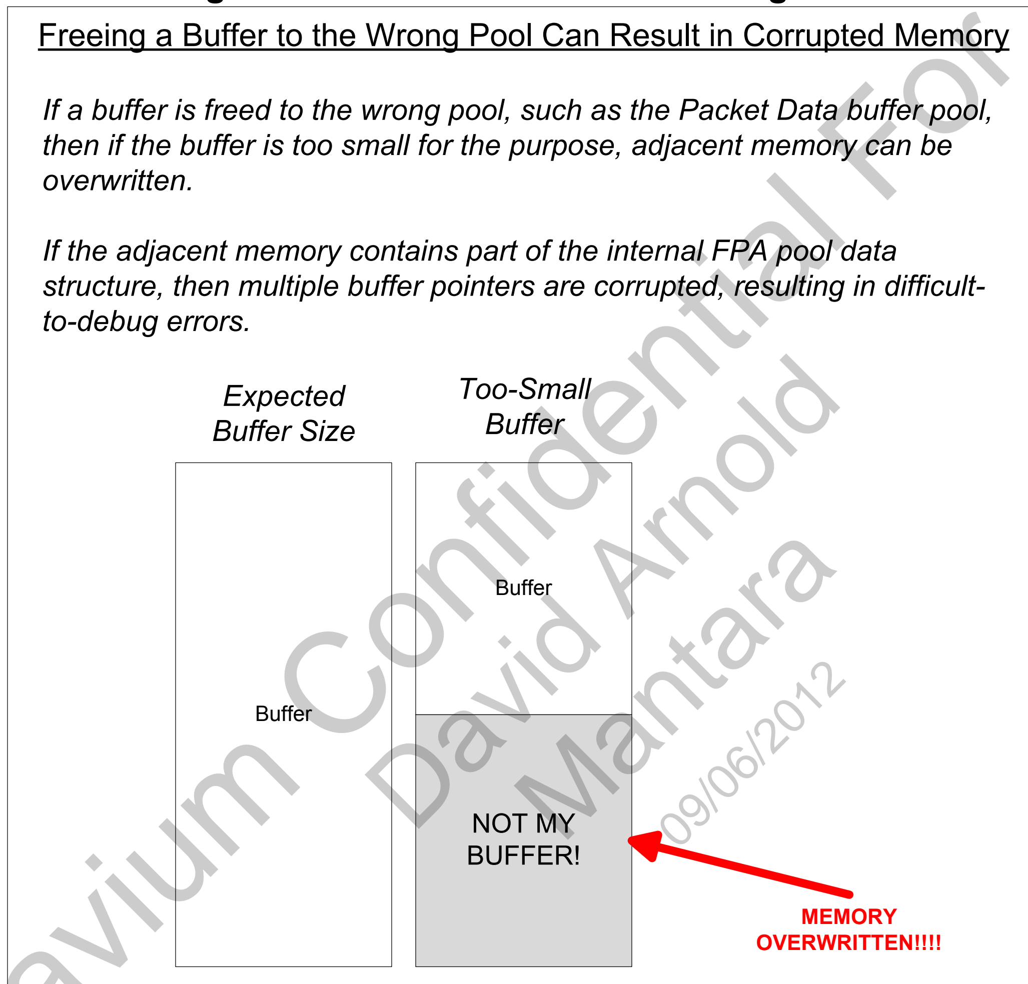
10.3.4 Buffer Freed to the Wrong Pool

If a buffer is freed to the wrong pool, then two problems can occur:

- The original pool can unexpectedly run out of buffers.
- If the freed buffer is smaller than other buffers in the pool, the requester can receive a buffer too small for the purpose and overwrite adjacent memory.

A similar problem can occur when one pool has different sizes of buffers, and the requester assumes they are all a larger size. Debugging these situations is not worth the effort: have each FPA pool contain only one buffer size, and be careful to not free buffers to the wrong pool.

Figure 10: Buffer Freed to the Wrong Pool



Cavium Confidential For David Arnold Mantara 09/06/2012

To avoid this, be careful when configuring hardware units which automatically free buffers (see Section 15 – "Configuring Units Which Allocate/Free FPA Buffers"), and examine software to verify that buffers freed by software are freed to the correct pool.

10.4 Buffer Freed More than Once

If the same buffer is accidentally freed more than once, then the buffer may later be allocated more than once, causing multiple units to write data to it. This will result in unpredictable system behavior.

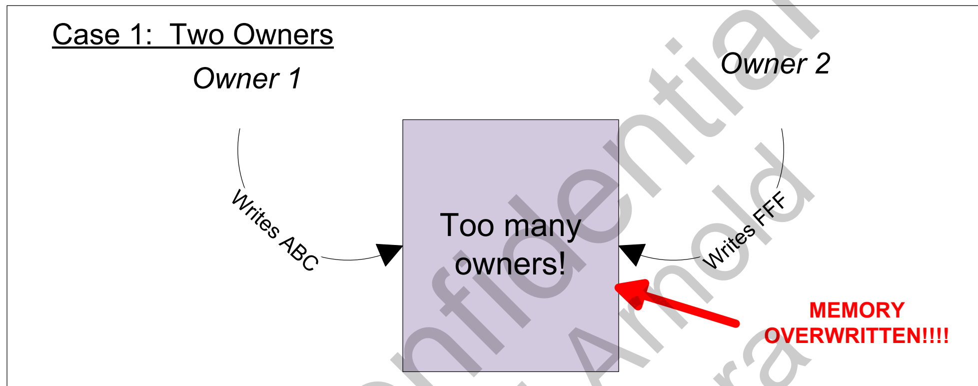
Figure 11: Buffer Accidentally Freed More Than Once

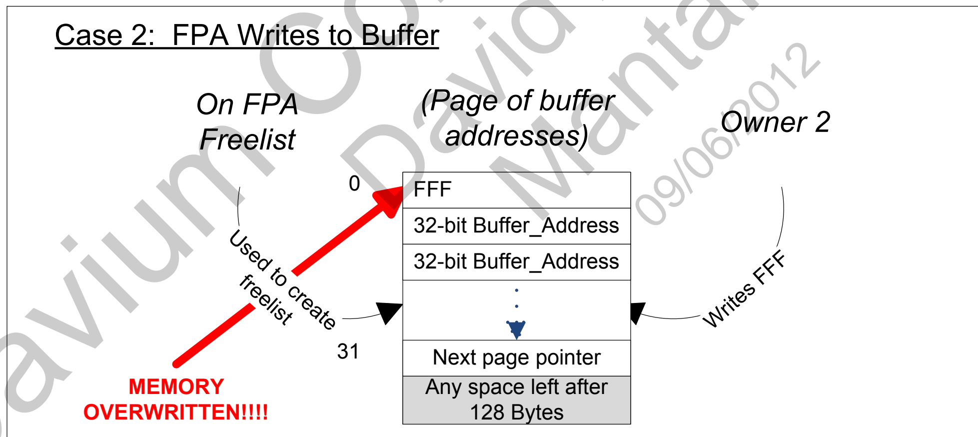
Accidentally Freeing the Same Buffer More than Once
Can Lead to Memory Corruption

If the same buffer is accidentally freed more than once:

Case 1: It can then be allocated to more than one new owner. The data in the buffer then become corrupted as multiple owners write to it.

Case 2: Worse yet, if one owner frees it and it becomes a page of buffers used by the FPA to manage the pool, and then an owner writes to it, the corruption affects not just one buffer, but the internal pool data structure.





Cavium Confidential For David Arnold Mantara 09/06/2012

11 Performance Tuning

In this section, advanced initialization items used in performance tuning are discussed.

Performance tuning includes adjusting the In-Unit Buffer Address Cache sizes and watermarks; ensuring the PIP/IPD will not run out of packets; and tuning the DWB counts. It is also worthwhile to prefetch buffers whenever possible (see the passthrough example for an example of prefetching), and to verify that there are sufficient Packet Data buffers.

For information on how to access registers and register fields, see the *Advanced Topics* chapter.

11.1 Enough Buffers

Verify that enough Packet Data buffers and Work Queue Entry buffers were allocated so that the IPD does not run out of buffers. If the IPD runs out of Work Queue Entry buffers, it will stop receiving packets on all ports until more buffers become available. See the *PIP/IPD* chapter for information on congestion causes and congestion-control mechanisms. For example, per-port backpressure and per-QoS RED/WRED can be used to help the highest-priority traffic continue to flow, while lower-priority traffic is backpressured or dropped.

11.2 Prefetch Buffers

Use the `cvmx_fpa_async_alloc()` function to prefetch buffers. This will prevent the core from stalling while waiting for the `cvmx_fpa_alloc()` function to return.

11.3 Initializing the Per-Pool Address Cache Allotment or Watermarks

In addition to the normal FPA initialization, the per-pool Address Cache allotment and watermarks may be initialized on some OCTEON models. This initialization is done by writing to the FPA Initialization Registers for each pool, including for the unused pools. (The default size is non-zero for unused pools, so they consume space unless the default value is changed.) Functions to set cache allotment or watermarks are not provided by SDK 2.0 API or earlier SDKs. Future SDKs may provide this support.

A larger address cache can be reserved for pools which have more frequent and/or large number of buffers requested, such as the Packet Data buffer pool or the WQE buffer pool (IPD prefetches about 128 of these at a time).

Remember that for each pool, two of the addresses are reserved for system use.

To initialize the per-pool Address Cache allotment and watermarks, modify the following registers:

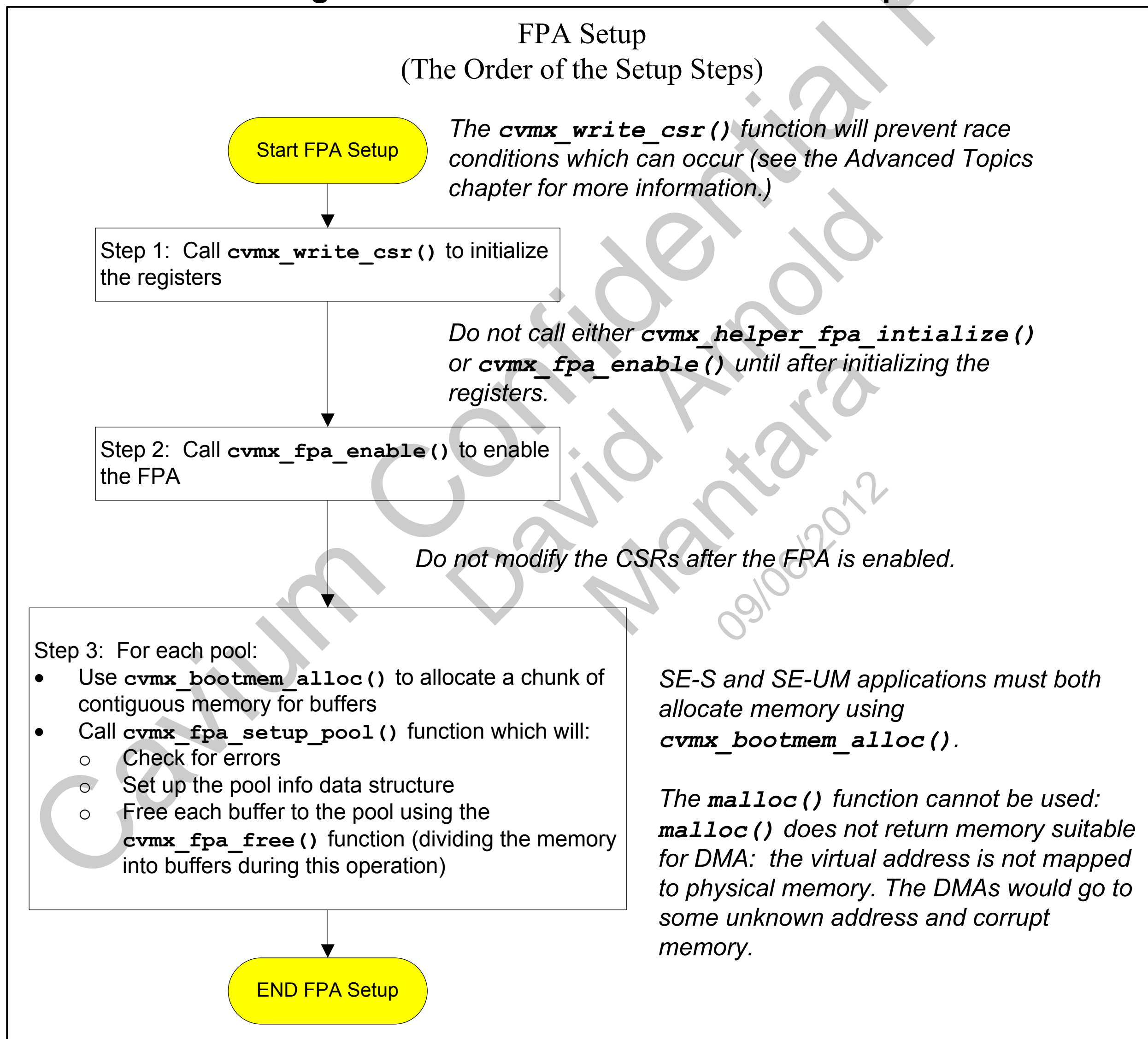
Cavium Confidential For David Arnold Mariana 09/06/2012

Table 18: Advanced Initialization Registers

Brief Description	Register	Fields
Per-Pool Allotment of In-Unit Buffer Address Cache		
The number of 32-bit entries in the In-Unit Buffer Address Cache (Address Cache) allocated for this pool. The sum of all the sizes for all pools cannot exceed the total size of the Address Cache.	`FPA_FPFx_SIZ` (one per FPA pool)	`FPF_SIZ`
Per-Pool Watermarks		
When to read in more buffers into a too-empty Address Cache.	`FPA_FPFx_MARKS` (one per FPA pool)	`FPF_RD`
When to write out buffers from a too-full Address Cache.	`FPA_FPFx_MARKS` (one per FPA pool)	`FPF_WR`

The initialization steps are shown in the following figure. (See Section 7.4.1.2 – "Example Code: Calling `cvmx_fpa_setup_pool()`" for an example of Step 3.):

Figure 12: Flowchart - FPA Unit Start-Up



Cavium Confidential For David Arnold Mantara 09/06/2012

11.4 Don't Write Back (DWB)

See the *Advanced Topics* chapter for more information on Don't Write Back (DWB).

11.5 Pool Number

Pool numbers are assigned when Simple Executive is configured. Pool numbers do not affect performance.

11.6 Performance Tuning Checklist

The following checklist is used for performance tuning. JBefore tuning performance, verify that the application does not make logical errors running `env-setup` with the `--checks` option. See Section 10 – "Debugging" for more information.

Table 19: Performance Tuning Checklist

PERFORMANCE TUNING CHECKLIST
Enough Buffers
□ Verify there are sufficient Packet Data Buffers and Work Queue Entry Buffers so the IPD will not run out of buffers.
Prefetch Buffers
□ Prefetch buffers before they are needed (see the *Configuration* chapter).
In-Unit Buffer Address Cache
□ Verify all of the available In-Unit Buffer Address Cache space has been used. Make sure unused pools are not consuming any space.
□ The default size of each pool's In-Unit Buffer Address Cache is 256 addresses. If a pool has fewer than 254 addresses, then reconfigure the size to fit the number of buffers in the pool plus 2 reserved addresses.
□ Verify the highest priority pool has the largest In-Unit Buffer Address Cache.
Watermarks
□ Verify the watermarks for each pool are tuned for best performance with your application.
□ Verify the read watermark has been adjusted so that new buffers read into the In-Unit Buffer Address Cache are always present when needed (no delay waiting for buffers to arrive).
Don't Write Back (DWB)
□ Verify DWB is being used appropriately. Using DWB on buffer free may or may not improve system performance by providing more clean L2 Cache Lines, and also by preventing some unnecessary writes to DRAM. It is recommended that you do not use DWB until you have analyzed the performance bottlenecks.
□ Verify DWB count matches the number of cache lines which may have been modified (is not higher than necessary). See the *Advanced Topics* chapter for more information.

Cavium Confidential For David Arnold Mantara 09/06/2012

12 Advanced Code Review Checklist

This checklist is for advanced users, including those who have gone beyond the Simple Executive. See Section 8 – "Basic Code Review Checklist", which also contains items for the advanced user to review.

Table 20: Advanced Code Review Checklist

ADVANCED CODE REVIEW CHECKLIST
Advanced Hardware Unit Configuration: In-Unit Buffer Address Cache and Watermarks:
□ Verify the configuration registers for per-port Address Cache allotment and watermarks were set up in order (Pool 0, 1, 2…).
□ Verify the *sum* of all the `FPA_FPFx_SIZ` values for all the pools is <= the size of the In-Unit Buffer Address Cache.
□ Verify `FPA_FPFx_SIZ` is divisible by 2.
□ Verify the read watermark (`FPA_FPFx_MARKS[FPF_RD]`) is at least 16.
□ Verify the write watermark (`FPA_FPFx_MARKS[FPF_WR]`) is at least `FPA_FPFx_SIZ - 34`.
□ Verify the read watermark (`FPA_FPFx_MARKS[FPF_RD]`) is not higher than `FPA_FPFx_SIZ`.
□ Verify the write minus read watermark is at least 34.
□ Verify that pools with an In-Unit Buffer Address Cache size of zero are not being used.
Hardware Unit Enable:
□ Verify that all configuration register writes have completed before enabling the FPA. Use the The `cvmx_write_csr()` function to write configuration registers to avoid potential race conditions (see the *Advanced Topics* chapter for details).
DWB Counts Correctly Configured:
□ Verify DWB count does not exceed the number of cache lines in the buffer. Setting the count too high could cause neighboring L2 cache line(s) to not be written to L2/DRAM.
□ Do not set DWB count higher than number of cache lines modified in the buffer. Setting the count too high may affect the system performance by causing the IOB to send unneeded DWB commands.

Cavium Confidential For David Arnold Mantara 09/06/2012

13 Beyond the SDK – When not Using the Provided API

Although ninety percent of users are able to use the SDK as described in Section 7.3 – "Easy-to-Use Executive FPA API Functions", the following information is provided only for the few users who need further customization. Note that a thorough understanding of design issues is necessary or the custom application may fail. Such failures can be difficult to debug.

A through understanding of the hardware architecture and related issues is critical to writing correct code for OCTEON processors.

Note that all OCTEON hardware units require physical addresses, not virtual addresses.

13.1 Design Considerations

Before beginning the custom software, certain design issues need to be understood. If at all possible, use the Simple Executive, or modify the Simple Executive to meet the application requirements. There are many possible errors which can occur in custom software (see Table 20 – "Advanced Code Review Checklist).

13.2 Enable the FPA and Populating the Pools

1. Enable the FPA (`FPA_CTL_STATUS[ENB]`).
2. Then read one of the RSL registers, such as `FPA_QUEn_AVAILABLE`, to force the register write to complete. Now the FPA is ready for use. See the *Advanced Topics* chapter for more information about this requirement.
3. For each pool:
    - Allocate DRAM memory: Allocating memory is easiest to do by multiplying the size of the buffers and number of buffers. `BUF_SIZE` should be (*N* * 128 bytes): an integer multiple of `CVMX_CACHE_LINE_SIZE` in size.
    - `pool_memory = BUF_SIZE * NUM_BUFS`
    - Note that PIP/IPD *always* gets its Packet Data buffers from pool 0: this is not configurable.
4. For each pool, populate the pool by freeing the buffers one at a time to the pool.
5. Initialize the registers for hardware units using the FPA pools as needed (see Section 15 – "Configuring Units Which Allocate/Free FPA Buffers"). If these registers are not set up properly, the FPA-managed buffers will not be allocated from or freed to the correct pool, and the DWB counts will be incorrect.
6. Once the buffers are in the pool, the core and the other hardware units may use them.

13.3 Synchronous Buffer Allocation

To allocate a buffer synchronously, read the register corresponding to the pool. Note that the address returned is a physical address, not a virtual address.

Cavium Confidential For David Arnold Martara 09/06/2012

Table 21: FPA Registers used in Buffer Allocate and Free Operations

Pool	Physical Address Range for Each Pool (used for allocate and free operations)
Pool 0	0x1 2800 0000 0000 - 0x1 280F FFFF FFFF
Pool 1	0x1 2900 0000 0000 - 0x1 290F FFFF FFFF
Pool 2	0x1 2A00 0000 0000 - 0x1 2A0F FFFF FFFF
Pool 3	0x1 2B00 0000 0000 - 0x1 2B0F FFFF FFFF
Pool 4	0x1 2C00 0000 0000 - 0x1 2C0F FFFF FFFF
Pool 5	0x1 2D00 0000 0000 - 0x1 2D0F FFFF FFFF
Pool 6	0x1 2E00 0000 0000 - 0x1 2E0F FFFF FFFF
Pool 7	0x1 2F00 0000 0000 - 0x1 2F0F FFFF FFFF

13.4 Asynchronous Buffer Allocation

To allocate a buffer asynchronously, use the IOBDMA operation. It is essential to set up scratchpad memory which will be the target of the IOBDMA. See the *Configuration* and *Advanced Topics* chapters for more information on IOBDMA. Asynchronous allocation can result in more efficient code: the code does not stall waiting for the allocation request to return.

13.5 Freeing a buffer

When freeing a buffer, the address for the store operation is a combination of the base address of the pool register used in the free operation and a bitwise OR of the DRAM address of the buffer. The resultant address contains the address of the buffer to be freed in bits <39:0>. The pool is selected by bits <42:40>. Bits <48:43> tell the chip that it's an FPA operation.

The data written with the store is the number of cache lines in the buffer to be marked for DWB.

For example, to free the buffer at DRAM address 0x0 0004 1000 0000 in which 2 cache lines are dirty to FPA pool 3, write the value 0x2 to address 0x1 2B04 1000 0000.

See Table 21 – ": FPA Registers used in Buffer Allocate and Free Operations" for the base address of the pool register used in the free operation.

Before freeing the buffer, be sure to issue the `syncws` instruction to make sure all writes to the buffer complete before the buffer is freed. See the *Configuration* and *Advanced Topics* chapters for more information on `syncws`.

14 FPA Registers

The FPA Registers provide additional functionality beyond the API. See Section 14 – "FPA Registers" and the *Advanced Topics* chapter for more information on accessing registers and register fields.

The registers are documented in greater detail in the Hardware Reference Manual. An overview is provided here. Note that for some registers, one is provided for each pool. In this case the name of

Cavium Confidential For David Arnold Martara 09/06/2012

the register appears in the table containing an "*n*". For example: `FPA_QUEn_AVAILABLE`. "*n*" can be any value between 0-7, inclusive.

Note that in some OCTEON models, the In-Unit Buffer Address Cache is configurable. OCTEON models which allow configuration of this unit have the following registers:

`FPA_FPFx_SIZE` and
`FPA_FPFx_MARKS`

If these registers are not present (for example: CN30xx and CN31XX), then the In-Unit Buffer Address Cache is 512 addresses, with each pool using 64 addresses. The Read watermark is set to 16. The write watermark is set to 56. None of these values are configurable.

More information about the system's memory and registers can be found in the *Advanced Topics* chapter.

To access the registers:

- The FPA's Major DID = 5.
- The FPA has eight Sub-DIDs (0-7), one for each pool.
- The Major DID and Sub-DID, combined, provide the physical address of the registers (see the *Advanced Topics* chapter for more information.
- To convert the address to virtual memory:
  For *xkphys*:
  ```
  (Virtual_address = (1 << 63) | Physical Address)
  ```
- The CSRs are shown in Table 22 - "FPA Register Summary". For exact field bits, refer to the Hardware Reference Manual (HRM).

The following two tables show the register and register field definitions provided by the Hardware Reference Manual.

Cavium Confidential For David Arnold Mantara 09/06/2012

Table 22: FPA Register Summary

Register	Address	Detailed Description
FPA_FPF0_MARKS	0x0001180028000000	Set watermarks for this pool's Buffer Pointer Cache
FPA_FPF1_MARKS	0x0001180028000008	Set watermarks for this pool's Buffer Pointer Cache
FPA_FPF2_MARKS	0x0001180028000016	Set watermarks for this pool's Buffer Pointer Cache
FPA_FPF3_MARKS	0x0001180028000024	Set watermarks for this pool's Buffer Pointer Cache
FPA_FPF4_MARKS	0x0001180028000032	Set watermarks for this pool's Buffer Pointer Cache
FPA_FPF6_MARKS	0x0001180028000040	Set watermarks for this pool's Buffer Pointer Cache
FPA_FPF6_MARKS	0x0001180028000048	Set watermarks for this pool's Buffer Pointer Cache
FPA_FPF7_MARKS	0x0001180028000038	Set watermarks for this pool's Buffer Pointer Cache
FPA_INT_SUM	0x0001180028000040	Interrupt Summary (reason for interrupt)
FPA_INT_ENB	0x0001180028000048	Interrupt Enable
FPA_CTL_STATUS	0x0001180028000050	Control and Status register
FPA_FPF0_SIZE	0x0001180028000058	Set size of this pool's In-unit Buffer Pointer Cache
FPA_FPF1_SIZE	0x0001180028000060	Set size of this pool's In-unit Buffer Pointer Cache

Register	Address	Detailed Description
FPA_FPF2_SIZE	0x0001180028000068	Set size of this pool's In-unit Buffer Pointer Cache
FPA_FPF3_SIZE	0x0001180028000070	Set size of this pool's In-unit Buffer Pointer Cache
FPA_FPF4_SIZE	0x0001180028000078	Set size of this pool's In-unit Buffer Pointer Cache
FPA_FPF5_SIZE	0x0001180028000080	Set size of this pool's In-unit Buffer Pointer Cache
FPA_FPF6_SIZE	0x0001180028000088	Set size of this pool's In-unit Buffer Pointer Cache
FPA_FPF7_SIZE	0x0001180028000090	Set size of this pool's In-unit Buffer Pointer Cache
FPA_QUE0_AVAILABLE	0x0001180028000098	Number of Free Buffers Available for this Pool
FPA_QUE1_AVAILABLE	0x00011800280000A0	Number of Free Buffers Available for this Pool
FPA_QUE2_AVAILABLE	0x00011800280000A8	Number of Free Buffers Available for this Pool
FPA_QUE3_AVAILABLE	0x00011800280000B0	Number of Free Buffers Available for this Pool
FPA_QUE4_AVAILABLE	0x00011800280000B8	Number of Free Buffers Available for this Pool
FPA_QUE5_AVAILABLE	0x00011800280000C0	Number of Free Buffers Available for this Pool
FPA_QUE6_AVAILABLE	0x00011800280000C8	Number of Free Buffers Available for this Pool
FPA_QUE7_AVAILABLE	0x00011800280000D0	Number of Free Buffers Available for this Pool
FPA_BIST_STATUS	0x00011800280000E8	Status of Power On Self-Test - zero = no problems
FPA_QUE0_PAGE_INDEX	0x00011800280000F0	The index of the current page of pointers for this pool.
FPA_QUE1_PAGE_INDEX	0x00011800280000F8	The index of the current page of pointers for this pool.
FPA_QUE2_PAGE_INDEX	0x0001180028000100	The index of the current page of pointers for this pool.
FPA_QUE3_PAGE_INDEX	0x0001180028000108	The index of the current page of pointers for this pool.
FPA_QUE4_PAGE_INDEX	0x0001180028000110	The index of the current page of pointers for this pool.
FPA_QUE5_PAGE_INDEX	0x0001180028000118	The index of the current page of pointers for this pool.
FPA_QUE6_PAGE_INDEX	0x0001180028000120	The index of the current page of pointers for this pool.
FPA_QUE7_PAGE_INDEX	0x0001180028000128	The index of the current page of pointers for this pool.
FPA_QUE_EXP	0x0001180028000130	Set if the read page owner or index is wrong.
FPA_QUE_ACT	0x0001180028000138	Set if the read page owner or index is wrong.

Cavium Confidential For David Arnold Mantara 09/06/2012

In the table below, the fields used to initialize the pools are highlighted in yellow.

Table 23: FPA Key Register Field Summary

Register *(key configuration registers are highlighted)*	Field Name	Default SDK Value	Field Description
FPA_CTL_STATUS	MEM0_ERR	0	Used in manufacturing to verify memory tests work.
FPA_CTL_STATUS	MEM1_ERR	0	Used in manufacturing to verify memory tests work.
FPA_CTL_STATUS	ENB	0	Enable FPA functionality.
FPA_CTL_STATUS	USE_STT	0	When 0, the FPA stores pointers to DRAM using STT, bypassing L2 cache (this is preferred).
FPA_CTL_STATUS	USE_LDT	0	When 0, the FPA loads pointers from DRAM using LDT, bypassing L2 cache (this is preferred).

Register *(key configuration registers are highlighted)*	Field Name	Default SDK Value	Field Description
FPA_CTL_STATUS	RESET	0	We don't support use of this reset. Instead, reset the whole chip.
FPA_FPFx_SIZE	FPF_SIZ	256	The size of the In-unit Buffer Address Cache for this pool. The sum of all the sizes cannot exceed 2048.
FPA_FPFx_MARKS	FPF_RD	64	When to read in more buffers into a too-empty cache.
FPA_FPFx_MARKS	FPF_WR	196	When to write out buffers from a too-full cache.
FPA_INT_ENB	FED0_SBE	0	Enable the interrupt
FPA_INT_ENB	FED0_DBE	0	Enable the interrupt
FPA_INT_ENB	FED1_SBE	0	Enable the interrupt
FPA_INT_ENB	FED1_DBE	0	Enable the interrupt
FPA_INT_ENB	Q*n*_UND	0	Enable the interrupt
FPA_INT_ENB	Q*n*_COFF	0	Enable the interrupt
FPA_INT_ENB	Q*n*_PERR	0	Enable the interrupt
FPA_INT_SUM	FED0_SBE	n/a	Used in manufacturing. Should not fail in the field.
FPA_INT_SUM	FED0_DBE	n/a	Used in manufacturing. Should not fail in the field.
FPA_INT_SUM	FED1_SBE	n/a	Used in manufacturing. Should not fail in the field.
FPA_INT_SUM	FED1_DBE	n/a	Used in manufacturing. Should not fail in the field.
FPA_INT_SUM	Q*n*_UND	n/a	Set when the pool's page count available goes negative (The pool is corrupted).
FPA_INT_SUM	Q*n*_COFF	n/a	Set when the pool's stack end tag is present and the count available is greater than pointers present in the FPA. (The pool is corrupted.)
FPA_INT_SUM	Q*n*_PERR	n/a	Set when the pool's address read from the stack in the L2C does not have the FPA ownership bit set. (The pool is corrupted.)
FPA_QUE*n*_PAGES_AVAILABLE	QUE_SIZ	n/a	The number of free buffers available in the pool.
FPA_QUE*n*_PAGE_INDEX	PG_NUM	n/a	The index of the current page (is also the number of pages of buffer pointers in the pool).
FPA_QUE_EXP	EXP_INDX	n/a	Expected page index read from memory (latched on PERR). The page index is an FPA-internal number.
FPA_QUE_EXP	EXP_QUE	n/a	Expected FPA Pool number (queue) read from memory (latched on PERR)
FPA_QUE_ACT	ACT_INDX	n/a	Actual page index (latched on PERR). The page index is an FPA-internal number.
FPA_QUE_ACT	ACT_QUE	n/a	Actual FPA pool number (queue). (latched on PERR)
FPA_BIST_STATUS	FDR	n/a	Used in manufacturing to test internal memory.
FPA_BIST_STATUS	FFR	n/a	Used in manufacturing to test internal memory.
FPA_BIST_STATUS	FPF1	n/a	Used in manufacturing to test internal memory.
FPA_BIST_STATUS	FPF0	n/a	Used in manufacturing to test internal memory.
FPA_BIST_STATUS	FRD	n/a	Used in manufacturing to test internal memory.

Cavium Confidential For David Arnold Mantara 09/30/2012

15 Configuring Units Which Allocate/Free FPA Buffers

The following table shows the hardware-level requirements for CN54/55/56/57XX. (The DFA information is from CN58XX.) To get requirements for a specific OCTEON model, see the HRM.

Table 24: DFA Unit

DFA		
API Function	`cvmx_initialize_dfa()`	
Pool	`DFA_DIFCTL[POOL]`	Used to free buffers (`CVMX_FPA_DFA_POOL`)
Size	`DFA_DIFCTL[SIZE]`	(`CVMX_FPA_DFA_POOL_SIZE / 8`)
Minimum Size	128 bytes (required by FPA)	A minimum of 128 bytes. Size should be a multiple of 128.
DWB	`DFA_DIFCTL[DWBCNT]`	Number of cache lines to DWB in each buffer (`CVMX_FPA_DFA_POOL_SIZE / 128`)
First command buffer in linked list	`DFA_DIFRDPTR[RDPTR]`	

Table 25: IPD Unit

IPD WQE Buffers		
API Function	`cvmx_helper_global_setup_ipd()`	
Pool	`IPD_WQE_FPA_QUEUE[WQE_QUE]`	Used to allocate buffers
Size	n/a	
Minimum Size	128 bytes	
DWB	n/a	
IPD Packet Data Buffers		
API Function	`cvmx_helper_global_setup_ipd()`	
Pool	always required to be 0	Used to allocate buffers
Size	`IPD_PACKET_MBUFF_SIZE[MB_SIZE]` (actual buffer can be larger, see the *PIP/IPD* chapter for details).	
DWB	n/a	

Table 26: PCI/PCIe DMA Engine

PCI/PCIe DMA (SDK uses PKO Command Buffers for PCI/PCIe DMA)		
API Function	`cvmx_dma_engine_initialize()`	
Pool	`NPEI_DMA_CONTROL[FPA_QUE]`	Used to free buffers (`CVMX_FPA_OUTPUT_BUFFER_POOL`)
Size	`NPEI_DMA_CONTROL[CSIZE]`	(`CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE / 8`)
Minimum Size	128 bytes (16 64-bit words)	Size should be a multiple of 128; otherwise, don't use DWB.
DWB	`NPEI_DMA_CONTROL[DWB_ICHK]`	(`CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE / 128`)
First command buffer in linked list	`NPEI_DMA(0..4)_IBUFF_SADDR`	

Cavium Confidential For David Arnold Mantara 06/06/2012

Table 27: PKO Unit

PKO Packet Data Buffers		
API Function	`cvmx_helper_global_setup_ipd()`	
Pool	Set by hardware to pool 0	Used to free buffers. Always required to be 0.
Size	Size is derived from the instruction	
Minimum Size	n/a	
DWB	`PKO_REG_FLAGS[ENA_DWB]`	Enables DWB. Unit reads command buffer to get DWB count
PKO Command Buffers		
API Function	`cvmx_pko_initialize_global()`	
Pool	`PKO_REG_CMD_BUF[POOL]`	Used to free buffers (`CVMX_FPA_OUTPUT_BUFFER_POOL`)
Size	`PKO_REG_CMD_BUF[SIZE]`	Note: In the SDK, the size of the PKO command buffers is set to an odd number of 64-bit words. This allows the normal two-word command to stay aligned and never span a command word buffer. (`(CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE / 8) -1`)
Minimum Size	128 bytes (16 64-bit words)	Size should be a multiple of 128; otherwise, don't use DWB.
DWB	`PKO_REG_FLAGS[ENA_DWB]`	(`CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE / 128`)

Table 28: RAID Unit

RAID Instruction Buffers - (SDK uses PKO Command Buffers for RAID)		
API Function	`cvmx_raid_initialize()`	
Pool	`RAD_REG_CMD_BUF[POOL]`	Used to free buffers (`CVMX_FPA_OUTPUT_BUFFER_POOL`)
Size	`RAD_REG_CMD_BUF[SIZE]`	(`CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE / 8`)
Minimum Size	128 bytes (16 64-bit words)	Size should be a multiple of 128.
DWB	`RAD_REG_CMD_BUF[DWB]`	(`CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE / 128`)
First command buffer in linked list	`RAD_REG_CMD_BUF[PTR]`	

Cavium Confidential For David Arnold Mantaro 09/06/2012

Table 29: TIMER Unit

Timer		
API Function	`cvmx_tim_setup()`	
Pool	`TIM_MEM_RING1[CPOOL]`	Used to free buffers (`CVMX_FPA_TIMER_POOL`)
Size	`TIM_MEM_RING1[CSIZE]`	(`CVMX_FPA_TIMER_POOL_SIZE / 8`)
Minimum Size	128 bytes (16 64-bit words)	Size should be a multiple of 128. Otherwise, don't use DWB.
DWB	`TIM_REG_FLAGS[ENA_DWB]`	
First command buffer in linked list	`TIM_MEM_RING0[BASE]`	
Number of buckets in the ring	`TIM_MEM_RING0[BSIZE]`	

Table 30: ZIP Unit

ZIP Instruction Buffers - (SDK uses PKO Command Buffers for ZIP)		
API Function	`cvmx_zip_initialize()`	
Pool	`ZIP_CMD_BUF[POOL]`	Used to free buffers (`CVMX_FPA_OUTPUT_BUFFER_POOL`)
Size	`ZIP_CMD_BUF[SIZE]`	Minimum size = 9 64-bit words (`CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE / 8`)
Minimum Size	128 bytes (required by FPA)	Size should be a multiple of 128; otherwise, don't use DWB.
DWB	`ZIP_CMD_BUF[DWB]`	(`CVMX_FPA_OUTPUT_BUFFER_POOL_SIZE / 128`)
First command buffer in linked list	`ZIP_CMD_BUF[PTR]`	

Cavium Confidential For David Arnold Mantara 09/06/2012