

OCTEON®

Programmer's Guide

The Fundamentals



OCTEON®

Programmer's Guide

The Fundamentals

Contents of this document are subject to change without notice.

Part Number: CN_OCTEON_PRG_GUIDE_Vol1A

Cavium Networks Proprietary and Confidential DO NOT COPY

June 2009

OCTEON® Programmer's Guide

The Fundamentals

June Curtis

Published by Cavium Networks

PUBLISHED BY
Cavium Networks
805 East Middlefield Road
Mountain View, CA 94043
Phone: 650-623-7000
Fax: 650-625-9751
Email: sales@caviumnetworks.com
Web: <http://www.caviumnetworks.com/>
© 2005-2009 by Cavium Networks
All rights reserved.

No part of this manual may be reproduced in any form, or transmitted by any means, without the written permission of Cavium Networks.

Cavium Networks makes no warranty about the use of its products, and reserves the right to change this document at any time, without notice. Whereas great care has been taken in the preparation of this manual, Cavium Networks, the publisher, and the author assumes no responsibility for errors or omissions.

The data and illustrations found in this document are not binding. We reserve the right to modify our products in line with our policy of continuous product improvement. The information in this document is subject to change without notice and should not be construed as a commitment by Cavium Networks, Inc.

OCTEON® and NITROX® are registered trademarks of Cavium Networks. MIPS®, MIPS64®, and MIPS32® are registered trademarks of MIPS Technologies. cnMIPS® is a registered trademark of MIPS Technologies; Cavium Networks is a licensee of cnMIPS. CompactFlash™ is a registered trademark of SanDisk Corporation in the United States. RLDRAM® II is a registered trademark of Micron Technology, Inc. PCI Express®, PCIe®, and PCI-X® are registered trademarks of PCI-SIG. Linux® is a registered trademark of Linus Torvalds. RapidIO® is a registered trademark of the RapidIO Trade Association. All other trademarks or service marks referred to in this manual are the property of their respective owners.

Preface

About This Book

This volume of the *OCTEON® Programmer's Guide* provides fundamental information needed by software engineers to develop code to run on the OCTEON processor.

This volume contains the following chapters:

1. *Introduction*
2. *Packet Flow*
3. *Software Overview*
4. *SDK Tutorial*
5. *Software Debugging Tutorial*
6. *OCTEON Application Performance Tuning Whitepaper*
7. *Glossary*

The appendices relevant to a chapter are not collected at the end of the book, but instead are included with the corresponding chapter.

The chapters should be read in order, except for the appendices. Each chapter builds on information provided in the previous chapter.

If you have any suggestions for improvements or amendments, or believe you have found errors in this publication, please notify us at oct-prog-guide@caviumnetworks.com.

The following sections briefly introduce the contents of the *OCTEON Programmer's Guide, Volume 1*.

Chapter 1: Introduction

This chapter provides an overview of Cavium Network's OCTEON® processor family, and introduces key features provided by its members.

This chapter also provides a brief discussion of the advantages of some of the OCTEON processor's key features:

- Integrated hardware accelerators
- Per-core Security Coprocessors
- On-chip interconnects
- Special Cavium Networks-specific instructions
- Cache hierarchy

Chapter 2: Packet Flow

This chapter provides a detailed description of how the packet-management units offload and cooperate with the cores to accelerate packet flow through the OCTEON processor. This chapter includes details on how the Scheduling/Synchronization/and Order (SSO) Unit off loads the cores.

This chapter is divided the following sections:

- **Packet Flow Overview:** A typical packet flow (packet received, processed, transmitted) will be discussed, showing how the different functional blocks work together to create reliable, fast packet processing.
- **Special Hardware Features to Accelerate Packet Processing:** The key packet-management hardware acceleration features are discussed:
 1. management of packet classification and priority
 2. buffer management
 3. packet-linked locks
 4. management of packet order
- **The Schedule / Synchronization / Order (SSO) Unit:** The Schedule / Synchronization / Order Unit (SSO) is introduced. The SSO provides essential capabilities which are unique to the OCTEON processor; the SSO is the “heart” of OCTEON. Because of the SSO’s importance, the rest of this chapter introduces its hardware acceleration features, and describes how software may take advantage of them.

Chapter 3: Software Overview

This chapter provides an overview of the OCTEON processor’s software-related topics, including software architecture, multicore issues, scaling, and memory management.

The chapter introduces the following topics:

- cnMIPS® cores
- Simple Executive API (HAL)
- Different runtime environment choices
- Software architecture issues
- Application Binary Interfaces (ABIs) supported
- Tools: Cross-Development and Native Toolchains
- Physical Address Map and Caching on the OCTEON processor
- Virtual Memory
- Bootmem Global Memory
- Shared memory
- Bootloader
- Software Development Kit (SDK) code conventions

Throughout the chapter relevant SDK documents are referenced to help the reader find more detailed information.

Chapter 4: SDK Tutorial

This chapter introduces the Software Development Kit (SDK) from a hands-on perspective, provides a tour of the installed SDK, and also contains useful information for users new to embedded software development or new to Linux®.

This chapter is designed to augment the SDK documentation by providing a high-level view of the SDK and step-by-step instructions from installing the SDK to running example code on an evaluation board.

The hands-on sections in this tutorial are:

- System Administration Tasks
- Connect the Development Target
- Viewing the Target Board Console Output
- Gather Key Hardware Information
- Install the SDK
- Tour the Installed SDK
- Build and Run a SE-S Application (`hello`)
- Run `hello` on Multiple Cores
- Build and Run Linux
- Run a SE-UM Example (`named-block`)
- Run `linux-filter` as a SE-S Application (Hybrid System)
- Run `linux-filter` as a Linux SE-UM Application
- Run `linux-filter` as a SE-UM Application on Multiple Cores
- Creating a Custom Application

Chapter 5: Software Debugging Tutorial

This chapter provides information and hands-on steps to help users get started with embedded software debugging using GDB, including:

- hardware configuration
- debugging both standalone and PCI development targets
- multicore debugging

The hands-on sections in this tutorial are:

- Debug a SE-S Application: `hello`
- Debug the Linux Kernel
- Debug a SE-UM Application: `named-block`
- Debugging a SE-S Application on the OCTEON Simulator
- Debugging Linux on the OCTEON Simulator
- Building `vmlinux` to Run on the Simulator
- Running Linux User-Mode Applications on the Simulator

About OCTEON Application Performance Tuning Whitepaper

This whitepaper provides information on how to optimize software performance by taking advantage of the OCTEON processor's unique features.

This whitepaper describes common areas where changes can bring big performance improvements. Many of the performance improvement techniques presented in this document are industry-standard; others take advantage of Cavium Networks-specific hardware acceleration.

This whitepaper addresses both designing for high performance, and post-development performance tuning. Both single core and multiple-core (scaling) issues are discussed.

Performance tuning issues are separated into four sections:

1. Software Architecture for High Performance
2. Tuning the Minimum Set of Cores
3. Tuning Multicore Applications (Scaling)
4. Linux-Specific Tuning

Within each section, performance tuning choices are presented from the easiest to most difficult to implement.

Information is also provided on performance evaluation tools.

Performance tuning is both an art and a science. This whitepaper does not attempt to cover all the possibilities, only some of the more common ones.

Glossary

The glossary contains terms defined in this volume. Some common industry terms are also provided for convenience.

Softcopy of Chapters

OCTEON Programmer's Guide chapters are also available at the Cavium Networks support site at <https://support.caviumnetworks.com/>.

Where to Get More Information

Other resources include the extensive documentation supplied with the SDK, the *Hardware Reference Manual (HRM)*, whitepapers and application notes. All of these are available at the support site at <https://support.caviumnetworks.com/>. As new *OCTEON Programmer's Guide* chapters are published, they are made available at the support site.

MIPS® Architecture manuals are available from: <http://www.mips.com/>. The key manuals are:

- *MIPS64® Architecture for Programmers Volume I: Introduction to the MIPS64® Architecture*
- *MIPS64® Architecture for Programmers Volume II: The MIPS64® Instruction Set*
- *MIPS64® Architecture for Programmers Volume III: The MIPS64® Privileged Resource Architecture*

There are two excellent MIPS guides written by Dominic Sweetman. See *MIPS Run Linux* is currently the newer of the two books.

Sweetman, Dominic, *See MIPS Run*, ISBN-10: 1558604103; ISBN-13: 978-1558604100

Sweetman, Dominic, *See MIPS Run Linux*, ISBN-10: 0120884216; ISBN-13: 978-0120884216

Conventions Used in This Book

The following typographical conventions are used in this book:

<i>Italic</i>	is used for document names, new terms, special terms such as <i>root</i> , comments in code, notes in tables and figures, and for emphasis
Constant Width	is used for code, commands, contents of files, file names, and directory names
<i>Italic Bold</i>	is used for notes in text
Constant Bold	is used to indicate what to type on the command line, and for commands used in figures
System	is used for keyboard keys

Acknowledgments

Several key people from the Cavium Networks team contributed to this book, including representatives from hardware engineering, software engineering, architecture, technical marketing engineering, applications engineering, marketing, and sales.

Because of their efforts, each chapter provides experience, depth, and perspective which are not available to one person alone.

Summary Table of Contents

Preface	v
---------------	---

Chapter 1: Introduction

1 Introduction.....	1-2
2 Introducing the OCTEON Processor Family.....	1-2
3 Hardware-Acceleration Units	1-8
4 Packet-Management Accelerators	1-9
5 Per-Core Security Coprocessors	1-12
6 On-Chip Interconnects.....	1-12
7 Special Cavium Networks-Specific Instructions.....	1-16
8 Cache Hierarchy	1-17
9 Summary	1-18

Chapter 2: Packet Flow

1 Introduction.....	2-4
2 Packet Flow Overview.....	2-4
3 Hardware Features to Accelerate Packet Processing.....	2-15
4 The Schedule / Synchronization / Order (SSO) Unit.....	2-17

Chapter 3: Software Overview

1 Introduction.....	3-8
2 Introducing cnMIPS (Cavium Networks MIPS)	3-9
3 Introducing the Simple Executive API	3-10
4 Runtime Environment Choices for cnMIPS Cores.....	3-13
5 Combinations of Runtime Environments on One Chip	3-21
6 Software Architecture	3-36
7 Application Binary Interface (ABI).....	3-62
8 Tools	3-66
9 Physical Address Map and Caching on the OCTEON Processor.....	3-70
10 Virtual Memory	3-76
11 Allocating and Using Bootmem Global Memory.....	3-94
12 Accessing Bootmem Global Memory (Buffers).....	3-102

13	Accessing I/O Space	3-107
14	Simple Executive Standalone (SE-S) Memory Model	3-108
15	Linux Memory Model.....	3-117
16	Downloading and Booting the ELF File.....	3-129
17	SDK Code Conventions.....	3-140
18	Bootloader Historical Information.....	3-145

Chapter 4: Software Development Kit (SDK) Tutorial

1	Introduction.....	4-8
2	Overview.....	4-11
3	Hardware and Software Requirements	4-11
4	Hands-on: System Administration Tasks	4-14
5	Hands-on: Connect the Development Target	4-15
6	Hands-on: Viewing the Target Board Console Output.....	4-19
7	Hands-on: Gather Key Hardware Information	4-24
8	Hands-on: Install the SDK.....	4-25
9	Hands-on: Tour the Installed SDK	4-32
10	About Building Example Applications.....	4-49
11	Hands-on: Build and Run a SE-S Application (<code>hello</code>).....	4-54
12	Hands-on: Run <code>hello</code> on Multiple Cores.....	4-67
13	About the Bootloader.....	4-69
14	About Downloading the Application	4-73
15	About Booting SE-S Applications.....	4-76
16	About Building Linux.....	4-77
17	Hands-on: Build and Run Linux.....	4-84
18	Hands-on: Run a SE-UM Example (<code>named-block</code>)	4-88
19	About the <code>linux-filter</code> Example	4-88
20	Hands-on: Run <code>linux-filter</code> as a SE-S Application (Hybrid System)	4-92
21	Hands-on: Run <code>linux-filter</code> as a Linux SE-UM Application	4-98
22	Hands-on: Run <code>linux-filter</code> as a SE-UM Application on Multiple Cores.....	4-103
23	Hands-on: Creating a Custom Application.....	4-104
24	The Hardware Simulator.....	4-108
25	Appendix A: Introduction to Available Products	4-115
26	Appendix B: Linux Basics.....	4-120
27	Appendix C: About the RPM Utility	4-126
28	Appendix D: Other Useful Tools.....	4-130
29	Appendix E: U-Boot Commands Quick Reference Guide	4-131
30	Appendix F: ELF File Boot Commands Quick Reference	4-133
31	Appendix G: Null Modem Serial Cable Information	4-135
32	Appendix H: Query EEPROM to get Board Information	4-135
33	Appendix I: Updating U-Boot on a Standalone Board.....	4-137
34	Appendix J: TFTP Boot Assistance (<code>tftpboot</code>).....	4-144
35	Appendix K: Downloading Using the Serial Connection.....	4-148
36	Appendix L: Simple Executive Configuration	4-149

37	Appendix M: Changing the ABI Used for Linux	4-150
38	Appendix N: Contents of the Embedded Root Filesystem	4-150
39	Appendix O: Getting Ready to Use a Flash Card	4-152
40	Appendix P: Booting an ELF File From a Flash Card	4-154
41	Appendix Q: Using the Debian Root Filesystem	4-155
42	Appendix R: About <code>oct-pci-console</code>	4-157
43	Appendix S: About <code>oct-pci-reset</code> and <code>oct-pci-csr</code>	4-158
44	Appendix T: Multiple Embedded Root Filesystem Builds.....	4-159
45	Appendix U: How to Find the Process's Core Number	4-161

Chapter 5: Software Debugging Tutorial

1	Introduction.....	5-6
2	Getting Started Debugging	5-7
3	Building Applications and the Linux Kernel for Debugging	5-18
4	Debugging Applications in the Embedded Root Filesystem	5-20
5	Hands-On: Debug a SE-S Application: <code>hello</code>	5-22
6	About Debugging SE-S Applications or the Linux Kernel	5-29
7	Hands-On: Debug the Linux Kernel.....	5-48
8	About Debugging the Linux Kernel	5-58
9	Hands-On: Debug a SE-UM Application: <code>named-block</code>	5-59
10	About Linux User-Mode Application Debugging.....	5-66
11	EJTAG (Run-Control) Tools	5-71
12	About Debugging on the OCTEON Simulator.....	5-72
13	Appendix A: Common GDB Commands	5-84
14	Appendix B: Connecting Using a Terminal Server	5-86
15	Appendix C: How to Simplify the Command Lines	5-88
16	Appendix D: Graphical Debugger	5-89
17	Appendix E: Core Files	5-90
18	Appendix F: The <code>oct-debug</code> Script.....	5-94
19	Appendix G: Debian and the Cavium Networks Ethernet Driver	5-95

Chapter 6: OCTEON Application Performance Tuning Whitepaper

1	Introduction.....	6-5
2	Performance Tuning Overview.....	6-6
3	Performance Tuning Checklist	6-18
4	Hardware Architecture Overview	6-20
5	Software Architecture for High Performance.....	6-22
6	Tuning the Minimum Set of Cores	6-26
7	Tuning Multi-core Applications (Scaling).....	6-51
8	Linux-specific Tuning	6-56

Glossary	7-1
----------------	-----



Introduction

TABLE OF CONTENTS

TABLE OF CONTENTS	1
LIST OF FIGURES	1
1 Introduction.....	2
2 Introducing the OCTEON Processor Family.....	2
2.1 Target Applications	3
2.2 Key Features	3
3 Hardware-Acceleration Units	8
4 Packet-Management Accelerators	9
4.1 Packet Flow, Summarized	9
4.2 The Scheduling/Synchronization and Order Unit (SSO).....	10
4.3 Architectural Advantages of Work Groups	11
5 Per-Core Security Coprocessors	12
6 On-Chip Interconnects.....	12
6.1 The Coherent Memory Bus Interconnect	13
6.2 I/O Interconnect	14
7 Special Cavium Networks-Specific Instructions	16
8 Cache Hierarchy	17
9 Summary.....	18

LIST OF FIGURES

Figure 1: The OCTEON and OCTEON Plus Processor Overview	7
Figure 2: Packet Data Movement Over I/O Buses	15

1 Introduction

This chapter provides an overview of Cavium Network's OCTEON® processor family, and introduces key features provided by its members.

This chapter also provides a brief discussion of the advantages of some of the OCTEON processor's key features:

- Integrated hardware accelerators
- Per-core Security Coprocessors
- On-chip interconnects
- Special Cavium Networks-specific instructions
- Cache hierarchy

Note: Throughout this chapter, OCTEON model-specific hardware components are marked with an asterisk (*). See the *Hardware Reference Manual (HRM)* for your specific part number for more details.

2 Introducing the OCTEON Processor Family

The OCTEON family consists of three generations of software-compatible, highly integrated multicore products: OCTEON®, OCTEON® Plus and OCTEON® II (CN3XXX, CN5XXX, and CN6XXX, respectively). These processors are optimized to provide high-performance, high-bandwidth, and low power consumption.

The OCTEON processor can be used for control-plane applications, data-plane applications, or a hybrid of both. The OCTEON processor is an ideal solution for intelligent networking, wireless, and storage applications from 100 Mbps to 40 Gbps.

All OCTEON products share the same architecture, which enables software compatibility across the entire family. Individual OCTEON products vary and scale based on the:

- Number of integrated cnMIPS® cores
- Frequency of the cores and the internal interconnects
- Type and number of integrated I/O interfaces
- Type and number of integrated hardware acceleration units
- Size and associativity of the caches

Each OCTEON product offers a feature set which is optimized for the specific functionality and performance needs of the target applications. Software written for one OCTEON model will run on another OCTEON model as long as the required features are available.

Cavium Networks is actively engaged in adding new features and enhancements to the OCTEON family. For more information on the latest additions and the future roadmap, contact your Cavium Networks sales representative.

2.1 Target Applications

The OCTEON processors are used in a wide variety of OEM equipment. Some examples include routers, switches, unified threat management (UTM) appliances, content-aware switches, application-aware gateways, triple-play broadband gateways, WLAN access and aggregation devices, 3G, WiMAX and LTE basestation and core network equipments, storage networking equipment, storage systems, servers, and intelligent network adapters.

2.2 Key Features

This section provides a brief overview of key OCTEON features. Because the OCTEON II models are in development as this chapter is being written, some of the features listed are subject to change.

Note: *Features which are optional are marked with an asterisk (*).*

Up to 32 cores, up to 1.5 GHz: The OCTEON family of multicore processors supports up to 32 cnMIPS cores with speeds ranging from 300 MHz to 1.5 GHz.

- The OCTEON and OCTEON Plus models have from 1 to 16 cores, at speeds ranging from 300 MHz to 800 MHz.
- The OCTEON II models have up to 32 cores, with speeds up to 1.5 GHz.

Hardware Acceleration Units: Multiple hardware acceleration units are integrated into each OCTEON processor. These hardware acceleration units offload the cores, reducing software overhead and complexity. These acceleration units include:

- Packet-management accelerators
- Security accelerators
- Application accelerators
- Specialized accelerators

Dedicated DMA Engines: Dedicated DMA Engines are provided for each hardware unit which accesses memory. Additional memory-to-PCI PCIe®/PCI/PCI-X DMA Engines are present in some models.

High-Speed Interconnects: The hardware units and the cores are connected by high-speed interconnects. These interconnects run at the same frequency as the cores. Each interconnect is a collection of multiple buses with extensive pipelining and sophisticated hardware arbitration logic. The width and placement of the buses are optimized to streamline packet data flow, eliminating potential bottlenecks. Some of the OCTEON II models include a cross-bar hyper-connect to scale up to 32 cores and higher.

Industry-Standard Toolchains and Operating Systems: Industry-standard toolchains (GCC, GDB) and operating systems (including SMP Linux) have been modified to utilize the OCTEON processor's multiple cores, hardware acceleration units, and special Cavium Networks-specific instructions. Users can easily write C/C++ code, and can re-use legacy software. Programs written for MIPS64 and MIPS32 ISA are inherently supported.

Flexible Software Architecture: The hardware architecture allows flexible software architecture design, including the ability to group cores as desired to add more performance where it is needed. A common configuration is data-plane plus control-plane. In this configuration, one group of cores runs a data-plane application; another group of cores runs SMP Linux or another general-purpose operating system to provide the control-plane functionality. If needed, cores can be added to the data-plane, resulting in linear performance scaling.

Streamlined Software Development: Software development complexity is minimized by hardware acceleration units, flexible software architecture, standard MIPS64 ISA, and industry-standard toolchains and operating systems. A Cavium Networks Software Development Kit (SDK) is provided. The SDK includes the GNU C/C++ compiler and other development tools, C-language Application Programmer's Interfaces (APIs) to the hardware units, a simple executive which can run code on the cores without any operating system, and Cavium Networks SMP Linux. Optional software packages are available to support more complex features.

Packet-management Acceleration: Packet receive/transmit is automated by software-configurable packet-management accelerators. Accelerations include:

- Packet data buffers are automatically allocated and freed
- Layer-2 through layer-4 packet header parsing, exception checks and checksum calculation
- Up to 7-tuple flow classification with VLAN stacking support
- Packet data is automatically stored in L2/DRAM on ingress
- Packet ordering and scheduling is automatically managed by hardware without requiring explicit software locking
- Packet data transmission is managed by a hardware accelerator

TCP/UDP Acceleration: TCP/UDP acceleration features include:

- Packet-management accelerations
- Automated packet header checking on receive
- Automated TCP/UDP checksum generation on transmit
- Timer Unit supports efficient implementation of TCP retransmission

Per-Core Security Hardware Acceleration*: Common security algorithms are accelerated by optional per-core Security Engines. (See the *HRM* for a complete list of hardware accelerated algorithms.) In the following list, a double asterisk (**) denotes security algorithms which are not present in all Security Engines. The hardware accelerated algorithms include:

- Large multiply-and-accumulate unit for fast modular exponentiation needed for RSA and Diffie-Hellman operations
- Security hash algorithms:
 - MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512, AES XCBC HMAC
- Symmetric cryptographic algorithms:
 - 3DES and DES in ECB and CBC modes
 - AES in CBC, ECB, CTR, LRW, ICM, XTS, GCM, and CCM modes
 - RC4
 - KASUMI** (OCTEON Plus, OCTEON II)
 - SNOW 3G** (OCTEON II)
 - SMS4** (OCTEON II)
- Asymmetric key operations:
 - RSA, DSA, DH
- TKIP Operations: TKIP
- Galois field multiplication (used in both security, such as SNOW 3G, and RAID calculations)

Per-Core CRC Engines: CRC generation is accelerated by the per-core CRC Engines:

- Hardware CRC calculation (up to 32 bits): For example, CRC10 accelerates AAL2 and CRC32 accelerates AAL5 protocol processing.
- CRC hardware also accelerates ROHC (Robust Header Compression) protocol processing and iSCSI checksum calculation/verification.

FIPS Certification Support: Other features which facilitate high-level FIPS (Federal Information Processing Standards) certification include:

- NIST-certified algorithms
- A cryptographically secure Random Number Generator (RNG) hardware unit. This unit has been designed to handle upcoming FIPS standards.
- Secure on-chip memory* for security keys which cannot be accessed through I/O interfaces
- A pin for zeroing out all the stored keys
- Restricted PCI host access

Storage Application Acceleration: Storage applications are accelerated by:

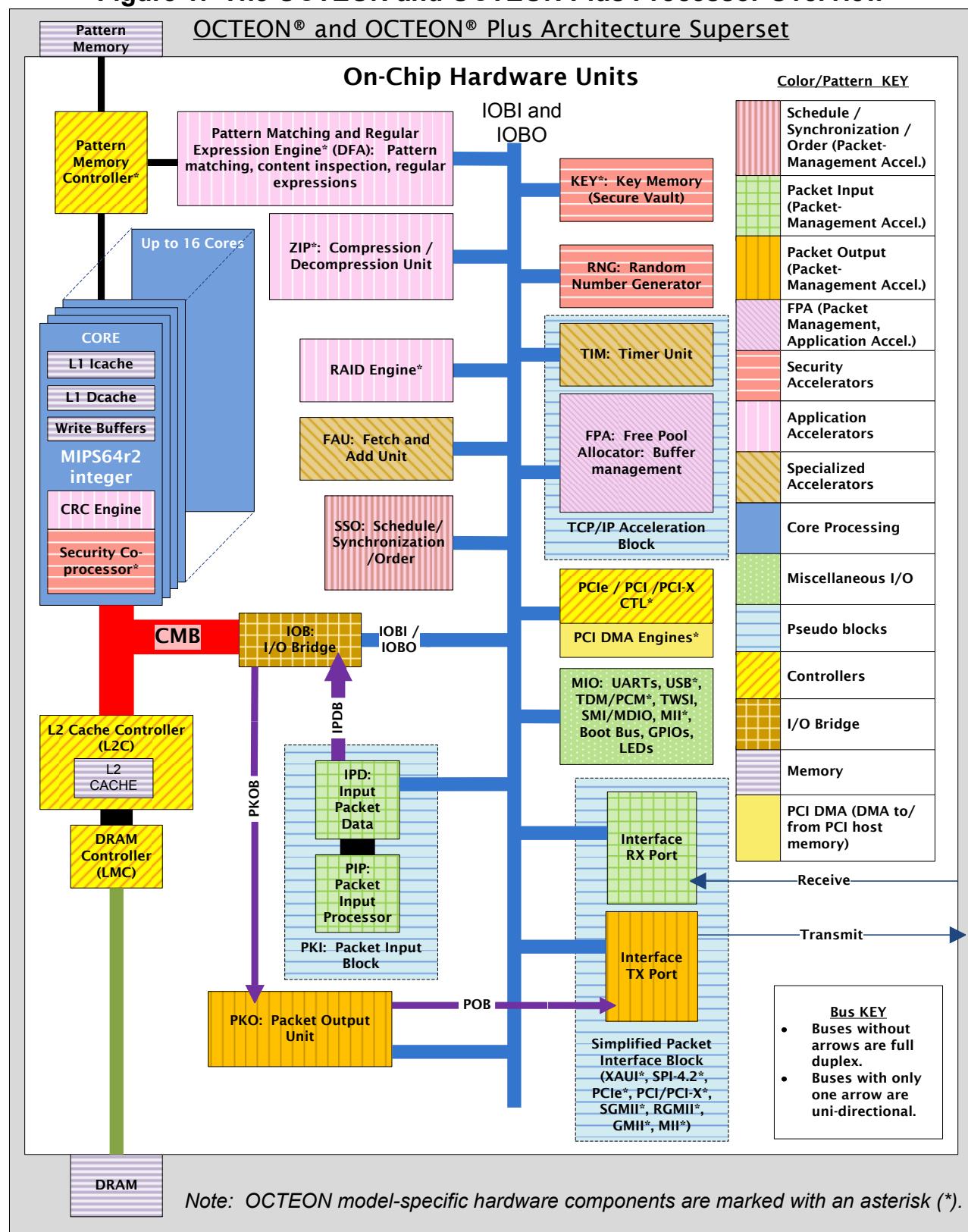
- RAID Engine*: RAID/XOR Acceleration for RAID 5 and RAID 6
- Per-core Security Engines*
- Galois field multiplication (per-core security acceleration) can also be used for RAID calculations.
- De-duplication acceleration

Other architectural features include the following list (optional features are marked with an asterisk (*)):

- MIPS64 release 2 integer Instruction Set Architecture (ISA)
- Additional Cavium Networks-specific instructions, enhancing the MIPS core to create the cnMIPS core. Most of these instructions are automatically generated by the C/C++ compiler.
- Dual-issue ALU with additional security acceleration coprocessor units. Combining two instruction issues together with additional security units, it is possible for more than two operations to be simultaneously executing in a given cycle.
- A cache hierarchy including:
 - L2 cache with ECC protection, ranging from 256 KB to 2 MB (OCTEON II: up to 4 MB), shared by the cores and I/O subsystem
 - L1 instruction cache (Icache) with parity protection, 32 KB (OCTEON II: up to 37 KB), per core
 - L1 data cache (Dcache) with parity protection, 8 KB to 16 KB (OCTEON II: up to 32 KB), per core
- A 32-entry to 64-entry TLB (OCTEON II: up to 128-entry) which supports:
 - variable page sizes from 4K to 256 MB
 - read and execute inhibit per-page options (used to protect against overflow attacks and malicious code)
- Memory options include:
 - DDR2 from DDR2-400 up to DDR2-800 for OCTEON and OCTEON Plus
 - DDR3 up to DDR3-1600 for OCTEON II
- Per-core Write Buffer with aggressive write combining, reducing unnecessary traffic on the buses by limiting the number of writes to memory
- Industry-standard I/O Interfaces: XAUI*, SPI-4.2*, PCIe*, PCI/PCI-X*, SGMII*, RGMII*, GMII*, MII*, (OCTEON II: serial RapidIO®* (sRIO), and Interlaken*)
- Support for NOR and NAND* flash
- Boot from NOR flash (CN52XX supports boot from NAND)
- Misc I/O Including: UARTs, USB 2.0* (including PHY), TDM/PCM*, TWSI, SMI/Mdio, MII*, Boot Bus, GPIOs, LEDs
- PCIe/PCI/PCI-X DMA Engines* to DMA to/from PCI host memory or from memory to memory
- Pattern Memory Controller*: used to connect pattern memory to the cores and to the Pattern Matching and Regular Expression Engine.

The following figure shows the OCTEON processor's on-chip hardware units in an idealized way because it shows all of the model-specific hardware components in one superset. The OCTEON II processor features are not included in this diagram. The OCTEON II processors will contain many of the features shown in the figure below, and will also contain new and enhanced features.

Figure 1: The OCTEON and OCTEON Plus Processor Overview



3 Hardware-Acceleration Units

Hardware-acceleration units are divided into groups for this discussion:

- Packet-management accelerators
- Security accelerators
- Application accelerators
- Specialized accelerators

Packet-Management Accelerators:

- SSO Unit – Schedule/Synchronization and Order Unit: This unit manages packet scheduling and ordering.
- FPA Unit – Free Pool Allocator Unit: This unit manages pools of free buffers, including Packet Data buffers.
- PIP Unit – Packet Input Processor Unit: This unit works with IPD to manage packet input.
- IPD Unit – Input Packet Data Unit: This unit works with PIP to manage packet input.
- PKO Unit – Packet Output Unit: This unit manages packet output.

Security Accelerators:

- RNG Unit – Random Number Generator
- KEY* Unit: This unit provides and manages secure on-chip memory which can be used to store a hardware key, and can be reset using an external pin.
- Per-Core Security Coprocessor* (Security Engine): This unit is a special coprocessor used to accelerate security algorithms. There is one Security Coprocessor per core.

Application Accelerators:

- Pattern Matching and Regular Expression Engine*: This unit is used to perform string matching. The unit has different names on different OCTEON models, for example Deterministic Finite Automata (DFA). Users store rules in the attached pattern memory. In the OCTEON II and the NITROX® deep packet inspection (dpi) products, the pattern matching accelerators have been enhanced to include non-deterministic finite automata (NFA) functionality. Using NFA results in up to 4 times higher performance, support for very complex expressions, and significantly lower pattern memory requirements.
- ZIP Engine*: This unit is a compression/decompression engine which provides DEFLATE compression/decompression as defined in RFC 1951, ALDER32 checksum for ZLIB as defined in RFC 1950, CRC32 checksum for GZIP as defined in RFC 1952.
- RAID Engine*: This unit provides RAID/XOR Acceleration for RAID 5 and RAID 6.
- Per-Core CRC Engine: This unit is used to accelerate CRC generation.
- FPA Unit – Free Pool Allocator Unit: This unit manages pools of free buffers, including Packet Data buffers. It can be used as a general buffer manager, not only as a Packet Data buffer manager.

Specialized Accelerators:

- FAU – Fetch and Add Unit: This unit is used to add a number to a memory location, and can be used to manage counters.
- TIM – Timer Unit: This unit provides timers, which can be used for TCP timeouts as well as other more general purposes.

All of these hardware units offload work, freeing the cores to focus on essential packet processing.

Note: Hardware versus software CRC and hashing are discussed in more detail in the *OCTEON Application Performance Tuning Whitepaper*.

4 Packet-Management Accelerators

The packet-management accelerators automatically handle an enormous amount of packet processing, offloading the cores from many time-consuming responsibilities. These hardware units are responsible for packet receive, buffering, buffer management, flow classification, QoS (Quality of Service), and transmit processing. All of these functions are highly configurable (at per virtual port granularity), and can be customized using software to access the configuration registers.

Packet-management accelerators manage the following functions, without assistance from the cores.

- Manage list of free packet data and other buffers. (FPA)
- Packet receive and transmit through networking or PCI type interfaces
- Automatic packet data buffer allocation and freeing, and buffer management
- Layer-2 through layer-4 packet header parsing, exception checks and checksum calculation
- Up to 7-tuple flow classification with VLAN stacking support
- Congestion avoidance option, based on multi-priority RED algorithm
- Packet ordering and scheduling is automatically managed by hardware without requiring explicit software locking
- Traffic management with strict priority and/or weighted round-robin scheduling for packet transmit
- 8 levels of hardware-managed QoS for the input ports
- Up to 16 levels of hardware-managed QoS for each output port

4.1 Packet Flow, Summarized

The PIP and IPD units work together to receive a packet and perform early processing on it. Each packet is represented as “work” for the cores to do, and is represented by a Work Queue Entry (WQE) data structure.

The PIP and IPD are responsible for many layer-2 through layer-7 processing requirements such as exception checks and TCP/UDP checksum verification.

The PIP/IPD hardware units:

1. Verify the IPV4 checksum and payload checksums for TCP and UDP (if applicable).
2. Classify the flow. The QoS and group values are set.
3. Obtain packet data buffers and WQE buffers from the FPA.
4. DMA the packet data into memory using a dedicated bus.
5. Send the WQE to the correct QoS queue in the SSO. The WQE includes the first 96 bytes of the packet, so the core can begin to work on it immediately after receiving the WQE.

The PIP/IPD hardware also implements a per-QoS-queue random early discard (RED) algorithm, and a per-port threshold algorithm. These algorithms allow the PIP/IPD to discard input packets if necessary.

The SSO is responsible for scheduling the work to cores, and for maintaining the packet ingress order.

Cores may request work from the SSO either asynchronously (the core continues to do other work while the instruction completes), or synchronously (the core waits for the instruction to complete). Typically, cores minimize idle time by requesting the work before it is actually needed.

The PKO is responsible for packet transmission. When packet processing is complete, the core notifies the PKO that the packet is ready for transmission. The PKO manages transmission priority.

The PKO:

1. DMAs the packet data from memory into its internal memory over a dedicated bus.
2. Optionally computes TCP and UDP payload checksum and inserts it into the packet's header on egress.
3. When the packet is ready to transmit, the PKO sends the packet data from its internal memory to the output port over a dedicated bus.
4. Optionally frees the packet data buffer back to the FPA after the transmission is complete.

Details on the packet-management accelerators are provided in the *Packet Flow* chapter, including a discussion of how they work together to offload the cores.

4.2 The Scheduling/Synchronization and Order Unit (SSO)

The SSO unit enables scalable use of the multiple cores, maximizing parallel processing. It schedules work for the cores to do based on QoS priority, and work group. The cores are completely freed of this responsibility. In addition to scheduling, the SSO maintains packet order. The SSO also provides a locking mechanism to protect critical regions. The principal advantage of this locking mechanism over spin locks is that the cores continue to work while the SSO manages locking constraints. It would take considerable software coding and runtime cycles to implement the SSO's sophisticated scheduling mechanism in software.

The SSO's design supports the goal of flexible software architecture. The SSO's features work effectively with pipelining, modified pipelining, and run-to-completion software architectures.

Scheduling:

- When cores are ready for more work (packets to process), they request work from the SSO. The SSO is responsible for scheduling the highest priority work to each core.
- The scheduling algorithm is highly flexible and software tunable, allowing the user to customize scheduling to optimize application performance.

Synchronization:

- Packets can either be processed in parallel or one at a time. In typical packet processing, many operations may be done in parallel, increasing application throughput.
- When packet processing requires obtaining a shared resource (for example, a “critical region” in the code), locked access is needed. The SSO manages this access, offloading software. After the critical processing completes, the packet is processed in parallel again. These locks are referred to as *packet-linked locks*, and are discussed in detail in the *Packet Flow* chapter.
- The SSO manages the synchronization of the packets as they move through these processing phases.

Order:

- Whether the packet is processed in parallel or one at a time, the SSO maintains the packet’s ingress order so that it may be transmitted in order.

4.3 Architectural Advantages of Work Groups

Each group can have from 0 to all of the cores. Cores can be in more than one group. The number of cores in each group can easily be changed by software, allowing the application to dynamically adapt to varying workload requirements. When a core requests more work to do, the SSO will only schedule work from an appropriate group to the core.

Work groups are used to balance the processing load among the different cores, providing architectural flexibility. Work groups can be used, for example, to assign a set of cores to run the data-plane application, and another set of cores to run the control-plane application. The work group value of a packet is set automatically on ingress, and can be changed by software. If the data-plane application needs to route a packet to the control-plane application, it simply changes the packet’s work group value, and sends the work back to the SSO to be rescheduled.

The exact number of work groups depends on the OCTEON model. For example, in CN58XX, there are up to 16 groups, and there are 16 cores.

This feature provides the following benefits:

- Divide Data Plane and Control Plane Responsibilities: Groups can be used to divide cores into belonging to either the data plane or the control plane. The packet can be routed from the control plane to the data plane by changing the group value.
- Scaling: Applications can be easily modified to meet different performance targets by using OCTEON models with different numbers of cores. Cores are simply added to or removed from existing work groups.
- Reduce Latency: Tasks that are latency sensitive (must complete quickly) can be assigned to dedicated cores. Tasks which take longer to complete can be assigned to different cores, so they cannot cause head-of-line blocking.
- Partition Responsibilities: Users can dedicate certain cores for handling hardware interrupts, and avoid scheduling long latency tasks on these cores. As a result, interrupt response latency can be minimized.

5 Per-Core Security Coprocessors

Each core has a dedicated Security Coprocessor which can be used to accelerate security applications and hash generation. Once the core issues an instruction to the coprocessor, the core can continue to do other work while the coprocessor completes the instruction, or the core can wait for the coprocessor to complete the task.

Because each Security Coprocessor is dedicated to the core, there is no contention for this resource, increasing the determinism in performing security operations.

There are three primary advantages for integrating these Security Engines into each cnMIPS core:

- Current Industry Trends: Security processing technology is becoming an integral part of all networking applications and all kinds of networking devices. Every element in a network, end to end, will become security aware and capable. As a result, cnMIPS cores, with integrated security acceleration, offer the most flexible and scalable solution for implementing networking applications.
- Performance: The per-core Security Coprocessors provide optimal performance: there is no overhead or additional latency required to transfer data to and from the security acceleration hardware for processing. This is especially critical for smaller packets.
- Flexibility: The OCTEON security acceleration architecture can easily support novel modes for existing cryptographic algorithms, even modes that have not been defined today. This support can be added via a software upgrade without any hardware change. For example, suppose a new mode were invented for using the AES cryptographic algorithm, that used a different feedback algorithm, or that used a different operation to create the ciphertext. Such a mode could be easily supported with minor software changes.

6 On-Chip Interconnects

The on-chip interconnects join the different integrated units together. There are two key interconnects on the OCTEON and OCTEON Plus processors:

1. Coherent Memory Bus (CMB): The CMB connects all of the cnMIPS cores, the L2 cache controller, and the I/O Bridge. (Note: Although the CMB contains the word “bus”, it is actually an interconnect which contains several buses, with sophisticated hardware arbitration.)
2. I/O Interconnect (IOI): The I/O Interconnect connects the remaining on-chip functional units: the I/O Bridge, hardware acceleration units, miscellaneous units such as the PCI DMA Engine, and the I/O interfaces.

The I/O Bridge is used to join the two interconnects.

The bulk of the data traveling on the interconnects is packet data, so the interconnects are optimized to handle packet data efficiently.

The interconnects are designed to:

- Maximize performance
- Support linear scaling when cores are added
- Deliver low-latency, high-determinism data transfers
- Minimize power consumption

Key features of the interconnect architecture include:

- The topology of the buses is carefully designed to align with packet processing flow, optimizing the movement of packet data among the various on-chip units.
- The split transaction and highly pipelined buses optimize bandwidth utilization, and avoid unnecessary over-provisioning.
- The data transfer latency is minimized and data transfer latency determinism is maximized as a result of focused connection of only the relevant data producers and consumers on each interconnect. (In some instances there are direct connections between the producer and consumer.)
- The bandwidth provisioning for each of the buses in the CMB or I/O interconnect is optimized based on the type of data transfers that the bus is responsible for in the packet processing flow.
- The interconnect architecture is scaled to best serve the number of cores on each OCTEON model.

The overall power consumption is minimized because:

- The optimized topology of the overall interconnect structure minimizes the paths and distance that data travels.
- The focused bandwidth provisioning avoids unnecessary over-provisioning of bandwidth.

See the *HRM* for details.

6.1 The Coherent Memory Bus Interconnect

The CMB interconnect is a collection of buses which connect all the cnMIPS cores, the L2 cache controller (L2C), and the I/O Bridge (IOB). (Note: The information in this section is for the OCTEON, OCTEON Plus, and some of the OCTEON II models. Some of the higher core count OCTEON II models will have a different interconnect architecture, which includes a cross-bar hyper-connect to scale up to 32 cores and higher.)

The CMB consists of several split-transaction, and highly pipelined buses. The data buses in the CMB are designed to meet the needs of the most demanding applications, and are over-provisioned for scalability to 16-cores and higher. The details of the CMB vary with the OCTEON model.

6.2 I/O Interconnect

The I/O interconnect connects the various on-chip functional units: the I/O Bridge, hardware acceleration units, miscellaneous units such as the PCI DMA Engine, and the I/O interfaces.

The I/O subsystem is connected together through a number of split-transaction and highly pipelined buses that run at core frequency. These buses together comprise a vast amount of bandwidth provisioned for internal data transfer. The exact buses and width vary depending on the OCTEON model.

The following figure shows an example of packet input and output using buses in the I/O interconnect. The following buses are used to transfer packet data:

Receive:

- IOBI – I/O Bus for Input: The IOBI carries the packet data from the input port to the IPD.
- IPDB – IPD Bus: Dedicated for DMA transfers from the IPD to L2 cache/DRAM.

Transmit:

- PKOB – PKO Bus: Dedicated for DMA transfers from L2 cache/DRAM to the PKO's internal memory.
- POB – Packet Output Bus: Dedicated for transfers from the PKO's internal memory to the output port(s).

Figure 2: Packet Data Movement Over I/O BusesPacket Data Movement Over I/O Buses

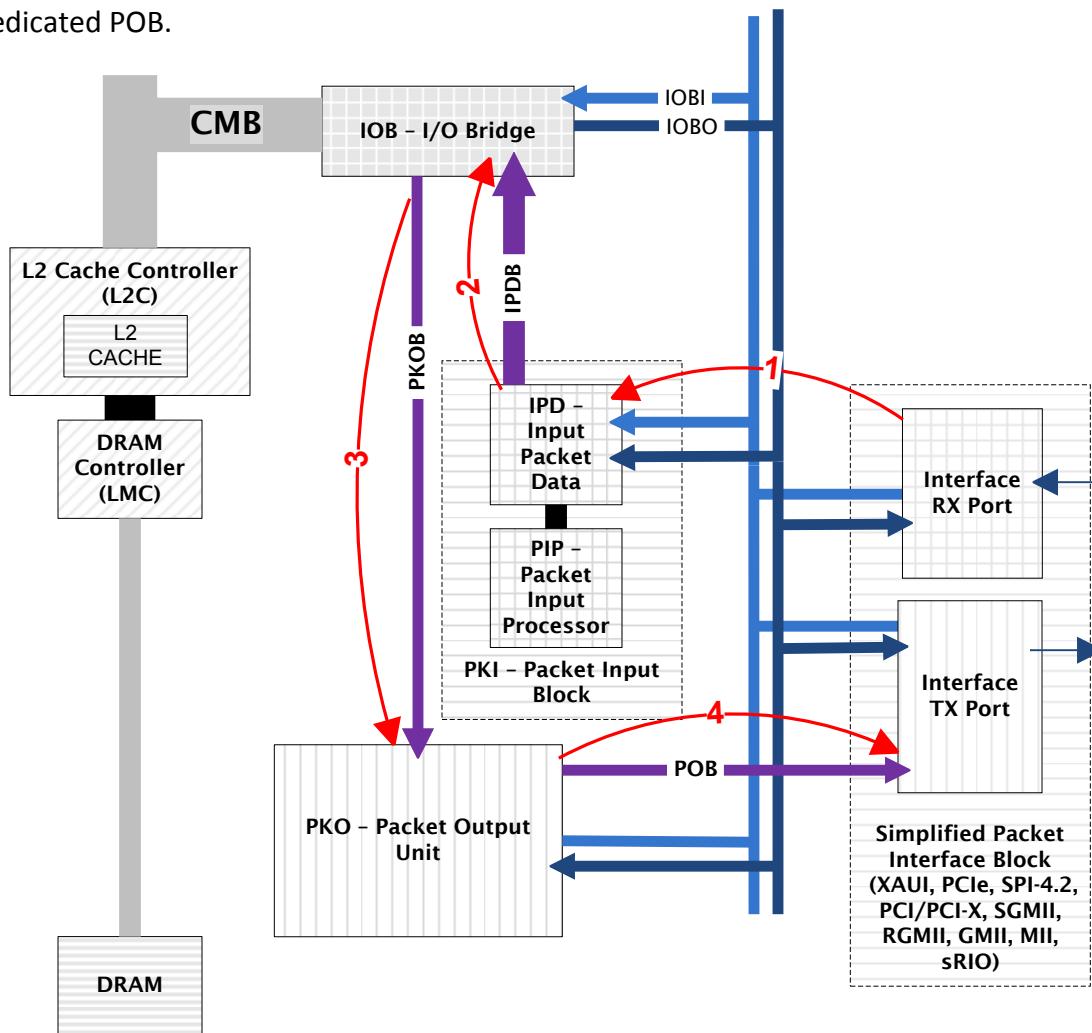
Most of the data traffic on the I/O interconnects is packet data. The I/O interconnect topology is carefully designed to optimize the flow of packet data:

Receive:

1. Receive packet data goes directly from the input interface to the IPD on the IOBI (the IPD is a second sink on the bus).
2. The IPD DMAs the packet data to L2 Cache/Memory via a dedicated IPDB.

Transmit:

3. The PKO DMAs the packet data from L2Cache/Memory to its internal memory on the dedicated PKOB.
4. The PKO sends the packet data from its internal memory to the output port on the dedicated POB.



7 Special Cavium Networks-Specific Instructions

Cavium Networks has added to the MIPS64 release 2 instruction set. The following list highlights some of the added instructions, but is not meant to be a complete list. For a complete list, see the *HRM*.

Note that the GNU C/C++ compilers supplied with the Cavium Networks SDK automatically generate most of these instructions. By re-compiling existing applications, users can automatically take advantage of these special instructions. Many instructions are also available through provided APIs.

Added instructions include:

- Bit field processing, extraction, and bit test branch instructions: MIPS64 release 2 provides a few bit-granularity instructions. Cavium Networks provides additional bit-granularity instructions. These instructions are used extensively in packet processing.
- Multiple instructions for controlling memory order: The MIPS64 ISA has a SYNC instruction for controlling memory order. Cavium Networks added several variations which provide finer memory order control for higher performance.
- Prefetch: The MIPS64 ISA provides a prefetch instruction which is used to load data to both L2 and L1 cache before it is actually needed. When the data is needed, it is already present, preventing the core from stalling while the data is fetched. Cavium Networks added prefetch instructions which allow the cores to:
 - prefetch data to L1 cache, bypassing the L2 cache (used to save space in the L2 cache, preventing eviction of cache blocks which are still needed)
 - prefetch to L2 cache without prefetching to L1 cache (used to prefetch data which will be needed by a different core than the requesting core)
- True unaligned loads and stores: The cnMIPS architecture processes unaligned data accesses without software intervention. In a traditional RISC processor core, unaligned data accesses result in an exception that the operating system needs to handle, significantly impacting system performance. The ability to process unaligned data also eases software reuse from legacy systems where unaligned data is allowed, saving the time and effort finding and fixing unaligned accesses.
- Atomic add to memory locations*: This instruction is used to atomically add a value to the value stored in a memory location. This instruction can be used to implement of large number of statistics counters in memory without explicit software locking. This instruction is not implemented in all OCTEON models.
- 32-bit, 64-bit population count: Population count instructions are used to determine number of bits that are set in a 32-bit or 64-bit piece of data. Population counts are useful in networking applications. For example, packet queues are typically managed as bit-masks with a set bit indicating that the corresponding queue is non-empty. Population count instructions can then determine the number of non-empty queues with a single instruction.

8 Cache Hierarchy

The OCTEON processor architecture offers the optimal caching policy for multi-core solutions, where efficient and low latency data sharing is critical to the overall performance.

The OCTEON processor cache hierarchy includes:

- Per-core L1 data cache
- Per-core L1 instruction cache
- Shared L2 cache (shared by all cnMIPS cores and the I/O sub-system)

The L1 data cache implements a hybrid write-through, write-back policy (using a write-buffer mechanism). The L2 cache implements a write-back policy.

The OCTEON processor architecture also offers many innovative cache-related features, including:

- Don't Write Back (DWB): The OCTEON processor provides the option not to write back selective data to L2/DRAM, to avoid unnecessary L2 data writes to memory. For example, the packet data in L2 cache can be discarded after the packet is transmitted; there is no need to store the data to L2/DRAM. In a conventional L2 cache design, all dirty data is written back to memory. This feature allows the user to tune the system to conserve memory bandwidth and power.
- L2 Cache Way Partitioning: The L2 cache ways can be partitioned among the cnMIPS cores and the I/O sub-system. This OCTEON feature enables intelligent management of the L2 cache to minimize cache pollution and the resulting loss of performance.
- Flexible Control of Data Movement: The OCTEON processor provides flexible control of moving data among the L2 cache, main memory, application acceleration engines, and I/O sub-system. For example, the OCTEON processor can be configured to automatically send:
 - the received packet header to the L2 cache
 - the packet data to main memory, bypassing the L2 cache
- Data Prefetch Instructions: The OCTEON processor provides multiple prefetch instructions to move data into L1 and/or L2 caches prior to the application needing the data. These instructions are used to avoid cache misses.

9 Summary

The OCTEON family of processors is highly optimized to achieve the highest performance per dollar and per Watt for a wide variety of networking, security, wireless, and storage applications. The OCTEON processor can be used for control-plane applications, data-plane applications, or a hybrid of both.

The OCTEON and OCTEON Plus processors have been designed into products from a significant majority of tier-1 OEM customers. They have delivered market-leading performance, along with low cost points and dramatically reduced power consumption.

Key features of the OCTEON family include:

- Custom-designed dual issue MIPS64 release 2 cores, with additional innovative Cavium Networks instructions added for improved performance.
- Linear scalability from 1 to 32 cores to provide the widest range of performance, power, and price options, while providing full software compatibility.
- Extensive integrated hardware acceleration units to dramatically improve overall application performance and automatically load-balance and synchronize processing.
- High-speed interconnects along with powerful DMA Engines, which are designed to optimize the flow of packet data through the OCTEON processor.
- Integrated I/O interfaces and memory controllers, which enable reduced bill of materials (BOM) cost and smaller form-factor designs.
- Simple software architecture based on standard C/C++ code, GNU toolchains, industry-standard operating systems, and optimized software stacks.

Software development complexity is minimized by hardware acceleration units, flexible software architecture, standard MIPS64, industry-standard toolchains and operating systems, packet-linked locks, and built-in scaling support.

OCTEON processors provide a proven, industry-leading solution for embedded developers looking to achieve fastest time-to-market with leading product performance and features.

Packet Flow

TABLE OF CONTENTS

TABLE OF CONTENTS	1
LIST OF TABLES.....	3
LIST OF FIGURES	3
1 Introduction.....	4
2 Packet Flow Overview.....	4
3 Hardware Features to Accelerate Packet Processing.....	15
3.1 Hardware Management of Packet Classification and Priority.....	15
3.2 Hardware Management of Buffer Pools: The Free Pool Allocator (FPA) Unit.....	15
3.2.1 Allocating a Buffer	16
3.2.2 Freeing a Buffer	16
3.3 Hardware Management of Packet-Linked Locks	16
3.4 Hardware Management of Packet Order	17
4 The Schedule / Synchronization / Order (SSO) Unit.....	17
4.1 Phase 1: Packet Input.....	18
4.1.1 Ingress Order	18
4.1.2 Packet Data Buffer.....	18
4.1.3 5-Tuple.....	18
4.1.4 Flow	20
4.1.5 Tuple Hash Value	20
4.1.6 Tag Value, First Tag Value	20
4.1.7 Tag Type (TT), First Tag Type.....	20
4.1.8 Tag Tuple.....	20
4.1.9 ORDERED Tag Type: Parallel Processing	20
4.1.10 ATOMIC Tag Type: Serialized Processing: Accessing Critical Regions	21
4.1.11 NULL Tag Type: Unordered, Not Serialized, Not Synchronized	21
4.1.12 Quality of Service (QoS) Value.....	22
4.1.13 Group (Grp)	22
4.1.14 Work Queue Entry (WQE)	22
4.1.15 The add_work Operation.....	23
4.1.16 QoS Input Queues.....	23
4.1.17 Phase 1 Summary:	24
4.2 Phase 2: SSO Schedules New Work to the Core.....	24
4.2.1 SSO Work Descriptors	25
4.2.2 Cached Input Queues and Overflow Input Queues	25
4.2.3 The get_work Operation.....	26

4.2.4	Core State Descriptor.....	27
4.2.5	Scheduled.....	28
4.2.6	Descheduled.....	30
4.2.7	In-Flight.....	30
4.2.8	Tag Tuple.....	30
4.2.9	In-Flight Queues	30
4.2.10	ORDERED Tag Type: Parallel Processing	33
4.2.11	ATOMIC Tag Type: Locking Critical Regions.....	33
4.2.12	NULL Tag Type: Unordered.....	33
4.2.13	Choosing the Next WD to Schedule; Skipping Un-schedulable WD.....	34
4.2.14	Phase 2 Summary	35
4.3	Phase 3: Lock Critical Region: One-at-a-time Access	36
4.3.1	The <code>switch_tag</code> Operation (Tag Switch).....	36
4.3.2	Switch Tag Sequence.....	36
4.3.3	Core's Switch Complete Bit	37
4.3.4	Initial In-Flight Queue	38
4.3.5	Target In-Flight Queue	38
4.3.6	Tag Switch Processing.....	38
4.3.6.1	Tag Switch Processing Steps.....	38
4.3.6.2	Tag Switch from ORDERED to ATOMIC	40
4.3.6.3	Tag Switch from ATOMIC to ORDERED	42
4.3.6.4	Tag Switch from ORDERED to ORDERED	42
4.3.7	Phase 3 Summary	43
4.4	Phase 4: Unlock Critical Region and Resume Parallel Processing	44
4.5	Phase 5: Packet Output	44
4.5.1	PKO Output Ports	44
4.5.2	PKO Output Queues	44
4.5.3	PKO Output Queue to Port Mapping.....	45
4.5.4	Selecting the PKO Output Queue	45
4.5.5	Freeing the WQE Buffer.....	45
4.5.6	Locking the PKO Output Queue.....	46
4.5.7	Transmitting Packets in Ingress Order	46
4.5.8	Writing to the Output Queue, then Freeing the Lock	46
4.5.9	Freeing the Work Descriptor and Releasing the Lock	46
4.5.10	PKO DMAs the packet to the TX Port	46
4.5.11	Freeing the Packet Data Buffer.....	46
4.5.12	Phase 5 Summary	47
4.6	Workflow Model: One Flow	47
4.7	Workflow Model: Multiple Flows.....	48
4.8	Summary.....	48

LIST OF TABLES

Table 1: ORDERED, ATOMIC, and NULL Tag Types: Example Use	22
Table 2: Example Packet Processing Phases	51

LIST OF FIGURES

Figure 1: Packet Flow Diagram Part 1: PACKET INPUT	6
Figure 2: Packet Flow Diagram Part 2: SSO AND CORE PROCESSING	7
Figure 3: Packet Flow Diagram Part 3: PACKET OUTPUT	8
Figure 4: Steps 1 and 2 Shown in Detail	9
Figure 5: Steps 3 and 4 Shown in Detail	10
Figure 6: Steps 5 and 6 Shown in Detail	11
Figure 7: Steps 7 and 8 Shown in Detail	12
Figure 8: Steps 9 and 10 Shown in Detail	13
Figure 9: Steps 11 and 12 Shown in Detail	14
Figure 10: The 5-Tuple Fields in IPv4 TCP/IP Header	19
Figure 11: The First Two Words of the Work Queue Entry	22
Figure 12: The add_work Operation	23
Figure 13: Simplified View of SSO Input Queues	24
Figure 14: Simplified View of the Work Descriptor Data Structure	25
Figure 15: Simplified View of Cached Input Queues and Overflow Input Queues	26
Figure 16: The get_work Operation	27
Figure 17: Simplified View of the Core State Descriptor Data Structure	28
Figure 18: Core State Descriptors Shown as Scheduled and Unscheduled	29
Figure 19: Data Structures After a Successful get_work Operation	29
Figure 20: In-Flight Queues	31
Figure 21: Core State Descriptor: Scheduled	32
Figure 22: ORDERED Tag Types Execute in Parallel, ATOMIC Tag Types Wait for the Lock ..	34
Figure 23: Un-Schedulable Work Descriptors: ORDERED versus ATOMIC Tag Types	35
Figure 24: The Core's Switch Complete Bit	38
Figure 25: Tag Switch from ORDERED to ATOMIC	41
Figure 26: Tag Switch from ORDERED to ORDERED	43
Figure 27: PKO Output Queues and Output Ports	45
Figure 28: ATOMIC Tag Used to Guarantee Transmission in Ingress Order	47
Figure 29: Packet Processing Phases and Sequential Tag Switch Operations	50

1 Introduction

The OCTEON processor contains several functional blocks which work together to manage packet flow through the processor. In this document, the word *processor* refers to the entire chip, with all of the different functional hardware blocks including all of the cores on the chip.

This chapter is divided into several parts:

- In Section 2 – “Packet Flow Overview”, a typical packet flow (packet received, processed, transmitted) is discussed, showing how the different functional blocks work together to create reliable, fast packet processing.
- In Section 3 – “Hardware Features to Accelerate Packet Processing”, key hardware acceleration features are discussed:
 - 1) hardware management of packet classification and priority
 - 2) hardware buffer management
 - 3) hardware packet-linked locks
 - 4) hardware management of packet order
- In Section 4 – “The Schedule / Synchronization / Order (SSO) Unit”, the Schedule / Synchronization / Order unit (SSO) is introduced. The SSO provides essential capabilities which are unique to the OCTEON processor: it is the “heart” of OCTEON. Because of the SSO’s importance, the rest of this chapter introduces its hardware acceleration features, and describes how software may take advantage of them. The SSO unit provides the following key functions:
 - 1) Schedule: Schedules packets to be processed based on Quality of Service (QoS) priority.
 - 2) Synchronization: Provides hardware support for synchronization by providing packet-linked locks. These locks can be used to protect critical regions, or serialize packet transmission in ingress order.
 - 3) Order: Maintains ingress order during all packet processing phases. Ordered packets may be processed in parallel, but the SSO keeps track of their ingress order. When they switch to a different processing phase, or request a packet-linked lock, the switch processing is done based on ingress order.

Note: In this document the *core* means software running on any or all of the multiple cores available on your processor model.

Note: It is intended that this chapter is read in order, because the information presented throughout chapter will increase in complexity. The concepts presented in this chapter are used throughout the book. Understanding the material in this chapter is essential to understanding other chapters in the book.

2 Packet Flow Overview

This section introduces the key functional blocks on the OCTEON processor as used in packet processing, and how a packet moves through the processor.

The figures on the following pages illustrate the packet flow through an OCTEON processor.

The first three figures are an overview of the packet flow. After these figure, the packet flow is detailed in a series of separate figures. Both the overview and detailed views cover the same material, but the buses involved are easier to see in detailed views.

There are two pseudo blocks shown in these figures: the “Simplified Packet Interface Pseudo Block” and the “Packet Input (PKI) Pseudo Block”. These two pseudo blocks are included in the drawing to help the reader understand the various functions on the OCTEON processor.

- The Simplified Packet Interface pseudo block is a generic representation of the receive (RX) or transmit (TX) functions of any of several interfaces which can be used to receive packets. A packet interface can be any of: XAUI, PCIe, SPI-4.2, PCI/PCI-X, SGMII, RGMII, GMII, or MII, etc. All of these interfaces can send packets to the Packet Input Block. (Note: MII is only used as a packet interface on older OCTEON processors). The Packet Interface pseudo block has been greatly simplified and contains only two sub-blocks marked “Interface TX Port” and “Interface RX Port”.
- The Packet Input pseudo block consists of two blocks: the Packet Input Processor (PIP), and the Input Packet Data (IPD) block. This pseudo block receives and pre-processes data from the Simplified Packet Interface.

The packet flow is shown in three different parts in the following three figures.

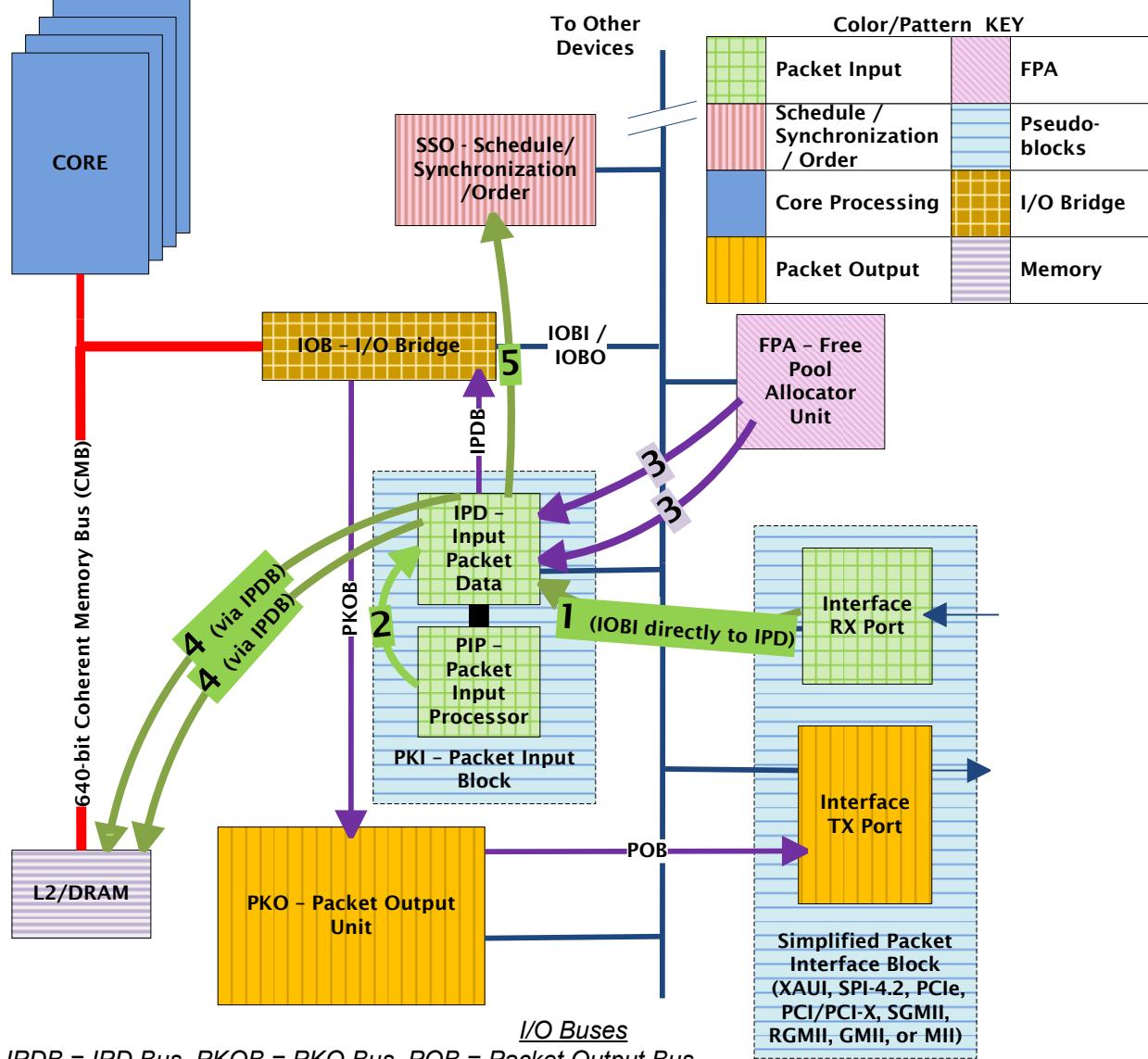
1. Packet Input
2. SSO and Core Processing
3. Packet Output

Note: These drawings show a simplified view of packet flow. In an actual system, many activities happen in parallel. For instance, the IPD prefetches buffers and the core prefetches packets while still processing the current packet. There are many special performance features which are not detailed in this chapter.

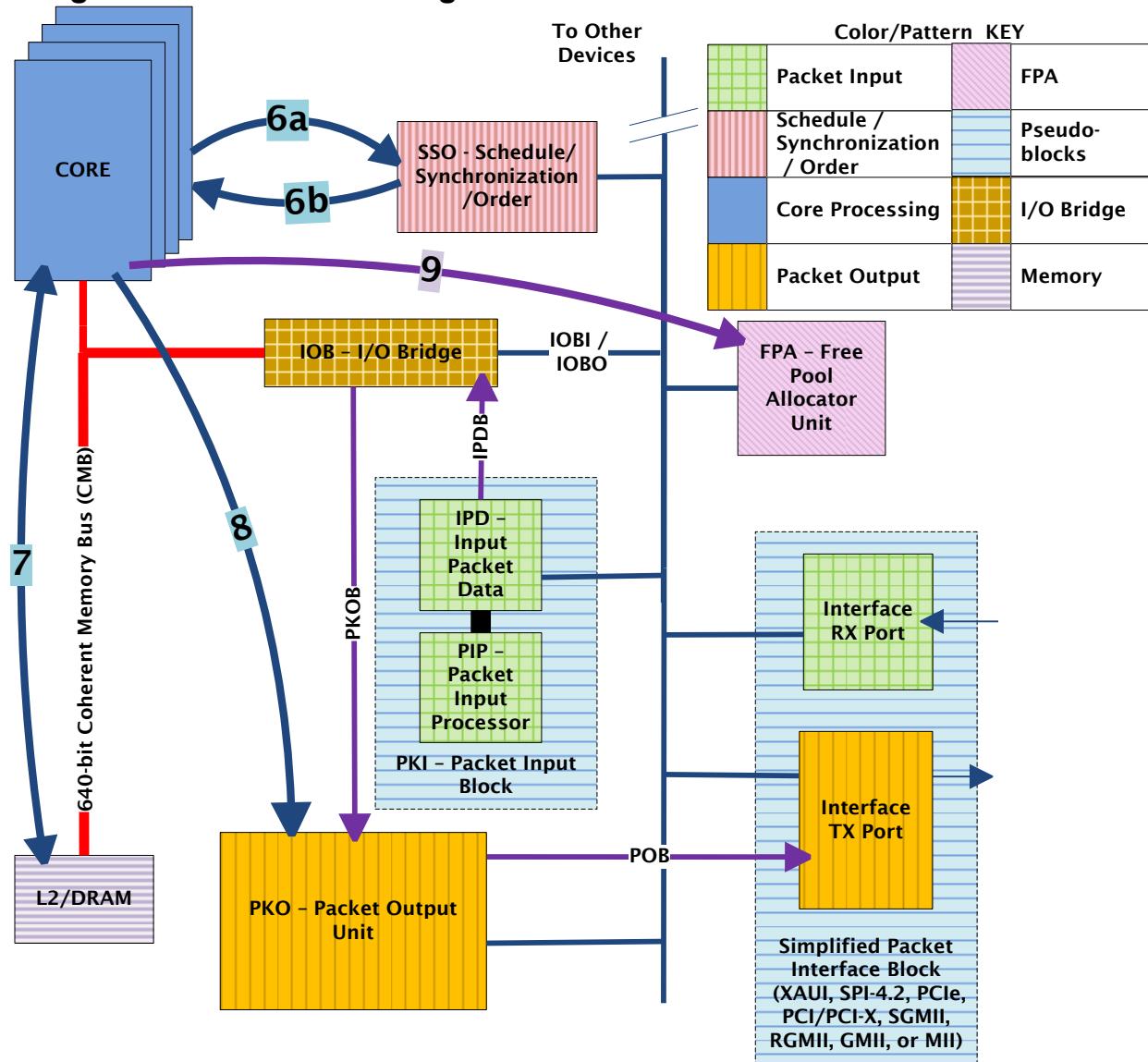
Note: These drawings assume the packet is not dropped due to configurable settings for Random Early Dropping (RED), or packet error. In RED, packets may be dropped at ingress if internal buffers are approaching full.

Note: The pointers referred to in this document contain physical addresses. Software Development Kit (SDK) functions such as `cvmx_pow_work_request_sync()` convert physical addresses to virtual addresses as needed.

Figure 1: Packet Flow Diagram Part 1: PACKET INPUT



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

Figure 2: Packet Flow Diagram Part 2: SSO AND CORE PROCESSING

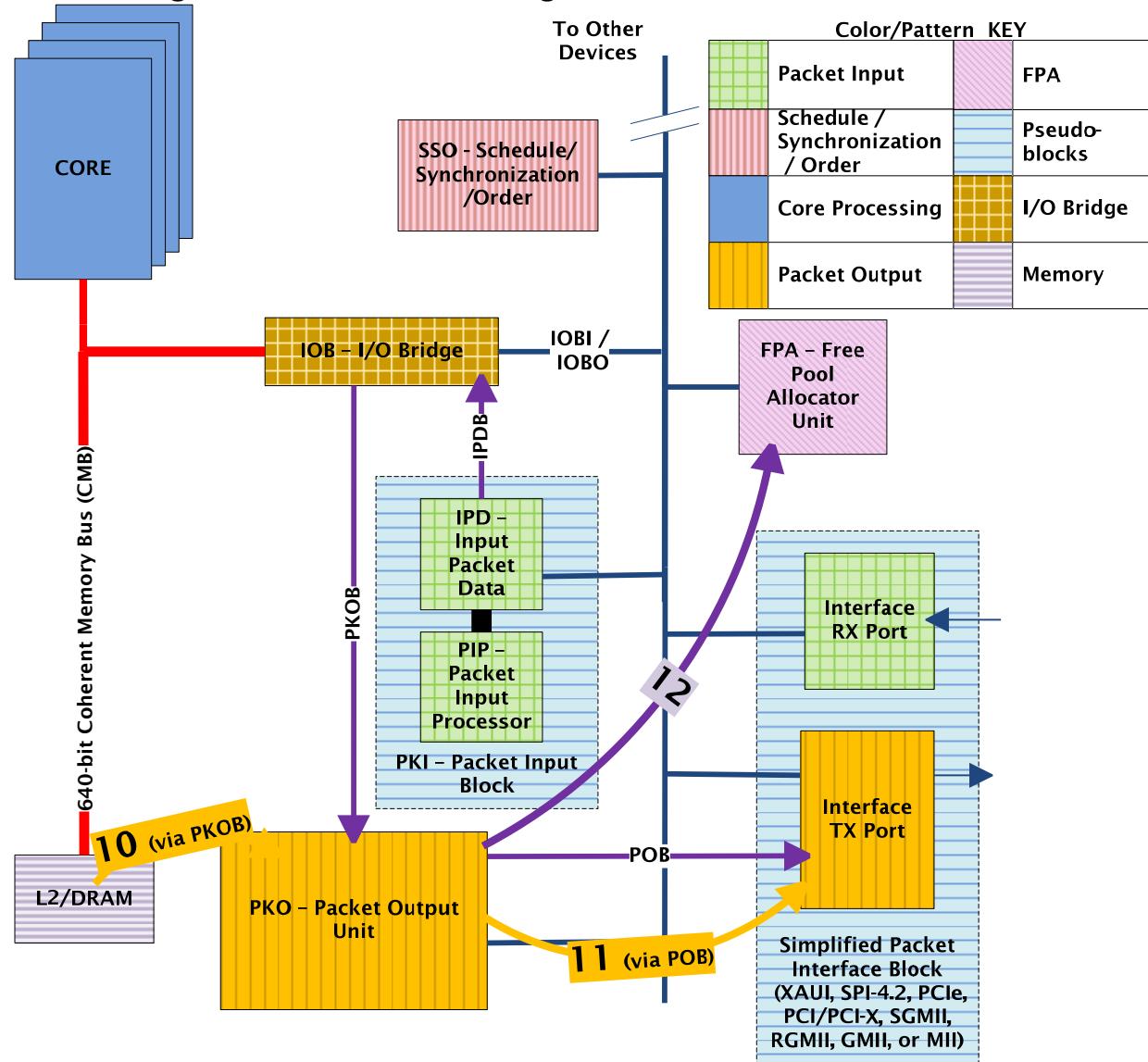
The SSO schedules the WQE based on QoS priority, ingress order, and current locks for that flow.

6a,6b. The core performs the `get_work` operation to get a new WQE pointer from the SSO. The WQE contains the Packet Data Buffer pointer.

7. The core processes the packet data, reading and writing the packet data in L2/DRAM.

8. After processing the packet data, the core sends the Packet Data Buffer pointer and the data offset to the appropriate Packet Output Queue in the Packet Output (PKO) Unit. The queue's configuration specifies the output port and packet priority. If needed, the packet is output in ingress order.

9. The core frees the WQE Buffer back to the FPA.

Figure 3: Packet Flow Diagram Part 3: PACKET OUTPUT


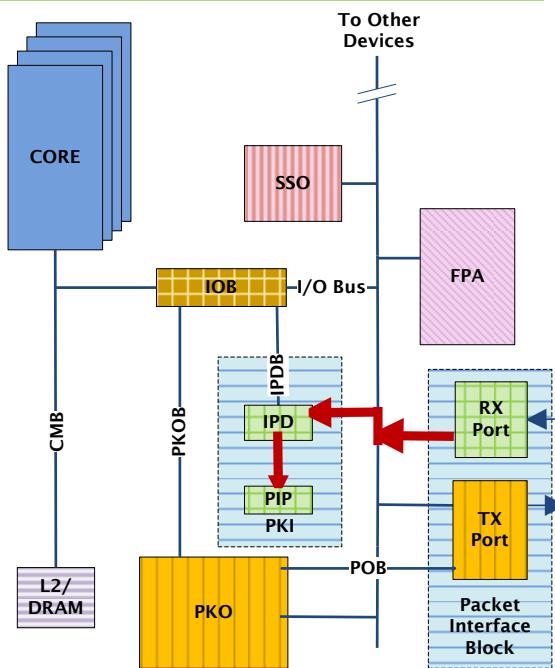
10. The PKO DMA's data from the Packet Data Buffer in L2/DRAM into its internal memory (via PKOB).
11. The PKO optionally adds the TCP or UDP Checksum, then sends the packet data from its internal memory to the Output Port (via POB). The Interface TX Port will transmit the packet. The PKO optionally notifies the core that the packet was sent.
12. The PKO frees the Packet Data Buffer back to the FPA.

The following drawings show the same material as the three above, but display the buses involved in the transactions. Understanding this information is useful in performance analysis: by visualizing the exact flow of the data in the processor, it is often possible to anticipate and avoid

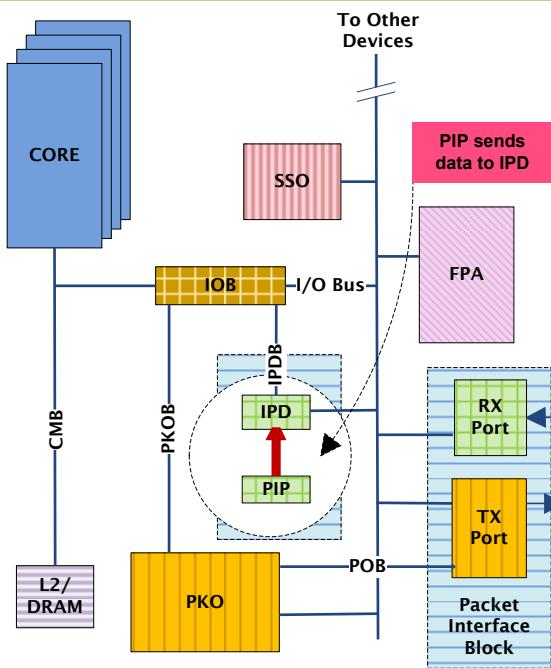
performance problems. Note that the special buses and DMA facilities provided for data transfer will keep the IOB from becoming overloaded.

Figure 4: Steps 1 and 2 Shown in Detail

1. After the Interface Rx Port receives the packet and checks it for errors, it passes the packet to the Input Packet Data (IPD) Unit (via the IOB). The IPD shares the data with the Packet Input Processor (PIP). These two units work together to process the input packet.



2. After the PIP performs the packet parsing, including any checks configured by software, it computes the data needed by the IPD for the Work Queue Entry (WQE) Fields (work flow and QoS).

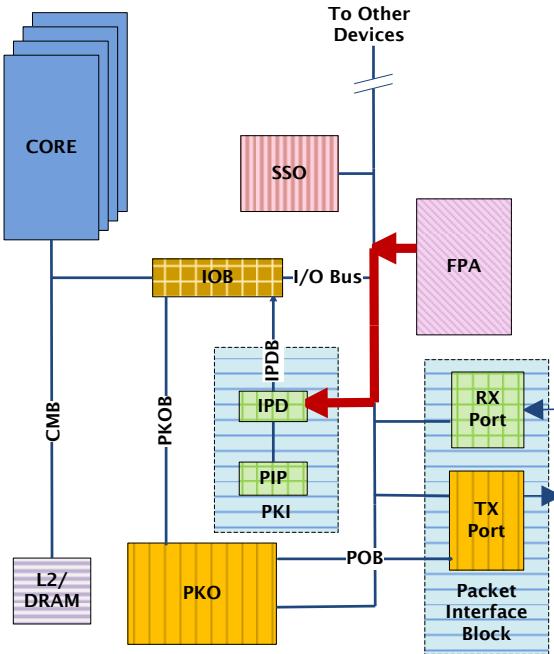


PACKET FLOW

Figure 5: Steps 3 and 4 Shown in Detail

PACKET FLOW

3. If IPD does not drop the packet (RED), it allocates a WQE Buffer and Packet Data Buffer from the Free Pool Allocator Unit (FPA). (The FPA manages the free buffers.)



4. The IPD writes the WQE fields to the WQE Buffer, and writes the packet data to the Packet Data Buffer (DMA via IPDB).

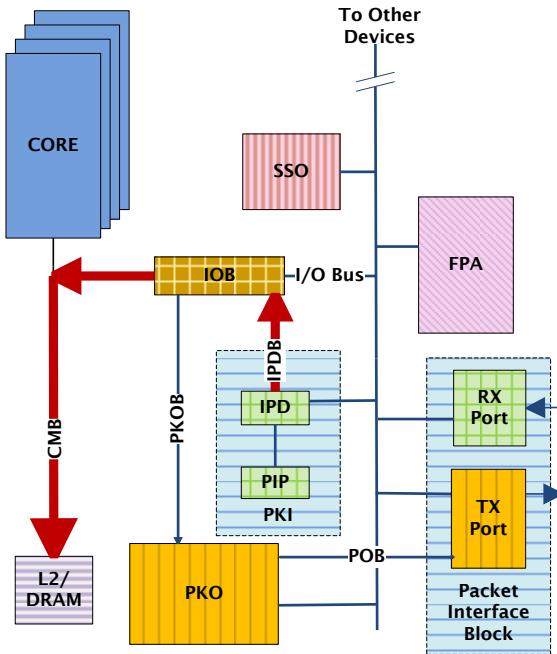


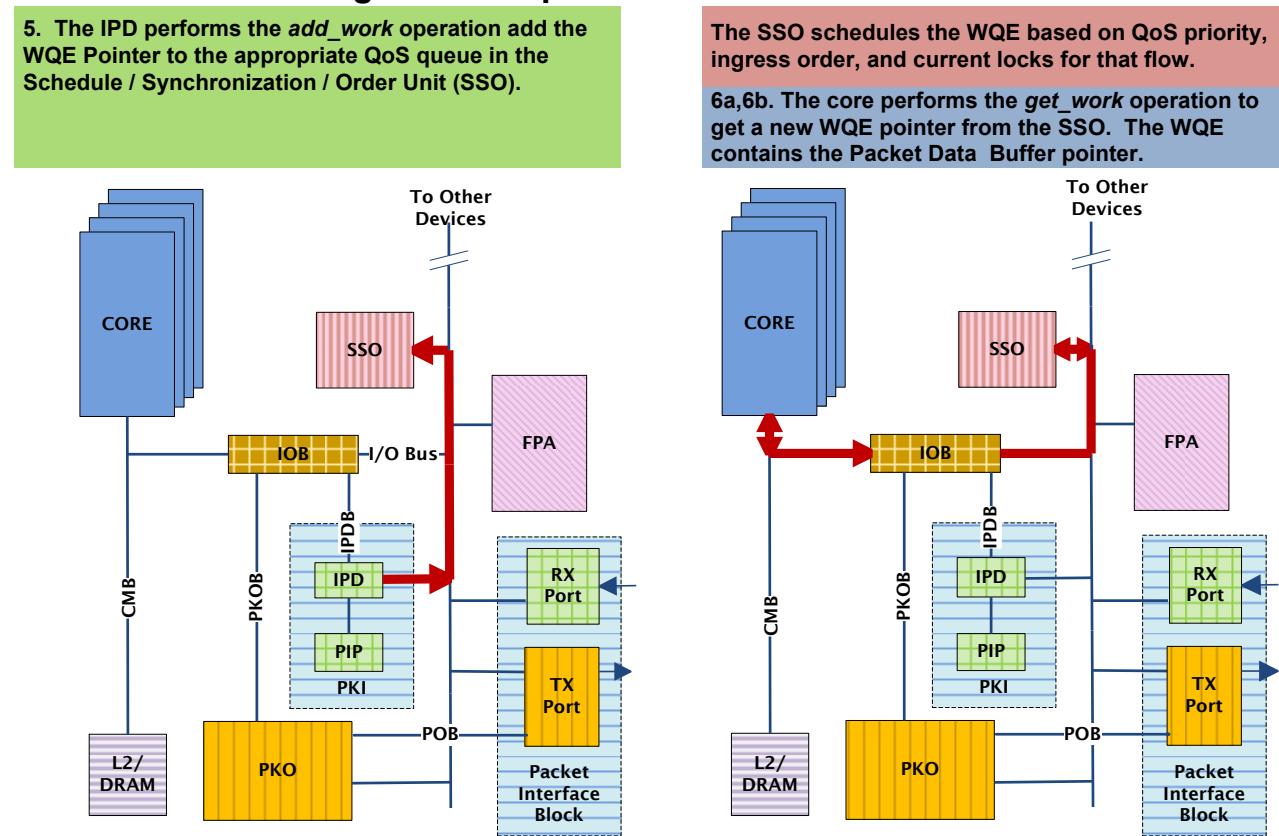
Figure 6: Steps 5 and 6 Shown in Detail

Figure 7: Steps 7 and 8 Shown in Detail

7. The core processes the packet data, reading and writing the packet data in L2/DRAM.

8. After processing the packet data, the core sends the Packet Data Buffer pointer and the data offset to the appropriate Packet Output Queue in the Packet Output (PKO) Unit. The queue's configuration specifies the output port and packet priority. If needed, the packet is output in ingress order.

PACKET FLOW

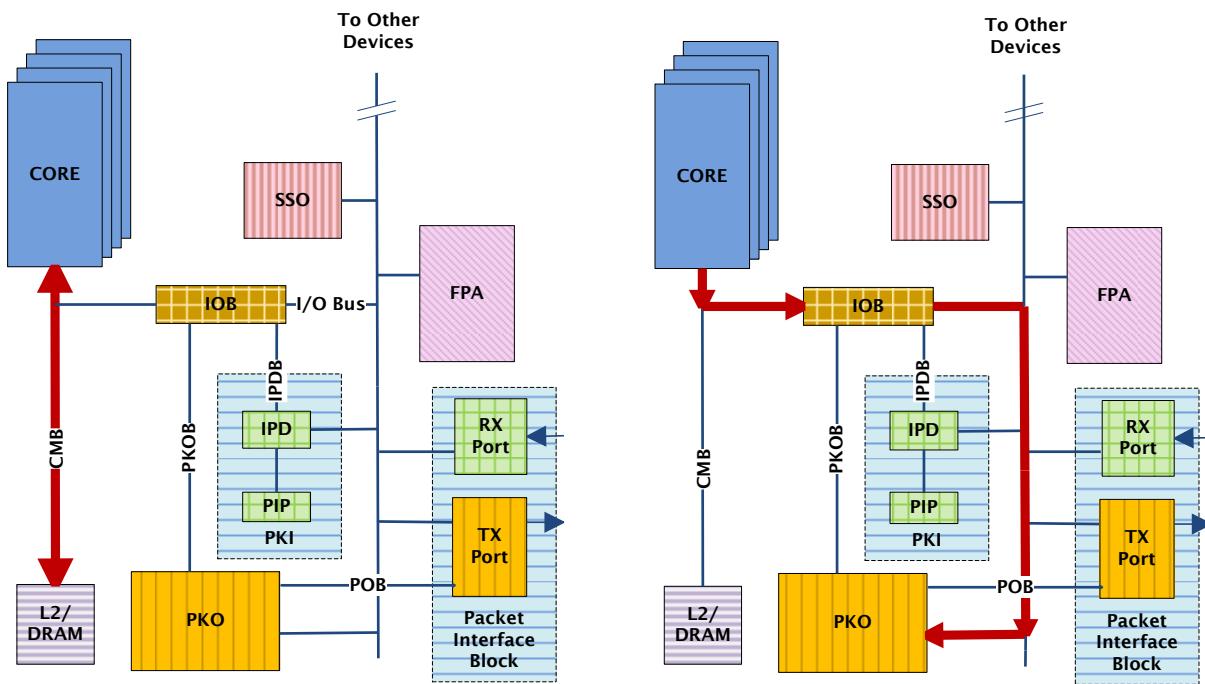


Figure 8: Steps 9 and 10 Shown in Detail

9. The core frees the WQE Buffer back to the FPA.

10. The PKO DMAs the data from the Packet Data Buffer in L2/DRAM into its internal memory (via PKOB).

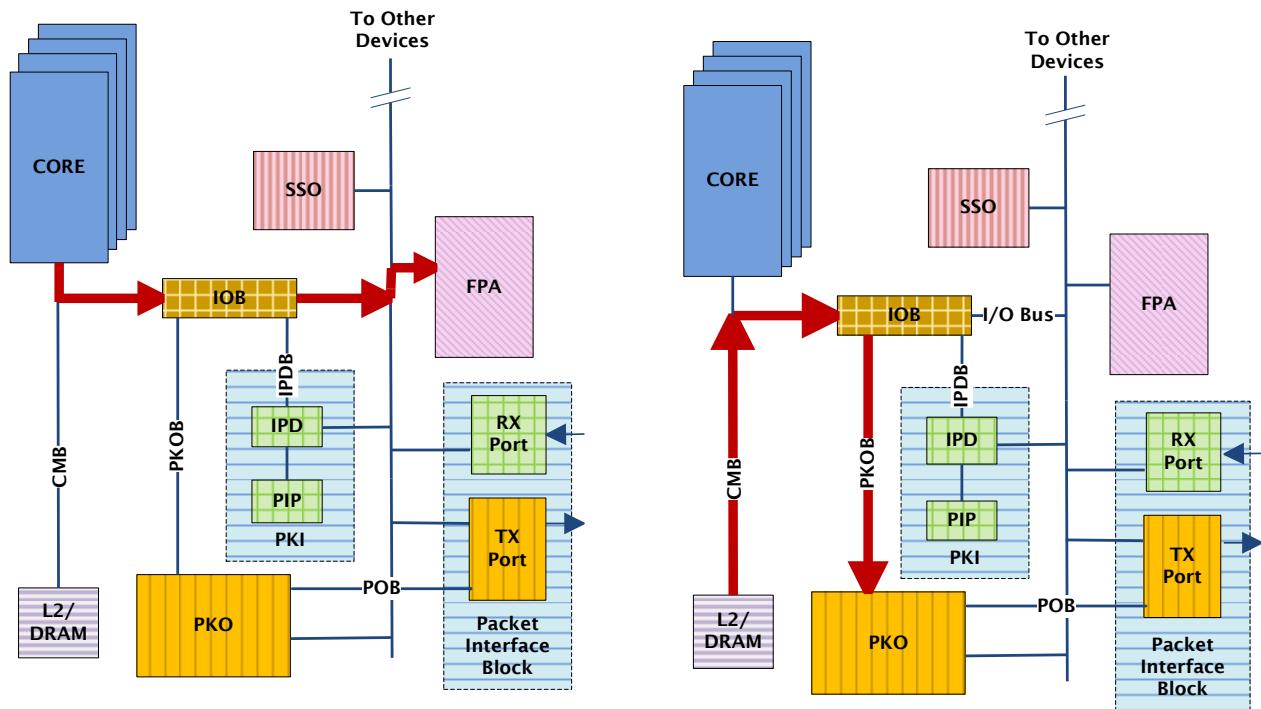
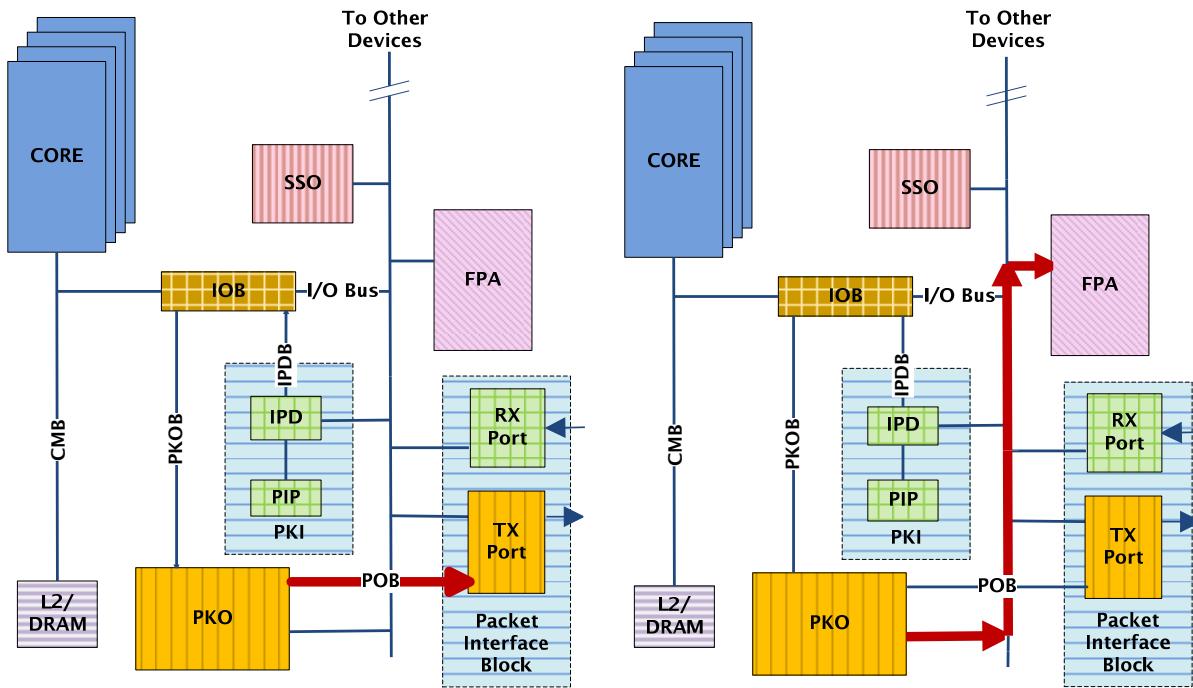


Figure 9: Steps 11 and 12 Shown in Detail

11. The PKO optionally adds the TCP or UDP Checksum, then sends the packet data from its internal memory to the Output Port. (via POB). The Interface TX Port will transmit the packet. The PKO optionally notifies the core that the packet was sent.

12. The PKO frees the Packet Data Buffer back to the FPA.

PACKET FLOW



3 Hardware Features to Accelerate Packet Processing

Before going into detail about packet processing, there are several important OCTEON features which are essential to accelerating packet flow. These are:

1. Hardware management of packet classification and priority
2. Hardware management of buffer pools
3. Hardware management of packet-linked locks
4. Hardware management of packet order

3.1 Hardware Management of Packet Classification and Priority

A key feature of the OCTEON processor is the hardware management of packet classification and priority.

Hardware classification and prioritization includes:

- Packets entering the processor are classified and given a QoS priority by the PKI, and then are put into the specified QoS Input Queue in the SSO. The priority of the QoS Input Queues is highly configurable.
- The core requests a packet from the SSO. The SSO returns the highest priority schedulable packet available. When the core receives the packet, it is ready to be processed. The core does not have to inspect packet priority. It also never receives a packet which is blocked waiting for a lock.
- After packet processing is complete, the core puts the packet on a PKO Output Queue for transmission. The PKO Output Queues are mapped to specific output ports on the Packet Interfaces. The priority of the PKO Output Queues is highly configurable.

This hardware processing removes packet flow bottlenecks by allowing cores to immediately work on packets in parallel, without the need to classify and prioritize the packets first.

This is a simplified view of the hardware management of packet priority. The SSO and other hardware units provide powerful and flexible scheduling capability. These capabilities are covered in detail in each hardware unit's chapter (see *OCTEON Programmer's Guide, Volume 2*).

3.2 Hardware Management of Buffer Pools: The Free Pool Allocator (FPA) Unit

Packet flow is very dependent on memory buffers. The Free Pool Allocator (FPA) unit manages the memory buffers for the other hardware units on the OCTEON processor.

These buffers are allocated from L2/DRAM at system initialization, and are arranged into up to 8 pools of buffers. Usually each pool contains buffers which are all the same size, and are all used for the same purpose. For instance, there are separate pools for the two types of buffers which are introduced below: Packet Data Buffers and Work Queue Entry Buffers.

The size of each pool is limited only by the amount of L2/DRAM allocated for the pool. Ideally, each pool size is configured carefully so there are always free buffers available.

The *FPA* chapter (in volume 2) contains details on how the buffers are configured and used, and provides information which can help prevent common configuration and coding errors.

3.2.1 Allocating a Buffer

A hardware unit, such as the IPD or core, may request buffers from a specific pool in the FPA. The FPA takes the buffer pointer off the specified pool, and gives the buffer pointer to the requester. This operation is called *allocating a buffer*.

Buffers may be pre-allocated from the FPA and stored by the requester until they are needed. This pre-allocation is used by both the IPD and the core to speed packet processing. The pre-allocation is asynchronous, leaving the core free to do other processing.

3.2.2 Freeing a Buffer

Many hardware units can return a buffer back to an FPA pool. The FPA takes the buffer pointer and places it in the specified pool. This FPA operation is called *freeing a buffer*. Although buffers may be returned to a pool other than the one from which they came, this is almost always a poor design. Freeing buffers to the wrong pool is a common software error.

3.3 Hardware Management of Packet-Linked Locks

One of the topics discussed in this chapter is hardware management of locks to protect critical regions. These locks are not generic locks: they are directly connected to packet processing, so are referred to as *packet-linked locks*.

During packet processing, critical regions such as code which reads/modified the TCP/IP control block, must to be protected with a lock. There is one TCP/IP control block per flow. Access to the lock must be in packet-ingress-order, not random-order or first-come-first-serve. The hardware locking facilities are perfect for packet-linked locking.

Hardware packet-linked locking features include:

- Offloads software: Hardware locking improves system efficiency by offloading software: The SSO is responsible for granting the lock, and figuring out which packet is next in line for the lock.
- Atomic access: Only one packet at a time is granted the lock.
- Lock is granted in packet ingress order: Often, the critical regions should be locked in the order packets were received, not the order the lock was requested. The SSO provides hardware locks which enforce this rule.
- Asynchronous lock acquisition: Another key feature is asynchronous lock acquisition. The core can request the lock before it is needed, finish up packet processing, and then check whether the lock has been granted. The core is notified when the lock is granted.

The packet-linked lock is also used to synchronize packet processing.

Note: Because the core can have only one packet assigned to it at a time, it can only hold/request one packet-linked lock at a time.

The ATOMIC tag type is used to provide packet-linked locks. This type of lock is explained in detail in Section 4 – “The Schedule / Synchronization / Order (SSO) Unit”.

3.4 Hardware Management of Packet Order

Another key OCTEON feature is hardware management of packet order.

When parallel processing is desired, multiple cores may work simultaneously on multiple packets from the same flow. When the packets change processing phases, or need access to a hardware lock, the SSO will force the packets to stay in ingress order. These features are used to maintain packet order from ingress to egress.

Hardware management of packet order is explained in detail in Section 4 – “The Schedule / Synchronization / Order (SSO) Unit”.

4 The Schedule / Synchronization / Order (SSO) Unit

The rest of the chapter is an introduction the Schedule / Synchronization / Order (SSO) unit. (Note that this unit is called “POW (Packet / Order / Work)” in the *Hardware Reference Manual*.)

The SSO is powerful, flexible, and essential. The information presented in this chapter is simplified, focusing on those features required to manage packet flow. These key features provide hardware acceleration to packet processing. Understanding these features is essential to writing high-performance code. The SSO chapter (in volume 2) will cover the features and SSO configuration in detail, and depends on this introductory material.

The focus of this discussion is how the SSO’s scheduling functions are used to:

- Prioritize: Guarantee cores work on the highest priority packets available.
- Reduce core stalling: Guarantee the new packet is given to work on is not blocked waiting for a lock. Support asynchronous operations which allow the cores to stay busy while hardware manages requests.
- Scale: Allow the cores to work on more than one packet in a flow at the same time whenever possible. Packets from different flows are always processed in parallel.
- Protect critical regions: only one packet in a flow is allowed to access the critical region at a time (ATOMIC access).
- Serialize access: Guarantee that critical regions are accessed by all packets in a flow in the order they were received, not the order the lock was requested, or in random order.
- Maintain ingress order: Guarantee that packets in a flow move through packet processing phases in the order they were received.

The SSO scheduling functions are implemented by concepts specific to Cavium Networks. The following simplified example introduces these concepts. This simplified example divides packet processing into several different phases:

1. Phase 1: Packet Input
2. Phase 2: SSO schedules new work to the core: cores work on packets in parallel
3. Phase 3: Lock critical region: one-at-a-time access
4. Phase 4: Unlock critical region and resume parallel processing

5. Phase 5: Packet Output

4.1 Phase 1: Packet Input

The PKI block receives the packet from the Packet Interface block, performs header checks and flow classification, and stores the packet data in L2/DRAM. The PKI block creates a data structure which contains the information needed by the SSO to manage the scheduling, synchronization, and ordering of the packet. The PKI block submits the packet information (which includes a pointer to the data) to the SSO. The SSO will put the packet information onto the selected QoS Input Queue.

The following key concepts and operations are needed to understand this phase of the packet processing.

4.1.1 Ingress Order

The *ingress order* is the order that packets from the same flow arrived at the PKI and were submitted to the SSO.

4.1.2 Packet Data Buffer

The *Packet Data Buffer* contains the received packet data. The IPD allocates the buffer from the FPA. (The FPA manages free buffers. The memory for the FPA's buffer pools is allocated at system initialization, and the buffers are put into buffer pools at that time.) The IPD then copies the packet data into the buffer. The buffer is usually freed back to the FPA buffer pool by the PKO after it reads the packet data into its internal memory.

4.1.3 5-Tuple

5-tuple is a common networking term which refers to classification of a packet by its IP protocol, IP source address, IP destination address, and (if present) source port and destination port.

For example, Figure 10 – “The 5-Tuple Fields in IPv4 TCP/IP Header” shows the 5-tuple fields for an IPv4 TCP/IP packet, with the 5-tuple fields highlighted. The sizes of five highlighted fields sum to 13 bytes of information.

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

Figure 10: The 5-Tuple Fields in IPv4 TCP/IP Header

0

15 16

31

ip_v version	ip_hl header length	ip_tos type of service	ip_len total length						
ip_id identification			ip_off flags and fragment offset						
ip_ttl time to live	ip_p protocol		ip_sum IP checksum						
ip_src 32-bit source IP address									
ip_dst 32-bit destination IP address									
options (if any) (if ip_hl >5)									
th_sport 16-bit source number		th_dport 16-bit destination port number							
th_seq 32-bit sequence number									
th_ack 32-bit acknowledgement number									
th_off 4-bit header length	th_x2 reserved (6 bits)	th_flags flags	th_win 16-bit window size						
th_sum 16-bit TCP checksum			th_urp 16-bit urgent offset						
options (if any)									
data (if any)									

4.1.4 Flow

Flow is a common networking term which refers to a uni-directional collection of packets which share the same 5-tuple.

4.1.5 Tuple Hash Value

The *tuple hash value* is commonly used in packet processing. A Cyclic Redundancy Check (CRC) algorithm is used to reduce the 13-byte 5-tuple to 16 bits. The resultant 16-bit value is referred to as the tuple hash value. The tuple hash value is used to identify the flow. The PIP is responsible for reading the packet header and computing the tuple hash value.

4.1.6 Tag Value, First Tag Value

The *tag value* is a 32-bit number. The first tag value is set by the PIP/IPD when the packet is received. There are three distinct parts of the first tag value:

- Bits 31-24 are always 0x0
- Bits 23-16 are either the number of the port which received the packet, or all 1s (ones).
- Bits 15-0 are the tuple hash value computed by the PIP.

(In the list above, bit 0 is listed on the right by convention. In the *Hardware Reference Manual*, bits 15 to 0 are written as <15:0>.)

In our simplified example, bits 15 to 0 are set to the tuple hash value. The goal is for the first tag value to uniquely identify a flow. If the first tag value is unique for each flow, the OCTEON processor can process the different flows in parallel (one flow does not interfere with the other). All 32 bits of the tag value can be changed by the core to move the packet through different phases of processing, or to specify the desired Output Queue.

4.1.7 Tag Type (TT), First Tag Type

The *tag type* is one of: ORDERED, ATOMIC, or NULL. The first tag type is set by the PIP/IPD when the packet is received. In our example, the first tag type is set to ORDERED, but the first tag type may be set to any tag type. The ORDERED, ATOMIC, and NULL types are discussed after the concept of a tag tuple is introduced.

4.1.8 Tag Tuple

The *tag tuple* is the combination of the tag type and the tag value.

It is very important to know that when software running on the core receives the packet to process, it can then change the tag tuple to different values to move the packet through processing.

Extensive information is provided later in this chapter about changing tag tuples and how different tag types affect SSO scheduling.

4.1.9 ORDERED Tag Type: Parallel Processing

Multiple packets from the same flow with an *ORDERED tag type* may be processed in parallel by multiple cores. Thus, “ORDERED” does *not* mean “only one packet at a time”. Allowing multiple

cores to work on packets from the same flow allows scaling: more cores working on packets in the same flow results in faster packet throughput.

An example of packets where the ORDERED type is sufficient is UDP packets where ordering is required, such as streaming media carried over RTP/SRTP. Another example is TCP/IP packets which are being routed through the processor.

The ORDERED type can also be used during the parts of packet processing which can be done in parallel.

The special ordering done by ORDERED tag types are explained in more detail later in this chapter.

4.1.10 ATOMIC Tag Type: Serialized Processing: Accessing Critical Regions

The *ATOMIC tag type* is used for locking. Only one packet with the same tag tuple can have the ATOMIC lock. ATOMIC tag tuple processing is serialized: one-at-a-time in ingress-order.

An example of packets where ATOMIC processing is needed is TCP/IP packets, especially where this server is the destination. In this case, the tag type is set to ATOMIC on ingress. Since different flows will have different tag tuples (by definition), the different flows are processed in parallel. Within the same flow, ATOMIC packets are processed one-at-a-time. New ATOMIC packets which are waiting for the lock will not be scheduled to cores, so the cores will not be stalled. The packets are only scheduled to cores when the lock is available.

The core can change the tag type of a packet. If the first tag type is ORDERED (for instance, UDP packets where ordering is required), the core can request a change in the tag type from ORDERED to ATOMIC. The SSO will grant the new ATOMIC tag type to requesters one-at-a-time. This provides the core with a hardware ATOMIC lock to protect critical regions. The new ATOMIC tag type is granted in ingress-order, not in request-order or random-order. While waiting for the lock, the packet may remain assigned to the core. The SSO will notify the core when the lock is granted.

Note: If the first tag type is set to ATOMIC, and the core never switches the tag type to ORDERED, the packet throughput is slower than for flows with an ORDERED tag type, since only one packet at a time per flow can be processed. Thus, ATOMIC tag types should only be used when necessary.

The special ordering done by ATOMIC tag types are explained in more detail later in this chapter.

4.1.11 NULL Tag Type: Unordered, Not Serialized, Not Synchronized

NULL tag types are neither ordered nor serialized: multiple cores can process multiple packets, and no packet order is maintained by the SSO.

Examples of packets where ordering may not be important include ICMP packets (such as ping packets), UDP packets where ordering is not required, and non-IP packets.

This chapter does not go into detail about the NULL tag type. The SSO chapter (in volume 2) will cover this topic in detail.

Table 1: ORDERED, ATOMIC, and NULL Tag Types: Example Use

Tag Type	Example Use
ORDERED	Any IP routing: Lookup, Network Address Translation, forward.
	UDP packets where ordering is required, such as streaming media carried over RTP/SRTP.
ATOMIC	TCP/IP packets, especially if this computer is the destination for the packets.
NULL	ICMP packets, for example ping packets.
	UDP packets where ordering is not required.
	Non-IP packets

4.1.12 Quality of Service (QoS) Value

The *Quality of Service (QoS) value* is a number (0-7) which represents the priority of the packet. When packets are received, the PIP/IPD computes the QoS number for the packet, and saves the value in the Work Queue Entry. There is no requirement for 0 or 7 to be the highest priority, and there is no requirement for the priority to be linear. (In general, priority is configurable. There is only one exception to this rule: when using the static priority feature for PKO Output Queues, the lower the queue index, the higher the priority. There is no requirement that the static priority feature be used.)

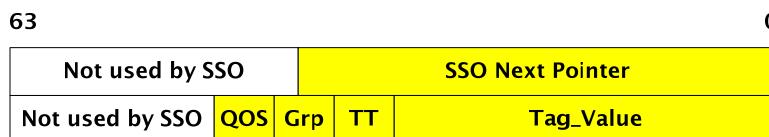
4.1.13 Group (Grp)

The *group (Grp)* field is not used in this simplified example. More detail on this field is provided in the *Software Overview* chapter.

4.1.14 Work Queue Entry (WQE)

The *Work Queue Entry (WQE)* is a data structure which contains the tag type, tag value, QoS value, group, and a pointer to the Packet Data Buffer. The IPD allocates the WQE Buffer from the FPA. The PIP/IPD fill in the WQE fields, then sends the WQE pointer to the SSO, using the `add_work` operation.

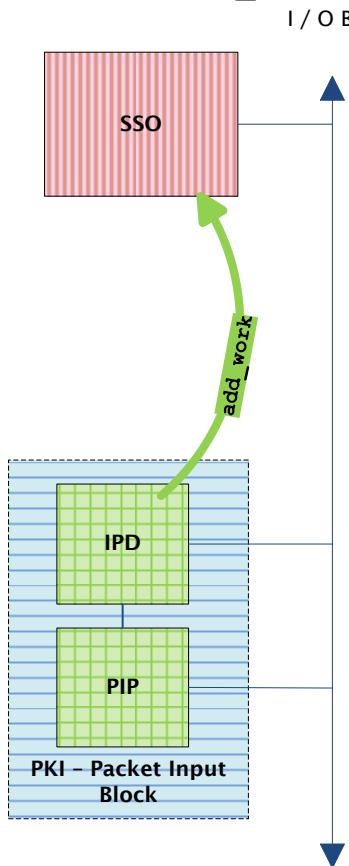
Figure 11: The First Two Words of the Work Queue Entry



4.1.15 The `add_work` Operation

The `add_work` operation is used to send a WQE pointer to the SSO. In this example, the PIP/IPD calls `add_work` to add the WQE to the appropriate SSO's Input Queue.

Figure 12: The `add_work` Operation



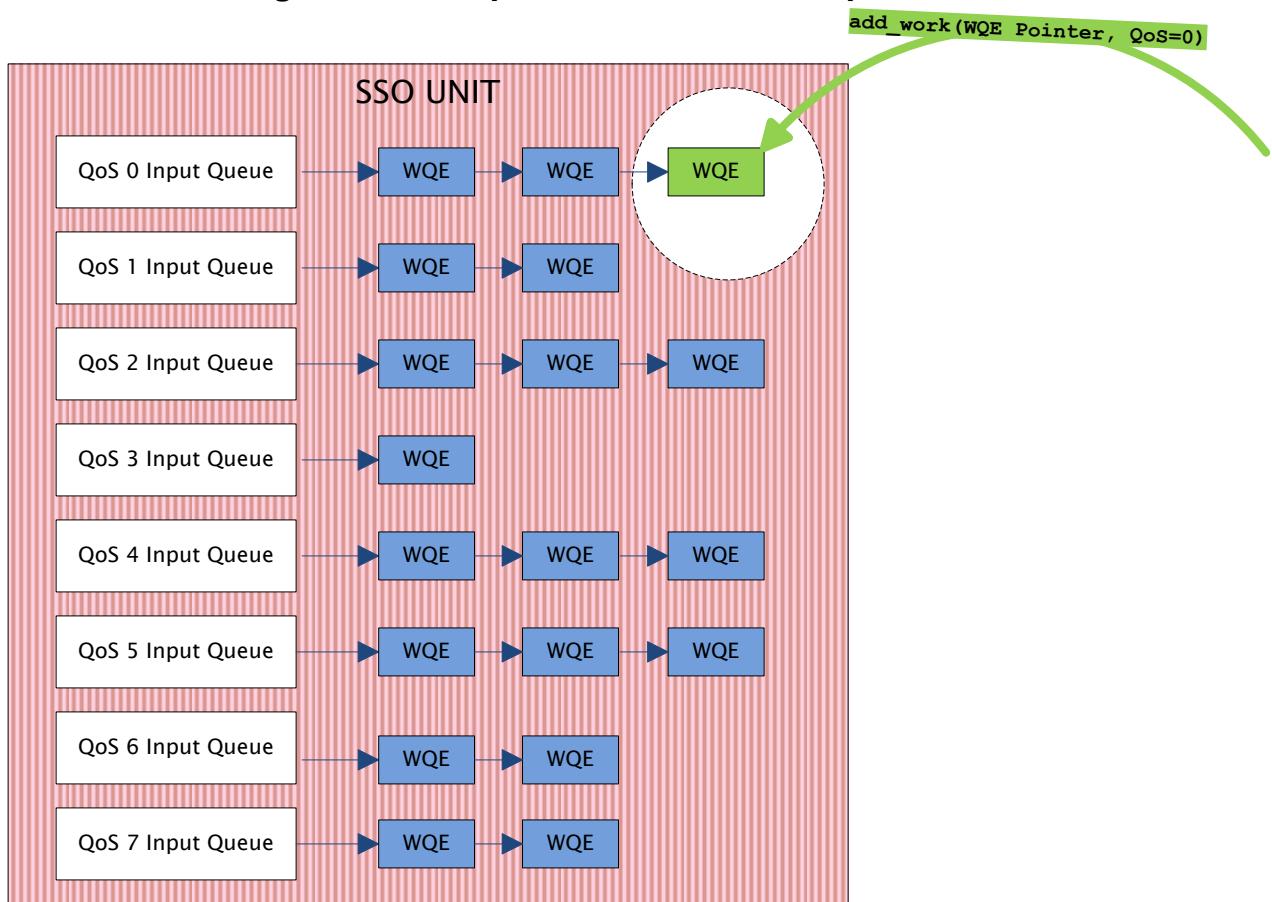
PACKET FLOW

4.1.16 QoS Input Queues

The SSO has 8 *QoS Input Queues* (0-7), one per QoS value. When a new WQE is added to the SSO, the WQE goes onto the Input Queue which matches its QoS value. When the WQEs are added to the SSO's Input Queue, the Next Pointer is used to link them into a list.

Note: This is a simplified view of the SSO's Input Queues! More detail is provided in the SSO chapter in Volume 2.

The next figure shows how the WQE field Next Pointer is used to create a linked list of Work Queue Entries, forming a QoS Input Queue. A SSO QoS Input Queue is also referred to in our manuals as a *work queue*.

Figure 13: Simplified View of SSO Input Queues


4.1.17 Phase 1 Summary:

In Phase 1, the PIP/IPD:

- receives the packet from the packet interface
- allocates a Packet Data Buffer from the FPA
- stores the data in a Packet Data Buffer
- allocates a Work Queue Entry (WQE) from the FPA
- writes tag value, tag type, QoS, group, and Packet Data Buffer pointer to the WQE
- performs an `add_work` operation to add the WQE pointer to the SSO, specifying the target QoS queue

These steps are shown in Figure 1 – “Packet Flow Diagram Part 1: PACKET INPUT”.

4.2 Phase 2: SSO Schedules New Work to the Core

The SSO receives the WQE pointer from the IPD, and adds it to the appropriate QoS queue. The priorities of the QoS queues are configured at system initialization. When the core requests more work, the SSO returns a pointer to the highest priority WQE which is schedulable.

The following key concepts and operations are needed to understand this phase of the packet processing.

4.2.1 SSO Work Descriptors

The SSO contains internal memory. Part of the internal memory has been used to create a limited number of Work Descriptors. Each Work Descriptor contains the key information needed by the SSO to schedule the work on a core, and to keep the packets in the correct order. The key fields in the Work Descriptor are: WQE pointer, tag value, tag type (TT), QoS and group (Grp). (The group field is not discussed in this example.)

Figure 14: Simplified View of the Work Descriptor Data Structure

WQE Pointer		
Tag_Value		
QoS	Grp	TT
Next_Descriptor		
Prev_Descriptor		
Next_TT	Next_Tag	
Switch Pending		

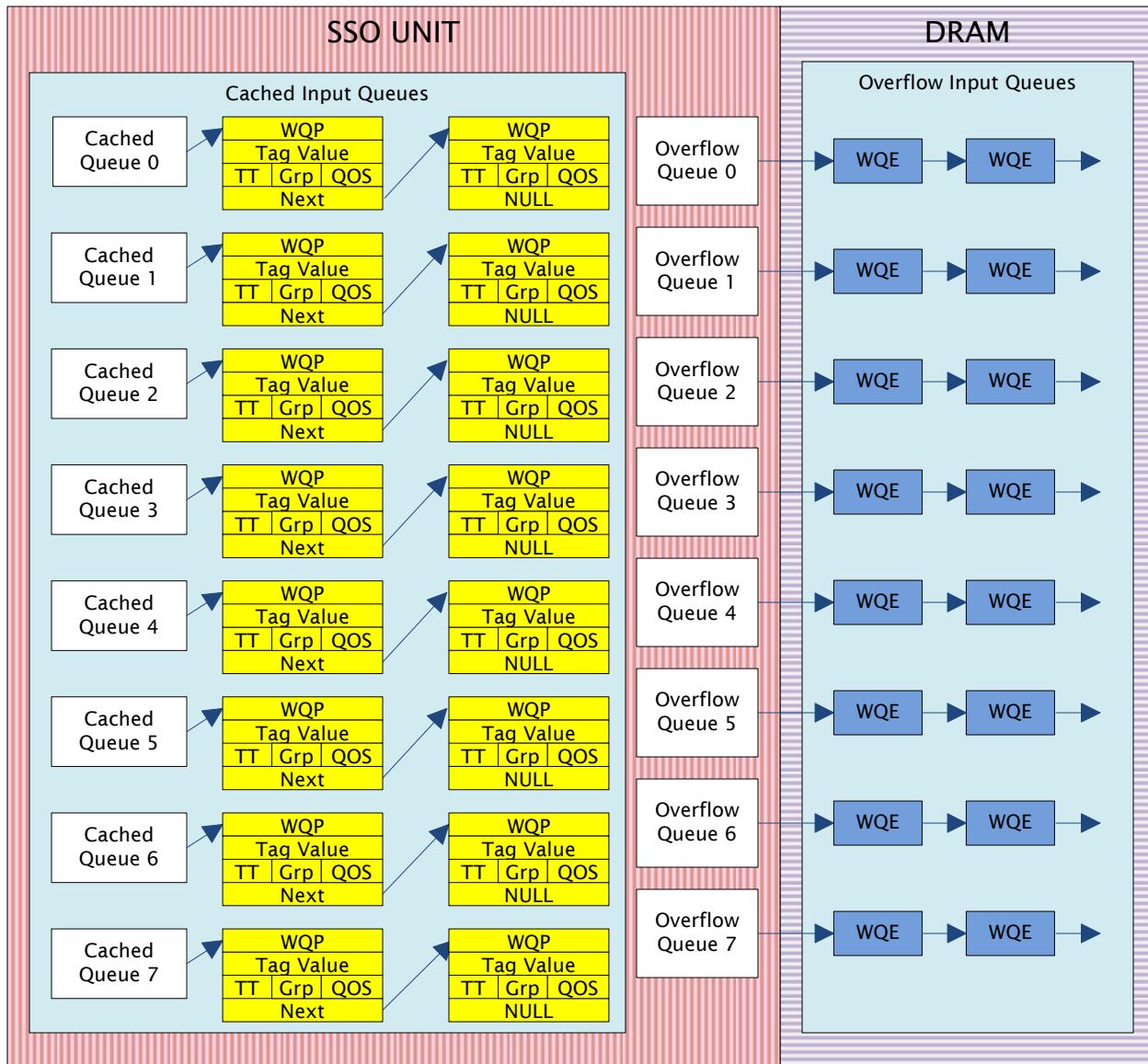
Note that software only reads the fields in this data structure when debugging. This internal information is not usually needed by the programmer.

4.2.2 Cached Input Queues and Overflow Input Queues

The SSO caches the head of each QoS queue in internal memory, one Work Descriptor per WQE. The portion of the QoS queue which is in internal memory is referred to as the *Cached Input Queue*. If there is not enough room in the Cached Input Queues, WQEs are stored in the *Overflow Input Queues*, located in DRAM. The Overflow Input Queues consist of WQEs linked together by their Next Descriptor pointers. Overflow Input Queues only exist if there is not enough room in the Cached Input Queues. The SSO refills the Cached Input Queues from the Overflow Input Queues when space becomes available.

Figure 15: Simplified View of Cached Input Queues and Overflow Input Queues

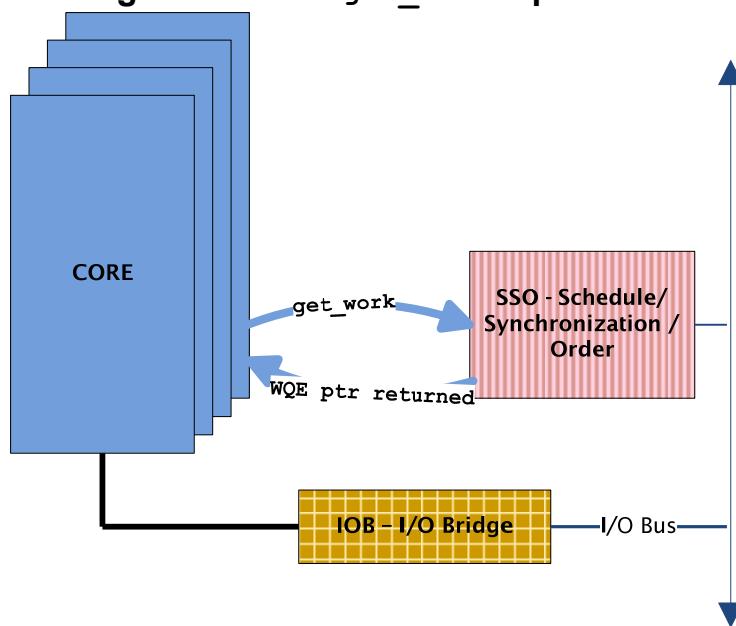
PACKET FLOW



4.2.3 The `get_work` Operation

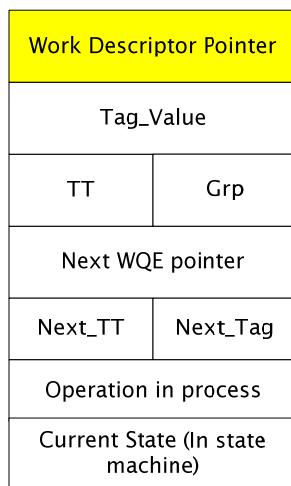
The core performs a `get_work` operation to get another WQE pointer from the SSO. If the operation is successful, the SSO returns a WQE pointer to the core. The SSO's scheduler is responsible for returning the highest priority schedulable WQE to the core.

Note that the Software Development Kit (SDK) provides functions to perform the various operations. For example, the SDK provides the `cvmx_pow_work_request_sync()` function to perform the `get_work` operation. This function converts the physical address returned by the `get_work` operation into a virtual address.

Figure 16: The `get_work` Operation

4.2.4 Core State Descriptor

Inside the SSO, there is one *Core State Descriptor* data structure for each core. When the core performs a successful `get_work` operation, a Work Descriptor is removed from the Cached Input Queue and assigned to the core. A pointer to the assigned Work Descriptor is stored in the Core State Descriptor. The Work Descriptor contains a pointer to the WQE. The `get_work` operation returns the WQE pointer to the core.

Figure 17: Simplified View of the Core State Descriptor Data Structure

Note the internal entry contents presented here are not precise. This information is needed only for debugging, and is provided here to help the reader visualize how the important fields are used.

4.2.5 Scheduled

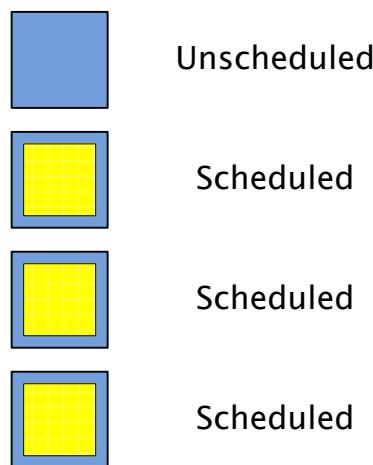
A Work Descriptor that has been assigned to a core is considered to be *scheduled*.

In the following figure, the Core State Descriptor is shown in blue. The Work Descriptor is shown in yellow. When a Work Descriptor has been scheduled to the core, the yellow Work Descriptor is shown as being inside the blue Core State Descriptor. This image will become useful in a later figure.

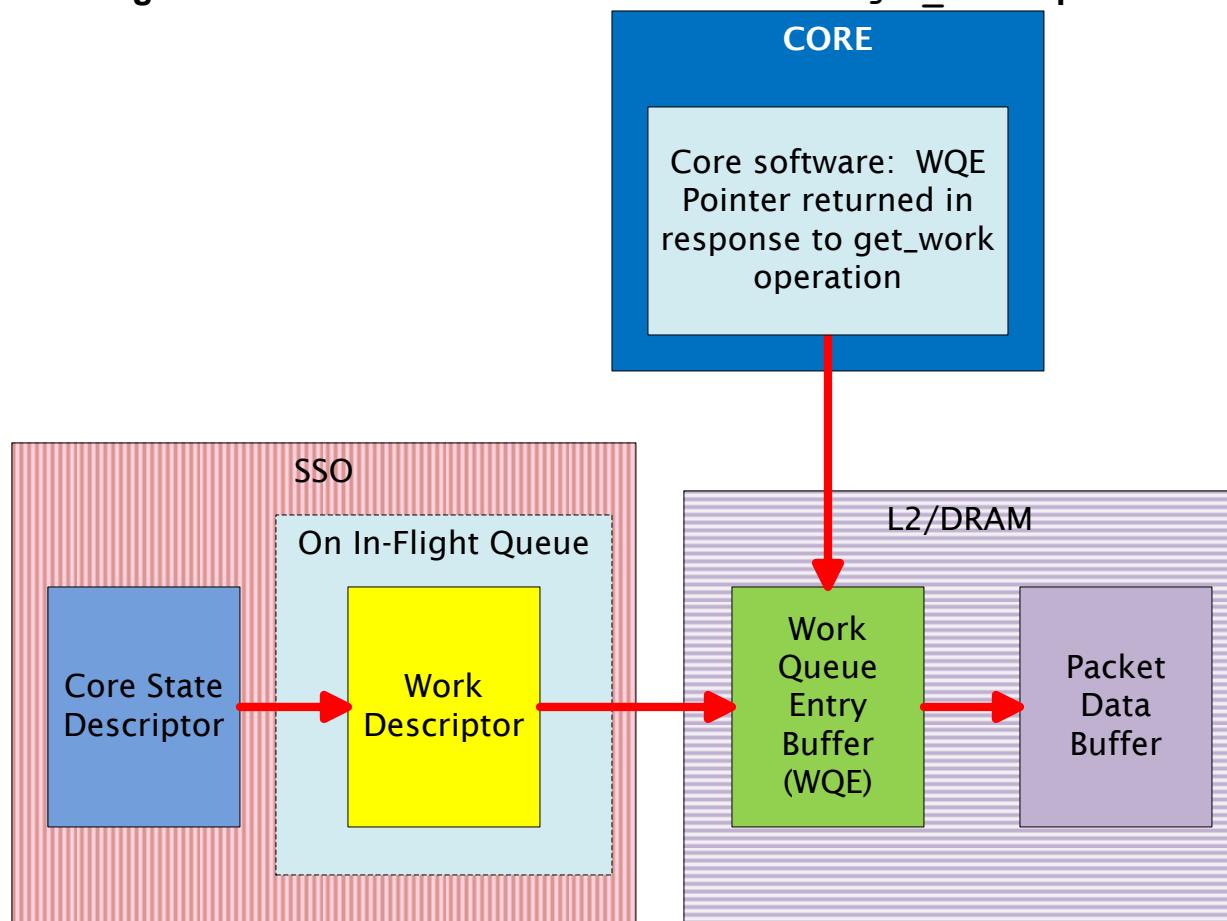
Note: In later figures, the Work Descriptor is shown in different colors; it will not always be shown in yellow.

Figure 18: Core State Descriptors Shown as Scheduled and Unscheduled

Example:
4 cores



PACKET FLOW

Figure 19: Data Structures After a Successful `get_work` Operation

4.2.6 Descheduled

The core may perform an operation to *deschedule* the Work Descriptor, so that the Work Descriptor is no longer assigned to the core. The SSO will reschedule the work descriptor to the same or a different core, using the normal scheduling criteria. A descheduled work descriptor which is runnable has a higher priority than a Work Descriptor which has never been scheduled. Descheduling Work Descriptors can hurt system performance by adding unnecessary processing steps. Descheduling is not shown in this simplified example. For more information on descheduling, see the *Software Overview* chapter and the *SSO* chapter (in volume 2).

4.2.7 In-Flight

Once a Work Descriptor has been scheduled on a core, it is considered to be *in-flight* until it is discarded by a subsequent `get_work` operation or a switch to the NULL tag type. Descheduled Work Descriptors are also considered to be *in-flight* since processing on the associated WQE has started, but has not completed.

4.2.8 Tag Tuple

The tag tuple was introduced in section 4.1.8.

In Phase 1, the tag type (ORDERED, ATOMIC, or NULL) was defined. The first tag value (a 32-bit number) was also defined. The first tag value is set by the PKI to the tuple hash value.

The combination of tag type and tag value is a tag tuple.

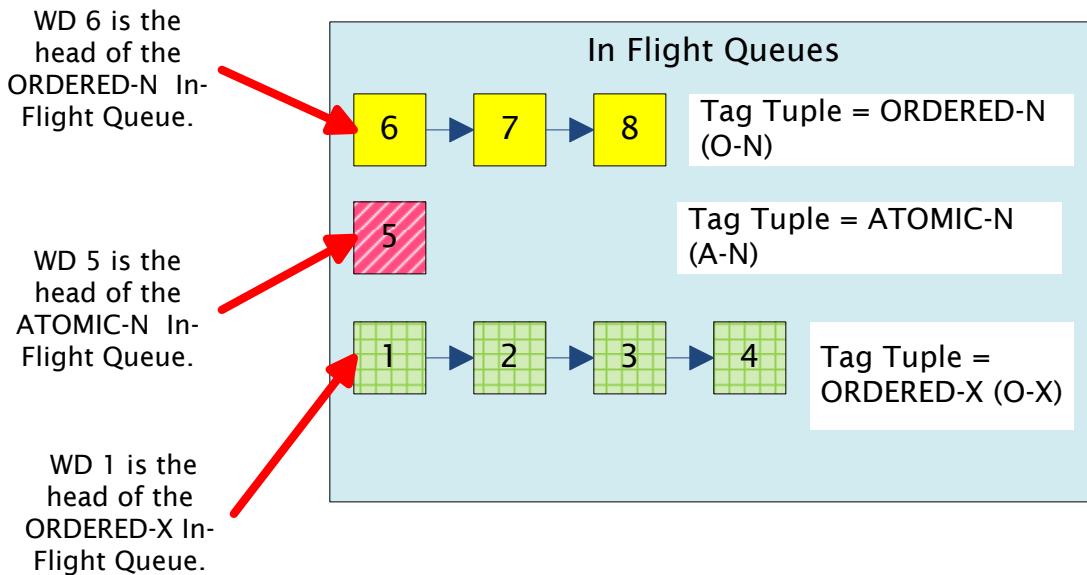
Software can change the tag tuple values to move packets from the same flow through different processing phases, including ATOMIC sections when packet-linked locking is needed.

For instance, if the tag type is “ORDERED” and the tag value is “5”, the tag tuple would be “ORDERED-5”. If the tag type is “ATOMIC” and the tag value is “5”, the tag tuple would be “ATOMIC-5”. Note that “ORDERED-5” and “ATOMIC-5” are different tag tuples, though in our simplified example, the packets in both *tag tuples* belong to the same flow: they have an identical *tag value* of “5”.

4.2.9 In-Flight Queues

The In-Flight Queues are internal to the SSO, and maintained by the SSO. They are essential to maintaining packet order, critical region locks, and packet serialization. (Note: In this discussion, descheduled in-flight Work Descriptors and in-flight descheduled queues are not discussed.)

When a Work Descriptor is scheduled on a core, it is put onto the In-Flight Queue which corresponds to its tag tuple. There is one In-Flight Queue per unique tag tuple. Thus, ORDERED-*N* and ATOMIC-*N* (where *N* is the tag value) are two different In-Flight Queues. Similarly, ORDERED-*N* and ORDERED-*X* are two different In-Flight Queues.

Figure 20: In-Flight Queues

Note: The numbers inside the boxes indicate the ingress order of packets in the flow, so "1" is the first in-flight packet, and "8" is the last in-flight packet from the same flow.

The in-flight Work Descriptors are linked together to create the In-Flight Queue: if there are no in-flight Work Descriptors for a particular tag tuple, there is no In-Flight Queue for that tag tuple.

Packets from the same flow are given the same first tag type and first tag value. As they become scheduled to cores, they are put onto the same In-Flight Queue. As they move through processing, the In-Flight Queue they are on may change. The packets from the same flow are kept together, and in ingress order, as they change In-Flight Queues. The In-Flight Queues correspond to both the unique flows and the flow's packet processing phases.

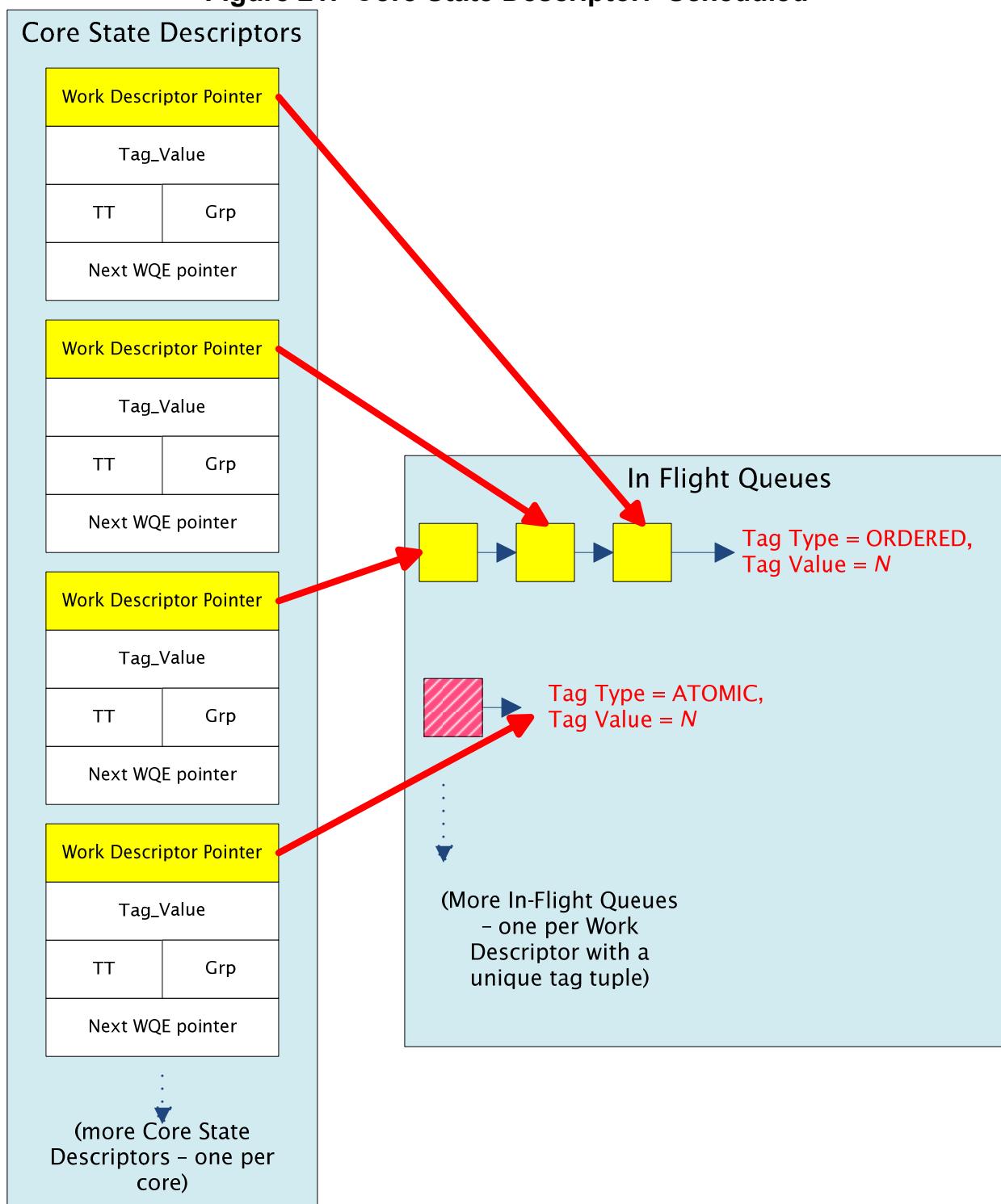
The following figure shows the Core State Descriptors for four cores which have all completed successful `get_work` operations.

The scheduled Work Descriptors are now on the appropriate In-Flight Queue. Note that packets from the same flow are on more than one In-Flight Queue: three packets are on the ORDERED-N queue and one packet is on the ATOMIC-N queue. (In this simplified example, packets are all in the same flow if they have the same tag value.)

Also, note that more than one packet from the same flow is in-flight simultaneously. This parallel processing increases packet throughput.

Figure 21: Core State Descriptor: Scheduled

PACKET FLOW



4.2.10 ORDERED Tag Type: Parallel Processing

The ORDERED tag type was introduced in section 4.1.9.

In this simplified example, the PIP sets the first tag type to ORDERED. Work Queue Entries with an ORDERED tag type may be processed in parallel by multiple cores. Thus, “ORDERED” does *not* mean “only one packet at a time”. Allowing multiple cores to work on packets from the same flow allows scaling: more cores working on packets in the same flow results in faster packet throughput.

The ORDERED tag type means that when a tag switch is requested, the switch will be completed in ingress order. Tag switches are explained in more detail later in this chapter.

4.2.11 ATOMIC Tag Type: Locking Critical Regions

The ATOMIC tag type was introduced in section 4.1.10.

Only one in-flight packet with the same tag tuple can have the ATOMIC tag type at a time. This is how packet-linked locks are implemented. The SSO will grant the lock in ingress order, not the order the request for the ATOMIC tag type was made.

Note: If the first tag type is set to ATOMIC, and the core never switches the tag type to ORDERED, the packet throughput for an individual flow may be slower than for ORDERED flows, since only one packet at a time per flow (identical tag tuple) can be processed. The overall system throughput, however, does not depend on only one flow: if there are 16 ATOMIC flows and 16 cores, 16 packets are processed simultaneously.

4.2.12 NULL Tag Type: Unordered

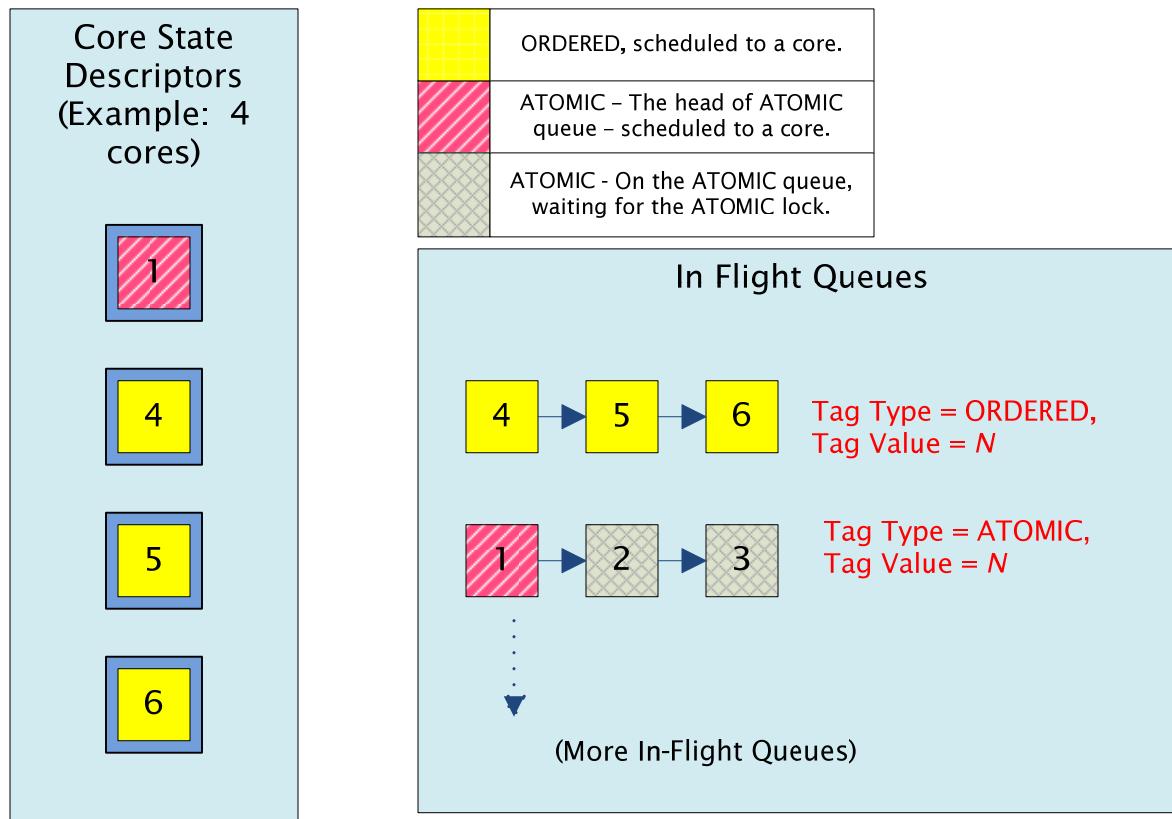
The NULL tag type was introduced in section 4.1.11.

A WQE can have a NULL tag type. If it is assigned a NULL tag type on ingress, then the SSO will assign it on a core, but will not put it in an In-Flight Queue. The NULL tag type means that the SSO will not keep the packets with the same tag tuple in ingress order. The core may change the tag type from ORDERED or ATOMIC to NULL.

Examples of packets where ordering may not be important include ping packets and non-IP packets. The NULL tag type is not used in this example. Switches involving NULL tag types are discussed in the *SSO* chapter (in Volume 2).

Figure 22: ORDERED Tag Types Execute in Parallel, ATOMIC Tag Types Wait for the Lock

PACKET FLOW



Note: The numbers inside the boxes indicate the ingress order of packets in the flow, so "1" is the first in-flight packet, and "6" is the last in-flight packet from the same flow. Note that multiple packets from the ORDERED in-flight queue are being processed by the cores simultaneously. Only one ATOMIC packet from the ATOMIC in-flight queue is being processed, the other packets have to wait.

4.2.13 Choosing the Next WD to Schedule; Skipping Un-schedulable WD

The SSO's scheduler is responsible for selecting the next highest priority Work Descriptor which can run on a core, and scheduling that Work Descriptor on a core which has performed the `get_work` operation.

The QoS priorities, combined with other tunable scheduling parameters, work together to set the priority of a piece of the Work Descriptor.

Regardless of priority, Work Descriptors in the Input Queue which have an ATOMIC tag type may not be schedulable (are un-schedulable). If earlier packets with the same tag tuple have or want the ATOMIC lock, the newer packet would have to wait. If this newer packet was assigned to a core,

the core would be idle waiting for the lock. It is better for the core to stay busy, so the new packet is un-schedulable at this time.

The un-schedulable Work Descriptors are skipped when the SSO is looking for new Work Descriptors to assign to the cores. Note that this is a simplified view of the SSO Scheduler. More details are provided in the SSO chapter (in Volume 2).

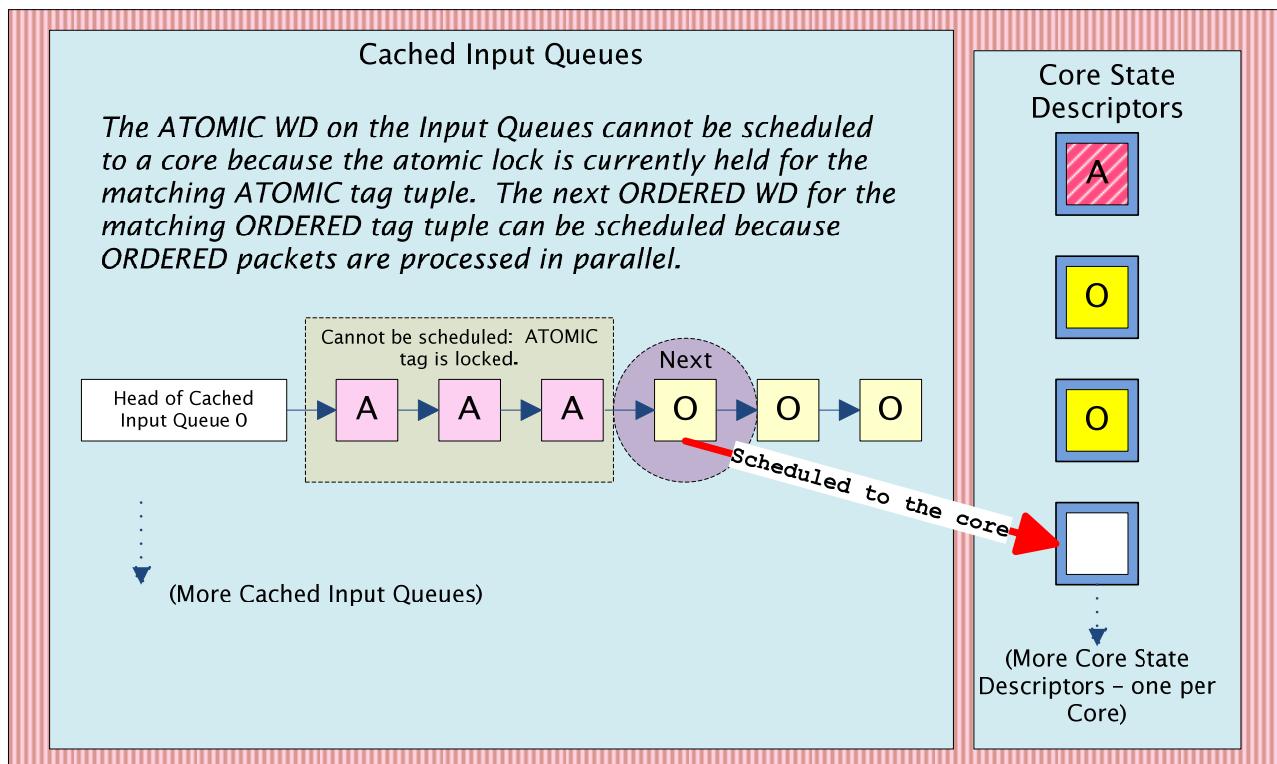
Figure 23: Un-Schedulable Work Descriptors: ORDERED versus ATOMIC Tag Types

SIMPLIFIED EXAMPLE: ONLY ONE INPUT QUEUE

KEY			
O	Unscheduled ORDERED WD	O	Scheduled ORDERED WD
A	Unscheduled ATOMIC WD	A	Scheduled ATOMIC WD

The ORDERED tag type can be scheduled to a core in parallel, is never un-schedulable.

The ATOMIC tag type: Cannot be scheduled to a core if the ATOMIC tag for its tag tuple is already “locked” (a prior packet with the same tag tuple has the lock or is waiting for the lock).



4.2.14 Phase 2 Summary

The SSO receives the WQE pointer from the IPD, and adds it to the appropriate QoS queue. The priorities of the QoS queues are configured at system initialization.

The SSO has internal memory which is used to create Work Descriptors and Core State Descriptors. The head of each Input Queue is cached in internal memory. The Cached Input Queue is a linked list of Work Descriptors. If there are not enough Work Descriptors to contain the entire Input Queue in internal memory, the queue will continue in DRAM as a linked list of WQEs: the Overflow Input Queue.

When a core performs a successful `get_work` operation, the SSO scheduling function takes the highest priority schedulable Work Descriptor off of the highest priority Cached Input Queue and schedules it to the core. A pointer to the scheduled Work Descriptor is stored in the Core State Descriptor. The Work Descriptor is put on the appropriate In-Flight Queue. The core receives a pointer to the WQE.

Work Descriptors which have an ORDERED tag type may be processed in parallel by multiple cores. Work Descriptors which have an ATOMIC tag type are processed one packet at a time, in ingress order.

These steps are shown in Figure 2 – “Packet Flow Diagram Part 2: SSO AND CORE PROCESSING” the step marked “SSO” in the text description, and steps 6a and 6b.

4.3 Phase 3: Lock Critical Region: One-at-a-time Access

Critical regions, such as code which modifies shared data structures, may be protected by using packet-linked locking, which is implemented by the ATOMIC tag type. When a core needs to access a critical region, it changes the Work Descriptor’s tag type from ORDERED to ATOMIC.

The following key concepts and operations are needed to understand this part of the packet flow.

4.3.1 The `switch_tag` Operation (Tag Switch)

A core may perform a `switch_tag` operation to change the Work Descriptor’s tag type, tag value, or both. Both may be changed with the same operation. This operation is referred to as a *tag switch*.

The core will perform a `switch_tag` operation from ORDERED to ATOMIC to request a lock (for instance to lock a critical region). To unlock, the core will perform a `switch_tag` operation to a different tag tuple.

The core may also perform a `switch_tag` operation from ORDERED-N to ORDERED-X, and other combinations.

4.3.2 Switch Tag Sequence

All the packets from the same flow need to switch tags tuples in the same sequence. For instance, all packets in the same flow need to switch from ORDERED-5 to ATOMIC-5 to ORDERED-6. If some packets do not switch tags in the same sequence, the packets may become out of order. If two steps in packet processing have the same tag tuple, then the packets will become mixed up: packet processing will not follow distinct processing steps.

Figure 29 – “Packet Processing Phases and Sequential Tag Switch Operations” illustrates the sequence: one flow moving in order through different processing phases.

4.3.3 Core's Switch Complete Bit

When the core performs the `switch_tag` operation, it requests the SSO change the Work Descriptor's tag tuple to the new tag tuple. The tag switch does not necessarily complete immediately. For both the ORDERED and ATOMIC tag types, the packet must become the head of the initial In-Flight Queue before it can switch. Additionally, when switching to the ATOMIC tag type, the packet may need to wait for the ATOMIC lock. The switch requirements for ORDERED and ATOMIC are detailed in the following sections.

The SSO will notify the core when the tag switch is complete.

The Cavium Networks-specific hardware register, 30, contains the status of the `switch_tag` request. The RDHWR instruction is used to read this register. Note this register is not the same as general purpose register 30.

When the core requests a tag switch, the core's Switch Complete Bit is set to zero. When the tag switch is complete, the SSO sets the Switch Complete Bit to one.

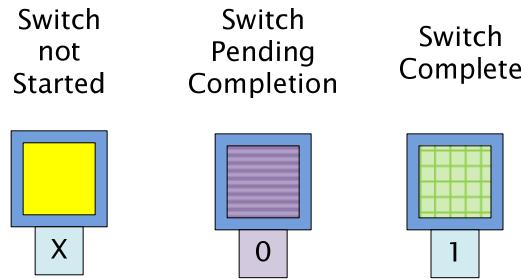
For example if the core performs a `switch_tag` operation to an ATOMIC tag type, it must wait for notification that the switch has completed. When the tag switch has completed, the core has been granted the ATOMIC lock and may safely access the critical region.

A core can have only one outstanding switch, so only one Switch Complete Bit is required per core.

The following figure illustrates the Core State Descriptor and the status of the Switch Complete Bit. In addition to showing the value of the Switch Complete Bit, the color of the scheduled Work Descriptor has been changed to alert the reader that the switch is incomplete. This illustration is used in figures later in this chapter. The specific meaning of the Work Descriptor's color is explained in the key included with each figure.

In the figure below:

- “X” means that the value is not important: the switch has not yet begun
- “0” means “Switch Pending Completion”, and
- “1” means “Switch Complete”.

Figure 24: The Core's Switch Complete Bit


4.3.4 Initial In-Flight Queue

The *initial In-Flight Queue* is the In-Flight Queue the Work Descriptor is on when the tag switch request begins.

4.3.5 Target In-Flight Queue

The *target In-Flight Queue* is the In-Flight Queue the Work Descriptor is on when the tag switch request completes. This queue corresponds to the requested tag tuple.

4.3.6 Tag Switch Processing

The tag switch operation has several steps. Each is handled by the SSO. Each step must complete before the switch is complete.

4.3.6.1 Tag Switch Processing Steps

Between the core's *request* to switch tags and the switch *completion*, there are discrete steps performed by the SSO. Understanding these steps will provide insight into appropriate use of tag switches, and the performance benefits and costs.

The tag switch Processing steps are:

1. The core requests a tag switch; the core's Switch Complete Bit is set to 0 (zero).
2. The Work Descriptor waits to become the *head* of the initial In-Flight Queue.
3. The Work Descriptor moves to the *end* of the target In-Flight Queue.
4. The Work Descriptor waits to be granted the target tag type:
 - A) If the target tag type is ATOMIC: the Work Descriptor waits to become the *head* of the target In-Flight Queue.
 - B) If the target tag type is ORDERED: the switch completes without additional delay. ORDERED tag types are processed in parallel.
5. After any waits are complete, the tag switch completes. The SSO sets switch complete to 1 (one).

In the following example, the core performs a `switch_tag` operation on the Work Descriptor numbered “4” from ORDERED to ATOMIC, without changing the tag value. The numbers inside the boxes correspond to packet ingress order. Packets 1-4 are all in the same flow. The “N” is a numeric tag value.

The steps are:

Initial State: The Work Descriptor shown with the number “4” has been scheduled to the core, and is on the In-Flight Queue corresponding to its tag tuple.

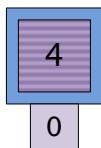
In this example, Work Descriptor “4” is on the ORDERED-N (O-N) In-Flight Queue.



Step 1: The core requests a tag switch; the core’s switch complete is set to 0:

The core performs a `switch_tag` operation. The core’s Switch Complete Bit is set to 0. Note that the Work Descriptor remains attached to the core during the switch.

Switch
Pending
Completion



Step 2: The Work Descriptor waits to become the head of the initial In-Flight Queue:

The Work Descriptor requesting the tag switch must wait until it becomes the head of the initial In-Flight Queue. Once the Work Descriptor is the head of the initial In-Flight Queue, it can move to the target In-Flight Queue.

In this example, Work Descriptor 4 must wait until it becomes the head of the queue before moving to the target In-Flight Queue. This process keeps packets in ingress order, and is part of the Ordering function of the SSO.



Step 3: The Work Descriptor moves to the end of the target In-Flight Queue:

Once the Work Descriptor is the head of the initial In-Flight Queue, the SSO moves it to the end of the target In-Flight Queue.

In this example, the target In-Flight Queue tag type is ATOMIC, the tag value is N . This is shown as “A-N”. The Work Descriptor labeled “1” is the head of the “A-N” In-Flight Queue. This Work Descriptor currently holds the ATOMIC lock. Work Descriptors 2, 3, and 4 have each performed a `switch_tag` operation to the new tag tuple “A-N”, and have, in turn, become the head of the initial In-Flight Queue. They have now successfully moved to the target In-Flight Queue.



Step 4: The Work Descriptor waits to be granted the target tag type:

- A) If the target tag type is ATOMIC: the Work Descriptor waits to become the head of the target In-Flight Queue.**

If the target tag type is ATOMIC, then the Work Descriptor must wait until it is the head of the target In-Flight Queue. This enforces the rule that each packet accesses the lock in ingress order. When the Work Descriptor becomes the head of the target In-Flight Queue, then the SSO grants it the ATOMIC lock and the tag switch is complete.



- B) If the target tag type is ORDERED: the switch completes without additional delay. ORDERED tag types are processed in parallel.**

If the target tag type is ORDERED the tag switch may complete immediately. In this example, the tag value for the packet four Work Descriptor has been switched from N to X, where N and X are both numeric tag values. There is no need to wait to get a lock. ORDERED packets are processed in parallel.

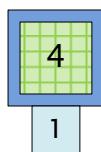
In this example, the target In-Flight Queue is ORDERED-X (O-X).



Step 5: Switch complete; core's Switch Complete Bit is set to 1

When the tag switch is complete, the core's Switch Complete Bit is set to 1 (one). The core may now access the critical region: it has the lock.

Switch
Complete



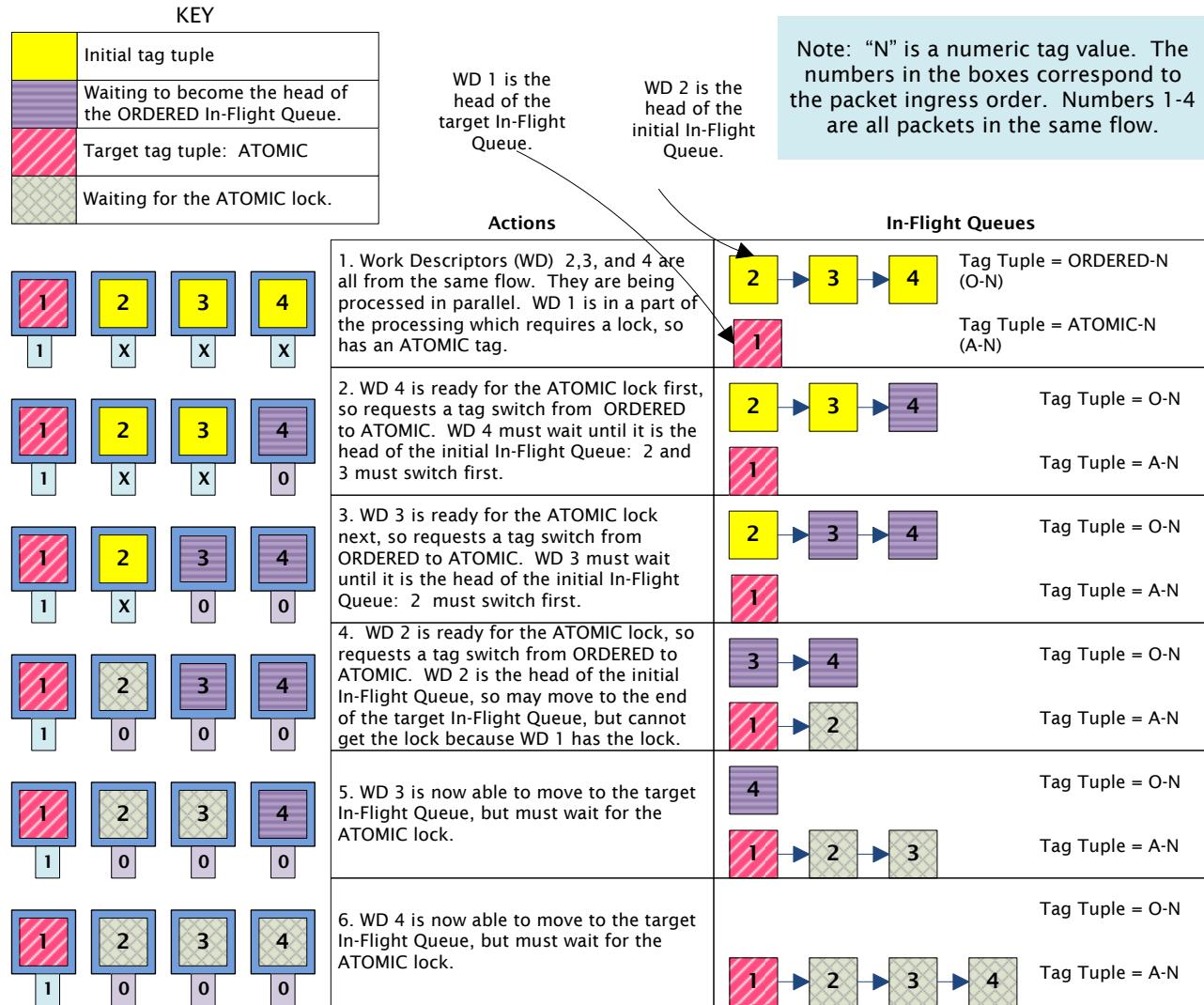
4.3.6.2 Tag Switch from ORDERED to ATOMIC

This operation is used to acquire an ATOMIC lock.

Note that a critical region is usually brief. The amount of time the core will have to wait for the lock depends on the length of the critical region and the number of waiters. If the wait is 30 cycles or less, the core can wait for the lock to be granted; there is no performance improvement from descheduling the work.

The figure below illustrates a tag switch from ORDERED to ATOMIC.

Figure 25: Tag Switch from ORDERED to ATOMIC



Note that the figure above does not show the lock being granted. Once the Work Descriptor is on the target In-Flight Queue, it waits for the ATOMIC lock to be released. The Work Descriptors are each granted the lock in turn. When the lock is granted, the SSO will set the associated core's Switch Complete Bit to "1".

4.3.6.3 Tag Switch from ATOMIC to ORDERED

This operation is used to release an ATOMIC lock. In this case, since only one Work Descriptor can hold the ATOMIC lock at a time, that descriptor is by definition the head of the initial In-Flight Queue. When the core performs the `switch_tag` operation, the Work Descriptor will move immediately to the target In-Flight Queue. Because the target In-Flight Queue is ORDERED, the Work Descriptors may be processed in parallel. Thus, the tag switch completes without requiring the Work Descriptor to be the head of the target In-Flight Queue. The SSO will set the switch complete bit when the Work Descriptor has completed the tag switch by moving to the target In-Flight Queue.

Note: If the switch is made from ORDERED to ATOMIC to ORDERED without changing the tag value, then the packet processing will become mixed up. It is essential that the new tag tuple be different from the original tag tuple when unlocking the critical region.

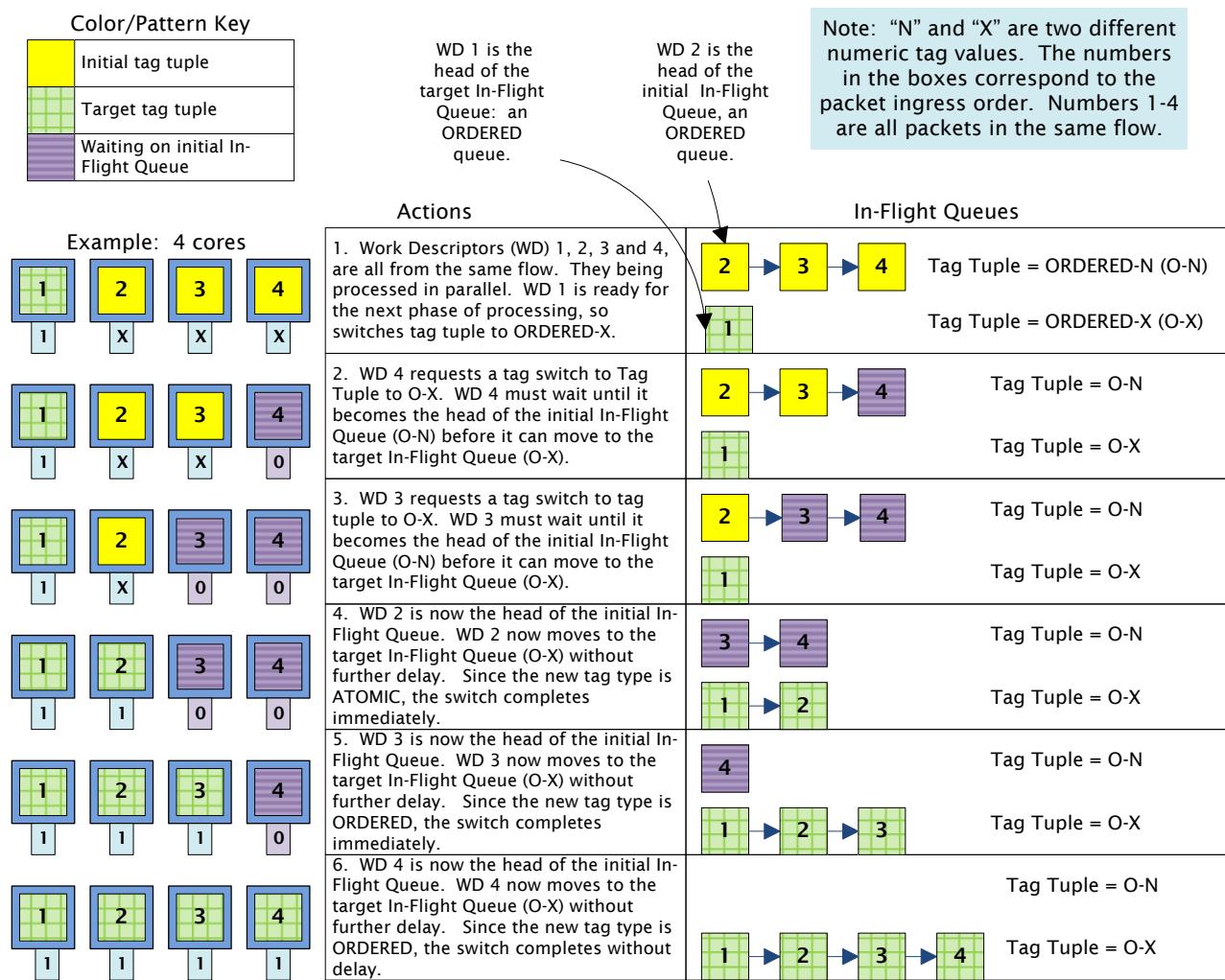
4.3.6.4 Tag Switch from ORDERED to ORDERED

This type of tag switch is rarely needed. It would be used, for example, to pipelining packet processing. Pipelining packet processing is not the highest-performance software design, and is not required on OCTEON processors. Tag switches are not free: they cost processing cycles, which can hurt system performance. In addition to the processing cycles, a delay in processing is introduced when the initial In-Flight Queue is ORDERED because all the *prior* packets must switch off the initial In-Flight Queue in order. Unless there is a *requirement* that all packets complete processing Phase 2 in order before moving to processing Phase 3, do not use the ORDERED to ORDERED tag switch.

The tag switch from ORDERED to ORDERED will *not* wait for a lock on the target In-Flight Queue, but *will* require that the Work Descriptor be the head of the initial In-Flight Queue before the switch may complete. Because of this requirement, the packets will move from one queue to the next in ingress order, which is part of the Ordering function of the SSO.

The figure below illustrates a tag switch from ORDERED-N to ORDERED-X.

Figure 26: Tag Switch from ORDERED to ORDERED



PACKET FLOW

4.3.7 Phase 3 Summary

In order to protect a critical region, the core performs a `switch_tag` operation from ORDERED to ATOMIC. The Work Descriptor moves from the initial In-Flight Queue to the target In-Flight Queue.

Other tag switch types were also discussed in this section, along with the rules for tag switches:

- When the *initial* In-Flight Queue is ORDERED, the Work Descriptor must become the head of the queue before it can move.
- When the *initial* In-Flight Queue is ATOMIC, the work descriptor is already the head.
- The Work Descriptor is moved from *head* of the initial In-Flight Queue to the *tail* of the target In-Flight Queue.
- When the *target* In-Flight Queue is ORDERED, the switch completes immediately.

- When the *target* In-Flight Queue is ATOMIC, the Work Descriptor must become the head of the queue before the switch can complete.

When the tag switch operation completes, the SSO sets the core's Switch Complete Bit. The core then has the ATOMIC lock and may access the critical region.

This phase is not shown or described in Section 2 – “Packet Flow Overvie” because that packet flow description is simplified.

4.4 Phase 4: Unlock Critical Region and Resume Parallel Processing

To resume parallel processing, the core will perform a `switch_tag` operation from ATOMIC

back to ORDERED. The tag value must be different than the original ORDERED tag value.

Remember that all packets in the flow must follow the same tag switch sequence or they may not stay in the correct order.

The information needed to understand this phase was covered in the Phase 3 section.

This phase is not shown or described in Section 2 – “Packet Flow Overvie” because that packet flow description is simplified.

4.5 Phase 5: Packet Output

When the core is ready to transmit the packet, it sends packet transmission commands to one of the prioritized PKO Output Queues.

The core uses the SSO's synchronization and ordering features to select the PKO Output Queue, lock it, and to guarantee packet transmission commands are sent to the PKO Output Queues in ingress order. (Note: sending packets to the PKO Output Queues in order is an *option*, not a *requirement*. In this example, packets from the same flow are transmitted in ingress order.)

The PKO will remove the commands from the tail of the PKO Output Queue, read the packet data from L2/DRAM, and DMA the data to the selected output port. The PKO can optionally notify the core when the packet is transmitted.

The following information is needed to understand this packet processing phase. More details on the PKO, including information on its configurable scheduling algorithm, is provided in the *PKO* chapter (in Volume 2).

4.5.1 PKO Output Ports

The PKO supports up to 40 *PKO Output Ports*, depending on the OCTEON model. Different ports correspond to the different hardware interfaces.

4.5.2 PKO Output Queues

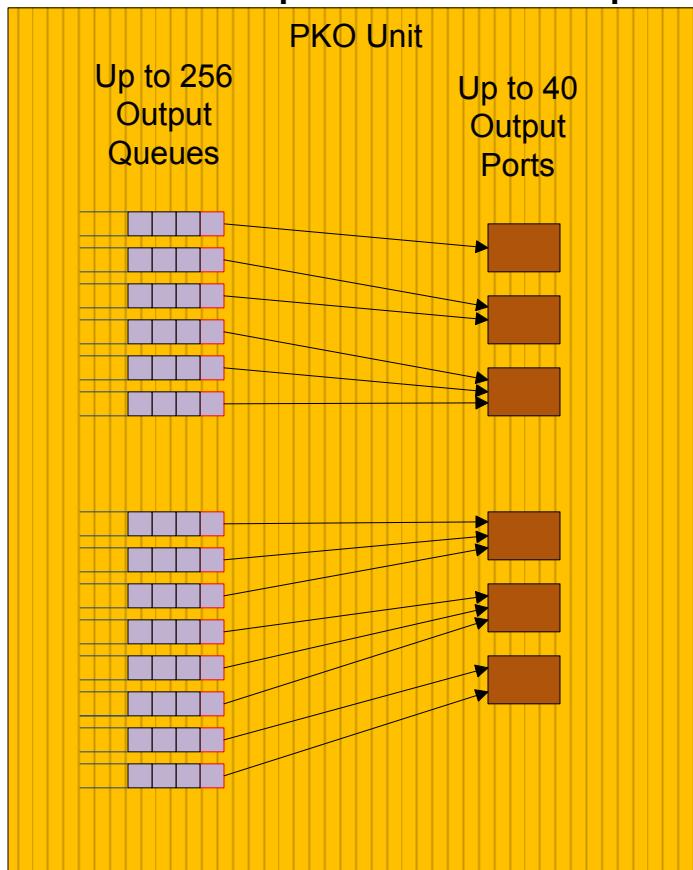
The PKO has up to 256 *PKO Output Queues*, depending on the OCTEON model. The Output Queues are mapped to the Output Ports. The Output Queues can have different priorities, which

are configured at system initialization time. To insure that all packets from the same flow are transmitted in ingress order, send them all to the same Output Queue.

4.5.3 PKO Output Queue to Port Mapping

Since there are more Output Queues than Output Ports, more than one Output Queue may be mapped to the same Output Port. The queue-to-port mapping is configured at system initialization time.

Figure 27: PKO Output Queues and Output Ports



4.5.4 Selecting the PKO Output Queue

Each Output Queue is identified by a unique tag tuple. The tag value which had identified the flow is now overwritten by a tag value which identifies the selected Output Queue.

4.5.5 Freeing the WQE Buffer

After creating the packet transmission commands, the core may free the WQE Buffer back to the FPA: it is no longer needed. This may be done before the commands are sent to the Output Queue.

4.5.6 Locking the PKO Output Queue

The core will perform a `switch_tag` operation to the tag value corresponding to the selected Output Queue, with tag type set to ATOMIC. This will lock the selected PKO Output Queue. Each Output Queue is identified by a unique tag value, thus each Output Queue will have its own lock.

4.5.7 Transmitting Packets in Ingress Order

Packet order is preserved through the packet processing if:

1. only ORDERED or ATOMIC tag types are used during processing (not NULL)
2. all packets in the same flow follow the same tag switch sequence
3. ATOMIC locking is used to guarantee that packets from the same flow are sent to the Output Queues one-at-a-time
4. all packets from the same flow are sent to the same PKO Output Queue

When the final `switch_tag` operation to the ATOMIC tag type is performed, the Work Descriptors are granted the lock in ingress order, thus the packet transmission commands are sent to the PKO in ingress order.

Note: All packets from the same flow must send their packet transmission commands to the same Output Queue to guarantee transmission in ingress order. In cases where ingress order is not important, the packet transmission commands may be sent to multiple Output Queues. In this case, the destination is responsible for re-assembling the packets into the correct order.

4.5.8 Writing to the Output Queue, then Freeing the Lock

Once the core has been granted the lock, it writes the commands to the Output Queue.

4.5.9 Freeing the Work Descriptor and Releasing the Lock

After the transmission commands have been written to the Output Queue, the core may free the Work Descriptor and release the packet-linked lock.

The `get_work` operation will:

- 1) free the Work Descriptor
- 2) release the packet-linked lock
- 3) return a new Work Queue Entry pointer to the core

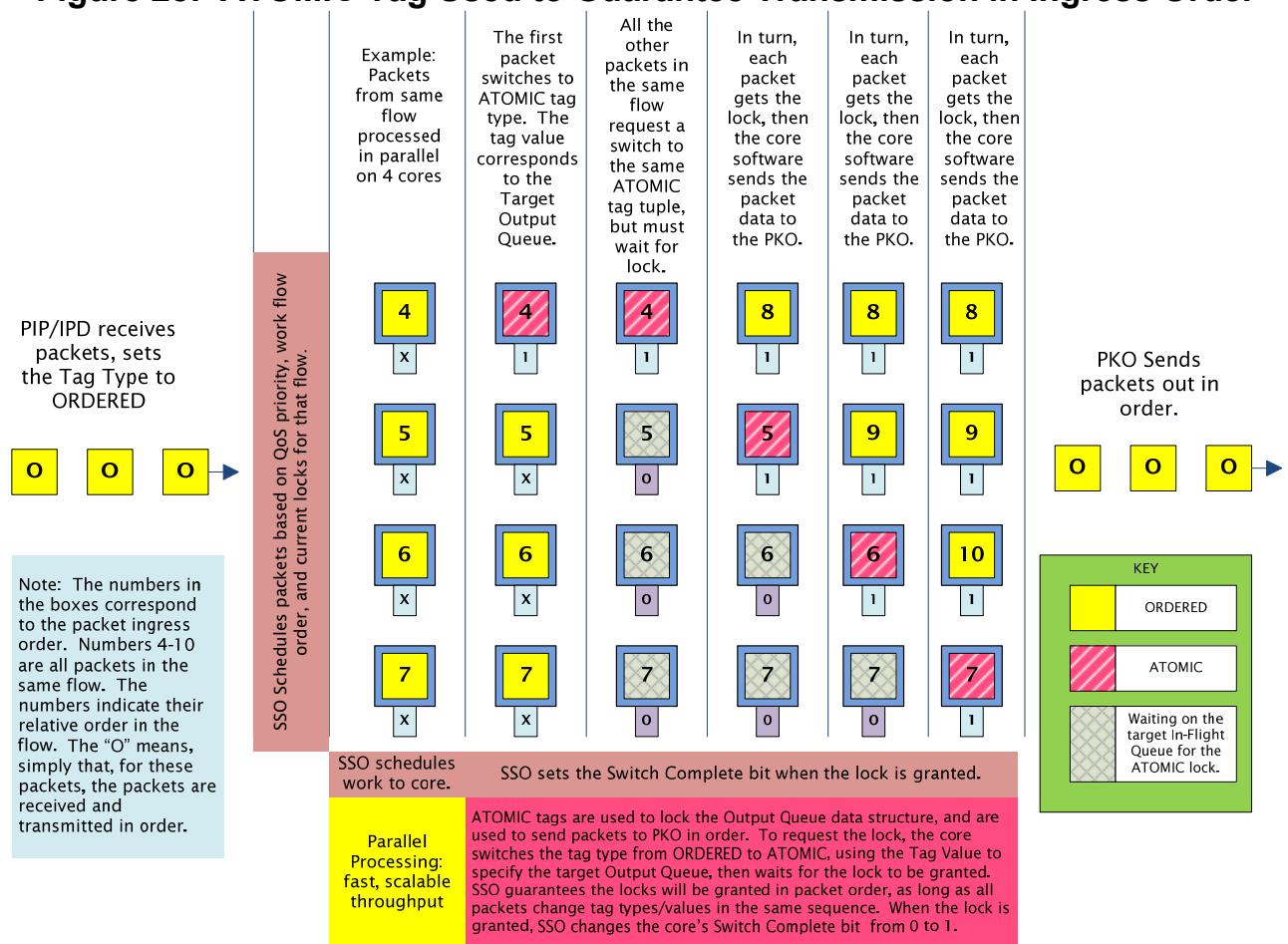
A `switch_tag` operation to the NULL tag type will also free the Work Descriptor and release the packet-linked lock, but will not provide a new WQE pointer.

4.5.10 PKO DMAs the packet to the TX Port

The PKO scheduler will remove the packet transmission commands from the tail of the Output Queue and DMA the packet data to the selected TX Port.

4.5.11 Freeing the Packet Data Buffer

The PKO will optionally free the Packet Data Buffer back to the FPA.

Figure 28: ATOMIC Tag Used to Guarantee Transmission in Ingress Order

4.5.12 Phase 5 Summary

After processing the packet data, the core switches the tag type and tag value. The tag type is set to ATOMIC; the target tag value corresponds to the selected Output Queue. This operation will lock the selected Output Queue. When the core has the lock, it sends the packet to the selected PKO Output Queue, and then releases the lock. The core frees the WQE and Work Descriptor. The PKO may optionally free the Packet Data Buffer after it reads the packet data into its internal memory.

If packets must be transmitted in ingress order, they must only be sent to one Output Queue. If order is not important, they may be sent to multiple Output Queues.

This phase is described in Figure 2 – “Packet Flow Diagram Part 2: SSO AND CORE PROCESSING” and Figure 3 – “Packet Flow Diagram Part 3: PACKET OUTPUT”.

4.6 Workflow Model: One Flow

Given these tools, it is now possible to control the packet workflow from Receive to Transmit.

In the typical workflow, tag tuples can be used to:

- Separate different flows.
- Keep packets in ingress order.
- Allow packets to be processed in parallel whenever possible.
- Protect critical regions with ATOMIC locks, while maintaining ingress order.
- Select the PKO Output Queue, lock the queue, and send packets to the PKO in ingress order.

4.7 Workflow Model: Multiple Flows

In this simplified example, only one flow was examined. In the ideal system, each flow has a unique tag value. Thus, multiple flows are processed in parallel, which is a very high-performance model. Flows may use independent packet-linked locks (for instance, there is one TCP/IP control block per flow, each protected by a different lock), or flows may share locks with other flows (for instance, use a common lock to access a shared PKO Output Queue).

4.8 Summary

The SSO uses tag values and tag types to manage multiple data flows while maintaining packet ingress order.

Each scheduled WQE has a corresponding Work Descriptor on the SSO's In-Flight Queue corresponding to the tag tuple. When the core performs a `switch_tag` operation, the Work Descriptor moves in ingress order from the initial In-Flight Queue to the target In-Flight Queue.

By using the ORDERED type, packets may be processed in parallel for high throughput. By using the ATOMIC type, critical regions are protected by packet-linked locks.

It is critical that all packets in the same flow go through the same tag switch sequence, or the ingress order will not be maintained.

Tag values and the ATOMIC tag type are used select and lock the PKO Output Queue, and to ensure the packets in a flow are sent to it in ingress order.

In the simplified example presented in this chapter, the packet processing was divided into five phases:

Phase 1: Packet Input

The packets in the flow are received in order (ingress order). The PIP/IPD performs header checks and flow classification. The IPD sets the first tag type and first tag value. In this example, the first tag type is set to ORDERED. The IPD allocates the needed buffers from the FPA. The PIP/IPD DMAs the packet data to the Packet Data Buffer in L2/DRAM. The IPD performs an `add_work` operation to add the WQE pointer to the SSO's Input Queue, based on the QoS value for the Work.

Phase 2: SSO Schedules New Work to the Core

Cores perform a `get_work` operation to get a Work Queue Entry (WQE) pointer. The WQE contains a pointer to the packet data. The core works on the packet data until it needs to lock a critical region, or transmit the packet. Because their first tag type is ORDERED, the packets can be processed in parallel by multiple cores.

Phase 3: Lock Critical Region: one-at-a-time access

When the core needs to lock a critical region, it performs a `switch_tag` operation, changing the tag type to ATOMIC. When the tag switch operation completes, the SSO sets the core's Switch Complete Bit. The core then has the ATOMIC lock and may access the critical region.

Phase 4: Unlock Critical Region and Resume Parallel Processing

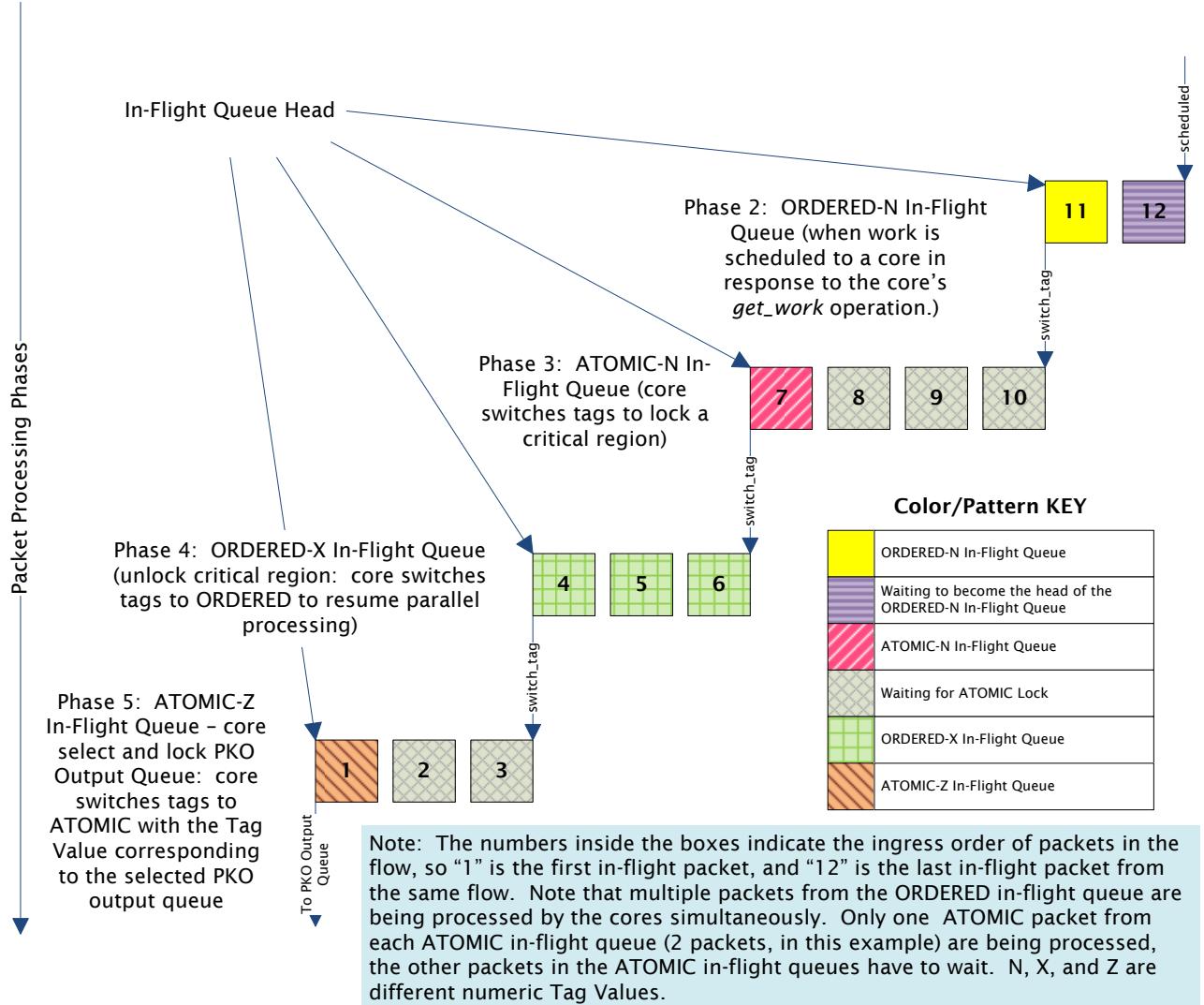
When the core needs to unlock a critical region, it performs a `switch_tag` operation, changing the tag type to ORDERED, carefully specifying a different tag value than was used in the original ORDERED tag tuple.

Phase 5: Packet Output

When the packet is ready for transmission, the core performs a `switch_tag` operation with tag type set to ATOMIC. The tag value is used to select the target PKO Output Queue. Once the core has been granted the ATOMIC lock, the core sends packet transmission commands to the PKO Output Queue. The PKO is responsible for managing packet priority, and sending the packet to the appropriate port in the packet interface. Packet order is guaranteed if all packets from the same flow are sent to the same PKO Output Queue.

The following figure shows an example of packets flowing through packet processing phases controlled by `switch_tag` operations. Phase 1 is not shown in this figure.

Figure 29: Packet Processing Phases and Sequential Tag Switch Operations



In the figure below, the simplified example packet processing phases are summarized, and shown with the associated tag type.

Table 2: Example Packet Processing Phases

Phase	Description	Details	Example Tag Type
Phase 1	Packet Input	The received packet is sent to the PIP/IPD which allocates needed buffers and DMAs the data to L2/DRAM. PIP/IPD sets the first tag type and tag value, and then gives the work to the SSO to schedule.	ORDERED
Phase 2	SSO schedules new work to the core: cores work on packets from the same flow in parallel	Cores requests work to do. The SSO schedules work to the cores. Multiple packets from the same flow are processed in parallel; multiple flows are processed in parallel.	ORDERED
Phase 3	Lock critical region: one-at-a-time access	Cores perform the <code>switch_tag</code> operation from ORDERED to ATOMIC to lock a critical region. The SSO manages the tag switch. Only one packet may hold the same ATOMIC lock at a time. Typically different flows have different locks, so multiple flows continue to be processed in parallel.	ATOMIC
Phase 4	Unlock critical region and resume parallel processing	Cores perform the <code>switch_tag</code> operation from ATOMIC to ORDERED to unlock a critical region. The SSO manages the tag switch. Once the switch is complete, multiple packets from the same flow are processed in parallel; multiple flows are processed in parallel.	ORDERED
Phase 5	Packet Output	Cores perform the <code>switch_tag</code> operation from ORDERED to ATOMIC to lock the PKO Output Queues. The tag value is used to select the PKO Output Queue. The SSO manages the tag switch. By using the ATOMIC tag type and sending all packets from the same flow to the same PKO Output Queue, packets are transmitted in ingress order.	ATOMIC

PACKET FLOW

Software Overview

SW OVERVIEW

TABLE OF CONTENTS

TABLE OF CONTENTS	1
LIST OF TABLES.....	5
LIST OF FIGURES	6
1 Introduction.....	8
1.1 Where to Get More Information	8
2 Introducing cnMIPS (Cavium Networks MIPS)	9
3 Introducing the Simple Executive API.....	10
4 Runtime Environment Choices for cnMIPS Cores.....	13
4.1 Performance Difference Between Simple Executive and Linux	14
4.2 Simple Executive	14
4.3 SMP Linux.....	15
4.3.1 Linux: <i>embedded_rootfs</i> File System	16
4.3.2 Linux: Debian File System	17
4.3.3 Linux Application Support	17
4.3.4 Cavium Networks Ethernet Driver	18
4.3.5 Simple Executive API Calls From Linux	18
4.3.6 CPU Affinity.....	20
4.3.7 Linux on Small Systems (Limited MBytes of Memory)	20
4.3.8 Running Multiple Linux Kernels on the OCTEON Processor	20
4.4 Hybrid Systems: Simple Executive and Linux Co-Existing.....	20
4.5 System Initialization	21
4.6 The Hardware Simulator.....	21
4.7 Other Runtime Environments	21
5 Combinations of Runtime Environments on One Chip.....	21
5.1 One-Core Runtime Choices.....	22
5.2 Multicore Runtime Choices.....	23
5.2.1 Easiest Configurations to Implement.....	23
5.2.2 Intermediate Configurations	23
5.2.3 Advanced Configurations	24
5.3 Application Entry Point and Startup Code	25
5.4 Booting SE-S or SE-UM Applications	27
5.5 Booting One ELF File on Multiple Cores: Load Sets	27
5.5.1 Starting SE-S Applications With the <code>bootoctl</code> Command.....	28
5.5.2 Starting Linux With the <code>bootoctllinux</code> Command.....	29
5.5.3 Starting SE-UM Applications With the <code>oncpu</code> Command	29
5.6 Booting Different ELF Files.....	32

5.7	Synchronizing Multiple Cores.....	32
5.7.1	Synchronizing Cores in the Same Load Set	33
5.7.2	Synchronizing Cores in Different Load Sets	33
5.7.3	SMP Linux Synchronization.....	34
5.7.4	Multiple SE-S or SE-UM ELF Files (Not Recommended)	34
6	Software Architecture.....	36
6.1	Control-Plane Versus Data-Plane Applications.....	36
6.2	Event-driven Loop (Polling) Versus Interrupt-Driven Loop	37
6.3	Using Work Groups in Packet Processing.....	38
6.3.1	Work Groups	38
6.3.2	Configuring the Per-Core Group Mask in the SSO Scheduler	39
6.4	Pipelined Versus Run-To-Completion Software Architecture	45
6.4.1	Comparing Run-To-Completion and Traditional Pipelining.....	45
6.4.2	A Quick Look at Packet Processing Math	46
6.4.3	Run-To-Completion.....	49
6.4.4	Traditional Pipelining	51
6.4.5	Modified Pipelining	52
6.5	Other Software Architecture Issues	54
6.5.1	Scaling	54
6.5.2	Code Locality: Reducing Icache Misses.....	55
6.5.3	Load-Balancing.....	57
6.6	Example: <code>linux-filter</code>	57
7	Application Binary Interface (ABI).....	62
7.1	ABI Choices.....	62
7.1.1	EABI (<code>OCTEON_TARGET=cvmx_64</code>): SE-S 64-Bit.....	62
7.1.2	N64 (<code>OCTEON_TARGET=linux_64</code>): SE-UM 64-Bit.....	62
7.1.3	N32 (<code>OCTEON_TARGET=cvmx_n32</code>): SE-S 32-Bit	62
7.1.4	N32 (<code>OCTEON_TARGET=linux_n32</code>): SE-UM 32-Bit	63
7.1.5	O32 (<code>linux_o32</code>) (Not Recommended)	63
7.1.6	Linux uclibc (<code>linux_uclibc</code>).....	63
7.1.7	Choosing the <code>OCTEON_TARGET</code>	63
7.2	64-Bit Porting Issues.....	63
8	Tools	66
8.1	GNU Cross-Development Toolchain	66
8.1.1	The Cavium Networks-Specific <code>cvmx_shared</code> Section	66
8.1.2	Link Addresses	68
8.1.3	Simple Executive Development Tools	68
8.1.4	Linux Development Tools	69
8.2	Native Tools (Run on the Target).....	69
8.2.1	Native tools and Simple Executive	69
8.2.2	Native tools and Linux.....	69
9	Physical Address Map and Caching on the OCTEON Processor.....	70
9.1	Physical Address Map	70
9.2	System Memory (DRAM) Addresses	72
9.3	I/O Space Addresses	72
9.4	Caching	74

9.5	Special L2 Cache Features: Partitioning and Locking	76
10	Virtual Memory	76
10.1	Virtual Address Translation.....	77
10.1.1	Mapping.....	77
10.1.2	The Translation Look-Aside Buffer (TLB)	78
10.1.3	Wired TLB Entries	78
10.2	Generic MIPS Virtual Memory Map.....	78
10.3	MIPS Virtual Memory Address Translation.....	79
10.3.1	Segments.....	80
10.3.2	Privilege Level (Mode) and Segments	81
10.4	Mapped and Unmapped Segments	82
10.4.1	Unmapped Segments	82
10.4.2	Mapped Segments.....	85
10.4.3	Addresses Versus Pointers.....	87
10.5	Virtual Memory onCavium Networks MIPS (cnMIPS).....	88
10.6	Cavium Networks-Specific cvmseg Segment	89
10.7	Accessing Application-Private System Memory.....	90
10.8	Summary of Virtual Address Space on cnMIPS	90
11	Allocating and Using Bootmem Global Memory.....	94
11.1	Using Global Bootmem	94
11.2	The malloc () and free () Functions and FPA Buffers.....	96
11.3	The cvmx_shared Section and FPA Buffers.....	97
11.3.1	The cvmx_shared Section is Not Always Shared	97
11.3.2	The cvmx_shared Section Should be Kept Small.....	99
11.4	Using Named Blocks to Share Memory Between Different Load Sets.....	100
12	Accessing Bootmem Global Memory (Buffers).....	102
12.1	Accessing Bootmem Global Memory From SE-S Applications	104
12.1.1	SE-S 64-Bit Bootmem Access.....	104
12.1.2	SE-S 32-Bit Bootmem Access.....	104
12.2	Accessing Bootmem Global Memory From Linux Kernel: 64-Bit	104
12.3	Accessing Bootmem Global Memory from SE-UM Applications	105
12.3.1	SE-UM 64-Bit Bootmem Access.....	105
12.3.2	SE-UM 32-Bit Bootmem Access.....	105
12.4	Bootmem Size in Different Access Methods.....	106
12.5	Using cvmx_ptr_to_phys () and cmvx_phys_to_ptr () Functions	107
13	Accessing I/O Space	107
13.1	Accessing I/O Space from SE-S Applications.....	107
13.1.1	SE-S 64-Bit I/O Space Access.....	107
13.1.2	SE-S 32-Bit I/O Space Access.....	107
13.2	Accessing I/O Space from Linux Kernel: 64-Bit	107
13.3	Accessing I/O Space from SE-UM Applications	107
13.3.1	SE-UM 64-Bit I/O Space Access	107
13.3.2	SE-UM 32-Bit I/O Space Access	108
14	Simple Executive Standalone (SE-S) Memory Model	108
14.1	Simple Executive Application Space.....	109
14.2	Simple Executive System Memory Access	109

14.2.1	Mapping of System Memory	109
14.3	Simple Executive I/O Space Access.....	113
14.4	Simple Executive Virtual Memory Configuration Options.....	113
14.4.1	CVMX_USE_1_TO_1_TLB_MAPPINGS.....	113
14.4.2	CVMX_NULL_POINTER_PROTECT	114
14.5	SE-S 32-Bit Applications	114
15	Linux Memory Model.....	117
15.1	Configuring Linux and the Effect on the Memory Model.....	117
15.1.1	Linux <i>cvmseg</i> (IOBDMA and Scratchpad) Size.....	117
15.1.2	SE-UM 64-Bit: Direct Access to I/O Space Via <i>xkphys</i>	118
15.1.3	SE-UM 64-Bit: Direct Access to System Memory Via <i>xkphys</i>	118
15.1.4	SE-UM 32-bit: Reserving a Pool of Free Memory.....	118
15.2	Linux Kernel Space and Simple Executive API Calls.....	120
15.3	Linux Memory Configuration Steps	120
15.4	Linux Kernel-Mode Virtual Address Space on the OCTEON Processor.....	124
15.5	Linux 64-bit User-Mode Virtual Address Space for OCTEON	126
15.6	Linux 32-Bit Virtual Address Space for OCTEON	127
16	Downloading and Booting the ELF File.....	129
16.1	Bootloader Memory Model	130
16.1.1	The Reserved Download Block	131
16.1.2	ELF File Maximum Download Size	131
16.1.3	The Reserved Linux Block	133
16.2	Booting the Same SE-S ELF File on Multiple Cores	135
16.3	Downloading and Booting Multiple ELF Files	137
16.3.1	Downloading by Re-using One Reserved Download Block	137
16.3.2	Downloading Using Two Different Reserved Download Blocks	138
16.4	Protection from Booting Multiple Applications on the Same Core	140
17	SDK Code Conventions.....	140
17.1	Register Definitions and Accessing Registers.....	140
17.1.1	Register Definitions	140
17.1.2	Register Typedefs	141
17.1.3	Accessing Registers Using Register Definitions and Data Structures.....	142
17.2	The <i>cvmx_sysinfo_t</i> Typedef	144
17.3	OCTEON Models	145
18	Bootloader Historical Information.....	145
18.1	Backward Compatibility for Linux ELF Files Built Under SDK 1.6	147

LIST OF TABLES

Table 1: Types of Cavium Networks-Specific Instructions	9
Table 2: OCTEON Hardware Units Overview.....	11
Table 3: Additional Simple Executive Support.....	12
Table 4: SE-S Application Entry Point and Startup	26
Table 5: Linux SE-UM Application Entry Point and Startup.....	27
Table 6: Setting the Cores's Group Mask in the SSO	40
Table 7: Key ABI Differences	64
Table 8: SE-S ABIs (N32, EABI64), Data Type Lengths, and Toolchain.....	64
Table 9: SE-UM ABIs (N32, N64), Data Type Lengths, and Toolchain	65
Table 10: Other ABI (O32), Data Type Lengths, and Toolchain	65
Table 11: Simplified View of I/O Space	73
Table 12: The 64-Bit Virtual Address Segments.....	91
Table 13: The 32-Bit Virtual Address Segments.....	92
Table 14: Bootmem Allocator Functions in SDK 1.8	95
Table 15: Summary of Access to System Memory and I/O Space.....	103
Table 16: Configuration Choices and Resultant Global Memory Limits.....	106
Table 17: Cavium Networks-Specific Linux menuconfig Options	120
Table 18: Accessing Register Fields.....	143

LIST OF FIGURES

Figure 1: Simple Executive Hardware Abstraction Layer (HAL)	10
Figure 2: Using Simple Executive API from Different Runtime Environments	13
Figure 3: Simple Executive Standalone Application (SE-S)	14
Figure 4: Simple Executive calls from Kernel Mode	15
Figure 5: Simple Executive User-Mode (SE-UM) Application	15
Figure 6: One Core Runtime Choices.....	22
Figure 7: Easiest Multicore Configurations.....	23
Figure 8: Intermediate Multicore Configurations.....	24
Figure 9: Advanced Multicore Configurations.....	25
Figure 10: SE-S Load Set	27
Figure 11: SE-UM Load Set.....	28
Figure 12: Booting SE-S Applications With the Bootoctl Command	29
Figure 13: SE-UM Applications Started With oncpu on Multiple Cores	31
Figure 14: Hybrid Load Sets.....	32
Figure 15: Multiple SE-S ELF Files (Not Recommended)	35
Figure 16: Multiple SE-UM ELF Files (Not Recommended)	35
Figure 17: SE-S Used for Both Control-Plane and Data-Plane Applications	36
Figure 18: Linux for Control-Plane and SE-S for Data-Plane Applications	37
Figure 19: The First Two Words of the Work Queue Entry.....	38
Figure 20: Each Core May Accept Work from Any and All Groups	39
Figure 21: Cores Can Receive Work Based on Their Group Mask	41
Figure 22: A Core is Idle if No Suitable Work is Available	42
Figure 23: Scheduling Previously Descheduled Work	44
Figure 24: Packet Processing Math	47
Figure 25: Run-To-Completion Versus Traditional Pipelining.....	49
Figure 26: Simplified Run-To-Completion Architecture	50
Figure 27: Scaling Run-To-Completion Architecture	51
Figure 28: Traditional Pipelining.....	52
Figure 29: Modified Pipelining	53
Figure 30: Modified Pipelining: Using Groups to Load Balance.....	53
Figure 31: Scaling the Data Plane	55
Figure 32: Using Code Locality to Reduce Icache Misses.....	56
Figure 33: Example: Linux-filter Drops a Broadcast IP Packet	59
Figure 34: Example: Linux-filter Forwards a Non-Broadcast IP Packet	61
Figure 35: Simplified Physical Address Map.....	71
Figure 36: Simplified View of Cache “miss” and “hit”	74
Figure 37: Prefetch Commands Used to Bypass Some Caches.....	75
Figure 38: Multiple Programs Have the Same Virtual Addresses.....	77
Figure 39: Generic MIPS Memory Map.....	79
Figure 40: 64-Bit Virtual Address: Segment Selector and SEGBITS	80
Figure 41: 32-Bit Virtual Address: Segment Selector and SEGBITS	81
Figure 42: The <i>xkphys</i> Window to Physical Address Space.....	83
Figure 43: The Small <i>kseg0</i> Window to Physical Address Space	84
Figure 44: <i>kseg0</i> and <i>kseg1</i> Access the Same Memory	85

Figure 45: 64-Bit Virtual Address Translation on MIPS.....	86
Figure 46: 32-Bit Virtual Address Translation on MIPS.....	87
Figure 47: OCTEON 64-Bit Virtual Address Space – Summarized	93
Figure 48: OCTEON 32-Bit Virtual Address Space - Summarized.....	94
Figure 49: Named and Unnamed Memory Blocks	96
Figure 50: <i>cvmx_shared</i> : Same and Different Load Sets	97
Figure 51: <i>cvmx_shared</i> : Inefficient SE-S Configuration	98
Figure 52: <i>cvmx_shared</i> : Inefficient SE-UM Configuration	99
Figure 53: Sharing Memory Between Different Load Sets	101
Figure 54: Simple Executive Size Limitation if 1:1 Mapping is Used.....	111
Figure 55: SE-S 64-Bit Virtual Memory Map.....	112
Figure 56: SE-S 32-Bit Virtual Memory Map.....	116
Figure 57: Linux Kernel Virtual Address Space	125
Figure 58: Linux 64-Bit SE-UM Virtual Address Space for OCTEON.....	127
Figure 59: Linux 32-Bit SE-UM Virtual Application Space on OCTEON.....	129
Figure 60: Creating an In-Memory Image	130
Figure 61: Downloading to the Reserved Download Block	132
Figure 62: The Bootloader Creates the In-memory Image.....	133
Figure 63: The Reserved Linux Block.....	134
Figure 64: Bootloader Memory Usage in SDK 1.7 and Above.....	135
Figure 65: The Power of One Load Set	136
Figure 66: Downloading Multiple ELF Files – Same Download Block	138
Figure 67: Downloading Two ELF Files Using Two Download Blocks	140
Figure 68: Bootloader Memory Usage in SDK 1.6 and Below	146

1 Introduction

This chapter provides a software overview. Additionally, certain hardware and software architecture topics are covered in this chapter.

The chapter will introduce the following topics:

- cnMIPS cores
- Simple Executive API (HAL)
- Different runtime environment choices such as standalone or user-mode, and combinations
- Software architecture issues
- Application Binary Interfaces (ABIs) *supported*
- Tools: cross-development and native toolchains
- Physical address map and caching on the OCTEON processor
- Virtual memory, including different views depending on runtime environment
- Bootmem global memory: how to allocate and access it.
- Shared memory
- Bootloader
- Software Development Kit (SDK) code conventions: registers and typedefs

This information is needed to understand the next chapter: the *SDK Tutorial*. The *SDK Tutorial* chapter provides details on how to boot and run applications. Two examples are run: `hello` and `linux-filter`.

This chapter is not designed to replace the documentation provided with the SDK, but merely to provide a high-level overview of the software provided with the SDK. Throughout the chapter relevant SDK documents are referenced to help the reader find more detailed information. See the *SDK Tutorial* chapter for information on how to access the SDK documentation. Note that if the information in the SDK conflicts with information in this chapter, it may be due to the SDK being more current than this chapter. The information provided with the SDK should be considered to be more accurate because the SDK documentation is updated with each release.

Before reading this chapter, please read the *Packet Flow* chapter. This chapter will provide background information on the basic hardware units and how they interact. This information is necessary to understand the Simple Executive API.

1.1 Where to Get More Information

The SDK comes with a large amount of documentation in html format. This documentation is located in the `docs` directory in the installed SDK. See the *SDK Tutorial* chapter for information on how to extract the SDK and locate the documentation.

2 Introducing cnMIPS (Cavium Networks MIPS)

Each OCTEON processor may contain between 1 and N cnMIPS cores, depending on the OCTEON model. When this chapter was written $N = 16$. In the future, the number of cores available may be higher.

The cnMIPS (Cavium Network MIPS) cores use the MIPS64 v2 instruction set, supporting both 32-bit and 64-bit processing.

Cavium Networks has added some custom instructions to accelerate common networking operations, such as bit test branch instructions or bit-field insert/extract. Because of these added instructions, only the tools provided with the SDK should be used to build software which will run on the OCTEON processor. When using the tools provided with the SDK, the optimizer uses these instructions automatically. The table below briefly describes the added functions. See the *OCTEON Hardware Reference Manual (HRM)* for more information. Note: there are about 3 pages of Cavium Networks instructions listed in the *HRM*.

Hardware floating point instructions are not implemented. Floating point instructions can be implemented by using the “soft float” option on the compiler (`gcc hello.c -msoft-float -o hello`).

Table 1: Types of Cavium Networks-Specific Instructions

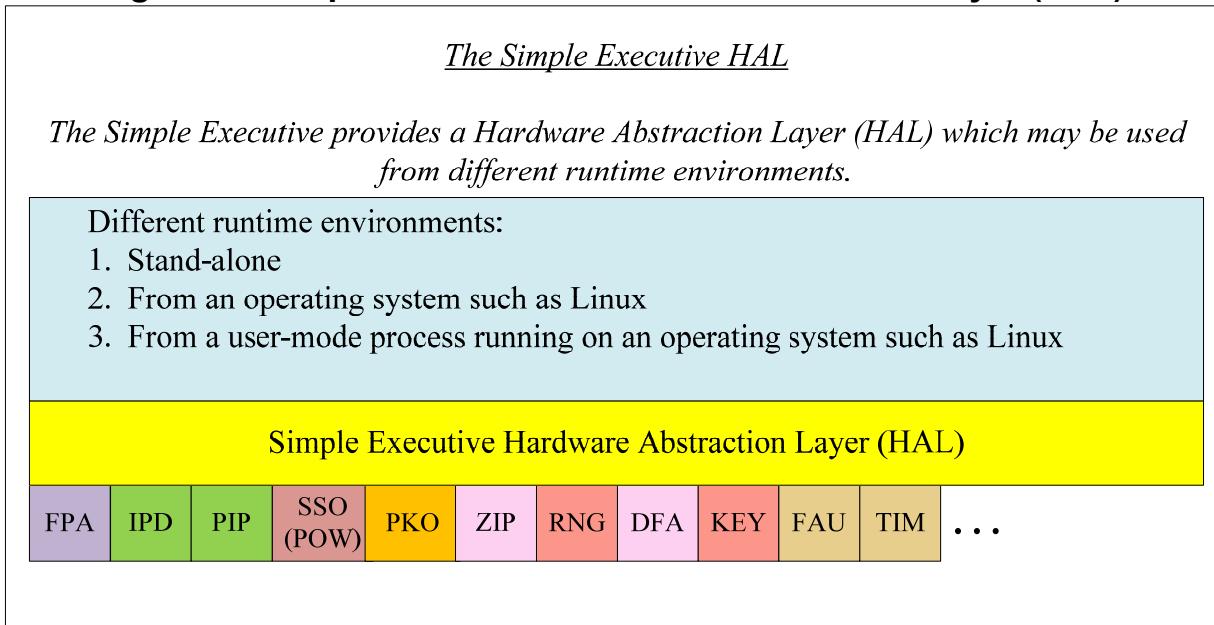
Instruction Categories
Unsigned byte add.
Bit-test branches
Cache manipulation instruction
Instructions to use the in-core 3DES coprocessor (must have the Security Engine)
Instructions to use the in-core AES coprocessor
Instructions to use the in-core CRC coprocessor
Instructions to use the Galois Field Multiplier
Instructions to use the in-core HSH coprocessor
Instructions to use the in-core KASUMI coprocessor
Instructions to use the in-core LLM coprocessor
Register-direct 64-bit multiply
Signed-bit field extract and clear/insert instructions
Instructions to move data to/from Cavium Networks-specific multiplier registers
Prefetch, Don't Write Back , Prepare for Store
Count the number of ones in a 32-bit (POP) or 64-bit (DPOP) variable
64-bit cycle counter. Fast SSO Switch access
32-bit and 64-bit store atomic add instructions

Instruction Categories
Set on equal; set on non-equal instructions
Memory reference ordering instructions (SYNCIOBDMA, SYNCs, SYNCW, SYNCWS)
Unaligned load/store instructions
Large multiply instructions

3 Introducing the Simple Executive API

The Simple Executive provides a Hardware Abstraction Layer (HAL) in the form of an Application Programming Interface (API) to the underlying hardware units. This API is a very thin layer of simple functions which access the CPU registers. It also provides some convenience routines for block initialization. The API can be used from both kernel and user mode.

Figure 1: Simple Executive Hardware Abstraction Layer (HAL)



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

- Basic units: FPA, IPD, PIP, SSO, and PKO
- Intermediate units: FAU and TIM
- Advanced units: LLM, ZIP, RNG, DFA, KEY, CIU, etc.

The following table provides an overview of the hardware units. Convenient access to these hardware units is provided by Simple Executive function calls and macros. Note that different chips have different features, so not all APIs are supported on all chips. In particular, DFA (Deterministic Finite Automaton – used in pattern matching) and LLM (Low Latency Memory – used to support DFA functions) are not provided on all chips.

Table 2: OCTEON Hardware Units Overview

<p><i>Note that different OCTEON models have different features: some hardware units are not available on all models.</i></p>	
Basic Hardware Units	
FPA	The Free Pool Allocator Unit manages up to 8 pools of free buffers which may be requested by other hardware units. The most common uses of the buffers are for Packet Data Buffers and Work Queue Entry Buffers.
PIP	The Packet Input Processing Unit receives the packet data from the Packet Interfaces, and perform basic error checking on the data.
IPD	The Input Packet Data Unit works together with the PIP to allocate needed buffers, and process the packet data. The IPD fills in the Work Queue Entry Buffer and the Packet Data Buffer. It then submits the Work Queue Entry Buffer to the SSO's QoS Input Queues. Requires FPA Packet Input Buffers and Work Queue Entry Buffers.
SSO	The Schedule/Synchronization/Order Unit maintains the QoS Input Queues, and manages scheduling work to cores. It also maintains the work order, and provides the support needed for packet-linked atomic locking.
PKO	The PKO manages packet output. Cores submit command words to its Output Queues. These command words include a pointer to the packet data to be transmitted. The cores then "ring" a doorbell to notify the PKO how many command words were written to the Output Queue. The PKO DMAs the packet data from the Packet Data Buffer to its internal memory, and sends it from there to the Packet Interfaces. This operation requires an FPA pool of Command Buffers.
Intermediate Hardware Units	
FAU	Fetch and Add Unit - a 2 KB register file supporting read, write, atomic fetch and add, and atomic update operations. This unit can be accessed from both the cores and the PKO. The cores use the FAU for general synchronization purposes.
TIM	Timers - requires FPA timer pool.
Advanced Hardware Units	
CIU	The Central Interrupt Unit controls the routing of interrupt sources to the cores, including mailbox and watchdog interrupts. Any interrupt source may be routed to any core.
DFA	Deterministic Finite Automata (DFA) unit, used for regular expressing parsing and acceleration. The chip must have the DFA hardware Unit.
LLM	Low Latency Memory - used for storing DFA graphs. The chip must have the DFA hardware unit, and the user-supplied LLM (Low Latency Memory).
ZIP	Compression/decompression unit. The chip must have the ZIP hardware unit.
RNG	Random Number Generator
KEY	8K of on-chip memory for holding security keys. This memory can be cleared using an external hardware pin.

Simple Executive API also includes functions and macros for:

- System memory allocation (bootmem)
- Synchronization between cores
- Spinlocks
- Reader-writer locks
- Atomic set, add, compare and store operations
- Barrier functions

Table 3: Additional Simple Executive Support

Note that different OCTEON models have different features: some functions are not supported on all models.

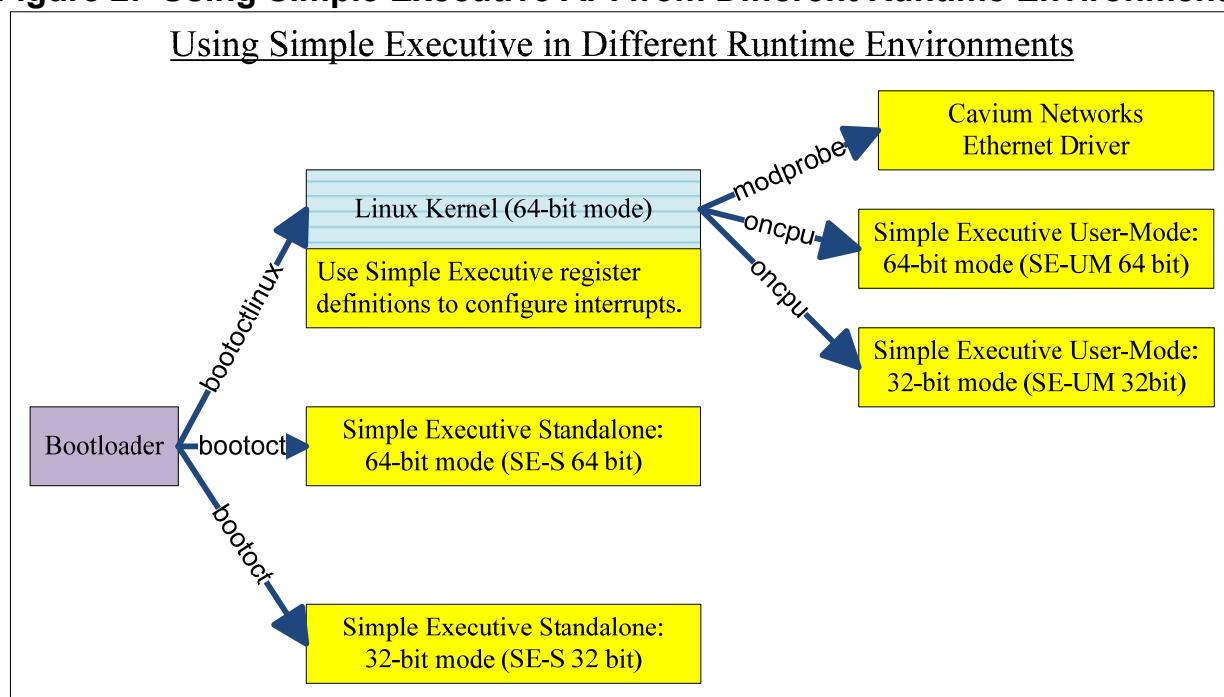
Synchronization Support	
Packet-linked Locks	Packet-linked locks are implemented by the SSO, and provide ATOMIC access to critical regions.
Basic Spinlocks	Non-recursive spinlocks.
Recursive Spinlocks	Recursive spinlocks
Atomic Operations	Atomic set, add, compare and store operations.
Reader/Writer locks	Multiple cores may hold read locks, while write locks are exclusive.
Barrier Functions	Barrier function which causes each core to wait until all cores reach the same instruction. (All cores running the same application.)
Coremask Functions	Coremask functions to select the first core to do the application initialization.
Memory Management Support	
Scratchpad access functions	Access core-local scratch pad memory (CVMSEG). Scratch pad used for local variables and for the results of IOBDMAs.
Bootmem functions	Used to allocate shared aligned memory. Usually used to allocate the memory used in FPA pools.
Utility Functions	
cvmx_user_app_init()	Mandatory function to initialize the Simple Executive application.
cvmx_get_core_num()	Queries a MIPS-standard register on the core to get the core number this instance.
cvmx_phys_to_ptr()	Convert physical address into a pointer containing a virtual address.
cvmx_ptr_to_phys()	Convert a pointer containing a virtual address into a physical address.
cvmx_sysinfo_get()	Access the global cvmx_sysinfo data structure (for instance, to synchronize cores)

More information about the Simple Executive functions and macros may be found in the SDK document “*OCTEON Simple Executive Overview*”.

Simple Executive functions and macros may be used either to create a standalone Simple Executive application, or may be called from drivers or applications running on an operating system kernel such as Linux. For instance, after the Linux kernel is booted, a Cavium Networks Ethernet driver may be started. This driver uses the Simple Executive API to configure the OCTEON hardware. Simple Executive User-Mode applications may also be started from Linux.

Both 32-bit and 64-bit modes are supported, although 64-bit mode should be used whenever possible.

Figure 2: Using Simple Executive API from Different Runtime Environments



4 Runtime Environment Choices for cnMIPS Cores

There are several choices for runtime environment. The three supplied by Cavium Networks are Simple Executive standalone mode, Linux, and the hardware simulator.

When running Simple Executive on multiple cores, the same ELF file is usually run on all of the cores. These cores are all started from one load command. The cores share the `.text` and read-only data (`.rodata`) sections. They also share `cvmx_shared` variables, and memory allocated with `bootmem_alloc`.

When running Linux on multiple cores (SMP), there is one kernel running, not one kernel per core. Linux applications are scheduled to run on different cores.

Simple Executive may be run on some of the cores, while Linux is run on the other cores (a hybrid system). In this case, the two ELF files are booted using two separate boot commands. The set of cores to run the program on is specified as an argument to the boot command.

Linux and Simple Executive both use the bootmem functions to allocate and free memory. Shared memory may be shared between the Linux and Simple Executive applications if the named block *bootmem_alloc* functions are used.

4.1 Performance Difference Between Simple Executive and Linux

Simple Executive run in standalone mode provides the lowest overhead and the greatest potential for scaling.

When running Simple Executive applications as Linux user-mode applications, although the OCTEON hardware has been configured to allow access to both hardware and memory without performance penalties, your application may still have noticeably slower performance than if it was run in standalone mode. Cache misses, TLB misses, and bus contention are more likely when running as a Linux user-mode application due to the large amounts of code and data needed for Linux. The Linux scheduler timer interrupt also periodically transfers focus to other tasks. The exact performance difference is application-dependent.

4.2 Simple Executive

Simple Executive provides an API to the hardware units. Simple Executive may be run Standalone (SE-S), or as a user-mode (SE-UM) application on an operating system such as Linux. When run as a user-mode application, different application startup code (`main()`) is called, and there are other minor porting items to consider.

All cores running a Simple Executive application which are started from the same load command share the *cvmx_shared* data section. For more information, see Section 8.1.1 – “The Cavium Networks-Specific *cvmx_shared* Section”. They also share the *.text* and read-only data (*.rodata*). They also share memory allocated with *bootmem_alloc*.

The following figure shows a representation of a core running Simple Executive in Standalone (SE-S) mode.

Figure 3: Simple Executive Standalone Application (SE-S)

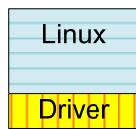
One core runs a Simple Executive Standalone (SE-S) application



Simple Executive calls may be made from kernel mode. For example, the Cavium Networks Ethernet driver, which runs on Linux, makes Simple Executive calls.

Figure 4: Simple Executive calls from Kernel Mode

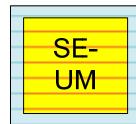
One core runs Linux with Cavium Networks
Ethernet Driver



The following figure shows a representation of a core running Simple Executive as a User-Mode application.

Figure 5: Simple Executive User-Mode (SE-UM) Application

One core runs a Simple Executive User-Mode (SE-UM) application on Linux



To use all available memory, SE-UM applications should be compiled for 64-bit mode. 32-bit mode is sometimes used, but can only access a limited amount of physical memory.

SE-S supports a single instance per core (there is no scheduler running). Note than an SE-S instance is not as complex as a process.

SE-S is very fast compared to SE-UM. There are no context switches, and all memory is mapped for fast access.

To get the maximum performance from the OCTEON processor: Whenever possible, design the application to use a 64-bit Simple Executive application.

4.3 SMP Linux

SMP (Symmetric Multi-Processing) Linux may be run on one or more cores. The file system is either the tiny embedded root file system (*embedded_rootfs*) or the large Debian file system. Usually, *embedded_rootfs* is used because it will fit into on-board flash. In some circumstances, such as during development, the larger Debian file system may be desired. The Debian file system must be used from either Compact Flash, or NFS.

When Linux is booted, the boot command (`bootoctlinux`) has an optional argument (`mem`) which is used to set the amount of memory allocated to Linux. The default is 512 MBytes. Setting “`mem=0`” will allow the kernel to use all the memory on the board. Note that setting “`mem=0`” will leave no bootmem available for applications running on other cores to allocate. The Linux driver will still allocate skbuff memory and populate the FPAs needed to send and receive packets.

Note: The default SDK configuration requires around 230 MBytes of system memory. Linux can be run with as little as 8 MBytes when the file system is in flash or Compact Flash.

4.3.1 Linux: *embedded_rootfs* File System

When running Linux with the embedded root file system (*embedded_rootfs*), the root file system is a RAM disk (in memory only). In this case, the ELF file is either stored in on-board flash, or downloaded from a host.

Note that when the system is powered off or reset, the ELF file is no longer in memory. It must be reloaded from flash or downloaded from a host.

The embedded root file system is used when there are no devices attached to the OCTEON processor to store the root file system for download to OCTEON. (For instance, if the ELF file cannot be loaded over the network or from an external device such as Compact Flash.) The Linux examples in the *SDK Tutorial* chapter will use *embedded_rootfs*.

Typically, the embedded root file system contains only the minimum number of files needed. To save space, the small utility set “BusyBox” is used instead of the normal Linux utilities. The BusyBox component in the embedded root file system is controlled by the makefile:

```
$OCTEON_ROOT/linux/embedded_rootfs/pkg_makefiles/busybox.mk.
```

There is one utility called `/bin/busybox`. The file is symbolically linked to other names to allow you to call the other “utilities”. When the utility is called by a different name, such as `cat`, it executes that function. BusyBox can be tailored to exclude any unneeded functions. This reduces the executable size, saving space.

The `BusyBox.txt` file has a list of the included “functions” (which then act as utilities).

From the `BusyBox.txt` file:

```
"COMMANDS
Currently defined functions include:

[, [[, addgroup, adduser, adjtimex, ar, arping, ash, awk,
basename, bbconfig, bunzip2, busybox, bzcat, cal, cat, catv,
chattr, chgrp, chmod, chown, chroot, chvt, cksum, clear, cmp,
comm, cp, cpio, crond, crontab, cut, date, dc, <text
omitted>"
```

Note that some utility options are not supported by these functions: options not usually needed in the embedded environment are not included. The exact options supported are detailed in the `BusyBox.txt` file.

More details may be found on the net at <http://www.busybox.net/about.html> or in `$OCTEON_ROOT/linux/embedded_rootfs/build/busybox-1.2.1/docs/BusyBox.txt`.

4.3.1.1 Adding Examples to *embedded_rootfs*

The example applications were added to Linux *embedded_rootfs* by instructions in package makefile `$OCTEON_ROOT/linux/embedded_rootfs/pkg_makefiles/sdk-examples.mk`.

For detailed directions on adding an application to the embedded root file system, see the SDK document “*Linux on the OCTEON*” in the section “*How to add a Package*”.

4.3.2 Linux: Debian File System

When running Linux with the Debian file system, the root file system is on a Compact Flash card. Other than some minor changes, the Cavium Networks version of Debian is a distribution from <http://www.debian.org/>, with some minor changes. Cavium Networks has not modified the utilities provided by Debian. If problems with the utilities occur, contact Debian for assistance.

When using the Debian file system, the kernel is booted off the Compact Flash card with the boot command option `root=/dev/cfa2`. This tells the kernel that the root file system is on the second partition on the Compact Flash card. Once the kernel has booted, the root file system is located on the Compact Flash Card. (Note that the “root” file system is mounted as “/” when the kernel is booted. In the Linux directory structure “/” is the “root” directory. All other directory paths are relative to this point.)

The Debian file system is useful for the large variety of programs provided.

For more information on running Debian Linux on the OCTEON processor, see the *SDK Tutorial* chapter, and the SDK document “*Running Debian GNU/Linux on OCTEON*”.

4.3.3 Linux Application Support

Both 32-bit and 64-bit Linux applications are supported by the cross development toolchain. Since OCTEON is a 64-bit processor, running in 64-bit mode is faster and more efficient, but is not required.

Note: *The kernel is always in 64-bit mode.*

To run an application as a Linux user-mode application, the application may be added either to `embedded_rootfs`, or to the Debian file system. Note that running SE-UM applications over NFS is not recommended. (See the note in Section 4.3.4 – “Cavium Networks Ethernet Driver”.)

Linux applications may make Simple Executive API calls. These Simple Executive files are not supported under Linux:

- `cvmx-interrupt.c`
- `cvmx-interrupt-handler.S`
- `cvmx-malloc.c`
- `cvmx-app-init.c` (this file is replaced with `cvmx-app-init-linux.c`)

When building an application using the Makefiles provided with the example code, if the target is a Linux target, the file `$OCTEON_ROOT/executive/cvmx.mk` will make the appropriate changes to the object files used in the build.

These applications also may not call `cvmx_malloc()` functions or `cvmx_zone()` functions.

4.3.4 Cavium Networks Ethernet Driver

A Cavium Networks Ethernet driver module is available to support Ethernet using either the GMII, RGMII, SGMII interfaces, SPI4 (with a SPI4000 daughter card), or XAUI. Different OCTEON models support different devices. The GMII, RGMII, and SGMII ports are Ethernet devices “eth0” through “ethN”. SPI4000 ports are devices “spi0” through “spiN”. XAUI devices are “xaui0” through “xauiN”. (N is the maximum number of devices supported by the system.)

To add the Cavium Networks Ethernet driver, use the `modprobe` command. (Note that the POW unit has been renamed to SSO in documentation, but is still referred to as POW in software.)

Arguments to the `modprobe` command include:

- `pow_receive_group`: only packets with this group number are received by the kernel. The default is “15”.
- `pow_send_group`: Linux creates a virtual Ethernet device not connected to any physical ports, named “pow0”. This device will accept work from the POW receive group and transmit using the POW send group. In the `linux-filter` example, this group is set to “14”. The `linux-filter` example is discussed in more detail in Section 6.6 – “Example: `linux-filter`”.

An example where `modprobe` is used is presented in the *SDK Tutorial* chapter.

Note: When the Cavium Networks Ethernet driver is in use, applications must not reconfigure the OCTEON hardware. The Ethernet driver configures the SSO, FPA, CIU, PIP, IPD, PKO, and FAU (the Fetch and Add Unit). Some examples such as “passthrough” also configure the hardware units. Running both the Cavium Networks Ethernet driver and an example which initializes the hardware will cause the crash and reset with an error similar to the following text:

```
Version: Cavium Networks OCTEON SDK version 1.7.2, build 244
Warning: Enabling FPA when FPA already enabled.
Fpa pool 0(Packet Buffers) already has 928 buffers. Skipping setup.
Fpa pool 1(Work Queue Entries) already has 960 buffers. Skipping setup.
Fpa pool 2(PKO Command Buffers) already has 124 buffers. Skipping setup.
Interface 1 has 4 ports (RGMII)
```

Similarly, if the file system is NFS-mounted, then the Cavium Networks Ethernet driver is loaded. Running a program such as the example program `passthrough` over NFS will not work because `passthrough` will reconfigure the OCTEON hardware and NFS will stop working.

More details may be found in the SDK document “Linux on the OCTEON” in the section “Kernel Ethernet Drivers”.

4.3.5 Simple Executive API Calls From Linux

Linux kernel and applications may both make Simple Executive API calls. When Simple Executive calls are made from Linux user space, the process is referred to as a Simple Executive User-Mode application (SE-UM).

According to the SDK documentation, applications using the Simple Executive libraries under Linux userspace must rename their `main()` function to match the prototype below. This allows Simple Executive to perform needed memory initialization and process creation before the application runs.

```
extern int appmain(int argc, const char *argv[]);
```

When building examples with the provided Makefiles, the file `$OCTEON_ROOT/common.mk` will redefine the word “`main`” to “`appmain`” if a Linux target is specified, for example:

```
ifeq (${OCTEON_TARGET},linux_64)
PREFIX=linux_64
CFLAGS_GLOBAL += -DOCTEON_TARGET=${OCTEON_TARGET} -mabi=64 -
march=octeon
-msoft-float -Dmain=appmain
```

This is why the `linux-filter` example does not contain two different `main()` calls (the string “`main`” becomes `main()` for SE-S applications, and `appmain()` for SE-UM applications).

The following are some of the key points to remember when writing applications to run both under the SE-S and SE-UM environments:

- Use `#ifdef __linux__` to make SE-UM-specific changes to the code.
- Be careful to use `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()`. The Simple Executive 1:1 TLB mappings allow you to be sloppy and interchange physical addresses with virtual address. This isn't true under Linux.
- If you're talking directly to hardware, be careful. The normal Linux protections are circumvented. If you do something bad, Linux won't save you.
- Most hardware can only be initialized once. Unless you're very careful this also means your SE-UM application can only run once.

The `linux-filter` example, which runs both as SE-S and SE-UM, includes some examples showing use of the `#ifdef __linux__` test, for example:

```
// if running on Linux, include file which contains definitions required
// for compatibility with the POSIX standard
#ifndef __linux__
#include <unistd.h>
#endif
```

The *SDK Tutorial* chapter contains a table showing the available example applications, and whether they may be run on Linux. Examples of Linux applications which use Simple Executive API calls may be found in the `/examples` directory after Linux is booted on the OCTEON processor.

More details may be found in the SDK document “*Linux Userspace on the OCTEON*” in the section “*Running Simple Executive Applications under Linux*”.

4.3.6 CPU Affinity

Use the `oncpu` Linux utility to control which core or set of cores the SE-UM application will run on.

A SE-UM application should *never* call `sched_setaffinity()`, unlike a generic Linux application which may call `sched_setaffinity()` to control the cores it uses.

Note: *SE-UM applications are not full Linux apps and should limit themselves to the features supplied by the Simple Executive.*

Details on using the `oncpu` utility with SE-UM applications are provided in Section 5.5.3 – “Starting SE-UM Applications With the `oncpu` Command”.

4.3.7 Linux on Small Systems (Limited MBytes of Memory)

To run Linux on a small system (256 MBytes or less), see the directions in the SDK document “*Linux on Small OCTEON Systems*”.

4.3.8 Running Multiple Linux Kernels on the OCTEON Processor

More than one Linux kernel can be run on the OCTEON processor. For more information, see the SDK document “*Linux on the OCTEON*” in the section “*Booting Two Separate Kernels on an EBT3000*”.

4.4 Hybrid Systems: Simple Executive and Linux Co-Existing

Linux may be run on a subset of the cores while Simple Executive is running on a different subset of cores.

More details may be found in the SDK document “*Linux on the OCTEON*” in the section “*Co-existing with Simple Executive Applications*”. Here are the general guidelines provided in that chapter:

1. Allocate shared memory using the bootmem allocator functions. These functions provide the needed locking so that two applications will not get the same memory.
2. Keep core dependencies generic. Instead of allocating cores by core ID, use `cvmx_sysinfo_get()` to get the bitmask of cores actually running your application. Use the `cvmx_sysinfo_t` field “`core_mask`” to determine how many cores are running your application, and use `cvmx_coremask_first_core()` to select the core for initialization tasks. An example of using these functions may be found in the *FPA* chapter (in Volume 2).
3. Choose a single application to perform hardware initialization. Many initialization tasks must only be performed once. When designing a hybrid system, choose which single instance is responsible for initialization.
4. Use OCTEON hardware for inter-application communication. Both the SSO (via groups) and the Fetch and Add Unit (FAU) can be used to provide fast hardware-based messaging.

Hybrid systems may also consist of other configurations.

Note that Linux does not support booting SE-S. SE-S ELF files must be started from the bootloader.

4.5 System Initialization

Note that only one operating system or SE instance is responsible for initialization.

When running only SE-S applications (in one load set), the first core in the load set is responsible for system initialization.

When running Linux and SE-S, normally the Linux kernel initializes the hardware through the Ethernet driver. Simple Executive applications must wait until this initialization is done before continuing. SE-UM applications which also initialize the hardware (such as passthrough) must not be run at the same time as the Cavium Networks Ethernet driver is running.

See Section 5.7 – “Synchronizing Multiple Cores” for more information.

4.6 The Hardware Simulator

The third runtime environment supplied by Cavium Networks is the Hardware Simulator. The simulator is useful when actual hardware is not available and it is also very useful for performance tuning. Performance tuning is most easily done using the tool Viewzilla. This tool analyzes the output of the simulator, so making sure the code will run on the simulator as well as on actual hardware is recommended for performance-critical applications.

See the whitepaper “*OCTEON_Performance_Tuning*” for more information.

All of the examples provided with the SDK run on the simulator.

4.7 Other Runtime Environments

In addition to the three runtime environments supplied by Cavium Networks, several open-source and proprietary operating systems are available. Contact your Cavium Networks representative for an updated list of choices.

5 Combinations of Runtime Environments on One Chip

The following figures show chips running combinations of runtime environments, without showing the control-plane/data-plane configuration.

Note: these figures are intended to show the flexibility of the OCTEON processor. A specific design does not have to exactly match the figures shown below.

5.1 One-Core Runtime Choices

The following choices are available if the OCTEON model has only one core:

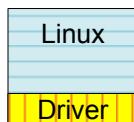
Figure 6: One Core Runtime Choices

Choices If Only One Core is Available

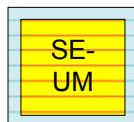
One core runs a Simple Executive Standalone (SE-S) application



One core runs Linux with Cavium Networks Ethernet Driver



One core runs a Simple Executive User-Mode (SE-UM) application on Linux



One core runs another OS



One core runs another OS with Simple Executive User-Mode application.



5.2 Multicore Runtime Choices

The following figures show different runtime choices for the OCTEON processor.

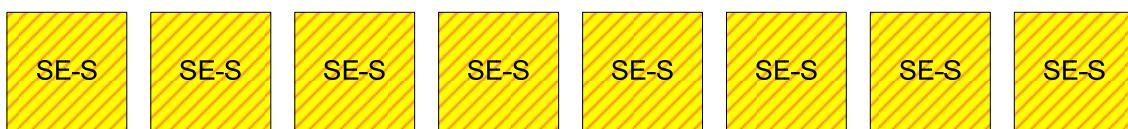
5.2.1 Easiest Configurations to Implement

The following configurations are the easiest to implement.

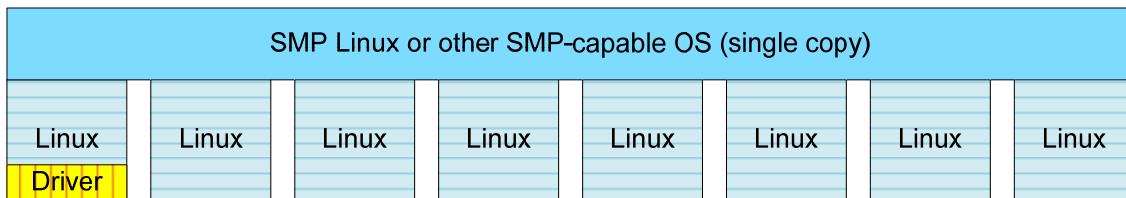
Figure 7: Easiest Multicore Configurations

Easy Multicore Implementations

Example of 8-core simple system running a Simple Executive Standalone application : 1 ELF file, 8 instances of SE-S



Example of 8-core simple system running Linux or other SMP-capable OS: a single copy of Linux runs on 8 cores

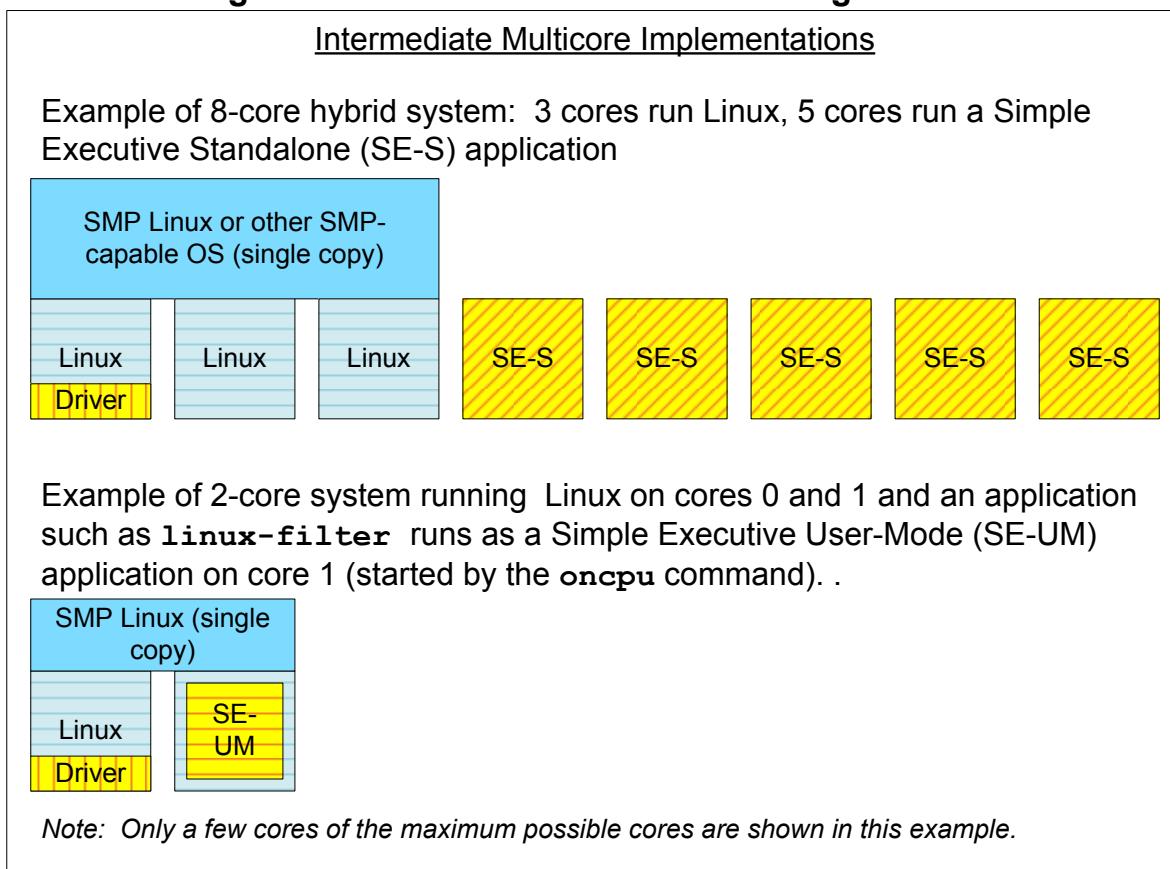


Note: Only a few cores of the maximum possible cores are shown in this example.

5.2.2 Intermediate Configurations

The following configurations in the midrange of complexity to implement.

Figure 8: Intermediate Multicore Configurations



Note: Although Linux is usually run on core 0, this is not a requirement.

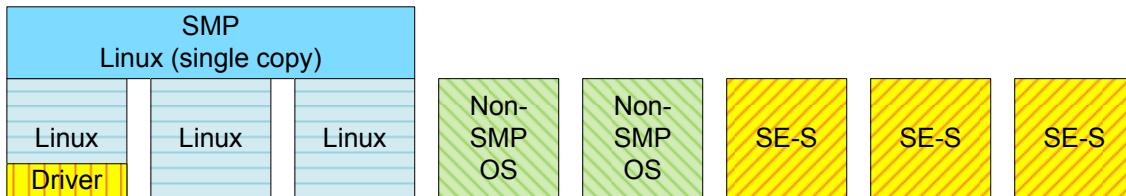
5.2.3 Advanced Configurations

The following configurations require advanced OCTEON processor knowledge, and careful resource management.

Figure 9: Advanced Multicore Configurations

Advanced Multicore Implementations (More Complex Resource Sharing)

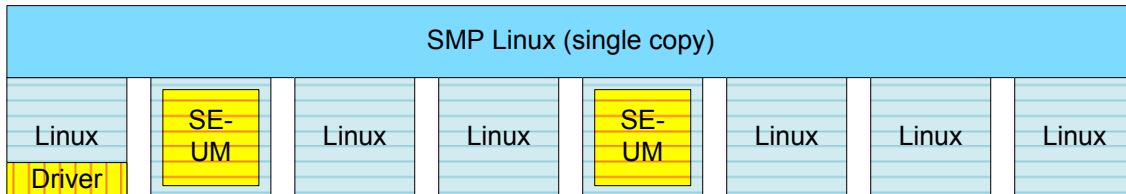
Example of 8-core hybrid system: 3 cores run Linux or another SMP OS; 2 cores run a non-SMP OS; 3 cores run SE-S



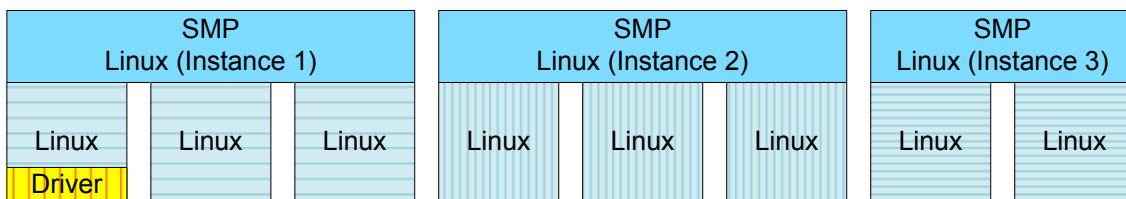
Example of 8-core simple system running a non-SMP OS: 8 separate instances of another OS



Example of 8-core simple system running Linux with 2 Simple Executive User-Mode (SE-UM) applications: a single Linux instance runs on 8 cores. An application such as `linux-filter` runs as a Simple Executive (SE-UM) application on cores 1 and 4 (started by the `oncpu` command).



Example of 8-core hybrid system: 8 cores may run 3 different Linux instances (advanced). Note only 1 Cavium Networks Ethernet driver may be run. (This configuration is not recommended.)



Note: Only a few cores of the maximum possible cores are shown in this example.

5.3 Application Entry Point and Startup Code

An application such as `linux-filter` may be compiled as either an SE-S or SE-UM application without modification.

The code executed when the application is started is not the same: when SE-S is the build target, the file `cvmx-app-init.o` is linked into the target. When Linux SE-UM is the build target, the file `cvmx-app-init-linux.o` is included instead. The makefile `$OCTEON_ROOT/executive/cvmx.mk` is responsible for making this change.

```

ifeq (linux,$(findstring linux,$(OCTEON_TARGET)))
OBJS_$(d) += \
    $(OBJ_DIR)/cvmx-app-init-linux.o
else
OBJS_$(d) += \
    $(OBJ_DIR)/cvmx-interrupt.o \
    $(OBJ_DIR)/cvmx-interrupt-handler.o \
    $(OBJ_DIR)/cvmx-app-init.o \
    $(OBJ_DIR)/cvmx-malloc.o
endif

```

Additionally, `main()` is renamed to `appmain()` if the example is build as a Linux SE-UM application. The makefile `$OCTEON_ROOT/common.mk` is responsible for making this change. See Section 4.3.5 – “Simple Executive API Calls From Linux”

The following two tables are a simplified view of the application entry point and startup functions.

The following table shows a simplified view of SE-S application entry point and startup functions.

Table 4: SE-S Application Entry Point and Startup

Simple Executive Standalone (SE-S) Entry Point and Startup Functions	
<code>__cvmx_app_init()</code>	Application entry point. Defined in <code>cvmx-app-init.c</code> .
<code>main()</code>	Defined in applicaton code such as <code>linux-filter.c</code> . Called after <code>__cvmx_app_init()</code> .
<code>cvmx_user_app_init()</code>	Called by <code>main()</code> , defined in <code>cvmx-app-init.c</code> .

The following table shows a simplified view of Linux SE-UM application entry point and startup functions.

Table 5: Linux SE-UM Application Entry Point and Startup

Simple Executive User-Mode (SE-UM) Entry Point and Startup Functions	
main()	Application entry point. Defined in <code>cvmx-app-init-linux.c</code> .
appmain()	Defined in application code such as <code>linux-filter.c</code> : "main" is aliased to "appmain" by <code>common.mk</code> .
<code>cvmx_user_app_init()</code>	Called by <code>appmain()</code> , defined in <code>cvmx-app-init-linux.c</code> .

5.4 Booting SE-S or SE-UM Applications

To boot Simple Executive applications:

- for SE-S applications: `bootoctl` bootloader command
- for SE-UM applications: `oncpu` Linux command or invoke the application from the command line (for example `./linux-filter`)

These commands may be used to boot on one or more cores. In the following section, booting on more than one core is discussed. Details of the `oncpu` command are provided in that section.

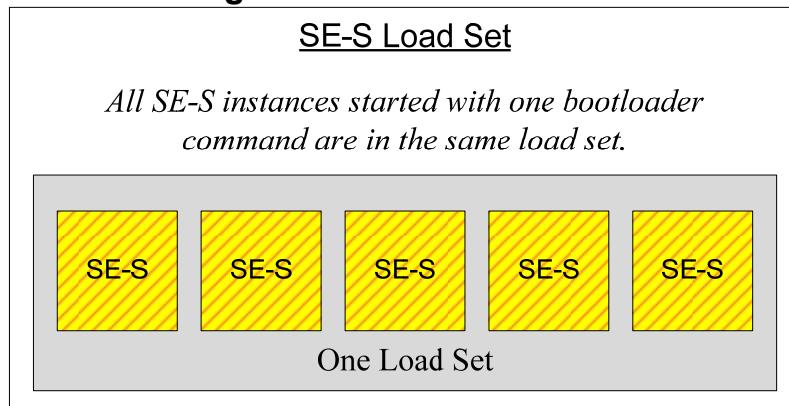
5.5 Booting One ELF File on Multiple Cores: Load Sets

Usually one Simple Executive application is run on multiple cores, booted by the same load command:

- for SE-S applications, using the same `bootoctl` bootloader command for all relevant cores
- for SE-UM applications, using the same `oncpu` Linux command for all relevant cores

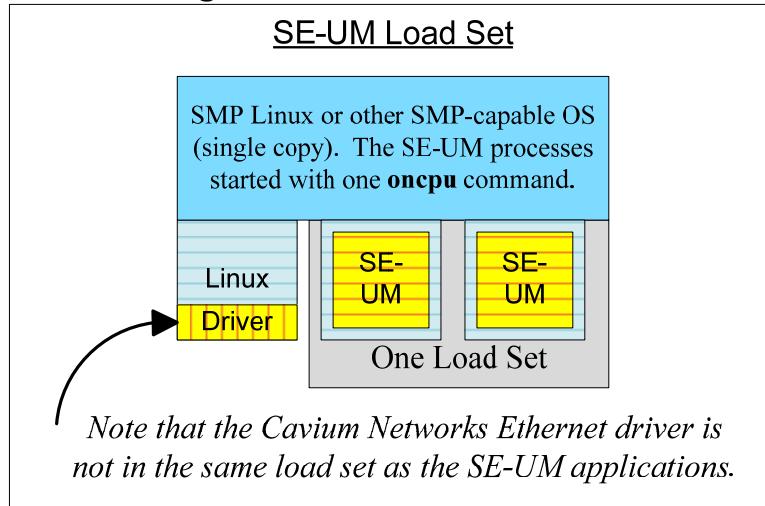
All cores booted by the same load command are in the same load set. The following figure shows cores running Simple Executive Standalone in a load set.

Figure 10: SE-S Load Set



The following figure shows cores running two Simple Executive User-Mode processes in a load set.

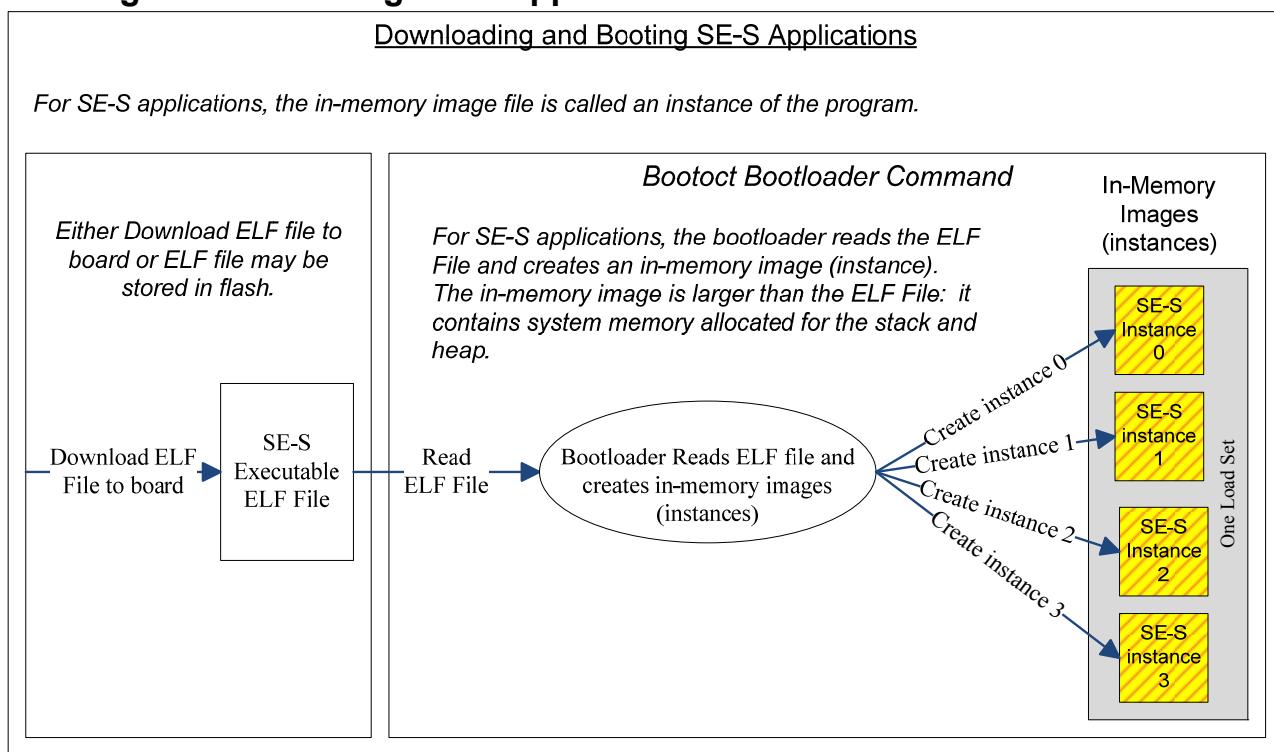
Figure 11: SE-UM Load Set



Load sets are discussed in more detail in Section 8.1.1.2 – “The *cvmx_shared* Section”

5.5.1 Starting SE-S Applications With the `bootoctl` Command

Starting Simple Executive Standalone applications with the `bootoctl` command is straightforward. An example is provided in the *SDK Tutorial* chapter. This command is discussed in more detail in Section 16.2 – “Booting the Same SE-S ELF File on Multiple Cores”.

Figure 12: Booting SE-S Applications With the `bootoctl` Command

SW OVERVIEW

5.5.2 Starting Linux With the `bootoctllinux` Command

Linux may be started with the `bootoctllinux` command is straightforward. An example is provided in the *SDK Tutorial* chapter. This command is discussed in more detail in Section 5.5.2 – “Starting Linux With the `bootoctllinux` Command”.

5.5.3 Starting SE-UM Applications With the `oncpu` Command

Usually the `oncpu` utility may be used to start a SE-UM application on Linux. The `oncpu` utility takes as arguments the core or coremask and name of the application to start. (The words CPU and Core are equivalent.)

```

oncpu <core> command
or
oncpu <coremask> command

```

`Core` is a decimal number from 0 to one less than the number of cores in the system; `Coremask` must be a hexadecimal number specified as `0xXXXX`. Core 0 is represented by the lowest bit in the mask.

Note: `oncpu` takes a virtual core number. This number can be different from the hardware core number. For instance, if SMP Linux is running on cores 4, 5, and 6, the kernels virtual core numbers are 0, 1, and 2.

To start a SE-UM application on core 5, the command would be:

```
oncpu 1 application
```

To start the application on all 3 cores, the command would be:

```
oncpu 0x7 application
```

In the traditional Linux use of `oncpu`, if the coremask contains more than one core, then the process may run on any of the cores in the coremask. This is a way of limiting the process to a subset of the available cores.

When `oncpu` is used to start a SE-UM application on multiple cores, the SE-UM application begins to run on only one core. Once the process begins to run, `main()` (defined in the Simple Executive source file `cvmx-app-init-linux.c`) will `fork()` one instance of the SE-UM application for each additional core, and use `sched_setaffinity()` to bind each process to one core. The result is one SE-UM process for each core. This is very different from traditional Linux applications where only one process is run on the cores in the coremask. See the next figure for an illustration of the difference between using traditional Linux processes and SE-UM processes with `oncpu`.

The set of processes created by one `oncpu` command is referred to as a *load set*. This set of processes shares the text, read-only data, and `cvmx_shared` sections. They also have set-awareness via the `sysinfo` data structure. This benefit is lost if multiple `oncpu` commands are used to start the same process on multiple cores. More information is provided on these features later in this chapter.

Note that while `linux-filter` is a good example of how `oncpu` may be used, the example `named-block` is not a good example. In the `named-block` code, once the forked process begins to run a test is made:

```
if (!cvmx_coremask_first_core(cvmx_sysinfo_get()->core_mask))
    return 0;
```

This test causes each program which is not the running on the *first* core to return without doing anything.

The processes may be seen using the `ps -ef` command on the target.

Note: If you run a SE-UM application without oncpu it will run on all cores under the control of Linux. The default coremask contains all cores under the control of Linux. This is therefore equivalent to calling oncpu with a coremask of all cores.

Figure 13: SE-UM Applications Started With `oncpu` on Multiple Cores

Using the `oncpu` command to start SE-UM Applications

After booting Linux, the `oncpu` command may be used to start SE-UM applications.

When more than one core is specified in the `coremask` argument to `oncpu`, one instance of the SE-UM application will be run on *each* specified core.

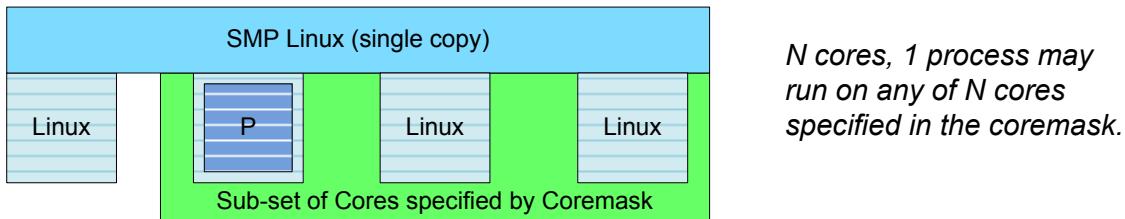
This special processing begins when the SE-UM application begins to run on a core. The function `main()` will `fork()` until a copy of the SE-UM process is running on every core which has the corresponding bit set in the `coremask`. The `main()` will call the function `sched_setaffinity()` to bind each SE-UM process to one core.

The set of SE-UM processes started by one load command is called a load set. All cores in the load set share `.text`, read-only data (`.rodata`), and the `cvmx_shared` section.

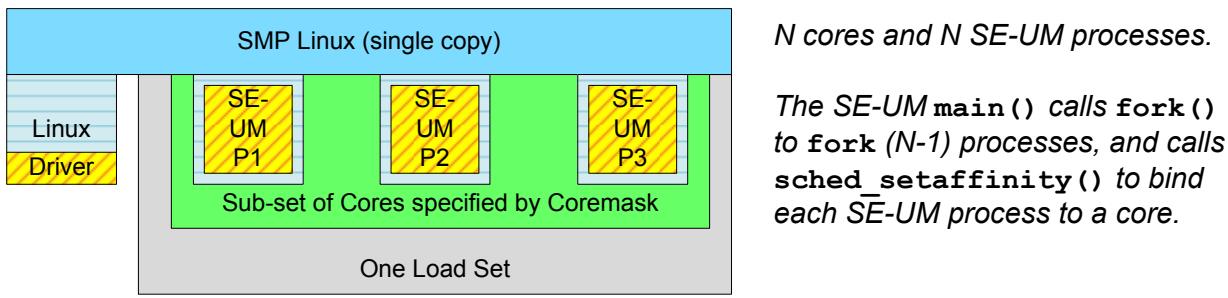
All cores in the load set have set-awareness through the `sysinfo` data structure.

If the SE-UM application is started from the command line (for instance:
`target# ./linux-filter`), then `main()` will start one instance of the SE-UM application on each SMP Linux core.

Traditional `oncpu` use: WITHOUT A SE APPLICATION: `oncpu 0xE non-SE_app`



Cavium Networks result of using `oncpu` WITH a SE-UM application: `oncpu 0xE SE_app`



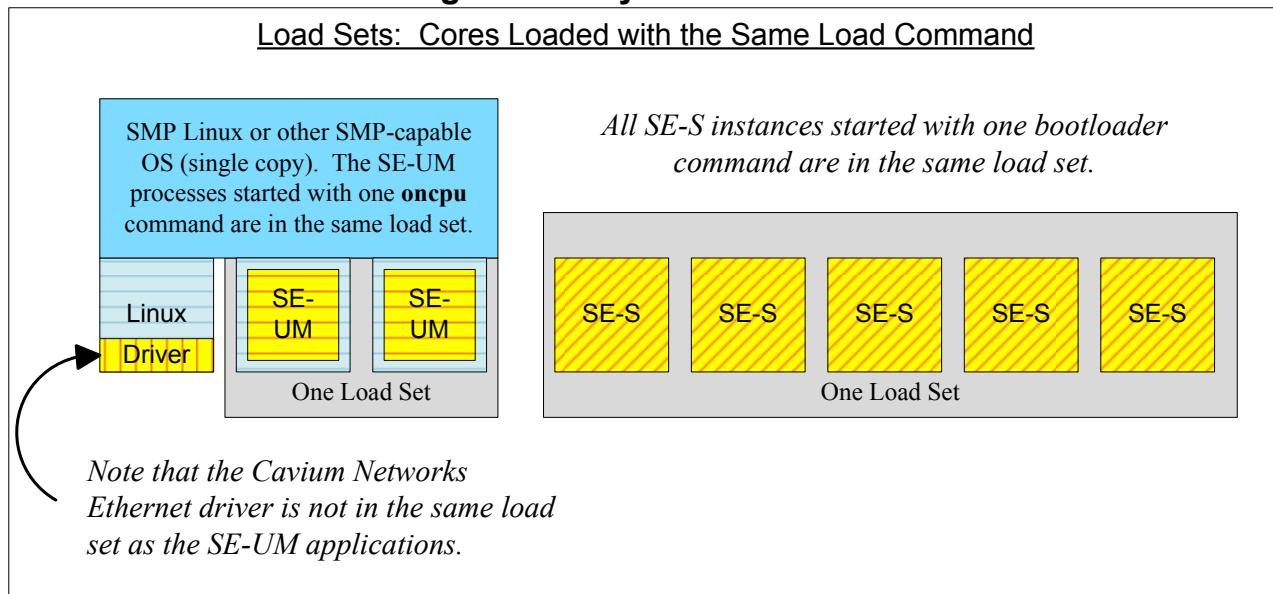
An example of using the `oncpu` command is presented in the *SDK Tutorial* chapter.

Details may be found in the SDK document “*Linux Userspace on the OCTEON*” in the section “*Controlling Core Affinity With `oncpu`*”.

5.6 Booting Different ELF Files

If the system is a hybrid system with both Simple Executive and Linux, the Linux ELF file is started separately on a different set of cores than the Simple Executive ELF file. Although Linux can start a Simple Executive User-Mode Application, the most efficient way to run Simple Executive is Standalone to avoid overhead added by Linux.

Figure 14: Hybrid Load Sets



The SDK Tutorial includes an example of booting two different ELF files (SMP Linux and `linux-filter`), and also an example of running `linux-filter` as a Simple Executive User-Mode Application.

If multiple load sets are used, as shown in the figure above, load the application on core 0 last. Once the application is loaded onto on core 0, the other cores come out of reset and begin to run their applications.

5.7 Synchronizing Multiple Cores

Synchronization between cores is critical, especially at system start-up when one core initializes the hardware, and the other cores must wait until the initialization is complete.

There are three different synchronization environments, depending on how the cores were loaded:

1. Between cores in the same load set
2. Between cores in different load sets
3. SMP Linux cores

This section will provide more detail on these differences.

5.7.1 Synchronizing Cores in the Same Load Set

The first synchronization environment is between cores started by the same load command, a “load set”. The `sysinfo` data structure for each of these cores includes common synchronization information.

All system initialization should be done by one core only, usually the first core in the core mask for the application (if all cores are in the same load set). For SE-S or SE-UM applications which are in the same load set, the function `cvmx_barrier_sync()` will cause the other cores to wait until the initialization is complete.

For example, the following code from the passthrough example checks whether the code is running on the first core in the core mask. If so, then the code initializes the hardware.

```

sysinfo = cvmx_sysinfo_get();
coremask_passthrough = sysinfo->core_mask;

/*
 * Elect a core to perform boot initializations, as only
 * one core should perform this function.
 *
 * cvmx_coremask_first_core returns 1 if this code is running on the first
 * core in the core mask.
 */
if (cvmx_coremask_first_core(coremask_passthrough))
{
    if ((result =
        application_init_simple_exec(packet_termination_num+64)) != 0)
    {
        printf("Simple Executive initialization failed.\n");
        printf("TEST FAILED\n");
        return result;
    }
}
/* Wait until all cores in the given core mask have reached
 * this point in the program execution before proceeding.
 */
cvmx_coremask_barrier_sync(coremask_passthrough);
.
.
```

5.7.2 Synchronizing Cores in Different Load Sets

The second synchronization problem is between cores started by different load commands. In this case, some special techniques are used. It is not possible to use bootmem global memory (discussed in Section 11 – “Allocating and Using Bootmem Global Memory”) to create a shared spinlock to use in initial synchronization because as of SDK 1.7.3, there is no function which allocates the memory and atomically initializes it to a specific value. Thus there is no way to initialize a spinlock for SE-S applications.

A common technique is to have the initializing core initialize IPD last. The other cores can check to see if the IPD has been enabled, as in the following code from the `linux-filter` example:

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

If there is further local initialization after the hardware initialization, the initializing application could send a message via “work” to the waiting application. The waiting application could wait for the IPD initialization, then perform the `get_work` operation to get the message that initialization is now complete.

5.7.3 SMP Linux Synchronization

When running Linux on multiple cores, the cores all jump to the start address of the kernel, then look at the core number. If the code is not running on the first core, the code spins waiting for the first core to finish initializing the hardware and then change a variable in memory which will bring all the other cores out of the loop at the same time.

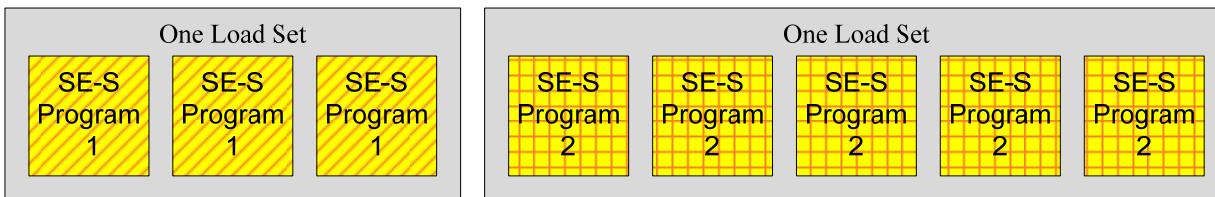
5.7.4 Multiple SE-S or SE-UM ELF Files (Not Recommended)

The following configuration will work, but is not recommended. In this configuration, two separate SE-S ELF files are booted, creating two different load sets. The two different load sets will not share the `sysinfo` data structure, making it difficult to synchronize the cores, adding coding complexity. In addition to this problem, more system memory is consumed because the different load sets cannot share the `.text` and read-only data (`.rodata`) segments of the code. For more information, see Section 11.3.1 – “The `cvmx_shared` Section is Not Always Shared”.

Figure 15: Multiple SE-S ELF Files (Not Recommended)

An Example of an Inefficient SE- S Configuration

Eight Cores run two different SE-S programs.



*This configuration is inefficient. If it is used, the Simple Executive processes will not have group awareness through their **sysinfo** data structures, making system startup and other times the cores need to synchronize with each other difficult to manage.*

This configuration also consumes more system memory: instances in different load sets will not share .text and read-only data (.rodata) sections.

Pipelining can be done without dividing the packet processing into different programs, one per core. Pipelining can be done with only one Simple Executive ELF file as shown in Figure 29 – “Modified Pipelining”.

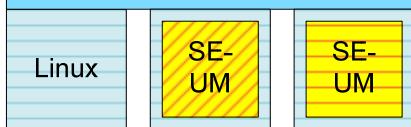
Similarly, multiple SE-UM ELF files are also not recommended, for the same reasons.

Figure 16: Multiple SE-UM ELF Files (Not Recommended)

An Example of an Inefficient SE- UM Configuration

Two Cores running different SE-UM programs.

SMP Linux or other SMP-capable OS (single copy). The SE-UM processes were started with two different **oncpu** commands.



*This configuration is inefficient. If it is used, the Simple Executive instances will not have group awareness through their **sysinfo** data structures, making system startup and other times the cores need to synchronize with each other difficult to manage.*

This configuration also consumes more system memory: the different instances will not share text and read-only data (.rodata) sections.

6 Software Architecture

When designing the software, it helps to separate two basic types of processing: normal packet processing (fast path), and exception processing (slow path).

Depending on the number of cores available, different configurations of cores devoted to either fast path or slow path processing can be used to optimize throughput.

This is a brief discussion of the issues and choices.

6.1 Control-Plane Versus Data-Plane Applications

Application functions may be divided into two categories: control plane (slow path), and data plane (fast path). The control plane usually handles exceptions. The data plane handles normal packet processing.

SE-S applications may be used for both control plane and data plane. SE-S applications provide the lowest overhead and highest potential for scaling. The next best solution (a typical solution) is SE-UM for control plane and SE-S for data plane.

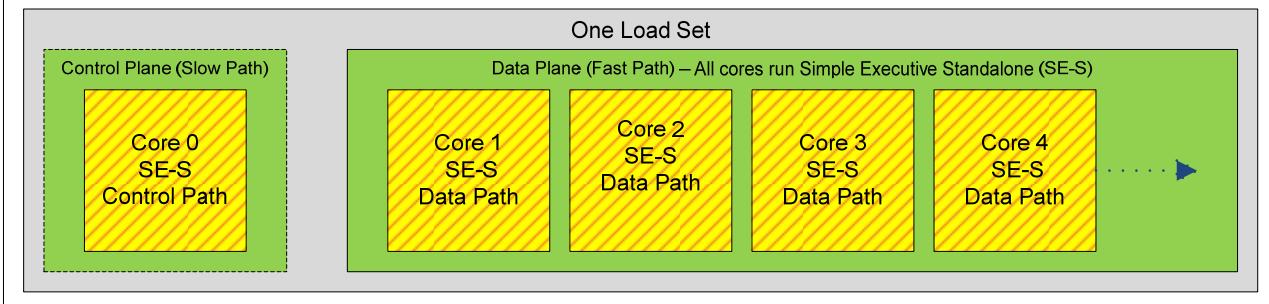
If necessary, SE-UM applications may be used for both control plane and data plane. This solution is sometimes necessary if there is only one core, and the application cannot be ported to Simple Executive.

The fastest multicore solution is to run one Simple Executive load set on all the cores. Note that only ONE Simple Executive ELF file has been downloaded to run on multiple cores, even if some cores are responsible for slow path and others responsible for fast path processing.

When running multicore applications, only one core does the initialization routine.

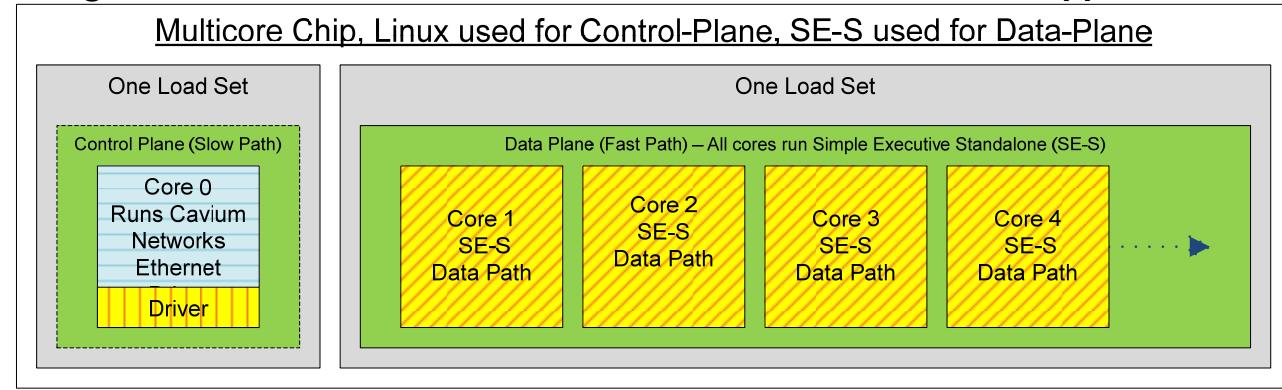
Figure 17: SE-S Used for Both Control-Plane and Data-Plane Applications

Multicore Chip, SE-S used for both Control-Plane and Data-Plane



Or Linux may run on one or more cores, with Simple Executive on the others.

Figure 18: Linux for Control-Plane and SE-S for Data-Plane Applications



6.2 Event-driven Loop (*Polling*) Versus Interrupt-Driven Loop

There are two different models for receiving packets to process: an event-driven loop (polling) or an interrupt-driven loop.

An event-driven loop looks like:

```
while (there is work do do)
{
    do the work
}
```

Typically, OCTEON programmers design software to use the event-driven loop. The Cavium Networks Ethernet Driver uses a hybrid of an interrupt-driven and event loop. In this loop, the driver sleeps when there is no work to do. When there is more work to do, an interrupt is sent to the driver. Then the driver processes all the work available until there is no more work to do.

The following code fragment shows the event-driven loop used in `linux-filter` when the code is run as a SE-S application:

```
while(1)
{
/* In standalone CVMX, we have nothing to do if there isn't work,
so use the WAIT flag to reduce power usage */
    cvmx_wqe_t *work = cvmx_pow_work_request_sync(CVMX_POW_WAIT);
    if (work == NULL)
        continue;
    . . .
```

The following code fragment shows an event-driven loop used in `linux-filter` when the code is run as a SE-UM application. Note that this code performs the `get_work` operation, bypassing the Cavium Networks Ethernet Driver.

```

while (1)
{
    cvmx_wqe_t *work = cvmx_pow_work_request_sync(CVMX_POW_NO_WAIT);
    if (work == NULL)
    {
        /* Yield to other processes since there is no work to do */
        usleep(0);
        continue;
    }
    . . .
}

```

The event-driven loop is a higher performance processing architecture than the interrupt-driven loop. In an event-driven loop, when the core is ready for work and work is available, it gets the work; when there is no work, the core loops looking for work to do. When using an interrupt-driven loop, there may be a delay between work available and the process being notified. SSO (POW) interrupts are configured based either on a time counter or the quantity of work available for a particular group (via the POW_WQ_INT_CNT registers). Instead of looping looking for work, the interrupt-handler thread exits, then is called again when the interrupt occurs. This not only can result in work being processed less quickly, but also results in more context switches, costing unnecessary system overhead.

The Cavium Networks Ethernet driver uses a modified interrupt-driven loop: once the interrupt occurs, the receive function performs the `get_work` operation to receive up to 60 packets, then exits. This is done to prevent the transmit function from being starved for CPU time. This code is also not as efficient as an event-driven loop. The Cavium Networks Ethernet driver code is located in `$OCTEON_ROOT/linux/kernel_2.6/linux/drivers/cavium-ethernet`.

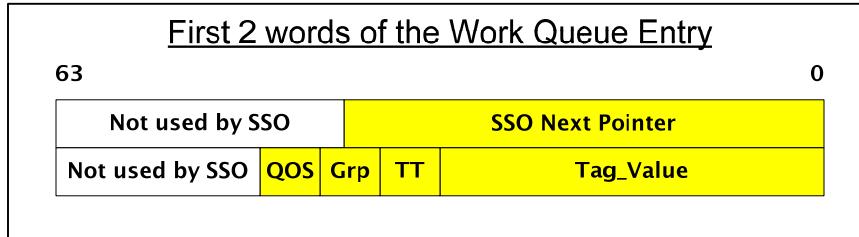
6.3 Using Work Groups in Packet Processing

Work Groups were previously mentioned in the *Packet Flow* chapter.

6.3.1 Work Groups

The Work Queue Entry data structure was introduced in the *Packet Flow* chapter. This data structure contains a field “Grp” which stands for Group (Work Group). The group number is set by the PIP/IPD Unit, based on the settings of its configuration register when the packet is received. Group values range from 0-Y where Y is one less than the number of groups supported by the OCTEON model.

Figure 19: The First Two Words of the Work Queue Entry



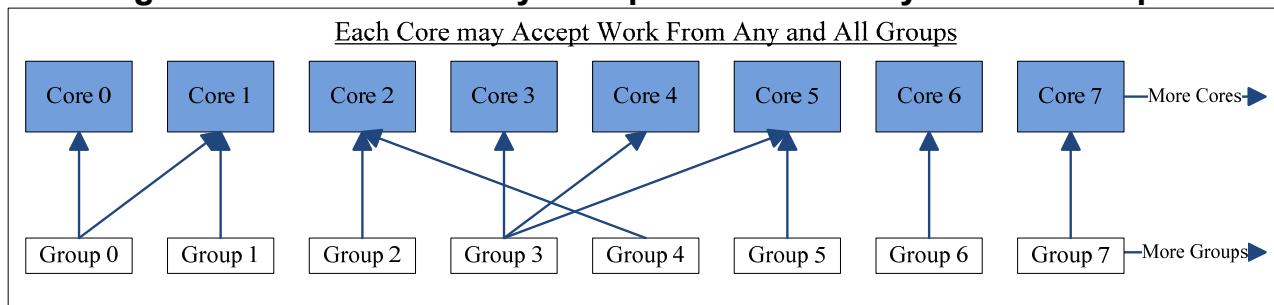
6.3.2 Configuring the Per-Core Group Mask in the SSO Scheduler

When a core performs a `get_work` operation, the request goes to the SSO Scheduler.

The SSO Scheduler maintains a per-core group mask. This group mask has one bit set for each group the core will accept work from. Cores may accept work from any or all work groups. When the scheduler receives the `get_work` request, it will schedule the highest priority WQE which is, based on its group, schedulable to the core.

Cores may receive work from any and all groups. Multiple cores may receive work from the same work group. This technique provides easy load-balancing, and also allows the creation of a special type of work, such as monitoring information, which can be processed by only one core.

Figure 20: Each Core May Accept Work from Any and All Groups



The simplest way to set the core's group mask is by using the Simple Executive function `cvmx_pow_set_group_mask()`. The arguments to this function are the core number and the `group_mask` for the core. An example of using this function is presented in the *SDK Tutorial* chapter.

The `cvmx_pow_set_group_mask()` function modifies the per-core SSO (POW) registers: `POW_PP_GRP_MSK(N)`, where N represents the core number: on a 16-core system N ranges from 0-15. (“PP” stands for “packet processor”, which simply means “core”.)

Inside the `POW_PP_GRP_MSK(N)` register, the field `GRP_MASK` is used to control which groups the core accepts work from. Each bit in the `GRP_MSK` represents a group: if bit 0 in the mask is set, group 1 work is accepted, and so on. There are Y groups. Typically (but not always), $N = Y$ (the number of groups matches the number of cores in the system). Each group is represented by a bit in the mask. Group 0 is represented by $1 \ll 0$. Group 15 is represented by $1 \ll 15$.

When the core performs the `get_work` operation, only work with a group number corresponding to a bit set in the core's `GRP_MSK` is returned.

In the following table, core 0 is configured to only receive work with group number 15. Core 1 is configured to receive work from groups 0 and 14. (The `linux-filter` example uses this configuration.)

Table 6: Setting the Cores's Group Mask in the SSO

Group Mask [GRP_MSK]																	Notes
Group	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0	
Core. ..	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Only group 15 work is schedulable to this core.
	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	Only groups 0 and 14 work is schedulable to this core.
	2																
	3																

Once the group mask is set, the scheduler will only return the highest-priority work which can be scheduled to this core.

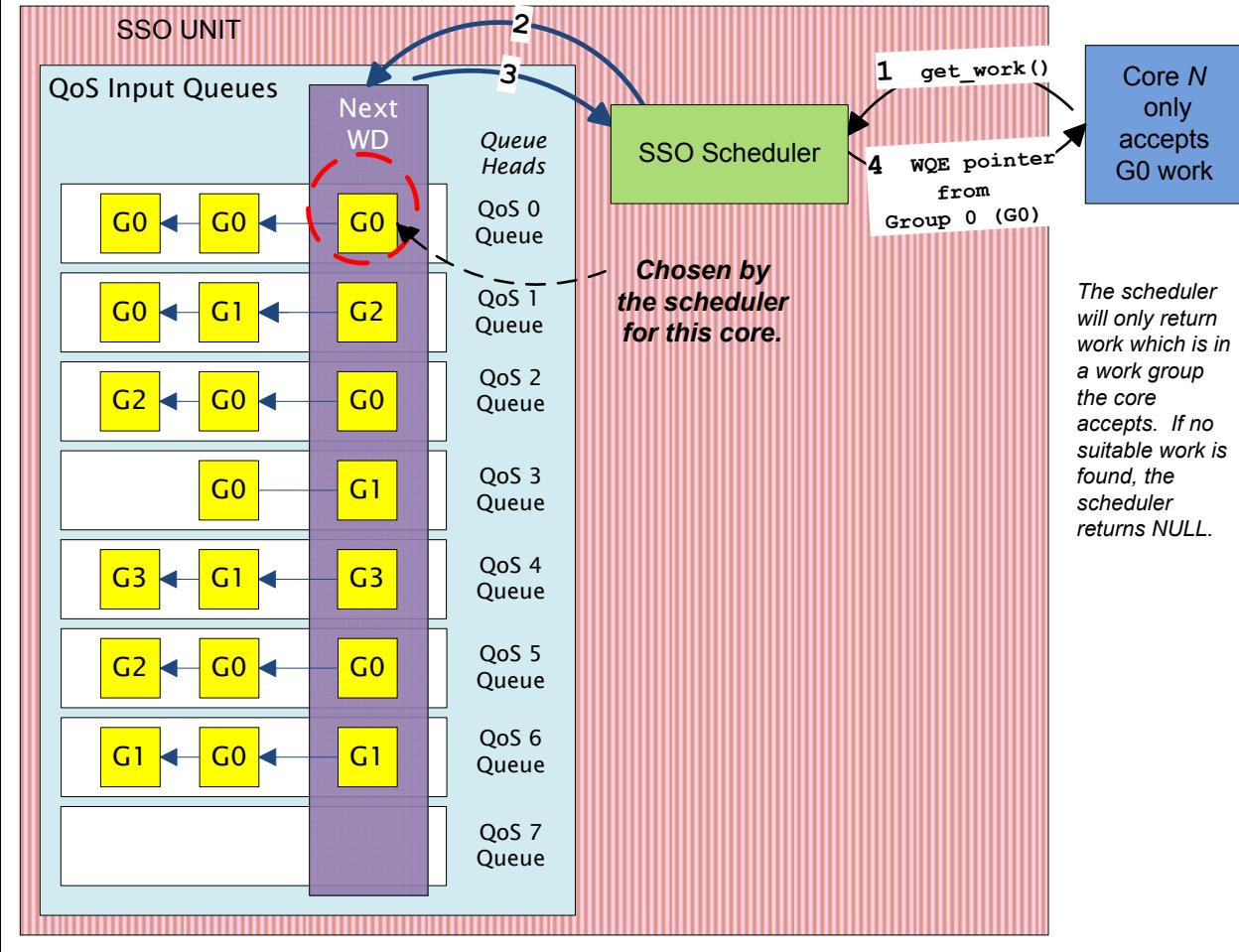
The cores may also be configured to accept work from a limited set of QoS Input Queues, and to adjust the priority of the QoS Input Queues they accept work from. Inside the `POW_PP_GRP_MSK(N)` register, the fields `QOS[N]_PRI` (one for each QoS priority) is used to control the QoS Input Queue priority for the core. A value of `0xF` prevents the core from receiving work for that QoS level.

In the following figure, the core will receive the first schedulable group 0 work in from the highest priority QoS Input Queue (as viewed from the core's `QOS[N]_PRI` field). This is a highly simplified view of SSO scheduling based on groups.

Figure 21: Cores Can Receive Work Based on Their Group Mask

The Cores can Receive New Work from Any or All Groups, Depending on their Group Mask

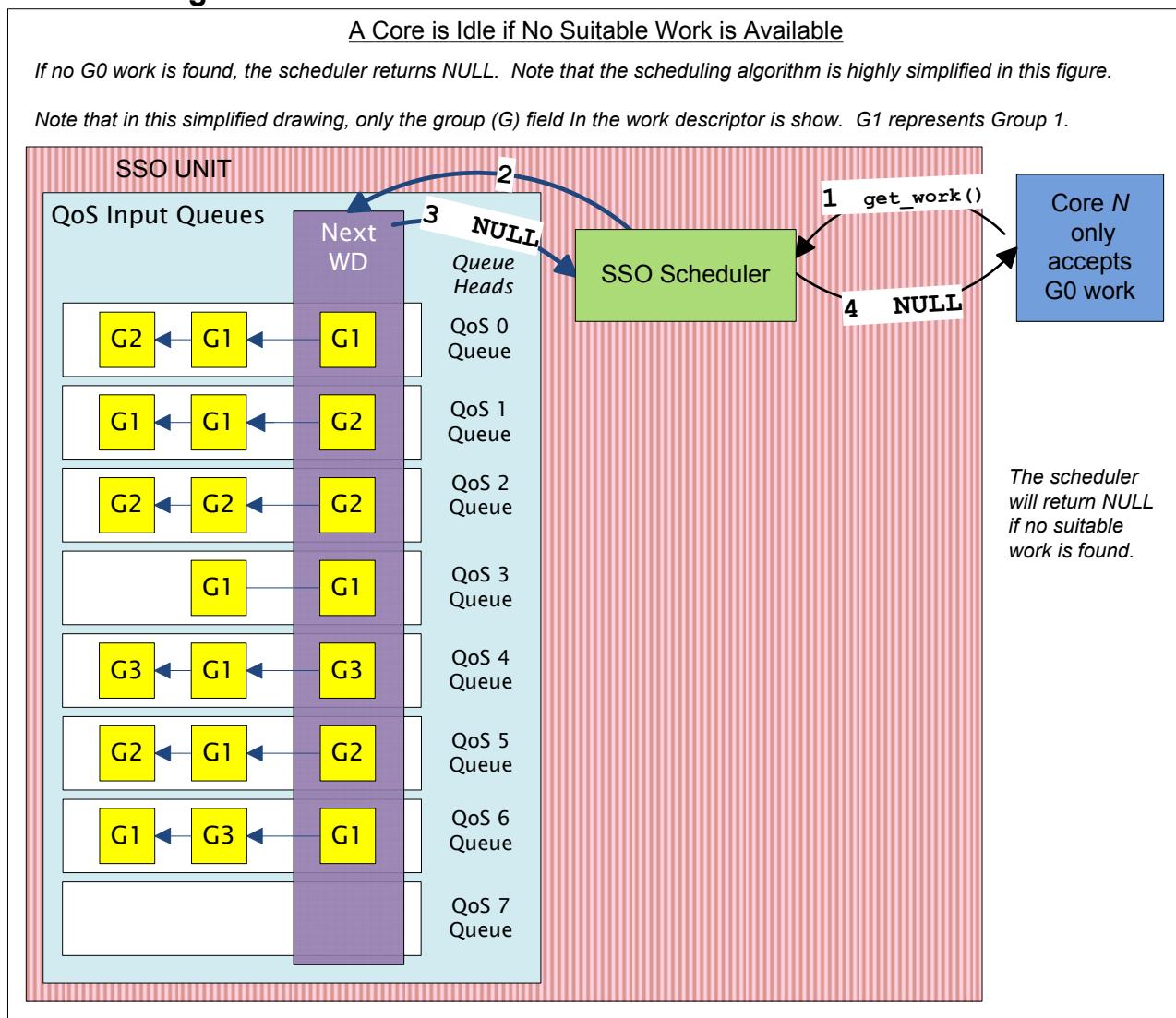
The first schedulable GO Work Descriptor is returned by the scheduler, in this example the WD is from QoS 0 Queue. Note that the scheduling algorithm is highly simplified in this figure.



SW OVERVIEW

Note that the core may be idle if there is work to do, but none of it is in a work group accepted by the core. Part of load-balancing is making sure the cores are as busy as possible. The core in the next figure can be configured to accept work from more groups.

Figure 22: A Core is Idle if No Suitable Work is Available



Groups may be used for many purposes: they are a flexible tool.

The PIP/IPD assigns the initial group number. After the work is assigned to a core, the core may change the group number by performing the `swtag_desched` operation.

6.3.2.1 Passing Work From One Core to Another Core

The following steps are used to pass work from one core to another core:

1. The `swtag_desched` operation deschedules the work from the core. The work remains in the In-Flight Queue so that ordering properties are maintained.
2. The corresponding Work Descriptor (WD) is unscheduled from the core and its state is set to Descheduled.
3. Once the WD is the head of its In-Flight Queue, a pointer to it is stored in the *Descheduled-Now-Ready List (DS-Now_Ready List)*. The WD can now be scheduled to a new core. (There is one DS-Now-Ready List per group. These lists contain only pointers to WDs which are ready to be rescheduled because each is the head of its In-Flight Queue.)
4. A new core will receive the now-ready WD when the core performs the `get_work` operation and the SSO schedules now-ready WD to the core.

The DS-Now-Ready List has a higher priority than the QoS Input Queue, which allows now-ready in-flight work to complete prior to new work.

This technique may be used to pass a packet from one core to another. For example, in `linux-filter`, groups are used to pass messages between data-plane and control-plane cores. This example is presented in Section 6.6 – “Example: `linux-filter`”.

Figure 23: Scheduling Previously Descheduled Work

Scheduling Previously Descheduled Work

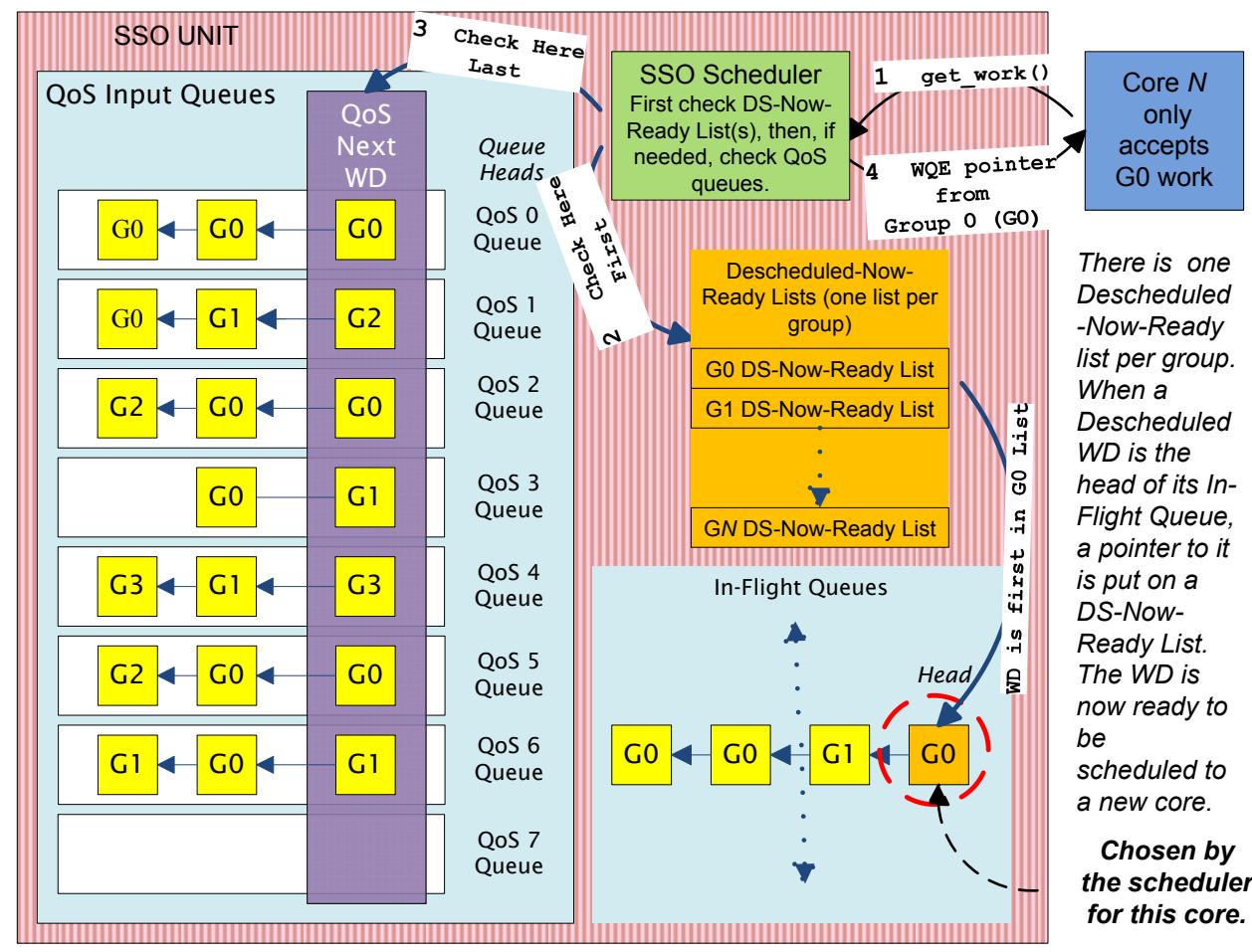
In this example, core N only accepts work from group 0.

SSO Scheduler Steps Shown in this Figure:

1. Core N performs the `get_work()` operation, accepting work only from group 0 (G0).
2. If the Group 0 Descheduled-Now-Ready List (DS-Now-Ready List) is not empty, the scheduler removes the entry from the DS-Now-Ready List and uses the corresponding Work Descriptor (WD) in step 4. The scheduler goes to step 4. (As shown in the figure.)
3. Else there are no entries on the G0 Ready List. The scheduler now examines the Next WD entries for the QoS Queues according to a configurable scheduling algorithm, looking for a WD which is suitable for the core. If the scheduler finds a suitable WD, it removes it from the QoS Queue. The scheduler goes to step 4.
4. If a suitable WD was found in either the DS-Now-Ready List or the QoS queues, the scheduler assigns the WD to the core and returns the WQE pointer to the core.
Else (no suitable WD) was found, the scheduler returns NULL.

Note: In this example only the group field (G) of the Work Descriptors (WD) are shown, and only one core is shown.

Note: This view of the SSO Scheduler is simplified: the details of the configurable scheduling algorithm are not shown.



6.4 Pipelined Versus Run-To-Completion Software Architecture

The OCTEON processor supports traditional pipeline, run-to-completion, and modified pipeline architectures. On some processors, system constraints can force the architecture into a pipelined model. For example, some processors can only run a limited number of instructions per core due to limited instruction memory per core. The OCTEON processor does not have this limitation.

Example software architectures supported include:

1. Run-to-completion: In run-to-completion architecture, each core performs all the functions, and the packet stays on the same core as it moves through the series of functions.
2. Traditional pipeline: In traditional pipeline architecture, each core handles one function and the packet moves through the pipeline, changing cores as needed to pass through the series of functions. The stages of the pipeline are bound to specific cores. On the OCTEON processor, when each core completes its part of the processing, it changes the packet's work group to a new value, and performs the `swtag_desched` operation to send the packet to the next core in the pipeline. The next core receives the packet when it performs the `get_work` operation.
3. Modified pipeline: On the OCTEON processor, because there is no limitation on code size, a modification of the traditional pipeline architecture can be used. A modified pipeline is one where any core can process any stage of the pipeline: the stages are not bound to specific cores. This modified architecture provides better load-balancing and scaling capabilities than traditional pipelining.

6.4.1 Comparing Run-To-Completion and Traditional Pipelining

Pipelining can be very nearly as efficient as run-to-completion, measured from a strict performance viewpoint.

The problems arise when writing and maintaining the software: pipe length adjustment, higher context switching overhead, and the need to re-tune the system after adding new functionality:

- For best performance, the processing time of each pipe stage must be about the same length, or else everything will stack up at the entry to the slowest stage. While that problem can be mitigated by adding cores to the slower stages (in modified pipelining), it's a long path to tune. In the future, when new functionality is added to a pipe stage then performance degrades, and the system must be re-tuned.
- Pipelining adds context switches (in this case, SSO tag switches) to each packet's path. A simple run-to-completion model can have 2-3 tag switches. Any pipeline model will have at least one tag switch per pipe stage plus ordinary overhead which will still probably be needed.
- Passing the packet from core to core will decrease utilization of the L1 data cache: each core will have to fault in new cache lines as it picks up a new packet. If the same core had continued operating on the packet, the data would still be in the L1 Dcache. The packet will probably still be in the L2 cache, but this is not as efficient as having it in the L1 Dcache. See Section 9.4 – “Caching” for a brief introduction to caching on the OCTEON processor.

- Due to the extra cycles spent in the context switch passing the packet from core to core, traditional pipelined architectures depend on optimizing the L1 instruction cache usage. The instruction/code size must be small enough to fit into the L1 instruction cache to avoid wasting cycles on cache misses.

A simpler run-to-completion model does not have the scaling and maintenance complexity, or the additional overhead of the pipelined model.

6.4.2 A Quick Look at Packet Processing Math

The following example uses a 750 MHz processor, and Ethernet packets with an IMIX average frame size of 353.8 bytes per frame.

To process packets at a line rate of 3.3 Mfps (Million frames per second), which is about 10 Gbps of Ethernet traffic when the bytes times per frame is 374 byte times per frame, there are only 299.2 ns per frame to complete packet processing. Every cycle is precious at this speed.

Figure 24: Packet Processing Math

Calculating Instructions per Packet

As shown in the math below, the number of instructions used to process one packet before the next is received can be small. Every cycle is precious at this speed.

Assumptions:

1. 10 Gpbs Line Rate (10,000,000,000 bits per second)
2. Data traffic is Ethernet
3. Assumed cnMIPS Instructions Per Cycle (IPC) is 1.3 MIPS/MHz (See Note 1)
4. OCTEON Processor with cnMIPS cores each running at 750 MHz

IMIX Average Frame Size:

$$\frac{7 \text{ frames} * \frac{64 \text{ bytes}}{1 \text{ frame}} + 4 \text{ frames} * \frac{570 \text{ bytes}}{1 \text{ frame}} * 1 \text{ frame} * \frac{1518 \text{ bytes}}{1 \text{ frame}}}{12 \text{ frames}} = 353.8 \frac{\text{bytes}}{\text{Frame}}$$

That rounds to 354 bytes per frame.

Each frame at the IMIX average size requires 374 byte times:

IMIX Average Frame Size + Preamble & SFD + Inter-frame Gap

354 bytes + 8 bytes + 12 bytes = 374 byte times per frame

(Preamble is 7 bytes, Start Frame Delimiter (SFD) is one byte.)

The frame rate is thus:

$$\frac{\text{Line Rate}}{\text{Frame size} + \text{Preamble \& SFD} + \text{IFG}}$$

$$\frac{10,000,000,000 \text{ bits}}{1 \text{ s}} * \frac{1 \text{ byte}}{8 \text{ bits}} * \frac{1 \text{ frame}}{374 \text{ bytes}} = 3.342 \text{ Mfps}$$

The available frame processing time, therefore, is:

$$\frac{1}{\text{Frame rate}} = \frac{1 \text{ second}}{3.342 \text{ Mfps}} = 299.2 \frac{\text{ns}}{\text{frame}}$$

From this, we can see how many CPU cycles are available per frame. Each core clock cycle period is:

$$\frac{1}{\text{Frequency}} = \frac{1 \text{ s}}{750 \text{ M cycles}} = 1.3333 \frac{\text{ns}}{\text{cycle}}$$

And the CPU cycles available are:

$$\frac{\text{Available Time per Frame}}{\text{Cycle Period}} = \frac{299.2 \text{ ns}}{1 \text{ frame}} * \frac{1 \text{ cycle}}{1.333 \text{ ns}} = 224 \frac{\text{CPU cycles}}{\text{frame}}$$

Last, the number of cnMIPS instruction per frame assuming an IMIX average, is

$$\frac{\text{CPU cycles per frame}}{\text{Instructions per cycle}} = \frac{224 \text{ CPU cycles}}{1 \text{ frame}} * \frac{1.3 \text{ instructions}}{1 \text{ CPU cycle}} = 291 \frac{\text{instructions}}{\text{frame}}$$

Note 1: The number of instructions executed per cycle may vary greatly depending on the application, compiler optimizations, cache sizes and cache utilization, and the locality of the code.

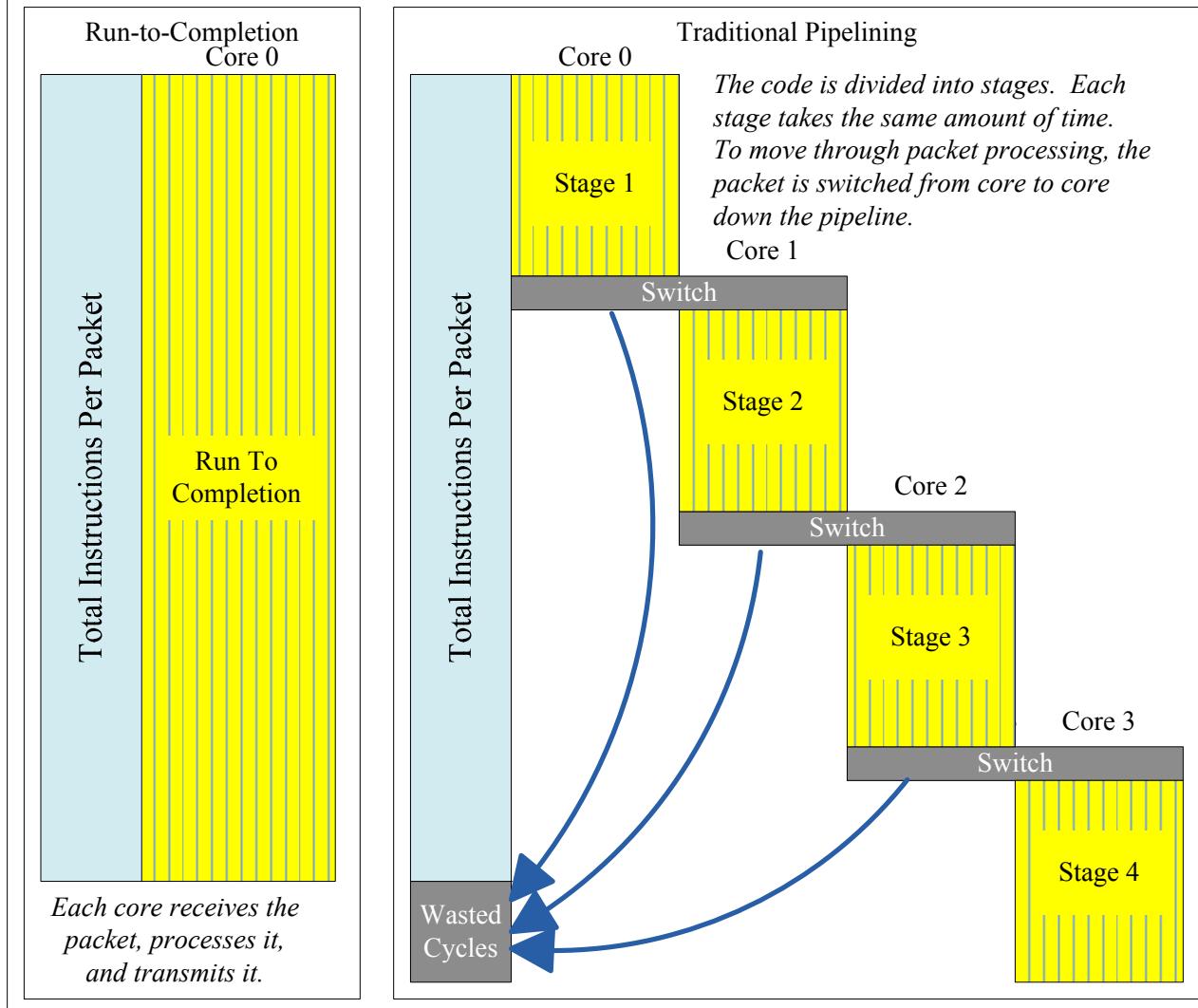
In a traditional pipeline, that means the first stage has to accept a packet every 224 cycles (on a 750 MHz core). There are a total of 3,584 core cycles (16 cores * 224 cycles in each stage) to complete packet processing on this packet (this extra processing time introduces a latency in packet processing). For example, assume the number of instructions per cycle is 1.3. At 1.3 instructions per cycle, 224 cycles is roughly 291 instructions per pipeline stage.

In this tight timeframe (10 Gbps), there is little time to do very much packet processing. To move the packet down the pipeline, the first core performs a `swtag_desched` operation to pass the WQE pointer to the next core. The receiving core performs a `get_work` operation to receive the WQE pointer. This is repeated for each stage in the pipeline. Spending unnecessary cycles on extra operations should be avoided if possible in order to achieve performance goals.

The run-to-completion model minimizes cycles spent on switches. Each core has 3,584 core cycles before it needs to accept another packet (assuming no switches occur).

Figure 25: Run-To-Completion Versus Traditional Pipelining

Run-To-Completion is Useful for High Performance Packet Processing Where Every Cycle Counts

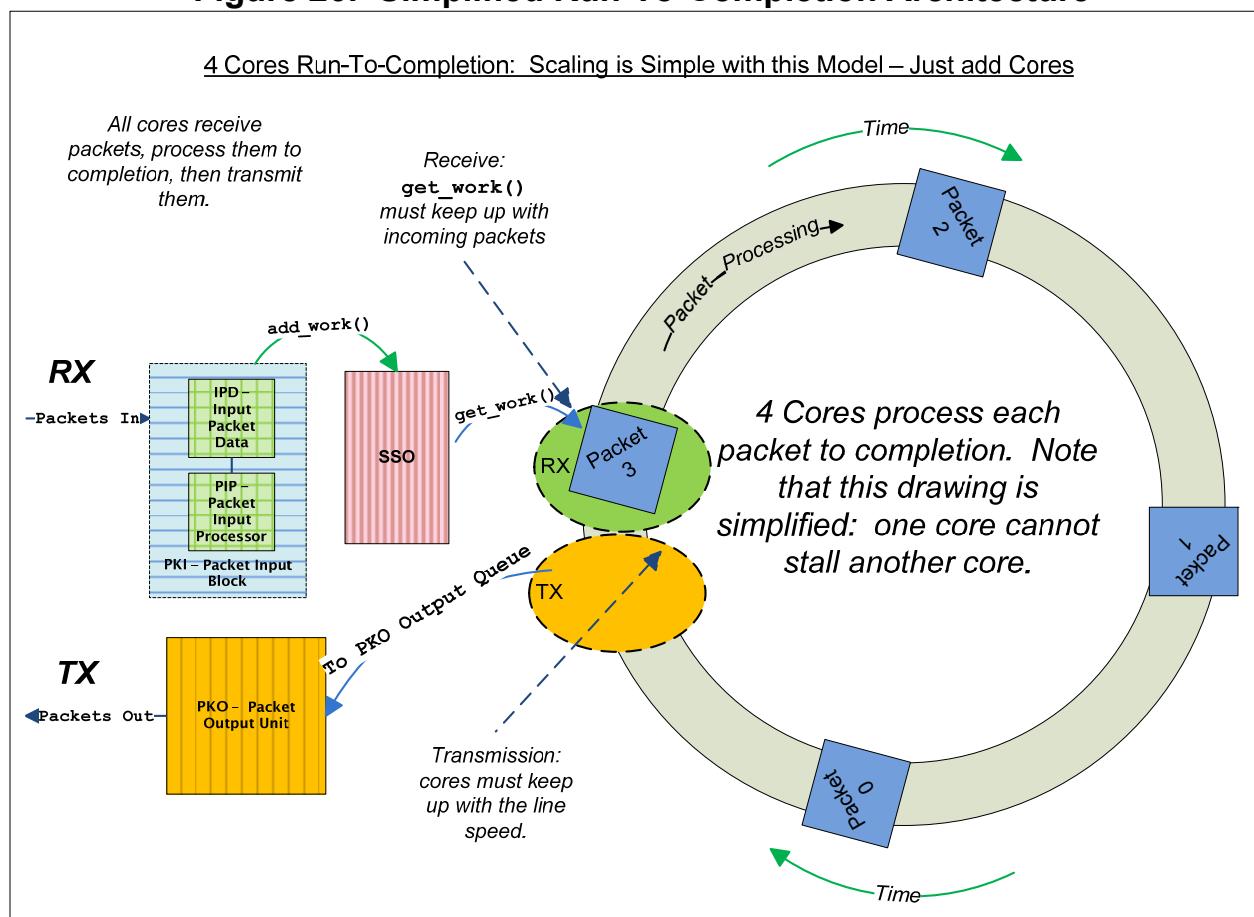


Note that as packet size increases, the packet rate drops dramatically. It is easier to keep up with the line rate when using larger packet sizes. There is a fixed per-packet overhead. Using a larger packet size will reduce the amount of overhead for the same data transfer.

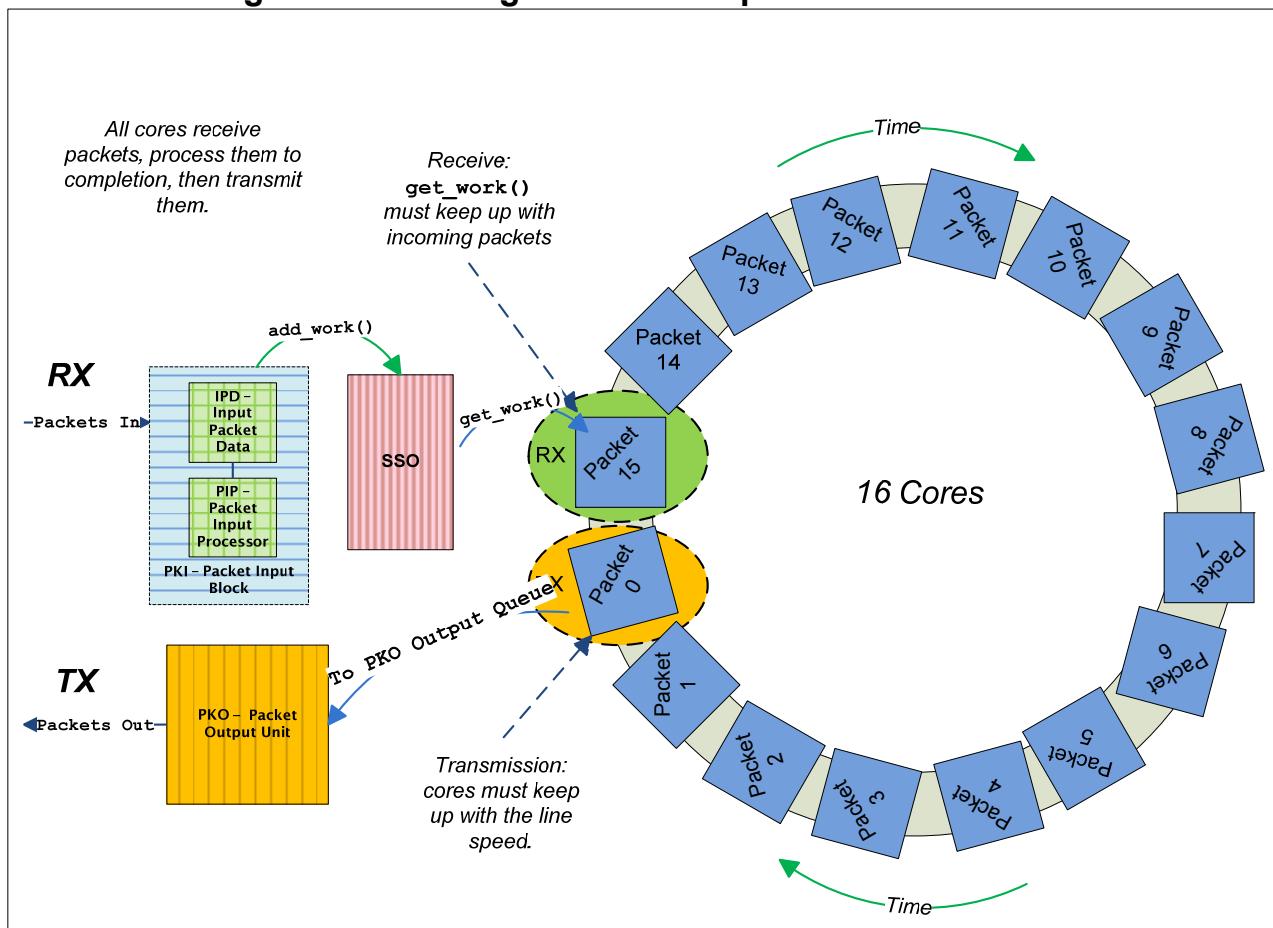
6.4.3 Run-To-Completion

In run-to-completion architecture, each data-plane core runs the same application. Each core may receive new work and process it to completion. One core cannot stall packet processing for the system, only for the single packet involved.

Figure 26: Simplified Run-To-Completion Architecture



Run-to-completion is easy to scale, as shown in the following figures. Simply add cores.

Figure 27: Scaling Run-To-Completion Architecture

SW OVERVIEW

Note that the run-to-completion model also keeps cores busy: if there is something to do, a core will get the work and do it.

Work groups and tag types can be used to route packets to specific cores.

In the figures above, no switches are shown. It is not unusual to use 2-3 switches in the run-to-completion model (for instance, the switch to the ATOMIC tag type to use packet-linked locking). Switches may or may not move the WQE to a different core.

Switches and work groups may be also used to send work between the control plane and data plane.

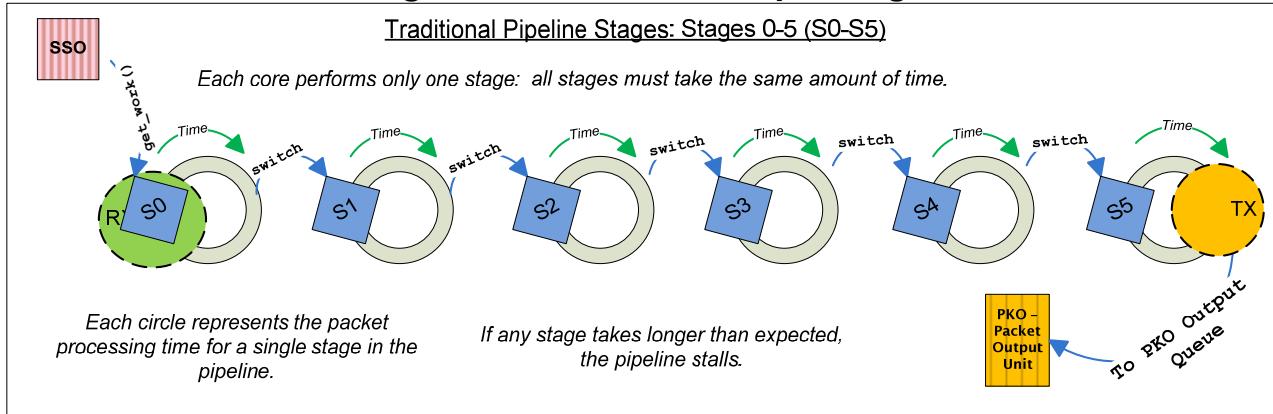
6.4.4 Traditional Pipelining

A simplified view of traditional pipelining is that each core handles part of packet processing, and the packet is passed from one core to the next until processing is complete.

On the OCTEON processor, this might be handled by having each core receive only one group. After each core completes its part of the packet processing, it performs the `swtag_desched`

operation (changing the packet's work group number) to pass the packet to the next core. In traditional pipelining, each core will only accept work from one group.

Figure 28: Traditional Pipelining



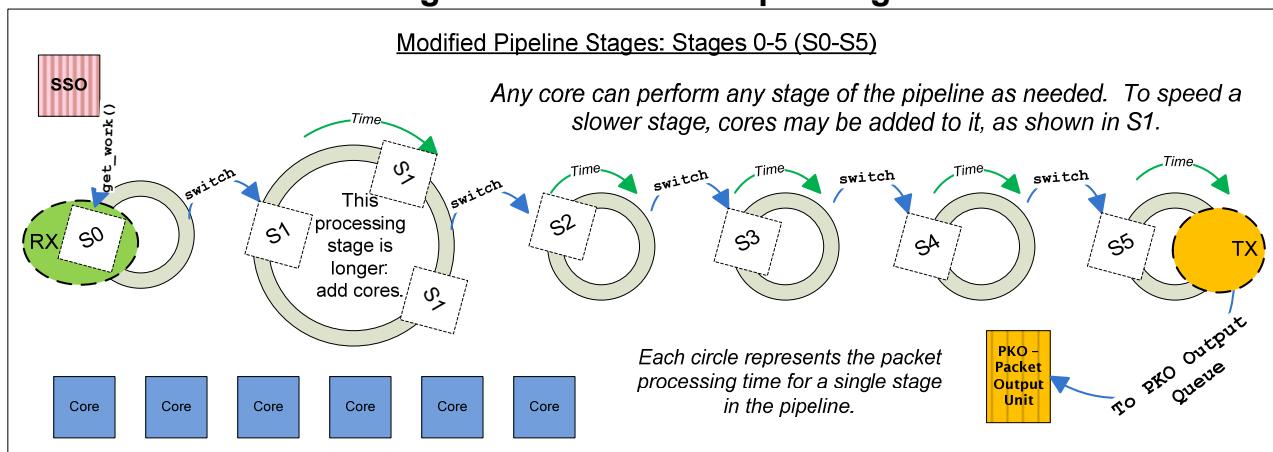
Note that it can be more efficient for one core to get a packet, and do packet processing all the way to completion instead of using stages. This will conserve the extra cycles spent on the `switch_desched` operation. This example is merely being used to illustrate the capability for modified pipelining. Run-to-completion is typically a higher performance architecture.

6.4.5 Modified Pipelining

If a pipelining architecture must be used on the OCTEON processor, the recommended architecture is the modified pipelining architecture.

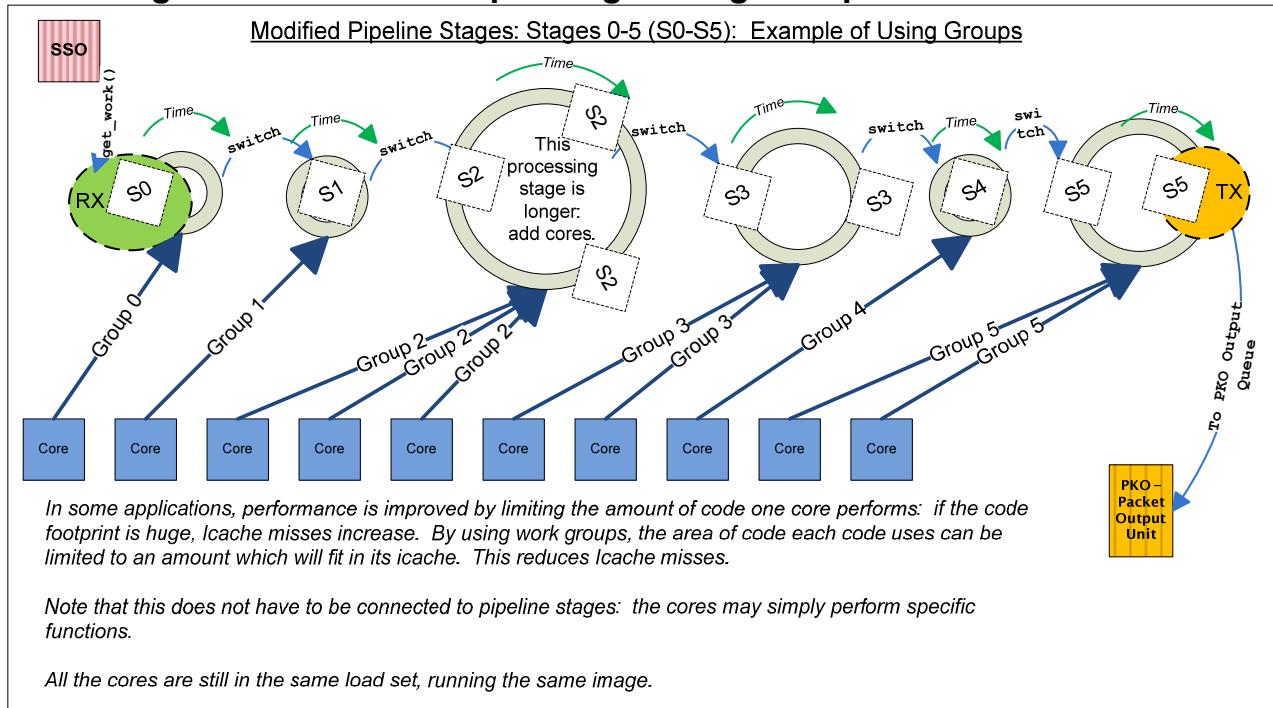
To use modified pipelining, cores may process more than one stage of packet processing. This is easy to implement by modifying the per-core group masks in the SSO.

In this model, each core can run the same application. After the `get_work` operation returns a WQE pointer to the core, the core can execute the appropriate function based on the packet's work group.

Figure 29: Modified Pipelining

SW OVERVIEW

The core's group mask can easily be modified to add or subtract groups. This makes load-balancing and adding new functionality simpler than the traditional pipelining model. This technique also allows cores to be in the same load set, with the shared data and synchronization advantages provided by a single load set.

Figure 30: Modified Pipelining: Using Groups to Load Balance

Note that modified pipelining, like traditional pipelining, has the disadvantage of cycles spent on `swtag_desched` and `get_work` operations. Load balancing is also still a problem when new functionality is added. The run-to-completion model is usually the easiest architecture to load-balance and scale.

6.5 Other Software Architecture Issues

6.5.1 Scaling

A key software architecture goal is to create software which scales well. Scaling refers to adding cores to a system to improve throughput. This is often needed as the system throughput needs increase over time.

In traditional pipelined processing, there is one function to a core: a static core allocation. This hard-coded architecture is difficult to tune for performance, to load-balance, and to scale.

Both run-to-completion and modified pipelining scale well. By using groups and tag types, software architecture can be created which scales well, is easy to tune for performance, and is easy to load-balance. A well-designed system will easily scale when cores are added.

Key elements of a well-designed system:

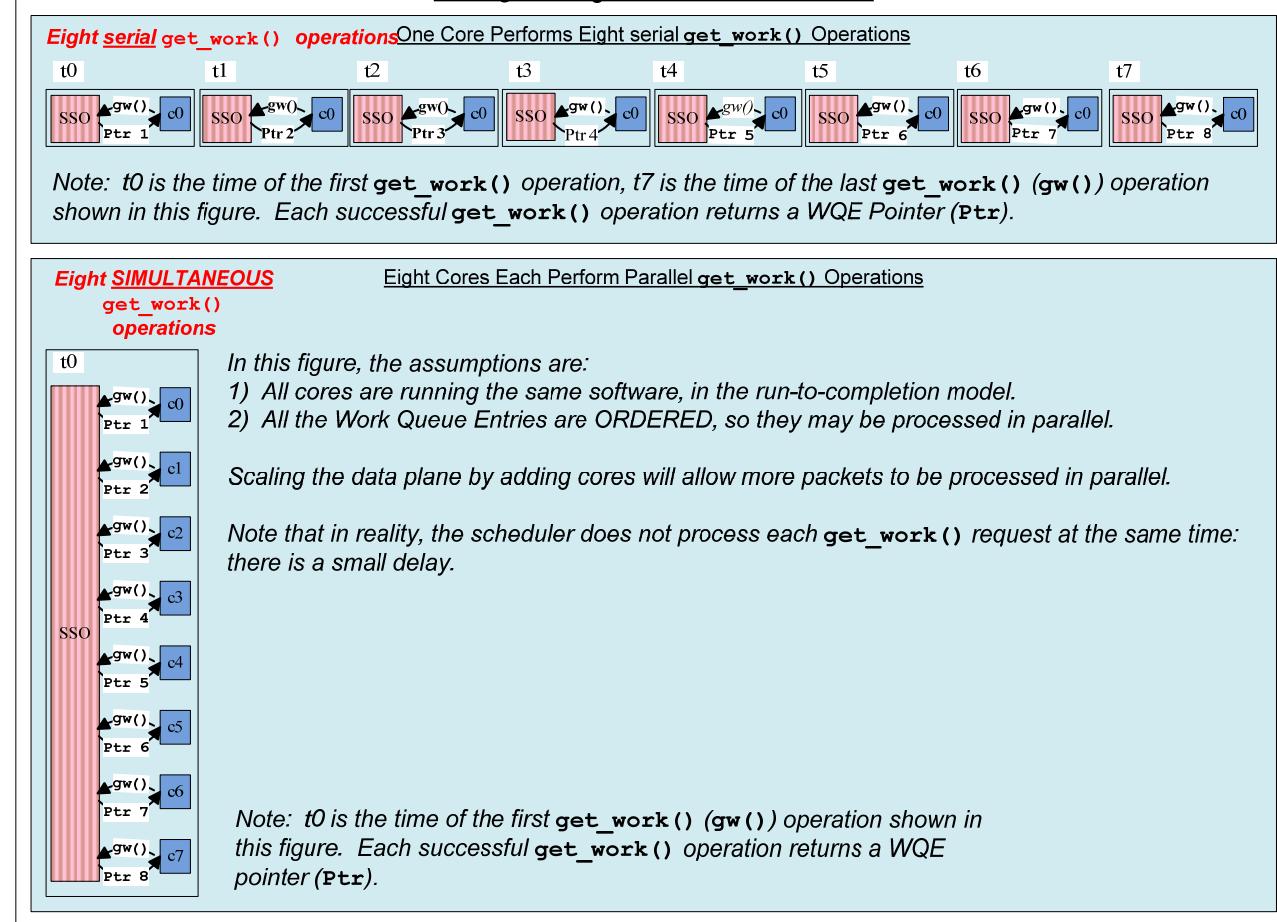
1. Locking: eliminate locks or minimize critical sections. Use packet-linked locks (via ATOMIC tags) when possible.
2. Cluster data which accessed at the same time into the same cache line so the core won't stall waiting for data. This is discussed in more detail in the OCTEON Performance Tuning Whitepaper. Clustering data can reduce contention on the shared bus by one third, or about 33%.
3. Use a polling (event-driven) loop instead of an interrupt-driven loop

In the following figure, the data-plane cores are scaled up from 1 to 7 cores. In the next section, the example `linux-filter` is discussed briefly. This is an example of an architecture designed for scaling: by running `linux-filter` on a load set of multiple cores, performance can be added to the data plane without changing the code.

Note that performance improvement from scaling depends on how much processing can be done in parallel versus how much processing must be serialized. For more details, see "Amdahl's Law" at <http://www.wikipedia.org/>. Designing an architecture which maximizes parallel processing will result in the best performance improvement with scaling.

Figure 31: Scaling the Data Plane

Scaling: Adding Cores adds Performance



SW OVERVIEW

More information on scaling and performance tuning can be found in the OCTEON Performance Tuning Whitepaper.

6.5.2 Code Locality: Reducing Icache Misses

On some large, very high-performance applications, reducing L1 Instruction Cache (Icache) misses can result in a significant performance improvement in some applications. This can be accomplished by improving code locality.

Each MIPS instruction is 4 bytes long. The size of the Icache varies with OCTEON model. If the Icache is 32 KBytes, then 8,192 instructions will fit into the Icache. Each cache line is 128 bytes (32 sequential instructions).

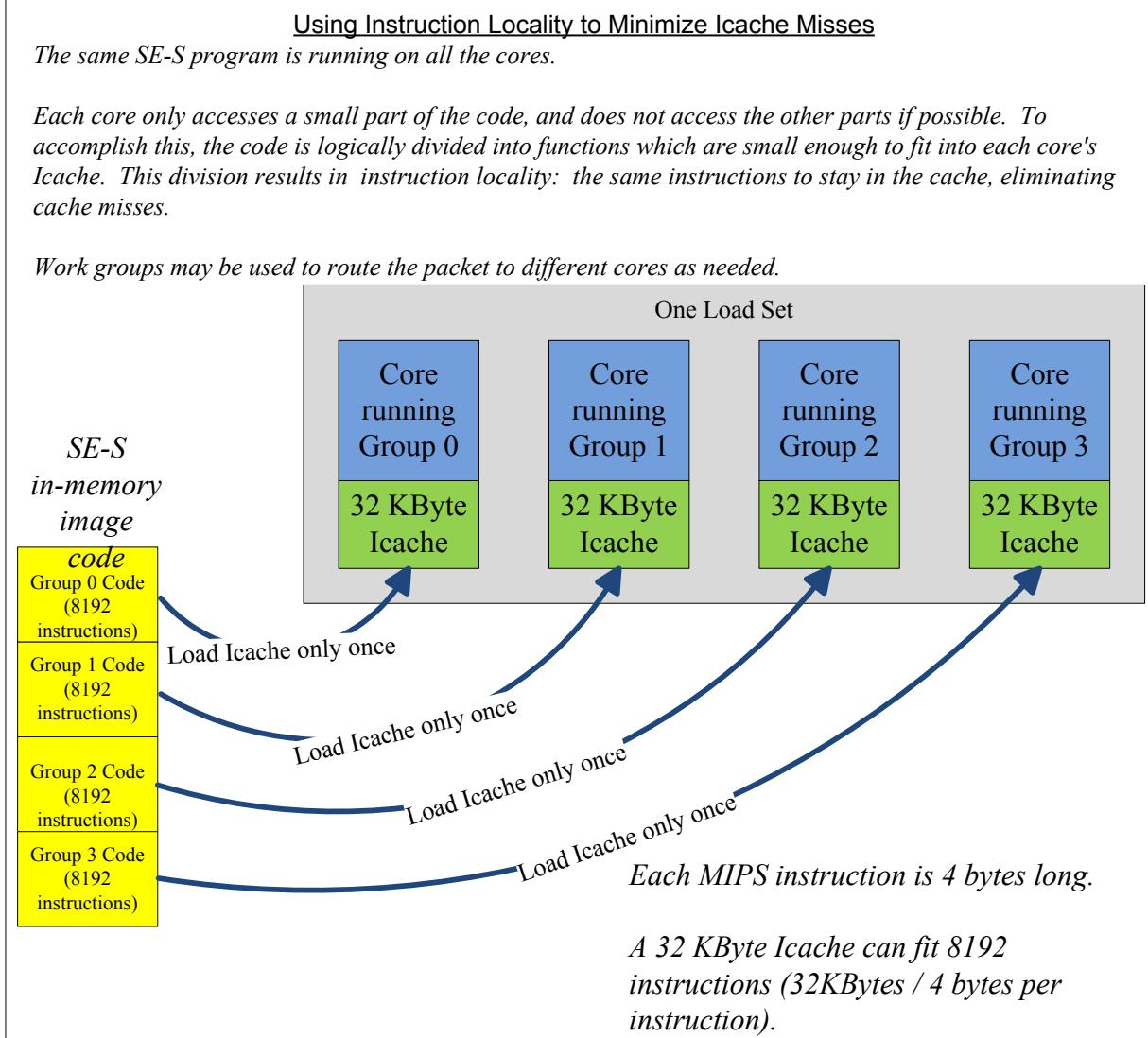
Once the core loads a set of instructions into Icache, if it continues to run only that set of instructions, the performance cost of Icache misses is 0.

For example, to take code locality to an extreme and reduce Icache misses to 0, the code is logically divided into functions which are small enough to fit into each core's Icache. This division results in instruction locality: the same instructions to stay in the cache, eliminating cache

misses. Work groups may be used to route the packet to different cores as needed. Note that performance analysis is needed to weigh the benefit of code locality versus the cost of any additional switches.

This extreme type of code locality can only be accomplished with only SE-S applications, not SE-UM applications. In the case of SE-UM applications, once the Linux Kernel runs (for instance, due to a timer interrupt), the kernel will displace the application code from the Icache.

Figure 32: Using Code Locality to Reduce Icache Misses



In the figure above, the cores are all in the same load set (running the same ELF file) but accept different groups and perform the different functions which match the group number. Each core may accept more than one group (perform more than one function).

In a less extreme situation, performance can be improved by increasing code locality while running a larger amount of code which does not fit in the Icache. For example, functions which are defined in the same source file are kept together by the linker. This may increase the possibility that they will share a cache block with another function needed by the same core. Similarly, when deciding which cores will perform which functions it is a good idea to look for opportunities to increase locality.

See Section 9.4 – “Caching” for a brief introduction to the L1 Cache.

6.5.3 Load-Balancing

Load-balancing is tuning the system so each core is working to its fullest capability. In a traditional pipelined system, load-balancing consists of making sure each processor uses the same amount of packet processing time, so one processor cannot stall the pipeline.

In the modified pipelining or the run-to-completion architecture which use the concept of work groups, load-balancing becomes simpler. For instance, in modified pipelining, the work can be divided into processing stages, with each stage represented by a group. To add more power to a processing stage, allow more cores to accept work with the group corresponding to the impacted stage. Similarly, underutilized cores may accept work from more work groups. This is shown in the figures in Section 6.4.5 – “Modified Pipelining”.

In the run-to-completion architecture, the different flows may be spread across the cores. For example, the PIP/IPD may be configured to assign the group number based on the tuple hash instead of the port number.

6.6 Example: `linux-filter`

An example of a design separating control path and data path, using a hybrid system is the `examples/linux-filter` example. This example shows a different use of work groups than modified pipelining: work groups are used to communicate between the data plane and control plane.

In this example, a Simple Executive application runs on one or more cores. Linux is also running on one or more cores. The cores running the Simple Executive application (`filter`) receive all incoming packets, check the packet type, and only send packets which are not IP broadcast to Linux.

Note that an ideal control path will only handle packets which are exceptions. In `linux-filter`, the control path is given packets which are not exceptions. This example is simplified and is intended only to illustrate packet filtering by a SE-S application, and passing packets between the control and data path. It is not intended for unmodified use in packet processing.

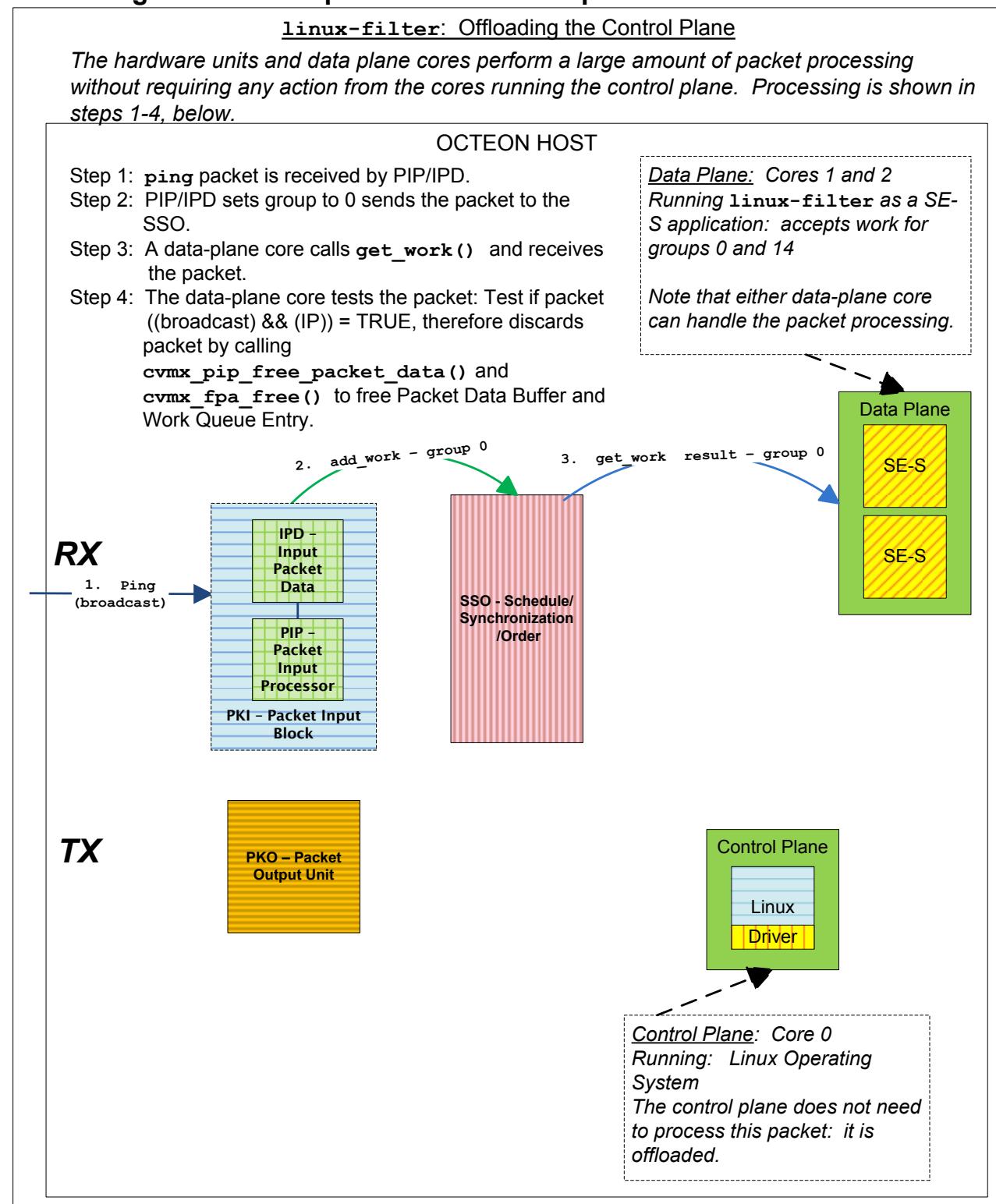
This program uses the idea of Work Groups to separate cores belonging to the fast path from those belonging to the slow path. Additionally, groups are used to identify the next processing phase. This example may be used as the base for an application which does similar processing.

In this example, Simple Executive (fast path) cores accept work for group 0 (new packet), and group 14 (response to packet received and processed by Linux). Linux cores (slow path) accept work for group 15.

The next two figures show `linux-filter` processing, without showing the rest of the OCTEON processor, or the connections between the hardware blocks. The packet interfaces are not shown in these figures. Although these figures show only two cores, many more cores may run `linux-filter` or Linux simultaneously.

In the first figure, the core receives an IP Broadcast Ping packet. The Simple Executive application, `linux-filter`, running on the fast path cores drops the packet (does not send it to the slow path (Linux) cores.

Figure 33: Example: Linux-filter Drops a Broadcast IP Packet

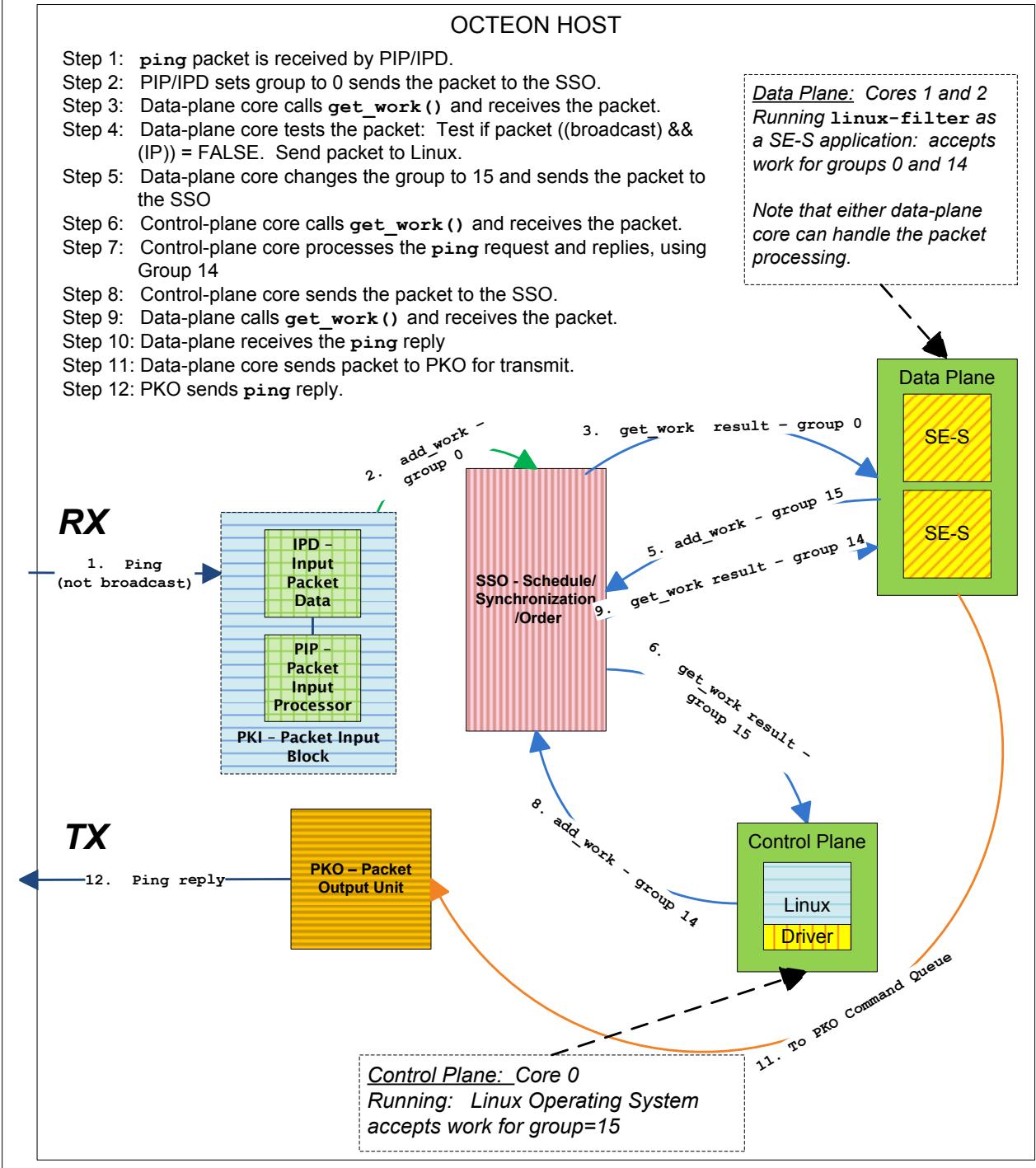


In the following figure, the Simple Executive application, `linux-filter`, accepts a non-broadcast ping packet, and forwards the packet to Linux. Linux sends a reply via `linux-filter`.

The exact details on how to run and test this example are presented in the SDK Tutorial chapter.

Figure 34: Example: Linux-filter Forwards a Non-Broadcast IP Packet**linux-filter:** Forwarding a Packet to the Control Plane

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



7 Application Binary Interface (ABI)

Several Application Binary Interfaces (ABIs) are supported for Simple Executive and Linux applications. Simple Executive and Linux applications may be compiled for 32-bits or 64-bits. The 64-bit mode is usually the highest performing choice. Note that the Linux Kernel is always 64-bit; only the Linux applications may be compiled for 32-bit mode.

To select the ABI, use the makefile option `OCTEON_TARGET`, as in “`make linux-filter OCTEON_TARGET=linux_64`”. The different choices for `OCTEON_TARGET` can be seen in the file `$OCTEON_ROOT/common.mk`.

For Simple Executive, the preferred ABI is EABI. For Linux, the preferred ABI is `linux_64`.

All the ABIs create ELF-format files.

7.1 **ABI Choices**

There are several ABI choices available. Which ABI is used depends on whether the application is 64-bit or 32-bit, and whether the application is run as a SE-S or SE-UM application. The *target* is the ELF executable file.

7.1.1 **EABI (OCTEON_TARGET=cvmx_64): SE-S 64-Bit**

The Simple Executive applications are created with this ABI. The matching toolchain is “`mipsisa64-octeon-elf-*`”. This ABI supports 64-bit registers and address space. This ABI is the default for Simple Executive.

7.1.2 **N64 (OCTEON_TARGET=linux_64): SE-UM 64-Bit**

The 64-bit Linux applications are created with this ABI. The matching toolchain is “`mips64-octeon-linux-gnu-*`” with the “`-mabi=64`” option. This ABI supports 64-bit registers and address space. This ABI is the default for Linux kernel and user space. The resulting binary is in ELF64 format.

For example, `linux-filter` can be compiled with this option. The resultant target file is `$OCTEON_ROOT/examples/linux-filter/linux-filter-linux_64`. This file may be added to the embedded rootfs. When `vmlinux.64` is booted, the file is automatically included in the `/examples` directory on the target, and has been renamed from `linux-filter-linux_64` to `linux-filter`.

7.1.3 **N32 (OCTEON_TARGET=cvmx_n32): SE-S 32-Bit**

Simple Executive 32-bit applications are created with this ABI.

The same ABI is used for SE-UM 32-bit applications, but the toolchain is different for SE-S 32-bit applications:

- The N32 toolchain for Simple Executive is “`mipsisa64-octeon-elf-*`” with the “`-mabi=n32`” command line option. This ABI supports 64-bit registers and 32-bit

address space. The resulting binary is in ELF32 format, whose symbol table is in DWARF format.

7.1.4 N32 (OCTEON_TARGET=linux_n32): SE-UM 32-Bit

Simple Executive Linux User Mode 32-bit (SE-UM 32-bit) applications are created with this ABI. The same ABI is used for SE-S 32-bit applications, but the toolchain is different for SE-UM 32-bit applications:

- The N32 toolchain for Linux is “mips64-octeon-linux-gnu-*” with the “-mabi=n32” option.

Note: SE-UM 32-bit applications are useful for compatibility with older code. Applications using large data structures may also get a benefit from pointers being smaller and taking less room. The downside is that there is much less memory available to 32-bit applications. These 32-bit Linux applications must use *reserve32*, a special region of free memory which is low enough to have 32-bit physical addresses (the “shallow end” of the memory pool). 32-bit SE-S applications do not use *reserve32*.

7.1.5 O32 (linux_o32) (Not Recommended)

The older O32 ABI is in ELF32 format with the symbol table in *.mdebug* (dot mdebug) format. All registers are treated as 32 bits. The 64-bit types are split into two separate registers.

Although the Cavium Networks compilers can *compile* o32 applications, they cannot *link* them: no o32 libraries are provided. To build o32 applications (which will NOT take advantage of Cavium Networks-specific instructions), use the Debian compiler.

7.1.6 Linux uclibc (linux_uclibc)

Linux applications are built with the smaller uclibc instead of glibc. The uclibc library is 32-bit only.

7.1.7 Choosing the OCTEON_TARGET

Linux code requiring large amounts of memory and the fastest possible access to OCTEON hardware should use the N64 ABI.

Linux code requiring many data structures dealing with pointers, but requiring only occasional hardware access should use the N32 ABI.

Some older applications and binaries may still use the O32 ABI, but it is recommended that they be upgraded to the N32 ABI.

A detailed discussion of Linux ABIs is located in the SDK document “*Linux Userspace on the OCTEON*”.

7.2 64-Bit Porting Issues

The key difference from a software porting perspective is in the following variables:

- 1) Size of long

2) Size of (void *)

With N32, be alert to automatic sign extension when loading 32-bit values into 64-bit registers. The N32 registers are 64 bits, but `sizeof (void *)` = 32 bits, so when loading a 32-bit value into a 64-bit register, it is automatically sign extended.

The following tables provide the ABIs, data type length and toolchain information for SE-S and SE-UM applications, as well as information on the older O32 ABI.

Table 7: Key ABI Differences

<i>(the most useful values are highlighted)</i>			
Data Type	O32	N32	N64 and EABI64
int	32 bits	32 bits	32 bits
long	32 bits	32 bits	64 bits
long long	64 bits	64 bits	64 bits
pointer	32 bits	32 bits	64 bits
register	32 bits	64 bits	64 bits

Table 8: SE-S ABIs (N32, EABI64), Data Type Lengths, and Toolchain

<i>(the most useful values are highlighted)</i>		
Application Type	SE-S 32-bit	SE-S 64-bit
Data Type	N32 (see Note 1)	EABI64
int	32 bits	32 bits
long	32 bits	64 bits
long long	64 bits	64 bits
pointer (void *)	32 bits	64 bits
register	64 bits	64 bits
Toolchain	mipsisa64-octeon-elf-*	
<i>Note 1: Function calls are not ABI-conformant in this toolchain.</i>		

Table 9: SE-UM ABIs (N32, N64), Data Type Lengths, and Toolchain

(the most useful values are highlighted)		
Application Type	SE-UM 32-bit	SE-UM 64-bit
Data Type	N32	N64
int	32 bits	32 bits
long	32 bits	64 bits
long long	64 bits	64 bits
pointer (void *)	32 bits	64 bits
register	64 bits	64 bits
Toolchain	mips64-octeon-linux-gnu-*	

Table 10: Other ABI (O32), Data Type Lengths, and Toolchain

(the most useful values are highlighted)	
Application Type	Other 32-bit
Data Type	O32
int	32 bits
long	32 bits
long long	64 bits
pointer (void *)	32 bits
register	32 bits (see Note 1)
Toolchain	Debian

Note 1: Registers are not 64-bits, unlike the other ABIs.

Two key things to remember when writing portable code:

1. Use `stdint.h` data types. Data types, such as `uint64_t`, are defined in this file in a portable way. This is the most important priority in writing portable code. In the executive and in examples, `stdint.h` is included indirectly: they include `cvmx.h` which includes `executive/cvmx_platform.h` which includes `stdint.h`.
2. When using `printf()` to print pointers, use “`%p`”.

Note: Be careful to use the functions `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()` when converting between physical addresses and virtual addresses. 90% of porting problems come from mistakenly using casts on physical and virtual addresses.

8 Tools

This section provides a quick overview of some of the tools. Tools are discussed in further detail in the *SDK Tutorial* chapter.

8.1 GNU Cross-Development Toolchain

Cross-development tools are tools run on the host machine to build object files which will run on the target machine.

In the tools/bin directory, there are two sets of tools including the cross compiler, linker, and libraries. One set is prefixed “`mipsisa64-octeon-elf`”; the other set is prefixed “`mips64-octeon-linux-gnu`”. These tools have been modified to support OCTEON-specific instructions to achieve maximum runtime performance, and support the Cavium Networks-specific section: `cvmx_shared`.

8.1.1 The Cavium Networks-Specific `cvmx_shared` Section

Cavium Networks toolchains support a `cvmx_shared` section, used to share small amounts of memory between cores started from the same load command.

8.1.1.1 Sections

When object files are created by the compiler, they are divided into different sections. Four common sections are `.text`, `.rodata`, `.data`, and `.bss`. The `.text` section is read-only executable code, `.rodata` is read-only data, `.data` is initialized data, and `.bss` is uninitialized data (which is initialized to 0 when the section is loaded). Because the `.text` section of an object file is read-only, multiple instances of the same object file may share this information in memory, which conserves system memory. This also allows the bootloader to collect sections with similar access permissions into the same block of memory (for instance `.text` and `.rodata` which are both read-only) allowing the system to use fewer TLB entries to map the program. (See Section 10 – “Virtual Memory”.)

The sections can be seen with the `objdump` command. Most of these sections can be ignored by the programmer. There is one which the programmer needs to be aware of, however, and that is the `cvmx_shared` section.

```
host$ mipsisa64-octeon-elf-objdump -h fpa
fpa:      file format elf32-bigmips

Sections:
Idx Name          Size    VMA       LMA       File off  Align
 0 .reginfo      00000018 10000000 10000000 0001c058 2**2
               CONTENTS, READONLY, LINK_ONCE_DISCARD
 1 .init         00000028 10000018 10000018 00001018 2**0
               CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .text         00016058 10000040 10000040 00001040 2**3
               CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .fini         00000020 10016098 10016098 00017098 2**0
               CONTENTS, ALLOC, LOAD, READONLY, CODE
 4 .rodata        00003368 100160b8 100160b8 000170b8 2**3
               CONTENTS, ALLOC, LOAD, READONLY, DATA
 5 .eh_frame     00000404 10019420 10019420 0001a420 2**3
               CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .ctors         00000010 12000000 12000000 0001b000 2**3
               CONTENTS, ALLOC, LOAD, DATA
 7 .dtors         00000010 12000010 12000010 0001b010 2**3
               CONTENTS, ALLOC, LOAD, DATA
 8 .jcr          00000008 12000020 12000020 0001b020 2**3
               CONTENTS, ALLOC, LOAD, DATA
 9 .data          00000f88 12000028 12000028 0001b028 2**3
               CONTENTS, ALLOC, LOAD, DATA
10 .sdata         000000a8 12000fb0 12000fb0 0001bfb0 2**3
               CONTENTS, ALLOC, LOAD, DATA
11 .sbss          000000a0 12001058 12001058 0001c058 2**3
               ALLOC
12 .bss           00000458 120010f8 120010f8 0001c058 2**3
               ALLOC
13 .cvmx_shared_bss 000012b0 14000000 14000000 0001c058 2**3
               ALLOC
```

<The remaining sections are not shown here.>

8.1.1.2 The *cvmx_shared* Section

Both SE-S and SE-UM applications support the *cvmx_shared* section, a Cavium Networks-specific section which is used to provide a shared data space for applications started with the same boot (load) command or one *oncpu* command.

When cores are in the same load set, shared variables can be created at compile time by specifying the CVMX_SHARED attribute. Variables declared with the CVMX_SHARED attribute are put into a special section in the compiled file: *.cvmx_shared_bss*.

If the *cvmx_shared* section is large, the ELF file will also be large. This can cause problems, for instance during load time. For example, when running very large SE-S programs (which will consume above the virtual address 0x20000000) 1:1 mappings cannot be used. As an alternative, when large amounts of shared memory are desired, the variable stored in the *cvmx_shared* section should be only a pointer. At application initialization time, the initializing core can use the *bootmem* functions to allocate shared memory from memory outside the 256 MByte program space. The initializing core can then put the address of the allocated memory into

the CVMX_SHARED pointer. This will keep the application size small, while allowing a large amount of shared memory.

Usage of a CVMX_SHARED variable may be seen in the linux-filter example code:

```
CVMX_SHARED int intercept_port = 0;
```

The *cvmx_shared (.cvmx_shared_bss)* section can be seen with the objdump utility:

```
host$ mipsisa64-octeon-elf-objdump -h linux-filter
passthrough:      file format elf32-tradbigmips

Sections:
Idx Name      Size      VMA          LMA          File off  Align
 0 .reginfo   00000018  10000000  10000000  00036cd0  2**2
               CONTENTS, READONLY, LINK_ONCE_DISCARD
               <text omitted>
 13 .cvmx_shared_bss 00001358  14000000  14000000  00036cd0  2**3
               ALLOC
               <more text follows>
```

Note: The *bss* (Block Started by Symbol) section is the name of the data section which contains static variables which will initialized to zero by the ELF loader when it loads the program.

8.1.2 Link Addresses

Link addresses and section sizes for a specific application can be seen using the objdump utility.

For example, when linux-filter is built as a SE-S application:

```
host$ mipsisa64-octeon-elf-objdump -h linux-filter
linux-filter:      file format elf32-tradbigmips

Sections:
Idx Name      Size      VMA          LMA          File off  Align
 0 .reginfo   00000018  10000000  10000000  0001b6d8  2**2
               CONTENTS, READONLY, LINK_ONCE_DISCARD
 1 .init      00000028  10000018  10000018  00000098  2**0
               CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .text      000155b8  10000040  10000040  000000c0  2**3
```

8.1.3 Simple Executive Development Tools

The mipsisa64-octeon-elf-* tools are used to build Simple Executive Applications.

8.1.3.1 C/C++ Runtime Support for Simple Executive

The C/C++ runtime support for Simple Executive is specified in the SDK document “*OCTEON Simple Executive Overview*”.

8.1.4 Linux Development Tools

The `mips64-octeon-linux-gnu-*` tools are used to build Linux Applications.

The cross-development tools are discussed in more detail in the *SDK Tutorial* chapter.

8.2 Native Tools (*Run on the Target*)

Native tools are those which may be run on the target instead of on the development host.

8.2.1 Native tools and Simple Executive

Simple Executive does not have a file system. Only one application runs. Thus there are no native tools.

8.2.2 Native tools and Linux

Native tools are provided with both *embedded_rootfs* and Debian.

8.2.2.1 The *embedded_rootfs* Native Tools

Native Linux tools are usually located in `/bin`, `/sbin`, and `/usr/bin`.

8.2.2.2 Debian Native Tools

Two toolchains are provided with the Debian file system:

- The Debian native toolchain
- A Cavium Networks toolchain, optimized for the OCTEON processor.

These toolchains are used for native compiling.

The Cavium Networks native toolchain supports both 32-bit and 64-bit Linux applications. This toolchain implements the Cavium Networks-specific instruction set. See the *OCTEON Hardware Reference Manual* for instruction set details.

To compile O32 applications, use the Debian toolchain. Note that these applications cannot use the Cavium Networks-specific instruction set.

9 Physical Address Map and Caching on the OCTEON Processor

A brief introduction to hardware issues such as the physical address map and the concept of caching is provided in this section.

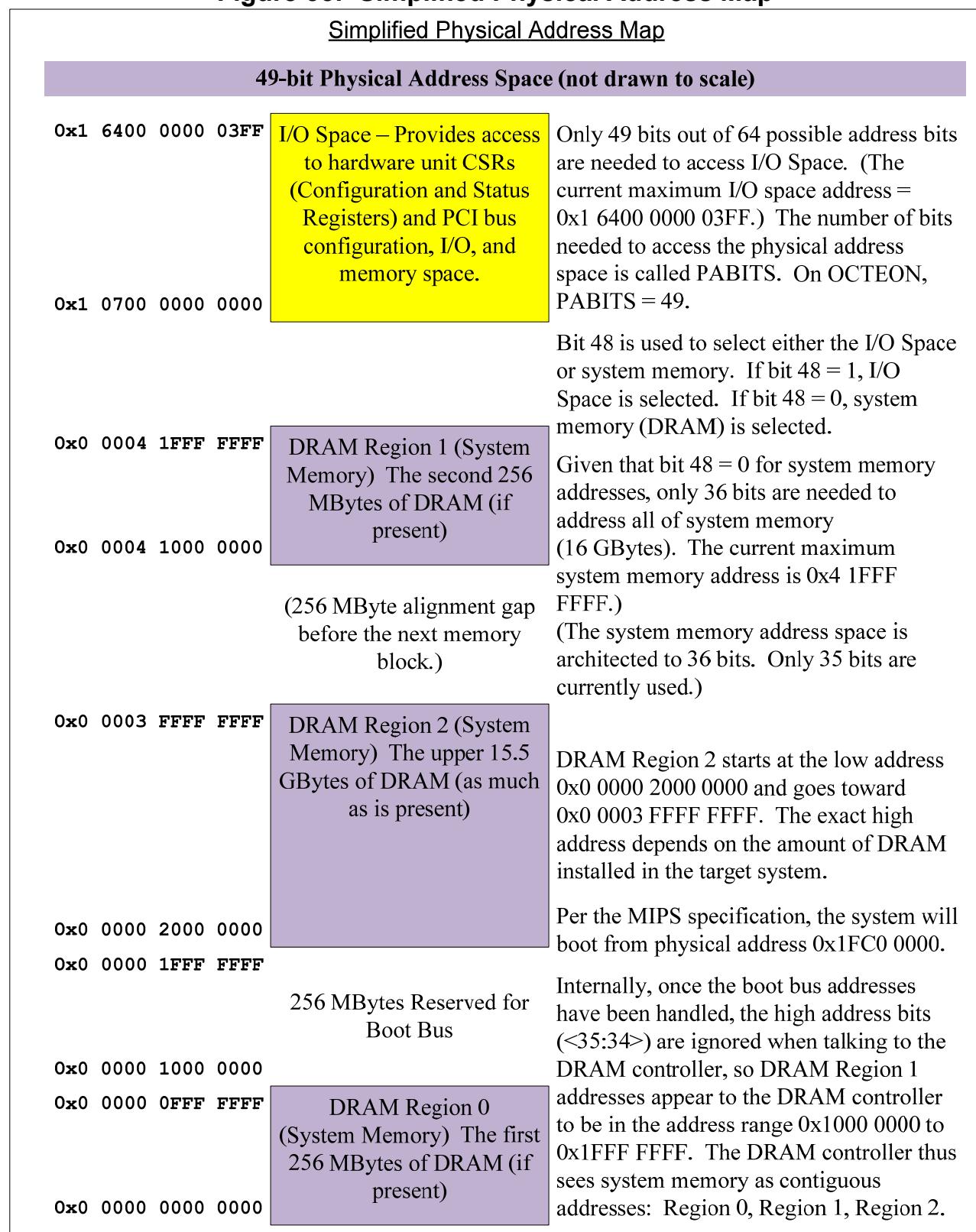
9.1 Physical Address Map

There are two key elements in the physical address map:

1. System memory (DRAM)
2. I/O space

Out of 64 possible Physical Address Bits (PABITS), only 49 bits (PABITS=49) are used to access the physical address space. These are bits <48:0>.

A simplified physical address map is shown in the next figure. Exact details may be found in the *OCTEON Hardware Reference Manual*.

Figure 35: Simplified Physical Address Map

9.2 System Memory (DRAM) Addresses

System memory (DRAM) is located starting at address 0 (zero). The amount of memory on the board is a design option. The default Linux configuration supplied with the SDK requires 230 MBytes of memory, so a minimum of 256 MBytes of system memory is recommended for this configuration. If less than 256 MBytes of system memory is available, see Section 4.3.7 – “Linux on Small Systems (Limited MBytes of Memory)” for instructions on how to configure Linux to require less system memory.

Note: The bootloader uses the first MByte of system memory. This space is needed even after the bootloader exits. This space is used by the bootmem functions.

There are up to three regions of system memory: DRAM Region 0, DRAM Region 1, and DRAM Region 2. (These regions are sometimes labeled as DR0, DR1, and DR2.) The actual memory map will vary depending on the amount of DRAM installed in the target board.

If physical address bit 48 is 0, the access is to system memory (DRAM). Out of the 49 PABITS, 36 bits (<35:0>) are architected to access all of system memory.

9.3 I/O Space Addresses

The I/O space contains the OCTEON processor configuration and status registers for the various hardware units and also contains the PCI configuration, I/O and memory space.

If physical address bit 48 is 1, the access is to I/O space.

Table 11: Simplified View of I/O Space

Physical Addresses		I/O Space Addressed
FROM	TO	
0x1 F000 0000 0000	0x1 F00F FFFF FFFF	FAU Operations
0x1 6000 0000 0000	0x1 6700 0000 03FF	SSO (POW)
0x1 5200 0000 0000	0x1 5200 0003 FFFF	PKO doorbell store operations
0x1 4F00 0000 0000	0x1 4F00 0000 07FF	IPD
0x1 4000 0000 0000	0x1 4000 0000 07FF	RNG Load/IOBDMA operations
0x1 3800 0000 0000	0x1 3800 0000 0007	ZIP doorbell store operations
0x1 3700 0000 0000	0x1 3707 FFFF FFFF	DFA NCB type CSRs and operations
0x1 2800 0000 0000	0x1 2F0F FFFF FFFF	FPA Pools Allocate/Free operations
0x1 2000 0000 0000	0x1 2000 0000 1FFF	KEY Memory operation
0x1 1F00 0000 0000	0x1 1F0F FFFF FFFF	NPI NCB type CSRs, doorbells
0x1 1B00 0000 0000	0x1 1EOF FFFF FFFF	PCI Bus Memory space
0x1 1A00 0000 0000	0x1 1AOF FFFF FFFF	PCI Bus IO space
0x1 1900 0000 0000	0x1 190F FFFF FFFF	PCI Bus Config/IACK/Special space
0x1 1800 F000 0000	0x1 1800 F000 07FF	IOB
0x1 1800 B800 0000	0x1 1800 B800 03FF	ASX1
0x1 1800 B000 0000	0x1 1800 B000 03FF	ASX0
0x1 1800 A800 0000	0x1 1800 A800 00FF	TRA
0x1 1800 A000 0000	0x1 1800 A000 1FFF	PIP
0x1 1800 9800 0000	0x1 1800 9800 07FF	SPX1, SRX1, and STX1
0x1 1800 9000 0000	0x1 1800 9000 07FF	SPX0, SRX0, and STX0
0x1 1800 8800 0000	0x1 1800 8800 007F	LMC
0x1 1800 8000 0000	0x1 1800 8000 07FF	L2C
0x1 1800 5800 0000	0x1 1800 5800 1FFF	TIM
0x1 1800 5000 0000	0x1 1800 5000 1FFF	PKO
0x1 1800 4000 0000	0x1 1800 4000 000F	RNM
0x1 1800 3800 0000	0x1 1800 3800 00FF	ZIP
0x1 1800 3000 0000	0x1 1800 3000 07FF	DFA
0x1 1800 2800 0000	0x1 1800 2800 01FF	FPA
0x1 1800 2000 0000	0x1 1800 2000 001F	KEY
0x1 1800 1000 0000	0x1 1800 1000 1FFF	GMX1
0x1 1800 0800 0000	0x1 1800 0800 1FFF	GMX0
0x1 1800 0000 0000	0x1 1800 0000 1FFF	MIO BOOT, LED, FUS, TWSI, UART, SMI
0x1 0700 0000 0000	0x1 0700 0000 08FF	CIU and GPIO

(Note this is an example of I/O Space. I/O space details are OCTEON model-specific.)

9.4 Caching

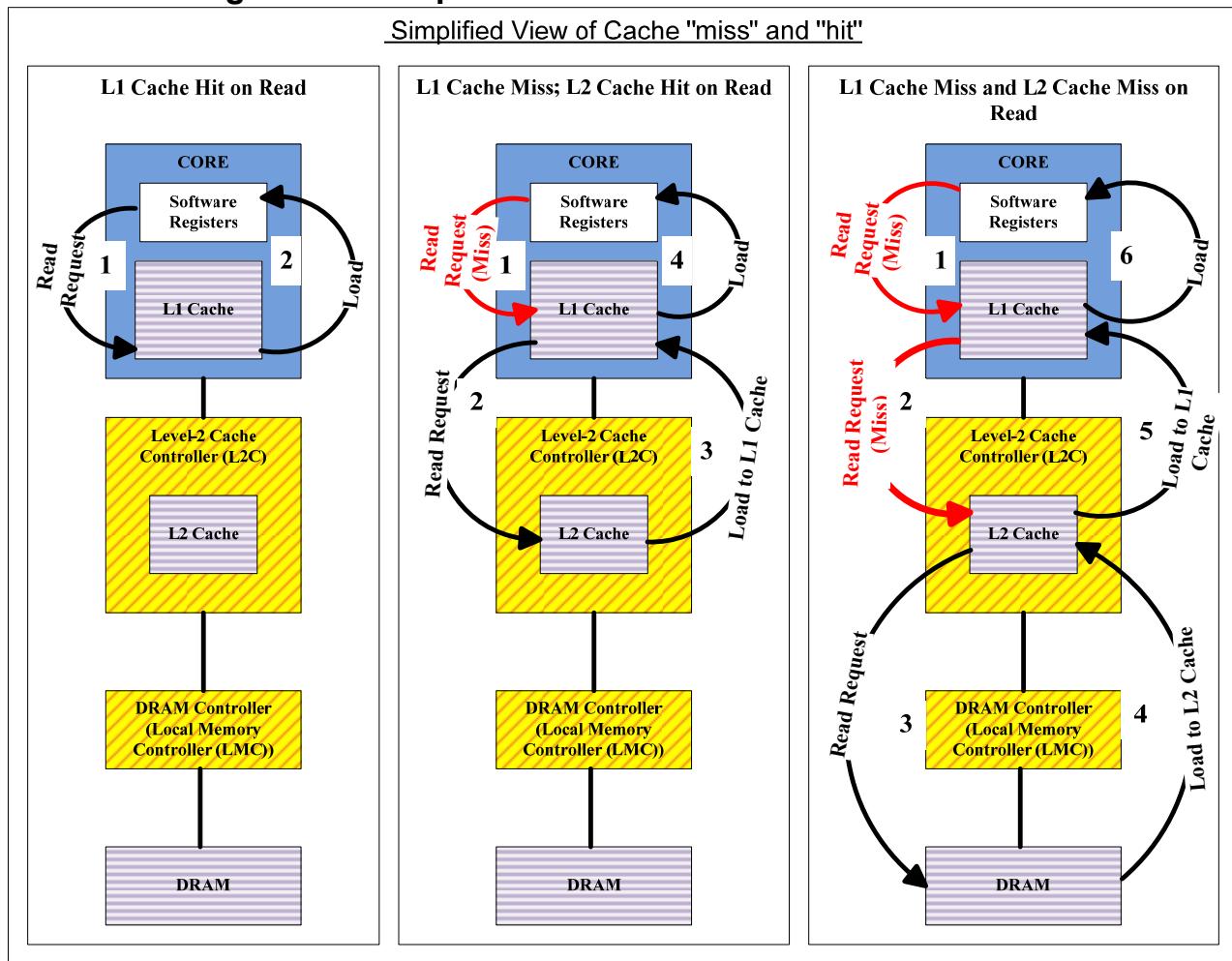
On the OCTEON processor, caching only applies to system memory (DRAM) accesses, not I/O space. Caching is used to improve system performance by providing core-local or chip-local fast memory which is used to cache (save a copy) of recently accessed data. This improves performance because accesses to on-chip cached system memory are lower latency than accesses to the external system memory (DRAM).

Caches on the OCTEON processor:

- Level-1 Data cache (Dcache) (per core)
- Level-1 Instruction cache (Icache) (per core)
- Level-2 (L2) cache (one shared by all the cores)

The following figure is a simplified view of a data load access, showing the difference between a *cache miss* and a *cache hit*.

Figure 36: Simplified View of Cache “miss” and “hit”



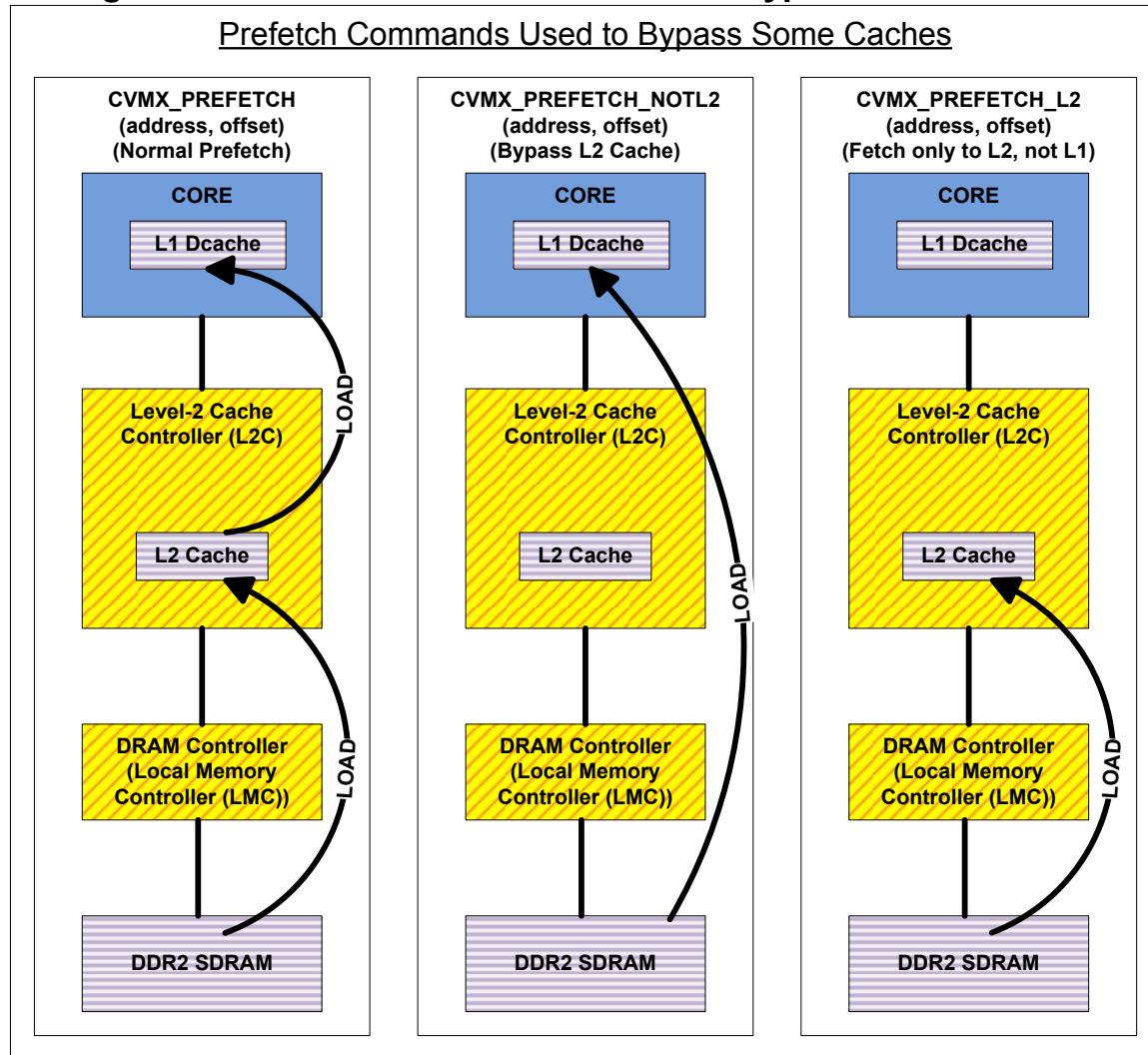
Note that the sizes of the L1 and L2 caches are limited. The specific sizes depend on the OCTEON model. The size of the L1 Dcache is also affected by the amount of Dcache set aside for *cvmseg*.

(The Cavium Networks-specific segment *cvmseg* is discussed in Section 10.6 – “Cavium Networks-Specific *cvmseg* Segment”.)

System memory *stores* are always cached.

The data returned by *load* instructions is usually cached in both L1 and L2 caches. As a performance improvement, prefetch commands may be used. Prefetch commands hide the fetch latency by requesting the data before it is needed. A normal prefetch loads the data into both the L1 and L2 caches. Some customers may wish to completely bypass the cache when accessing memory, especially when debugging hardware issues, however this is not an option. Special prefetch instructions are available which may bypass some, but not all, of the caches. Prefetch commands are not discussed in detail here. The following figure illustrates the prefetch instruction choices available.

Figure 37: Prefetch Commands Used to Bypass Some Caches



Prefetching into the L1 cache, bypassing the L2 cache, is useful to avoid “polluting” the shared L2 cache with data needed by only one core. This option should only be used if the data is read-only.

Prefetching into the L2 cache (but not the L1 cache) is only useful if the data will be needed by a core other than the one issuing the prefetch.

9.5 Special L2 Cache Features: Partitioning and Locking

The L2 cache controller provides two features that may be used in performance tuning: partitioning and locking.

Both partitioning and locking can be used to prevent one core from starving the other cores by causing excessive L2 traffic and causing other cores cache blocks to be evicted.

Partitioning can split the cache up into core-specific regions, so each core can only cause evictions from its own region.

Locking can be used to make a particular region of memory resident in the L2 cache, so it cannot be evicted. This feature is also used to speed access to this memory region for all cores.

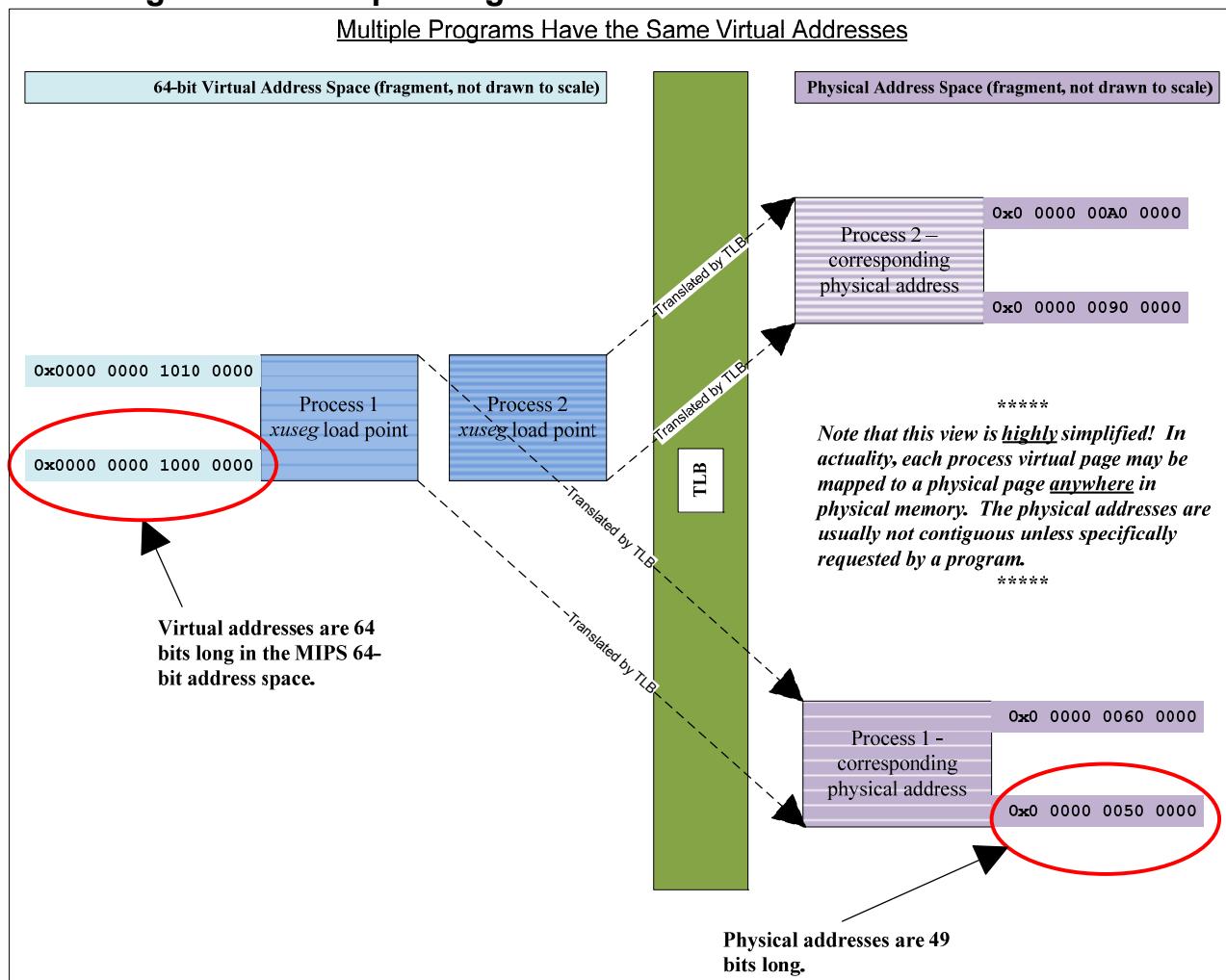
See the *OCTEON Hardware Reference Manual* for more details.

10 Virtual Memory

The goal of virtual memory is to make accessing physical memory and I/O space safer and more convenient.

Safety is provided when a process may only write to its own memory, not the memory of other processes. Because the user addresses are all mapped, the operating system can prevent the user from accessing memory inappropriately.

Convenience is provided so that, when the program is compiled, the linker may select the same hard-coded virtual address as starting address for each program. The hardware and operating system work together to translate identical virtual addresses into unique physical addresses, as shown in the following figure.

Figure 38: Multiple Programs Have the Same Virtual Addresses

SW OVERVIEW

10.1 Virtual Address Translation

In the traditional view of virtual addresses, addresses are *always* translated by the operating system working together with the MMU. This translation is referred to as *mapping*.

10.1.1 Mapping

There is a translation (mapping) between the physical and virtual address. Physical memory is mapped when accesses to it go through this translation process. This mapping allows multiple Linux applications to have the same starting address. Each virtual starting address is mapped to a different physical address. This is done so that when the file is compiled, the program addresses can be resolved at compile time instead of at load time.

Mapping requires, at a minimum, entries into a Translation Look-aside Buffer (TLB). Simple Executive Standalone applications use only the TLB for mapping; Linux uses a more complex memory management system (page tables and TLB miss handler). In this chapter, it is only necessary to know about the TLB.

10.1.2 The Translation Look-Aside Buffer (TLB)

The *Translation Look-aside Buffer (TLB)* is used to store a limited number of virtual-to-physical address mappings. There are either 32 or 64 entries in the TLB, depending on the OCTEON model. Each of these entries is a double entry. The 32-entry TLB can store 64 mappings. The 64-entry TLB can store 128 mappings. The sizes of the mapped pages may vary from 4 KBytes to 256 MBytes (all sizes which are powers of 2 in this range are allowed).

TLB entries contain an *Address Space ID (ASID)* (similar to a process ID (PID)). This identifies which process owns the TLB entry.

There is one TLB per core. It is shared by all the processes running on the core. In the Simple Executive, there is only one process per core, so TLB use is very simple. On Linux, many processes compete for the TLB entries.

10.1.3 Wired TLB Entries

Some entries in the TLB may be made permanent and not replaced by newer values. When a mapping is permanently saved in the TLB, the entry is considered to be “wired”.

Wired TLB entries may increase performance when the same page is accessed frequently: TLB miss exceptions will not occur for accesses within the wired region.

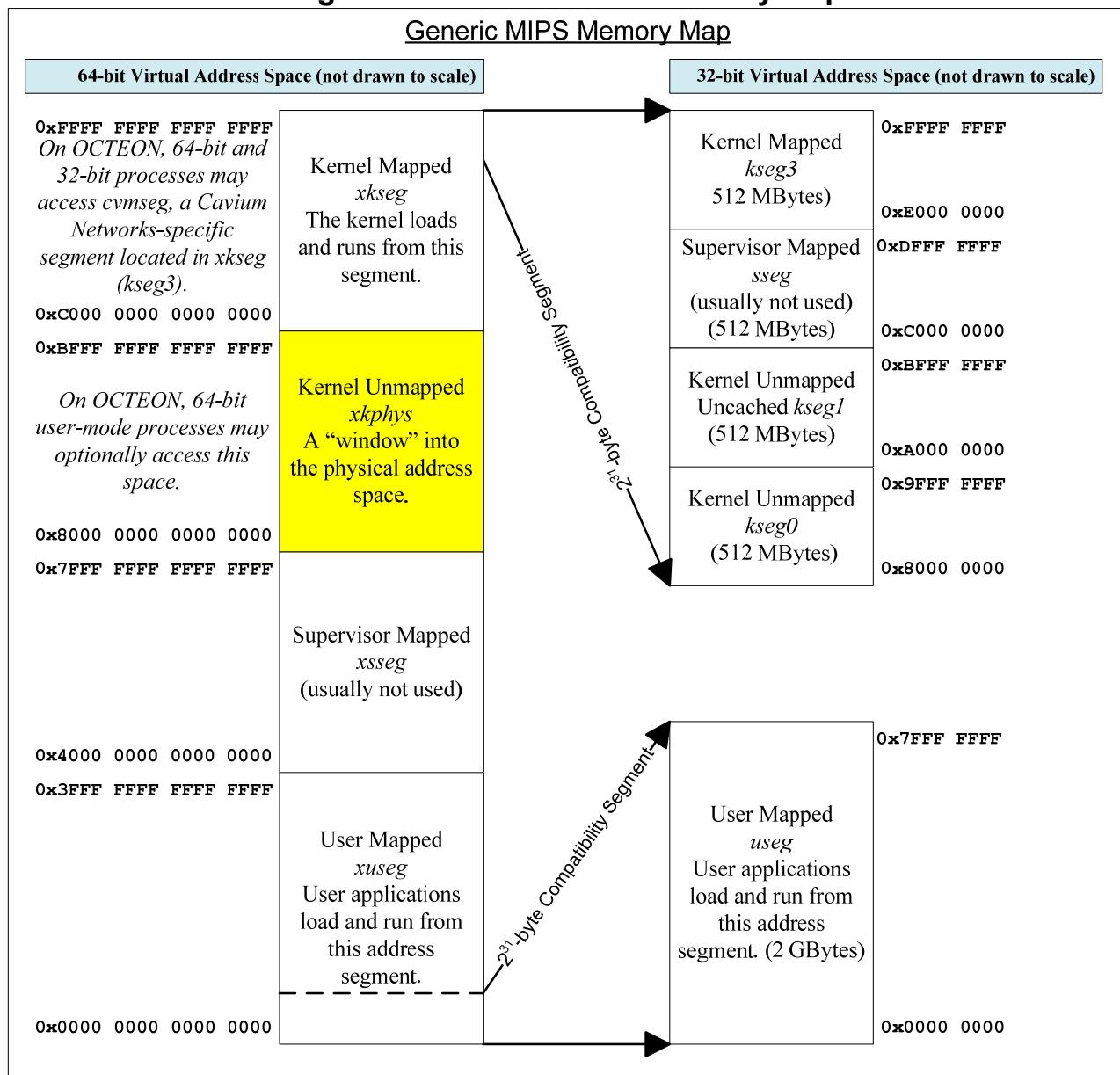
Wired TLB entries may also harm performance by reducing the number of TLB entries available for the other processes.

To determine the affect of wired TLB entries for the application, use profiling and performance tuning tools after the application has been written.

10.2 Generic MIPS Virtual Memory Map

The generic MIPS virtual memory map is shown in the figure below. The 64-bit address space contains a 32-bit compatibility region.

In the figure below, the *xkphys* segment is highlighted. This segment is particularly important because 64-bit software may use this segment to access physical memory and I/O space without mapping the virtual addresses.

Figure 39: Generic MIPS Memory Map

10.3 MIPS Virtual Memory Address Translation

MIPS virtual memory is divided into segments, not all segments are mapped (see Section 10.4 – “Mapped and Unmapped Segments”), and the MMU is streamlined.

Virtual address translation depends on:

1. The number of address bits in the address space: 64-bit or 32-bit address space
2. The segment addressed
3. The privilege level (mode): kernel, supervisor, or user

10.3.1 Segments

In MIPS architecture, the address space is divided into segments: it is not an undifferentiated virtual address space.

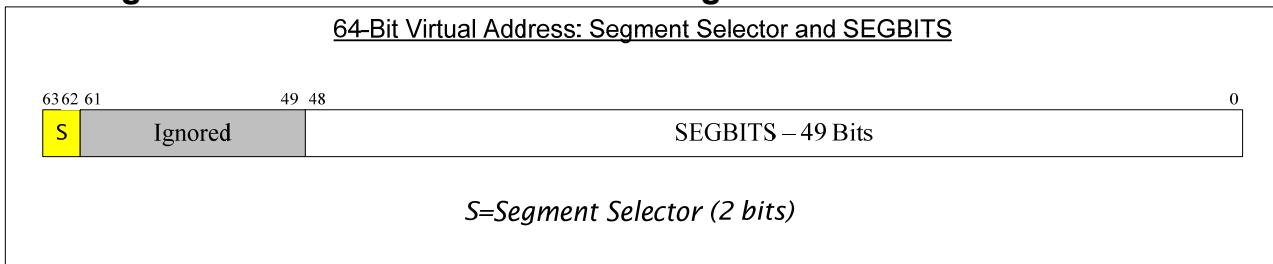
10.3.1.1 Segments: 64-Bit Virtual Address Map

In the 64-bit virtual address map, the high two bits of the virtual address (<63:62>) are used to select one of four segments. These address bits are always translated by the hardware, not the operating system.

Of the remaining 62 bits in the virtual address, some of the high bits are ignored if the processor does not support that many virtual address bits within a segment (*SEGBITS*).

On the OCTEON processor, SEGBITS equals 49, so only bits <48:0> of the virtual address define the address space within the segment. The remaining bits (<61:49>) are ignored.

Figure 40: 64-Bit Virtual Address: Segment Selector and SEGBITS

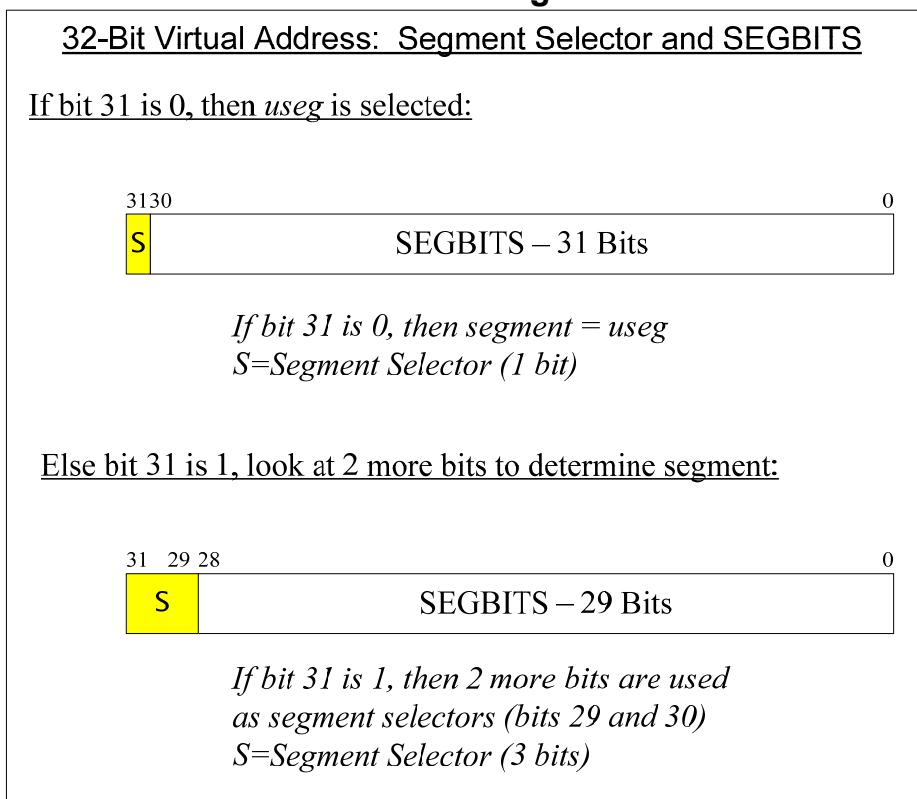


10.3.1.2 Segments: 32-Bit Virtual Address Map

In the 32-bit virtual address map:

- If the high bit (<31>) is 0, then the segment is *useg*. Within *useg*, the other 31 bits are not used as a segment selector.
- If the high bit (<31>) is 1, then 2 more bits are used as segment selectors (<30-29>).

The following figure illustrates the segment selector and SEGBITS for a 32-bit virtual address.

Figure 41: 32-Bit Virtual Address: Segment Selector and SEGBITS

SW OVERVIEW

10.3.2 Privilege Level (Mode) and Segments

There are three “modes” (privilege levels): user, supervisor, and kernel. The two most important modes are user and kernel (most Operating Systems ignore supervisor mode). Applications usually run in user mode. The kernel and drivers run in kernel mode.

On traditional processors, any virtual page can be mapped as any mode, and the mode bits are stored as part of the TLB entry. On MIPS, the virtual address space is divided into segments which are designed to correspond to the different runtime modes. For example:

- processes running in user mode use *xuseg*
- the kernel uses *xkseg*

Segments are also accessible to processes running in a higher mode, so *xuseg* is accessible to the kernel and drivers: they can access all legal addresses in the 64-bit or 32-bit virtual memory map.

In general, the user processes are restricted to *xuseg* (*useg*) addresses (any access outside *xuseg* will cause a trap). The Cavium Networks Linux port offers configurable options which may allow 64-bit user processes to access *xkphys* I/O or memory addresses. In addition, both 64-bit and 32-bit processes may access a special Cavium Networks-specific segment, *cvmseg*, which is in *xkseg* (or *kseg3* for 32-bit processes) virtual address segment.

The address space is divided into segments. Depending on the mode, different segments are visible to the process:

In 64-bit MIPS:

- User mode segments: *useg* (on the OCTEON processor *xkphys* may optionally be accessed in user mode by SE-UM 64-bit applications)
- Supervisor mode: *useg*, *xsseg* (usually not used)
- Kernel mode segments: *xuseg*, *xsseg*, *xkseg*, *xkphys*

In 32-bit MIPS:

- User mode segments: *useg*
- Supervisor mode segment: *useg*, *sseg* (usually not used)
- Kernel mode segments: *useg*, *sseg*, *kseg3*, *kseg0*, *kseg*

10.4 Mapped and Unmapped Segments

Depending on which segment is selected, MIPS also may interpret the SEGBITS part of the virtual address differently than traditional processors. On some traditional processors, all the virtual addresses are *always* mapped (translated by the operating system or TLB).

10.4.1 Unmapped Segments

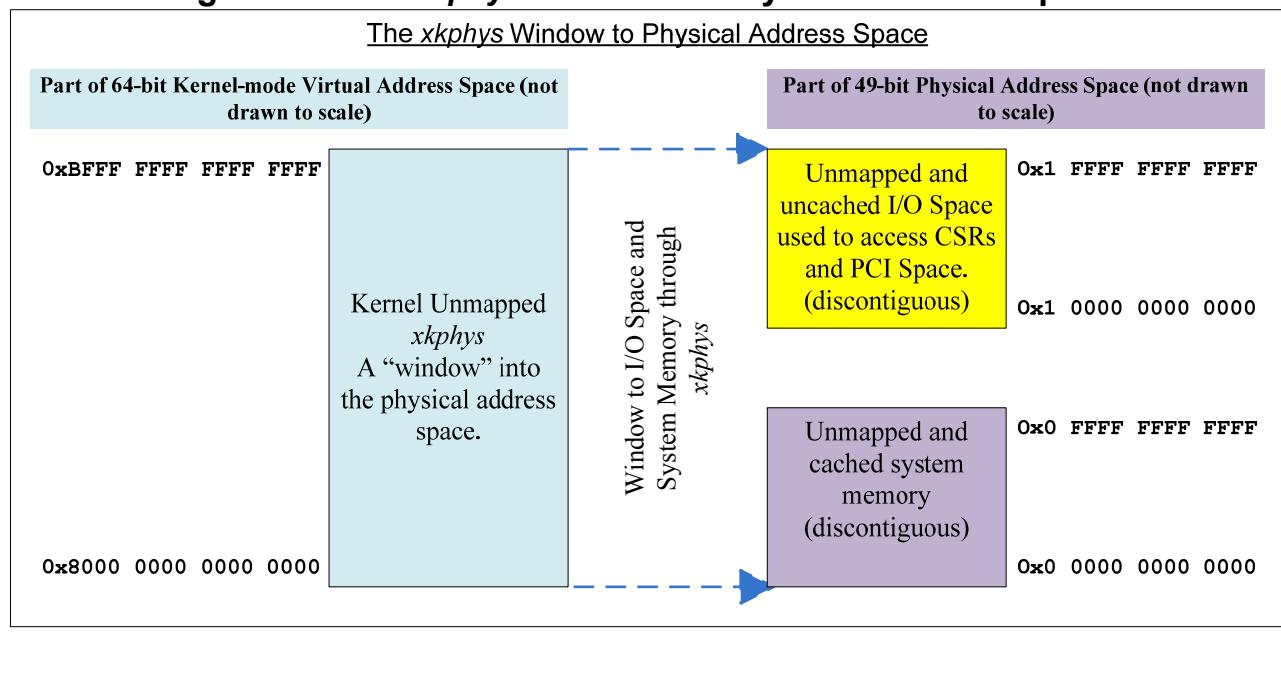
10.4.1.1 64-Bit Virtual Address Space: *xkphys*

On the OCTEON processor, both 64-bit kernel-mode processes and 64-bit user-mode processes may access physical memory and I/O space through the *xkphys* segment.

On MIPS, *xkphys* addresses are not mapped, and are *never* translated by the operating system or TLB. The *xkphys* addresses provide a “window” into the physical address space. The high bits are stripped off the virtual address, and the low PABITS (Physical Address BITS) are used as a physical address. On the OCTEON processor, PABITS is 49: bits <48:0>, matching the number of SEGBITS (49).

Note that the I/O space is selected if bit 48 of the physical address is “1”. Physical memory is selected if bit 48 of the physical address is “0”.

Figure 42: The *xkphys* Window to Physical Address Space



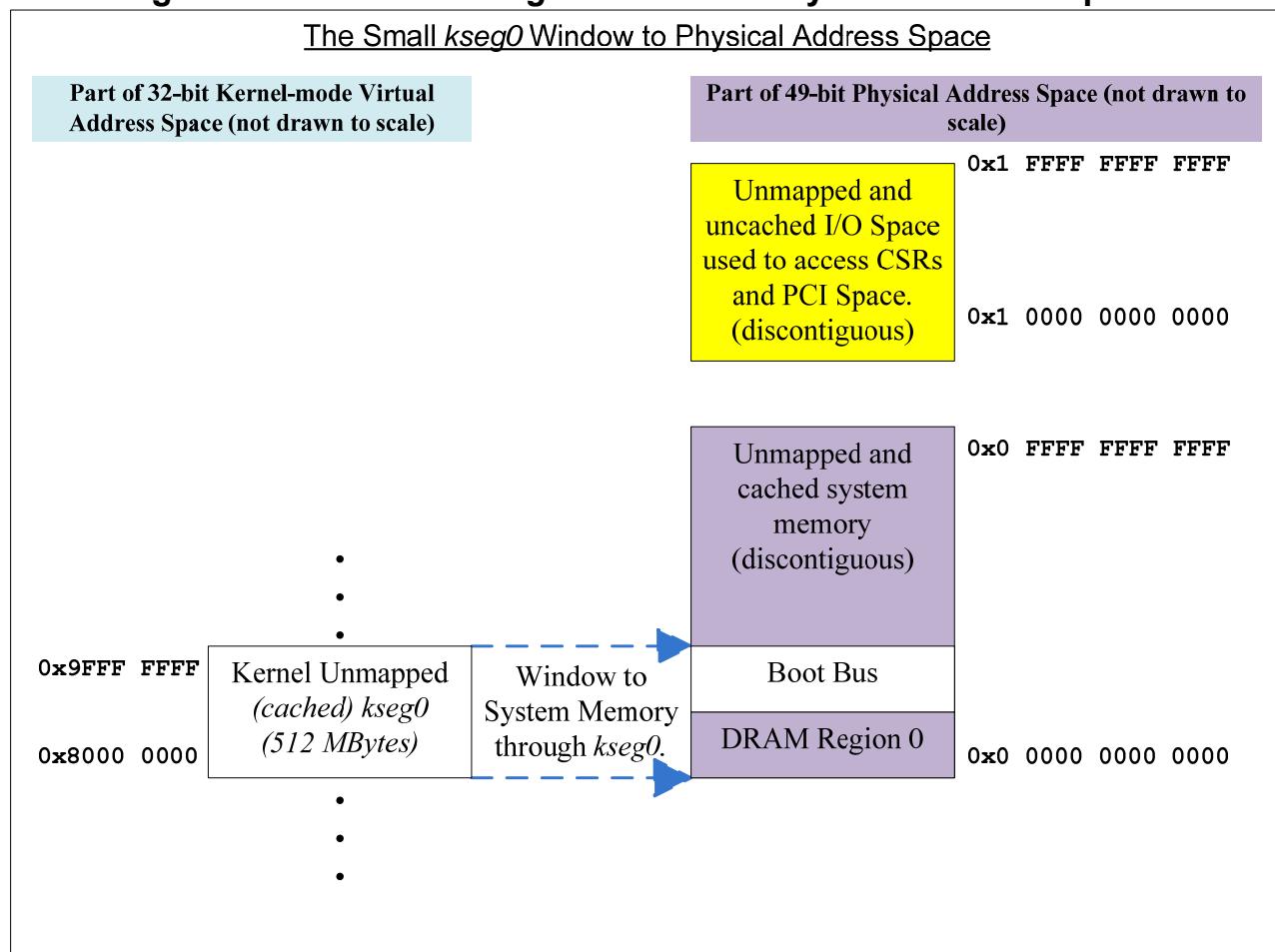
SW OVERVIEW

10.4.1.2 32-Bit Virtual Address Space: *kseg0* and *kseg1*

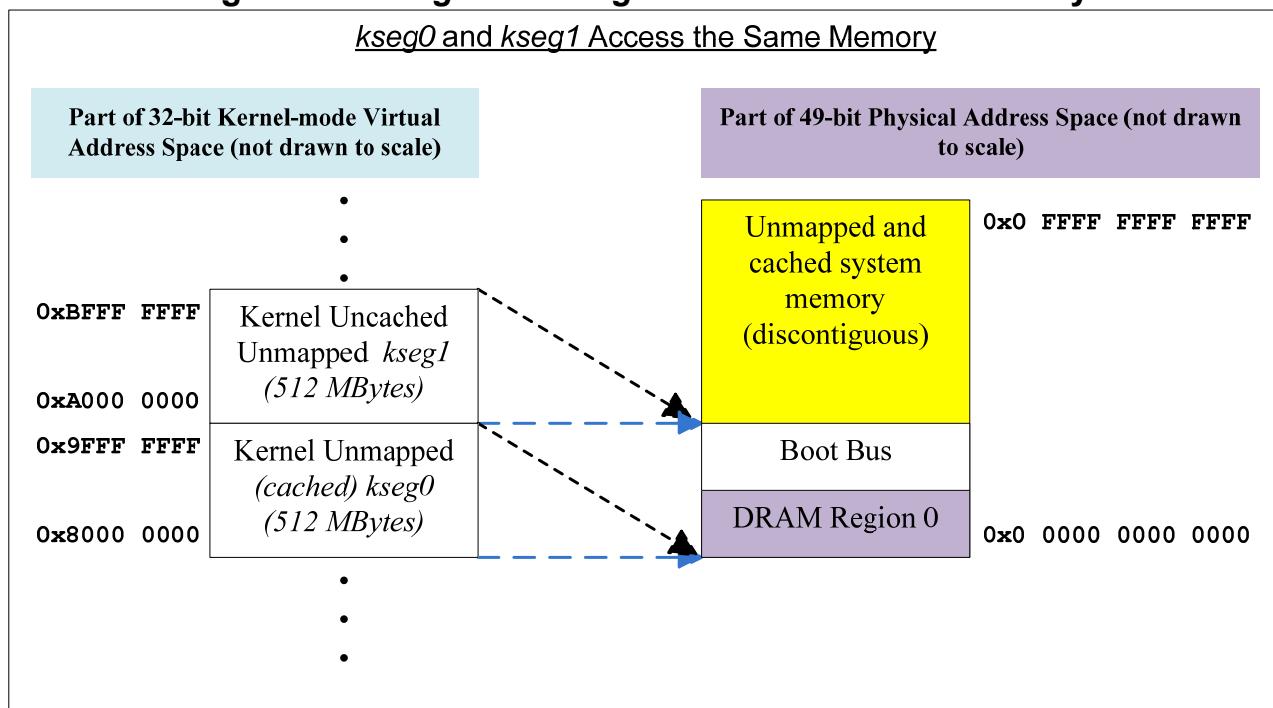
The 32-bit kernel-mode processes have a small window into physical address space though *kseg0*. This window is not large enough to reach the I/O space, and it can only reach the first 256 MBytes of DRAM (DRAM Region 0).

32-bit Simple Executive Standalone applications run in kernel mode and access physical memory through *kseg0* addresses.

Figure 43: The Small *kseg0* Window to Physical Address Space



Note that *kseg0* and *kseg1* access the same system memory. In the generic MIPS memory map, *kseg0* accesses are cached, and *kseg1* accesses are uncached. In the software provided with the OCTEON SDK, kernel-mode accesses to system memory are made through *kseg0*, not *kseg1*. Accesses to system memory on the OCTEON processor are always cached, even those made through *kseg1*.

Figure 44: kseg0 and kseg1 Access the Same Memory

SW OVERVIEW

32-bit Simple Executive User-Mode (SE-UM 32-bit) applications cannot access *kseg0*. Instead, they access system memory through memory mapped into *useg* (the *reserve32* area). The *reserve32* area is discussed in detail in Section 12.3.2 – “SE-UM 32-Bit Bootmem Access”.

10.4.2 Mapped Segments

On some traditional processors, the *Memory Management Unit (MMU)* consists of a TLB and hardware page tables which the operating system can read and write.

On MIPS, the MMU consists only of the TLB: page tables are optional and are implemented entirely in software.

When a program accesses a page which should be mapped, but the mapping is not found in the TLB, a *TLB miss* exception occurs. This exception causes the hardware to jump to a hardware vector and run a page fault handler. The page fault handler looks up the page in the page table, checks access permissions, and if access is allowed, it adds the mapping to the TLB, evicting a prior mapping if needed. Then the page access is retried and the access succeeds.

The following figure shows simplified address translation for the different segments in the 64-bit virtual memory map.

Figure 45: 64-Bit Virtual Address Translation on MIPS

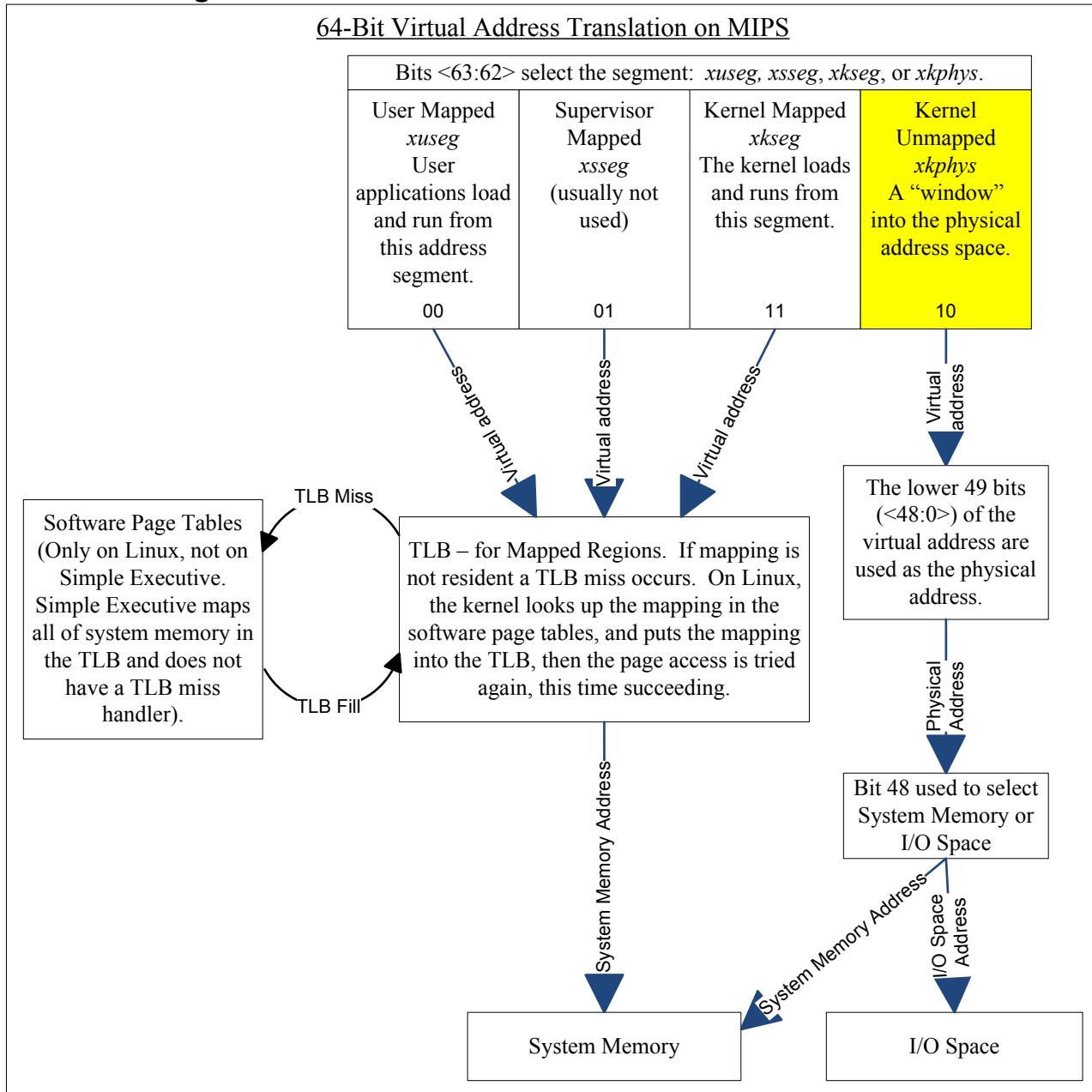
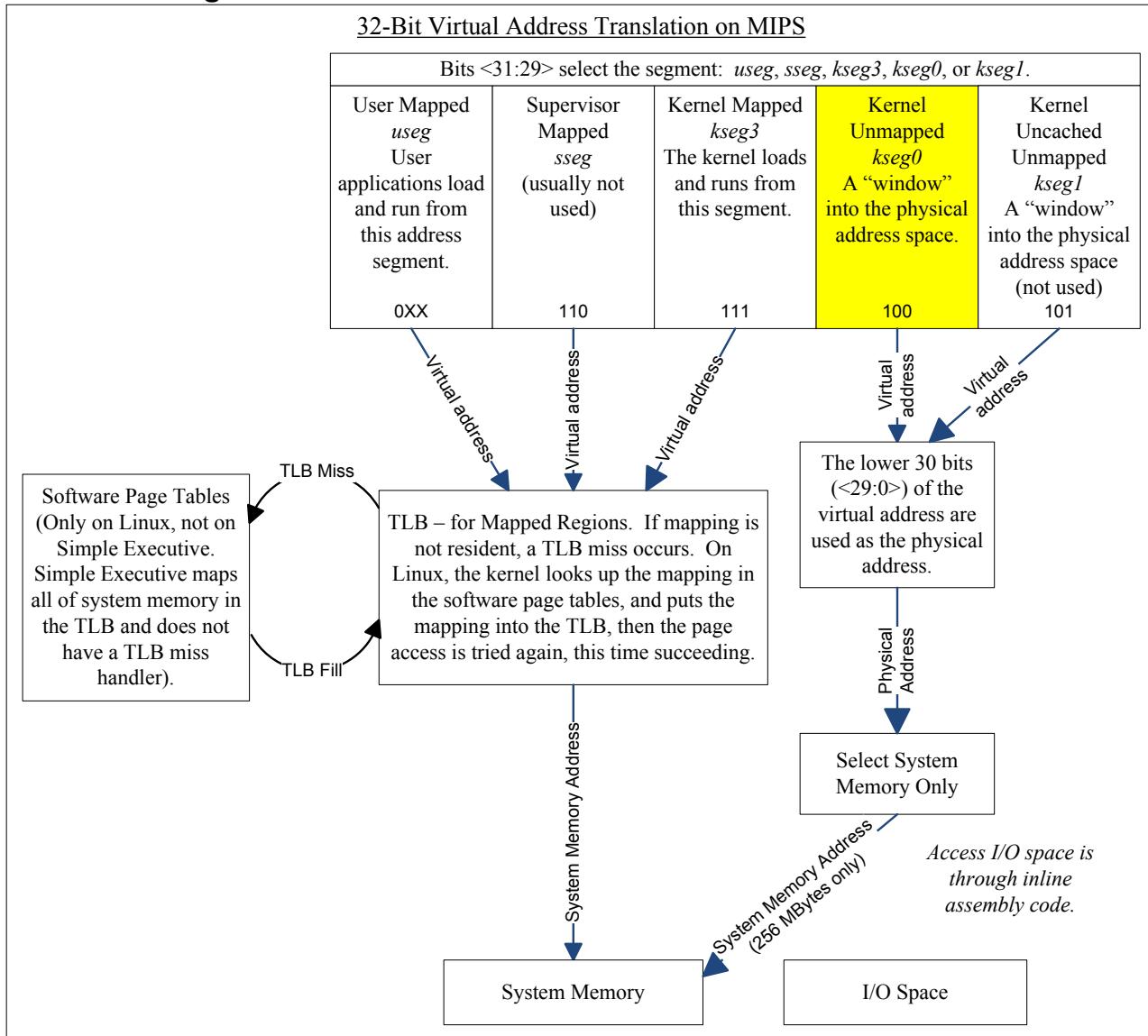


Figure 46: 32-Bit Virtual Address Translation on MIPS

SW OVERVIEW

10.4.3 Addresses Versus Pointers

In this document, the word *pointer* refers to a C or C++ data type which holds a virtual address, NULL, or an invalid address. The word *address* refers to a physical address.

The addresses used by a program are always virtual addresses. Virtual addresses are *not* the same as physical addresses, even if their 64-bit values are the same. Virtual addresses are always interpreted differently by the hardware (segment selector, ignored bits, and SEGBITS). C and C++ programs must therefore always use virtual addresses (pointers), not physical addresses, when accessing memory. Because of this requirement, the Simple Executive API functions such as

`cvmx_fpa_alloc()` use pointer arguments and return values, not addresses. These pointers contain virtual addresses which can be directly used by the application without further conversion.

At the hardware level, transactions requiring addresses use physical addresses. For instance, the “allocate” and “free” *operations* use the physical address of the buffer in DRAM, not a virtual address. The FPA is a hardware unit: it has no concept of the TLB or of virtual address space.

When accessing hardware registers directly, be aware that addresses sent and returned are physical, not virtual addresses. API functions to convert between the two types of addresses are provided: `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()`.

10.5 Virtual Memory on Cavium Networks MIPS (cnMIPS)

The virtual memory on the cnMIPS cores varies from the generic MIPS virtual memory map in the following areas:

- Caching
 - 1. System memory accesses are *always* cached, even those made through *kseg1* addresses.
 - 2. I/O memory accesses are *never* cached.
- Mapping
 - 1. I/O memory is *never* mapped unless explicitly mapped by the user.

In addition, the following virtual memory features were added:

- Special Access to *xkphys* for Linux Users
 - 1. 64-bit Linux applications may optionally access system memory and I/O space via *xkphys* addresses. This is a kernel configuration option. This option is discussed in more detail in Section 12.3 – “Accessing Bootmem Global Memory from SE-UM Applications”. Access to *xkphys* I/O Space or System Memory is controlled by a bit in the Coprocessor 0 (COP0) register *CvmMemCtl* (fields *XKIOENAU* and *XKMEMENAU*).
 - 2. 32-bit Linux applications may optionally reserve a pool of free memory which has physical addresses low enough for 32-bit applications to use. This memory is mapped into *useg*. An example where this is needed is when using FPA buffers: the function `cvmx_fpa_alloc()` returns the address of the allocated buffer, which must fit in 32 bits. This option is discussed in more detail in Section 12.3 – “Accessing Bootmem Global Memory from SE-UM Applications”.
 - a. This pool of reserved memory may optionally be mapped to ALL 64-bit and 32-bit processes on all cores running the same Linux kernel.
- The *cvmseg* segment
 - 1. There is a Cavium Networks-specific *cvmseg* segment. This segment is used for local scratchpad memory and for IOBDMA operations such as `cvmx_fpa_alloc_async()`. This special segment is discussed in more detail in Section 10.6 – “Cavium Networks-Specific *cvmseg* Segment” and in Section 11.3 – “The *cvmx_shared*”. Access to *xkphys* I/O space or system memory is controlled by a bit in the Coprocessor 0 (COP0) register *CvmMemCtl* [fields *CVMSEGENAU*, *CVMSEGENAK*, and *LMEMSZ*].

2. Linux User applications are allowed to access *cvmseg* (in *kseg3*) when doing *cvmseg* access operations. Running in kernel mode is not required. No special configuration is needed for this permission.

10.6 Cavium Networks-Specific *cvmseg* Segment

Part of the per-core data cache (Dcache) may be set aside for IOBDMA operations and scratchpad memory. The amount of Dcache used for *cvmseg* is set when either Simple Executive or Linux is configured.

Note that since space for cvmseg comes from Dcache, keeping the size of cvmseg to a minimum will help system performance by leaving more Dcache blocks available for the application.

The special *cvmseg* memory is be configured at build time for both Simple Executive applications and Linux.

It consists of two segments:

- CVMSEG LM
- CVMSEG IO

The CVMSEG LM memory consists of up to 54 cache blocks taken from the Dcache for this purpose (typically, only 2 or 4 cache blocks are used). Each cache block (cache line) is 128 bytes.

CVMSEG IO has only one valid address: 0xFFFF FFFF FFFF A200. A store instruction to this address starts an IOBDMA operation.

The data written in the IOBDMA instruction includes the CVMSEG LM offset (scratchpad location) where the result of the IOBDMA operation should be stored.

For example: `cvmx_fpa_alloc_async()` will start an IOBDMA operation which will get the address of a free buffer from the FPA, and store the buffer's address in the CVMSEG LM (scratchpad) memory.

The IOBDMA operations are asynchronous (the program does not wait for the result). When the program is ready to use the buffer, it issues a SYNCIOBDMA operation to make sure all the IOBDMA operations for that core have completed, and then retrieves the returned buffer address from the scratchpad.

Note: If an illegal address is provided in an IOBDMA instruction, or the requested number of bytes will exceed the allocated cache lines in CVMSEG LM, but within the range shown in the virtual address map, then the adjacent Dcache memory may be overwritten. (An address error will occur, but stores to these illegal addresses may not be stopped by the hardware, so they may corrupt the Dcache.)

The *cvmseg* addresses are in the *kseg3* address range, and are treated specially by the cnMIPS cores. (Although *cvmseg* is in *xkseg* when using a 64-bit address space, it is referred to as being *kseg3*. The 64-bit address space *contains* the compatibility space, so *kseg3* exists in the 64-bit address space, inside of *xkseg*.) When configured into the system (the default), load and store instructions access *cvmseg*. Otherwise, the access is a normal *kseg3* reference. Access to *cvmseg* is controlled by a bit in the Coprocessor 0 (COP0) register *cvmctl*.

When running Linux, the scratchpad memory is saved and restored on context switches.

10.7 Accessing Application-Private System Memory

Each application has private system memory. This private system memory is mapped into the application's virtual address space.

SE-S applications run in kernel mode, but are mapped into the *xuseg* or *useg* virtual address space, depending on whether they are 64-bit or 32-bit applications.

SE-UM applications run in user mode and are mapped into the *xuseg* or *useg* virtual address space, depending on whether they are 64-bit or 32-bit applications.

The Linux Kernel runs in kernel mode and is mapped into *xkseg*. It is always 64-bit.

10.8 Summary of Virtual Address Space on cnMIPS

The MIPS virtual address space is divided into segments. The 64-bit virtual address space contains a 32-bit compatibility mode address space.

The MIPS memory management unit is simplified, and consists only of a TLB. Page tables are optionally implemented in software.

Mapped: A virtual address is ‘mapped’ when access is through a TLB entry.

Cached: When a virtual address accesses system memory, the system memory is “cached” if it is stored in the L1 and/or L2 cache for fast access. On the OCTEON processor, all system memory accesses are cached.

In general, user-mode processes cannot access kernel-mode virtual address space. On the OCTEON processor, there are some exceptions to this rule and the generic MIPS virtual memory map.

Table 12: The 64-Bit Virtual Address Segments

Segment	Generic MIPS	OCTEON cnMIPS
	Mapped Segments	
<i>xuseg</i>	The <i>xuseg</i> segment is the user address space (mapped).	SE-S 64-bit applications run in kernel-mode, but are mapped into <i>xuseg</i> .)
<i>xsseg</i>	The <i>xsseg</i> segment is the supervisor address space (usually not used) (mapped).	The <i>xsseg</i> segment is usually not used in OCTEON cnMIPS.
<i>xkseg</i>	The <i>xkseg</i> segment is in the kernel address space (mapped).	The <i>xkseg</i> segment contains the OCTEON-specific <i>cvmseg</i> segment. User-Mode access is allowed to <i>cvmseg</i> .
	Unmapped Segments	
<i>xkphys</i>	The <i>xkphys</i> segment is in the kernel address space. It is an unmapped address space: a window into the physical address space: system memory and I/O space.	SE-UM 64-bit applications may be allowed access to <i>xkphys</i> addresses. SE-S 64-bit applications always have access to <i>xkphys</i> addresses (they run in kernel-mode). Accesses to system memory are <i>always</i> cached. Accesses to I/O space are <i>never</i> cached.
<p><i>Note:</i> The Linux kernel always runs in 64-bit mode. SE-UM and SE-S applications may run in either 64-bit or 32-bit mode. SE-S applications always run in kernel-mode.</p>		

Table 13: The 32-Bit Virtual Address Segments

Segment	Generic MIPS	OCTEON cnMIPS
	Mapped Segments	Mapped Segments
<i>useg</i>	The <i>useg</i> segment is the user address space (mapped).	OCTEON SE-S 32-bit applications run in kernel-mode, but are mapped into <i>useg</i> .
<i>sseg</i>	The <i>sseg</i> segment is the supervisor address space (usually not used) (mapped)	This segment is usually not used.
<i>kseg3</i>	The <i>kseg3</i> segment is in the kernel address space (mapped)	User-Mode access is allowed only to <i>cvmseg</i> in this segment.
<i>kseg0</i>	Unmapped Segments	Unmapped Segments
	The <i>kseg0</i> segment is in the kernel address space (unmapped, uncached)	Accesses to this segment access system memory which is <i>always</i> cached on OCTEON. SE-S 32-bit applications run in kernel-mode and access system memory through <i>kseg0</i> addresses.
<i>kseg1</i>	The <i>kseg1</i> segment is in the kernel address space (unmapped, cache attribute not defined)	Accesses to this segment accesses system memory which is <i>always</i> cached on the OCTEON processor.
<i>Note:</i> The Linux kernel always runs in 64-bit mode. Relative to a SE-UM 32-bit virtual address space, <i>cvmseg</i> is in <i>kseg3</i> . SE-S applications always run in kernel-mode.		

SE-S 64-bit applications run in kernel mode and are mapped to *xuseg*.

SE-S 32-bit applications run in kernel mode and are mapped into *useg*.

SE-UM 64-bit applications run in user mode and are mapped into *xuseg*.

SE-UM 32-bit applications run in user mode and are mapped into *useg*.

The Linux kernel is always 64-bit, runs in kernel mode, and is mapped into *xkseg*.

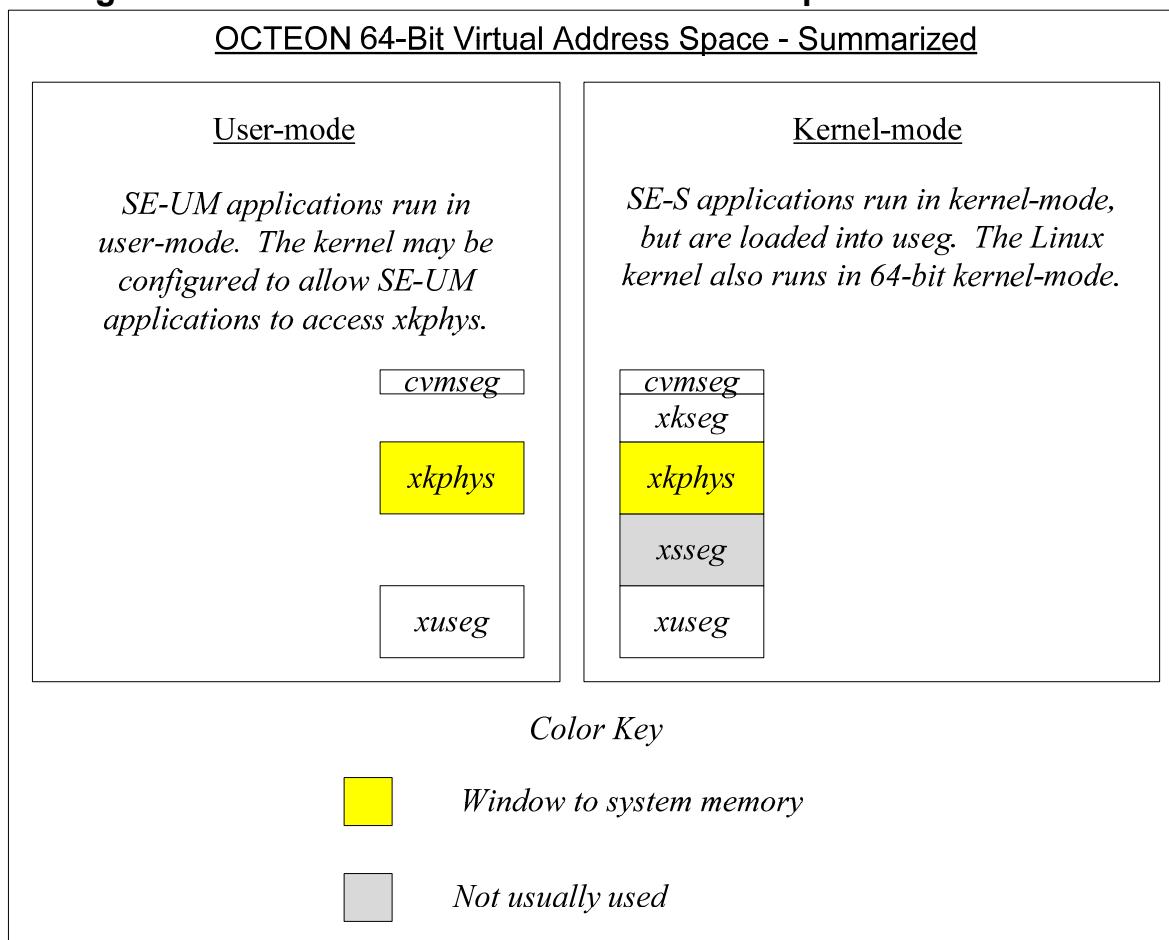
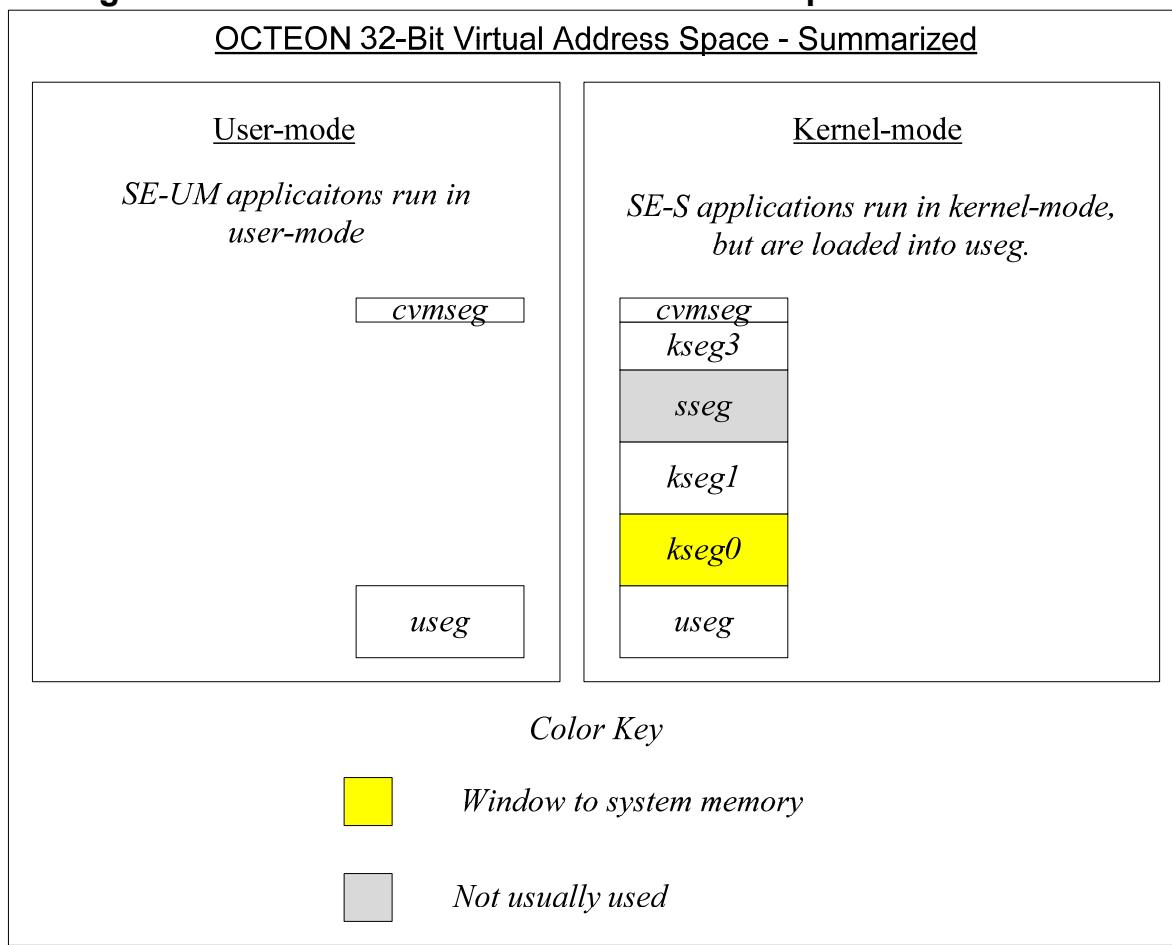
Figure 47: OCTEON 64-Bit Virtual Address Space – Summarized

Figure 48: OCTEON 32-Bit Virtual Address Space - Summarized



11 Allocating and Using Bootmem Global Memory

11.1 Using Global Bootmem

Large chunks of system memory are needed to create the FPA buffer pools. After the pools are created, the memory is usually shared between all cores on the processor, regardless of what in-memory images the cores are running. For instance: both Linux kernel-mode and user-mode processes, and Simple Executive Standalone processes read and write to Packet Data Buffers.

Processors may also need to allocate chunks of memory for other purposes.

At boot time, the bootloader creates a pool of all free memory, *bootmem*. This memory is managed by the bootmem allocator functions. These functions provide the needed locking so that two applications will not get the same memory, and return the appropriate virtual address of the allocated memory region.

Note that memory allocated via these functions is uninitialized: it is not guaranteed to be all zeroes.

Memory allocated via bootmem allocator functions is referred to as *bootmem global memory*.

The memory allocation functions are multicore safe: the free list will not become corrupted if different cores may simultaneous requests.

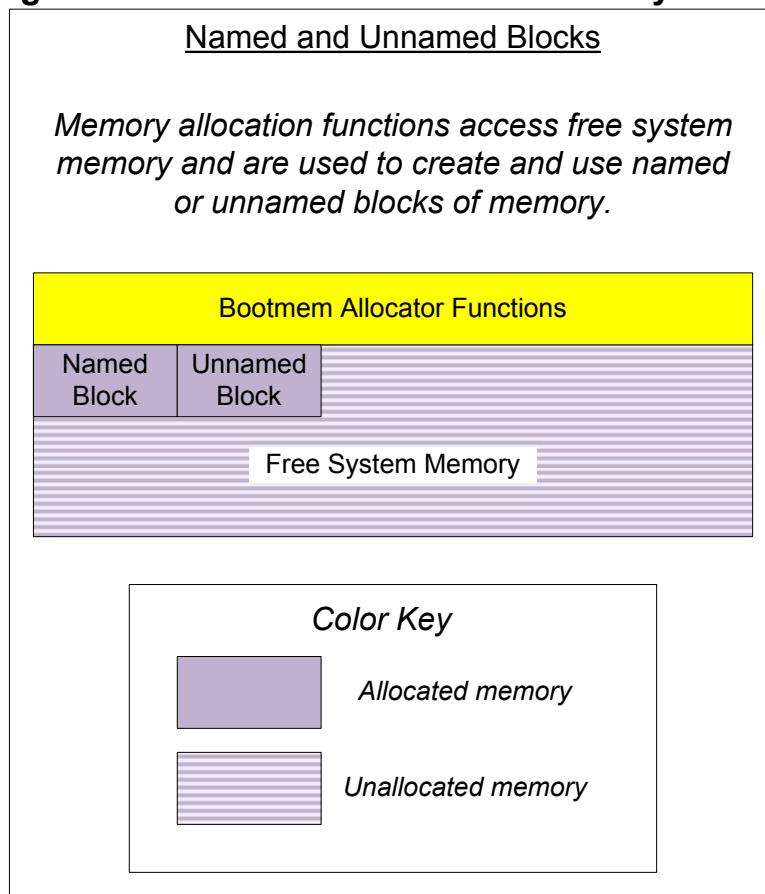
Table 14: Bootmem Allocator Functions in SDK 1.8

Function	Action
cvmx_bootmem_alloc()	Allocate a chunk of contiguous system memory (unnamed block). This memory cannot be freed. If enough contiguous memory is available to satisfy the request, the function returns a pointer to the start of the block, otherwise returns 0.
cvmx_bootmem_alloc_address()	Allocate a chunk of contiguous system memory (unnamed block). This memory cannot be freed. Specify the specific starting physical address desired. If the requested address has not already been allocated, and enough contiguous memory is available, the function returns a pointer to the start of the block, otherwise it returns 0.
cvmx_bootmem_alloc_named()	Allocate a chunk of contiguous system memory (named block), and name it. If the named block has not already been created, and enough contiguous memory is available to satisfy the request, the function returns a pointer to the start of the block, otherwise it returns 0. This memory block can be freed.
cvmx_bootmem_alloc_named_address()	Allocate a chunk of contiguous system memory (block), and name it. Specify the specific starting physical address desired. If the named block has not already been created, and the requested address has not already been allocated, and enough contiguous memory is available, the function returns a pointer to the start of the block, otherwise it returns 0. This memory block can be freed.
cvmx_bootmem_find_named_block()	Find a named block which has already been allocated. If the block is found, the function returns a pointer to the start of the block, otherwise it returns 0.
cvmx_bootmem_free_named()	Free the entire named block, and free the name.

Note that some of the allocation functions allow processes to allocate memory, but not free it. To be able to free the memory, named blocks must be used: `cvmx_bootmem_alloc_named()` and `cvmx_bootmem_alloc_named_address()`. Note that `cvmx_bootmem_free_named()` is for limited use to free temporary allocations, for instance the bootloader uses this function to free the Reserved Download Block. This function should not be used frequently: there is no memory defragmentation. If you need to free memory frequently, do not use bootmem functions.

As shown in the figure below, chunks of allocated bootmem are stored as either named or unnamed blocks. The bootmem allocator functions are responsible for managing both unallocated and allocated memory.

Figure 49: Named and Unnamed Memory Blocks



11.2 The `malloc()` and `free()` Functions and FPA Buffers

The C-library functions `malloc()` and `free()` only manage core-local memory. This memory can not be used for FPA buffers.

11.3 The *cvmx_shared* Section and FPA Buffers

There are two reasons why the *cvmx_shared* section may not be the best choice to create a large amount of shared memory, for instance for FPA buffers: the memory is not always shared, and it should be kept small to keep the ELF file and the in-memory image small.

11.3.1 The *cvmx_shared* Section is Not Always Shared

The *cvmx_shared* section provides shared memory between cores started with the *same* load command (the same load set):

- for SE-S applications, the same `bootoct` bootloader command
- for SE-UM applications, the same `oncpu` Linux command

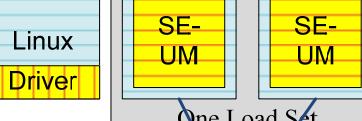
The cores started with the same load command are referred to as a *load set*. Note that each of the Simple Executive applications is a process, not a thread: global variables are not shared between cores.

The *cvmx_shared* section *cannot* be used to share memory between processes started with different load commands (different load sets).

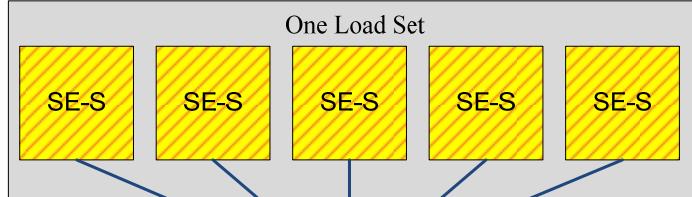
Figure 50: *cvmx_shared*: Same and Different Load Sets

cvmx_shared: Same and Different Load Sets

SMP Linux or other SMP-capable OS (single copy). The SE-UM processes started with one `oncpu` command.



All SE-S images started with one bootloader command are in the same load set.



Shared text, rodata, and *cvmx_shared* section.

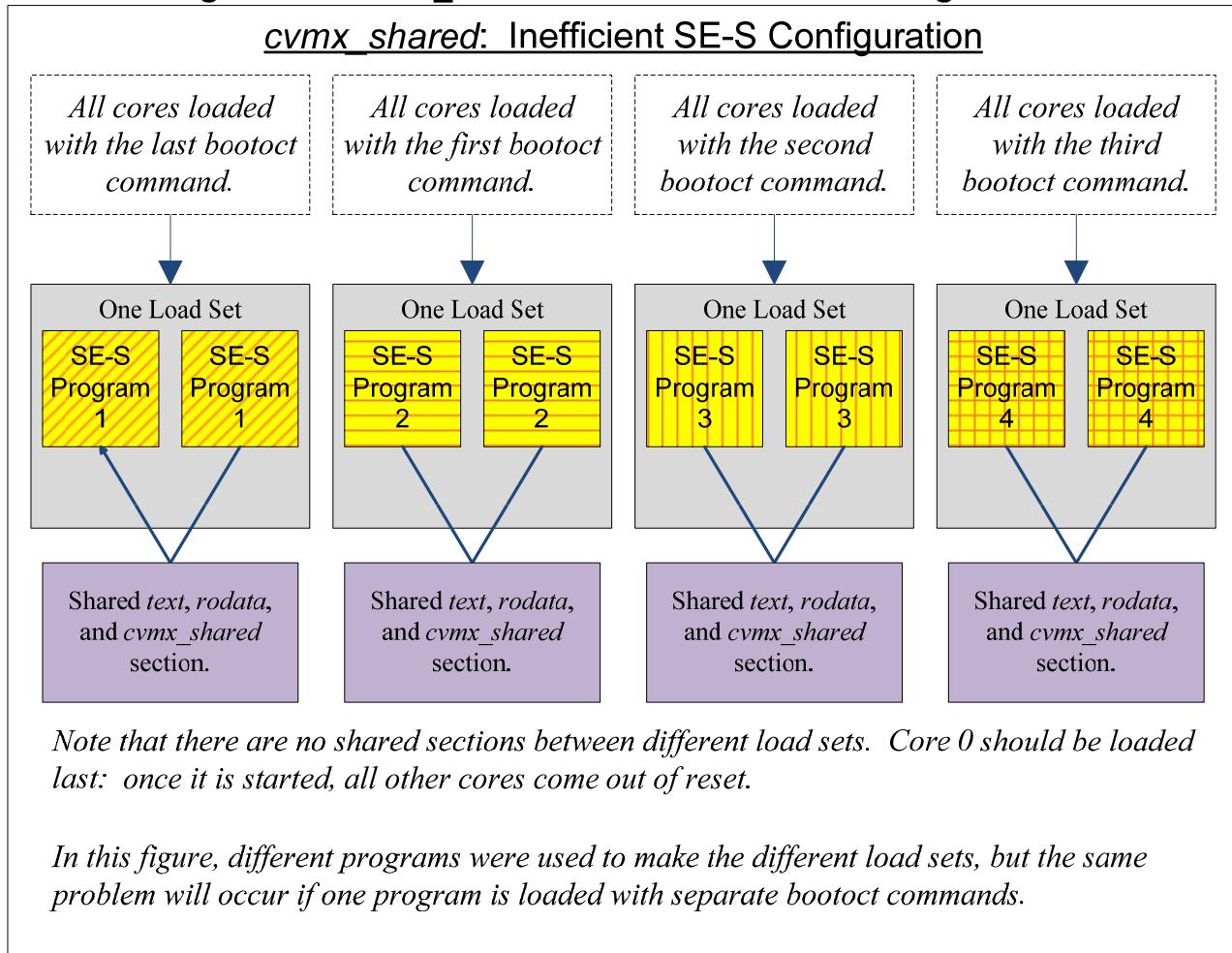
Shared text, rodata, and *cvmx_shared* section.

Note that the Cavium Networks Ethernet driver does not share the *cvmx_shared* section with the SE-UM applications.

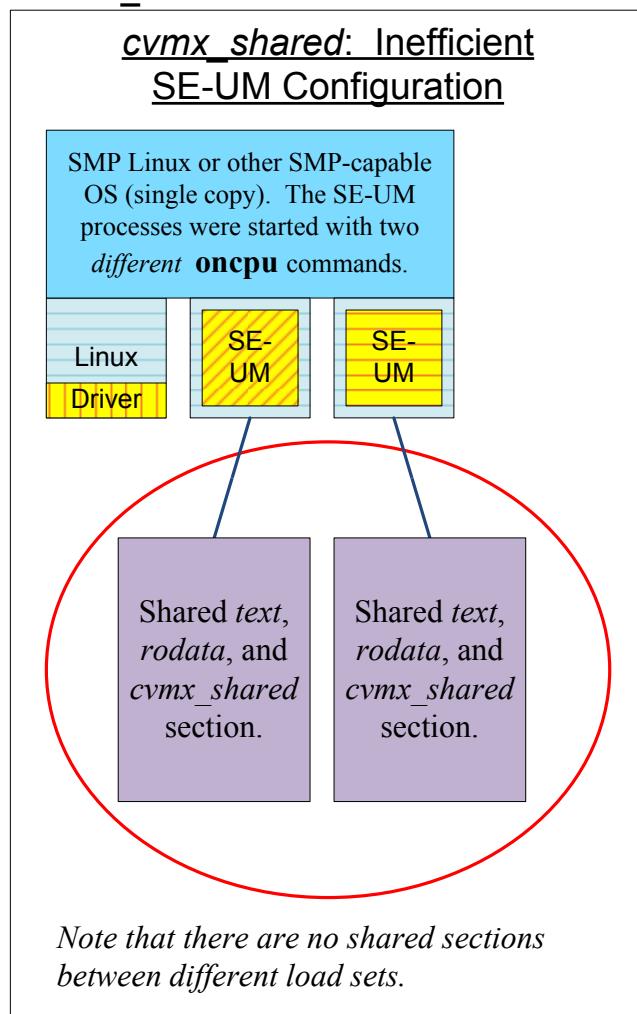
As shown in the figure above, one `oncpu` command is used to start multiple SE-UM applications on Linux so they will share the same load set.

As shown in the following figure, it is inefficient to load different SE-S applications because they will not share common sections: they will be in different load sets.

Figure 51: *cvmx_shared*: Inefficient SE-S Configuration



Similarly, it is inefficient to start SE-UM applications with two different `oncpu` commands.

Figure 52: *cvmx_shared*: Inefficient SE-UM Configuration

SW OVERVIEW

11.3.2 The *cvmx_shared* Section Should be Kept Small

It is not a good idea to use *cvmx_shared* to contain large amounts of shared memory. It is best to keep the size of the loaded ELF file small. The current (SDK 1.8) maximum ELF file download size is 256 MBytes. Also, some Simple Executive Standalone applications must fit into 256 MBytes of virtual memory (if 1:1 mapping is used). If a large *cvmx_shared* section has been created, the ELF file may not fit into virtual memory, causing the bootloader to fail. See Figure 54 – “Simple Executive Size Limitation if 1:1 Mapping is Used”.

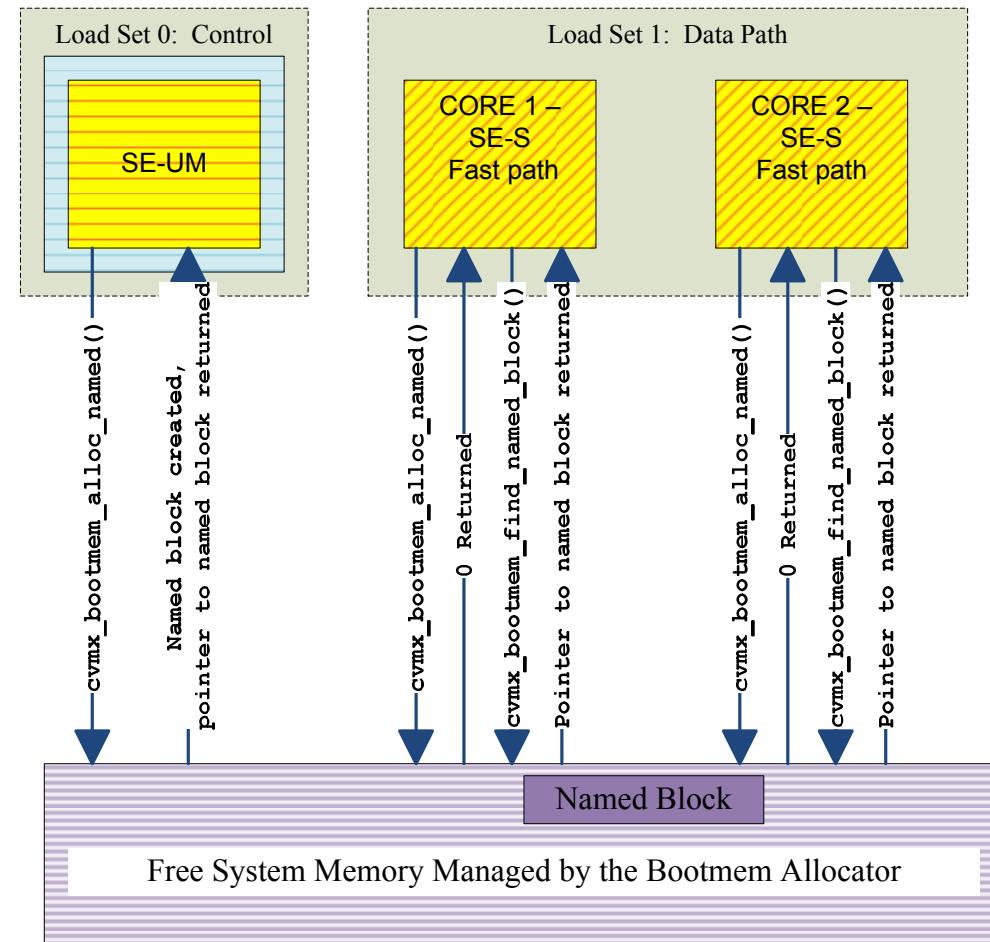
The best use of *cvmx_shared* is to create a pointer to shared memory, then allocate the memory on startup, and put the address into the *cvmx_shared* pointer. This keeps the size of the *cvmx_shared* section small, while still providing a large amount of shared memory.

11.4 Using Named Blocks to Share Memory Between Different Load Sets

To share system memory between cores running different load sets, use the *named block* bootmem allocator functions: `cvmx_bootmem_alloc_named()`, `cvmx_bootmem_find_named_block(name)`), etc.

By using named blocks, two different load sets such as Simple Executive and Linux may easily share memory:

- Both cores call `cvmx_bootmem_alloc_named()` to allocate memory and name it.
- The first core to make the function call creates the named memory block; all other cores which call the same function with the same named block will get a return value of “0”, which tells them that the named block has already been created.
- If the return value is “0”, they call `cvmx_bootmem_find_named_block(name)`) to get the address of the existing named block.

Figure 53: Sharing Memory Between Different Load SetsIsolated Processes Can Use Bootmem Functions to Share Memory

The first core to call `cvmx_bootmem_alloc_named()` creates the named block, and a pointer to the named block is returned. All other cores call the same function, and receive a return value of 0. Then they call `cvmx_bootmem_find_named_block()`, providing the name of the block, and receive a pointer to it in return.

The core running the Linux kernel is not shown in this picture.

The following code is from \$OCTEON_ROOT/examples/queue/queue.c.

```

/**
 * Gets a pointer to a named bootmem allocated block,
 * allocating it if necessary. This function is called
 * by all cores, and they will all get the same address.
 *
 * @param size    size of block to allocate
 * @param name    name of block
 *
 * @return Pointer to shared memory (physical address)
 *         NULL on failure
 */
void *get_shared_named_block(uint64_t size, char *name)
{
    void *ptr = cvmx_bootmem_alloc_named(size, 128, name);
    if (!ptr)
    {
        /* Either this core did not allocate it, or the allocation
request
        ** cannot be satisfied. Look up the block, and if that fails,
        ** then the allocation cannot be satisfied
        */
        if (cvmx_bootmem_find_named_block(name))
            ptr = cvmx_phys_to_ptr(
                cvmx_bootmem_find_named_block(name) ->base_addr);
    }

    return(ptr);
}

```

An example use of named blocks is to create a spinlock shared between different load sets.

12 Accessing Bootmem Global Memory (Buffers)

A simple example of accessing memory happens in packet processing. One process allocates memory for the FPA pools, divides it into buffers, and gives the buffers to the FPA to manage. The PIP/IPD automatically allocates Work Queue Entry Buffers and Packet Data Buffers. Any core can perform the `get_work` operation, which returns a Work Queue Entry Buffer. Now the core must access the buffer.

The most important thing to know about accessing memory is to use the functions `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()`. If these functions are used, all the complexity in the following discussion is hidden from the user.

Table 15: Summary of Access to System Memory and I/O Space

Runtime Environment	Virtual Address Space	Run Mode	Application -Private Memory	Bootmem Global Memory	I/O Space Access	Notes
SE-S: 64-bit no 1:1 mapping	64 bit	kernel mode	<i>xuseg (See Note 1)</i>	<i>xkphys</i>	<i>xkphys</i>	Preferred configuration - safest for porting.
SE-S: 64-bit with 1:1 mapping	64 bit	kernel mode	<i>xuseg (See Note 1)</i>	<i>xuseg</i>	<i>xkphys</i>	Using 1:1 mapping can result in porting problems if <code>cvmx_phys_to_ptr()</code> and <code>cvmx_ptr_to_phys()</code> are not used.
SE-S: 32-bit no 1:1 mapping	32 bit	kernel mode	<i>useg (See Note 1)</i>	<i>kseg0</i>	inline assembly code	This is the preferred configuration: safest for porting, but only 256 MBytes of memory are addressable through <code>kseg0</code> .
SE-S: 32-bit with 1:1 mapping	32 bit	kernel mode	<i>useg (See Note 1)</i>	<i>useg</i>	inline assembly code	Using 1:1 mapping can result in porting problems if <code>cvmx_phys_to_ptr()</code> and <code>cvmx_ptr_to_phys()</code> are not used.
Linux kernel and drivers	64 bit	kernel mode	<i>xkseg</i>	<i>xkphys</i>	<i>xkphys</i>	
Linux SE-UM: 64-bit	64 bit	user mode	<i>xuseg</i>	<i>xkphys</i>	<i>xkphys</i>	Kernel configuration option provides <code>xkphys</code> access to user-mode processes.
Linux SE-UM: 32-bit	32 bit	user mode	<i>useg</i>	<i>useg (reserve32)</i>	inline assembly code	A <code>reserve32</code> region is mapped into the address space of the process. Each application is limited to 2 GBytes of virtual address space.
<p><i>Note 1:</i> Although SE-S applications are run in kernel-mode, they use the <code>xuseg</code> or <code>useg</code> address space for application-private memory, depending on whether the application is 64-bit or 32-bit.</p>						

12.1 Accessing Bootmem Global Memory From SE-S Applications

12.1.1 SE-S 64-Bit Bootmem Access

64-bit SE-S applications may access bootmem global memory through either *xkphys* addresses or *xuseg* addresses.

12.1.1.1 SE-S 64-Bit: Access Via *xkphys* (NO 1:1 Mapping)

SE-S applications run in kernel mode which allows them access to the *xkphys* segment. In order to select this option, when configuring the Simple Executive, set CVMX_USE_1_TO_1_TLB_MAPPINGS to 0 (FALSE). No mapping step is needed because *xkphys* accesses are not mapped.

12.1.1.2 SE-S 64-Bit: Access Via *xuseg* (1:1 Mapping)

If CVMX_USE_1_TO_1_TLB_MAPPINGS is 1 (TRUE), then all of system memory is mapped into the process address space (*xuseg*) by `cvmx_user_app_init()`. All of bootmem global memory is accessible to any SE-S application: a separate mapping step is not needed because it has already been done.

This is discussed in more detail in Section 14.4 – “Simple Executive Virtual Memory Configuration Options”.

12.1.2 SE-S 32-Bit Bootmem Access

32-bit SE-S applications may access bootmem global memory through either *kseg0* addresses or *useg* addresses.

12.1.2.1 SE-S 32-Bit: Access Via *kseg0* (NO 1:1 Mapping)

SE-S applications run in kernel mode which allows them access to the *kseg0* segment. In order to select this option, when configuring the Simple Executive, set CVMX_USE_1_TO_1_TLB_MAPPINGS to 0 (FALSE). No mapping step is needed because *kseg0* accesses are not mapped.

12.1.2.2 SE-S 32-bit: Access Via *useg* (1:1 Mapping)

If CVMX_USE_1_TO_1_TLB_MAPPINGS is 1 (TRUE), then all of system memory is mapped into the TLB by `cvmx_user_app_init()`. The access is via *useg*. Note that the user will only be able to access the low addresses (within the 2 GByte *useg* address range). A separate mapping step is not needed because it has already been done by `cvmx_user_app_init()`.

This is discussed in more detail in Section 14.4 – “Simple Executive Virtual Memory Configuration Options”.

12.2 Accessing Bootmem Global Memory From Linux Kernel: 64-Bit

The Linux kernel-mode processes such as the kernel and drivers access bootmem global memory via *xkphys* addresses.

12.3 Accessing Bootmem Global Memory from SE-UM Applications

12.3.1 SE-UM 64-Bit Bootmem Access

The 64-bit Simple Executive User-Mode applications access bootmem global memory via *xkphys* addresses, not *xuseg* addresses. No mapping step is needed, because *xkphys* is a window to system memory. This is a configurable kernel option.

12.3.2 SE-UM 32-Bit Bootmem Access

The 32-bit Simple Executive User-Mode applications must access bootmem global memory via *useg* addresses, because *xkphys* addresses are outside the 32-bit virtual address space.

To allow the same code to be compiled as either a 32-bit or 64-bit application, some special processing will happen, hidden from the user. Without this special processing, the 32-bit SE-UM process would have to call `mmap()` to map the bootmem global memory into its address space. The code would have to be changed to handle this case, and runtime performance would be degraded.

The special processing involves setting aside the bootmem global memory during system start-up to preserve the lowest memory addresses for the 32-bit process to allocate using the bootmem functions.

- When the kernel is configured, a special *reserve32* named block is specified. The size of this named block is specified at configuration time.
- When the kernel is booted, it calls `cvmx_bootmem_alloc_named()` to allocate bootmem global memory for the *reserve32* named block. Because low memory addresses are allocated first, this action preserves the low memory addresses.
- After the kernel initializes the rest of memory, it frees the *reserve32* named block. The free list now contains a chunk of contiguous memory with low addresses.
- The previously reserved memory is now available to be the first block of free memory allocated by the bootmem allocator.

The user does not need to map *reserve32* into the process virtual address space: when the application runs, the Simple Executive function `main()` calls `mmap()` to map all of *reserve32*. (Note that this mapping includes system memory which has not been allocated by the process: thus the process has access to system memory which it does not own.)

Later when applications ask for memory there are two cases:

1. If a SE-UM 32-bit process calls `cvmx_bootmem_alloc()`, the function internally limits the range of memory to the addresses range of the original *reserve32* region. If enough contiguous memory cannot be found, the request fails. Typically, the SE-UM 32-bit process is responsible for allocating any shared memory which it needs to access, to guarantee that the allocated memory is within its address range. For example, a SE-UM 32-bit application which will use Packet Data Buffers must allocate the memory for them, and will usually initialize all of the FPA pools.
2. If a SE-S or SE-UM 64-bit processes calls `cvmx_bootmem_alloc()`, the function will attempt to get bootmem global memory from the address range in the original *reserve32* block

simply because it is first in the free list. If there is not enough contiguous memory to satisfy the request, the function will continue to search the free list for memory outside of the *reserve32* region.

This process will be discussed in more detail in Section 15.1.4 – “SE-UM 32-bit: Reserving a Pool of Free Memory”.

12.4 Bootmem Size in Different Access Methods

The 32-bit and 64-bit applications have different amounts of bootmem available, depending on the exact configuration.

The following table summarizes how system limits are affected by different configurations.

Table 16: Configuration Choices and Resultant Global Memory Limits

Application Type	Variations	Virtual Address Space - size	Load Image Maximum Size	Bootmem Global Memory Access	Bootmem Global Memory Accessible from the Application
SE-S Applications					
SE-S 64-bit	NO 1:1 Mapping	<i>xuseg - "unlimited"</i> (see Note 4)	"unlimited"	<i>xkphys</i>	ALL DRAM (see Note 2)
SE-S 64-bit	1:1 mapping	<i>xuseg - "unlimited"</i> (see Note 4)	256 MBytes (squeezed by mapped memory)	<i>xuseg</i>	ALL DRAM (see Note 2)
SE-S 32-bit	NO 1:1 Mapping	<i>useg - 2 GBytes</i>	2 GBytes max (see Note 1, Note 3)	<i>kseg0</i>	256 MBytes (See Note 2)
SE-S 32-bit	1:1 mapping	<i>useg - 2 GBytes</i>	256 MBytes (squeezed by mapped memory)	<i>useg</i>	no more than 2 GBytes (useg limit) (see Note 2)
Linux SE-UM Applications					
SE-UM 64-bit	N/A	<i>xuseg - "unlimited"</i> (see Note 4)	"unlimited" (see Note 1, Note 3)	<i>xkphys</i>	ALL DRAM (see Note 2)
SE-UM 32-bit	reserve32 - not wired	<i>useg - 2 GBytes</i>	2 GBytes minus reserve32 size	<i>useg: reserve32</i>	Blocks of DRAM in power of 2. Application load size must be less than 2 GBytes. (See Note 2)
SE-UM 32-bit	reserve32 - wired	<i>useg - 2 GBytes</i>	2 GBytes minus reserve32 size	<i>useg: reserve32</i>	512, 1024, or 1536 MBytes (see Note 2)
Notes					
Note 1: Huge load images may encounter problems loading. The maximum load size shown here is not guaranteed.					
Note 2: Bootmem size is limited by the amount of DRAM which is supported by and installed in the target system.					
Note 3: The current (SDK 1.8) maximum ELF image download size is 256 MBytes. The loaded image includes stack and bss, so the loaded image is larger than the ELF image file.					
Note 4: Although there is a limit to the size of <i>xuseg</i> , for practical purposes it is "unlimited".					

12.5 Using `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()` Functions

If conversion is needed between pointers and physical addresses, use the functions

`cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()`. This will allow the same code for both SE-S and SE-UM applications, and reduce porting complexity.

Since not using these functions can cause big problems for customers, the warning is repeated here:

Note: Be careful to use the functions `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()` when converting between physical addresses and virtual addresses. 90% of porting problems come from mistakenly using casts on physical and virtual addresses.

13 Accessing I/O Space

Various `cvmx` functions are used to hide any complexity in accessing the I/O Space:

```
static void cvmx_write_csr (uint64_t csr_addr, uint64_t val)
static void cvmx_write_io (uint64_t io_addr, uint64_t val)
static uint64_t cvmx_read_csr (uint64_t csr_addr)
static void cvmx_send_single (uint64_t data)
static void cvmx_read_csr_async (uint64_t scraddr, uint64_t csr_addr)
```

This section describes how the different accesses occur (the hidden complexity). For a summary, see Table 15 – “Summary of Access to System Memory and I/O Space”.

13.1 Accessing I/O Space from SE-S Applications

13.1.1 SE-S 64-Bit I/O Space Access

In Simple Executive Standalone (SE-S) applications run in kernel mode and access I/O space through `xkphys` addresses.

13.1.2 SE-S 32-Bit I/O Space Access

In Simple Executive Standalone (SE-S) applications run in kernel mode and access I/O space through inline assembly instructions. See Section 13.3.2 – “SE-UM 32-Bit I/O Space Access” for more information.

13.2 Accessing I/O Space from Linux Kernel: 64-Bit

The Linux kernel-mode processes such as the kernel and drivers access I/O Space via `xkphys` addresses.

13.3 Accessing I/O Space from SE-UM Applications

13.3.1 SE-UM 64-Bit I/O Space Access

The 64-bit Simple Executive User-Mode applications may access I/O Space via `xkphys` addresses. This option is configured into the kernel.

13.3.2 SE-UM 32-Bit I/O Space Access

I/O Space is accessed by using inline assembly instructions.

When using the functions `cvmx_read_csr()` and `cvmx_write_csr()`, all the complexity described below is hidden from the user. The technical details are included here for readers who need more detail.

Accessing I/O Space from 32-bit applications requires conversion between 32-bit pointers and 64-bit address values.

In the N32 ABI (used to compile SE-S 32-bit and SE-UM 32-bit applications), pointers are 32-bit values, and registers are 64-bit values. Since OCTEON hardware always uses 64 bits for memory access, and registers are 64-bit values, inline assembly can be used to bypass the 32-bit pointer limitation.

In O32 ABI (not recommended), pointers are 32-bit values, and registers (as viewed from the ABI) are 32-bit values. Hardware registers are always physically 64-bit values; it is just the O32 ABI that thinks they are only 32-bit values. Since O32 doesn't know about the 64-bit registers, it stores all 64-bit values in two separate registers. If the stored value is an address, to access the address quite a few assembly-language steps are needed:

1. Shift the high order bits into the upper bits of a register and add the lower bits.
2. Do the memory read, specifying the now 64-bit address in the 64-bit register.
3. Convert the 64-bit response into two 32-bit registers.
4. Make sure all registers touched are properly truncated to 32bits.
5. Return to C code.

A common error is forgetting step #4, because it is not obvious that you need to restore registers which are no longer needed.

This is why O32 is slower than N32 when doing CSR access.

The functions `cvmx_read64_uint()` and `cvmx_write_64_uint()` handle the special conversions required. The functions `cvmx_read_csr()` and `cvmx_write_csr()` are then thin wrappers around these functions.

14 Simple Executive Standalone (SE-S) Memory Model

Simple Executive Standalone (SE-S) applications run in kernel mode. All of the system memory is mapped, allowing Simple Executive applications full access to memory, including memory they do not own. 64-bit SE-S applications may also freely access the I/O space by using `xkphys` addresses. There are no context switches, and no TLB misses. SE-S applications are lightweight and fast.

On startup the bootloader and Simple Executive function `cvmx_user_app_init()` create a kernel-mode address space where *all* address mapping is complete by the time the application initialization routine completes. There are no expected TLB misses when running under SE-S: there is no exception handler. A TLB miss will cause the system to crash, because there is no TLB

miss handler for the hardware exception. The system would need to be reset or power cycled to recover.

Note: Even when virtual addresses are used, SE-S applications can overwrite memory they do not own because all system memory is mapped!

The file \$OCTEON_ROOT/executive/cmvx.mk will include the file \$OCTEON_ROOT/executive/cvmx-app-init.c when a Simple Executive target is specified on the “make” command line. This file includes the application initialization code cvmx_user_app_init().

14.1 Simple Executive Application Space

Applications are loaded into *xuseg* or *useg* at virtual address 0x1000 0000.

There is some stack overflow protection. When the bootloader allocates memory for the stack, it leaves the page below the stack unmapped, so that any access to this region will generate a TLB exception.

14.2 Simple Executive System Memory Access

Hardware units only use physical, not virtual memory addresses. A function such as cvmx_fpa_alloc() will convert the physical address into a virtual address as needed, returning a pointer to the buffer.

To access the corresponding physical address, this address must be converted to a physical address. Conversion functions are supplied by Simple Executive (cvmx_ptr_to_phys() and cvmx_phys_to_ptr()).

14.2.1 Mapping of System Memory

System memory may optionally be mapped 1:1 to the user’s address space, so that physical address 0 is virtual address 0. This configuration is not recommended, however for historical reasons it is currently the default.

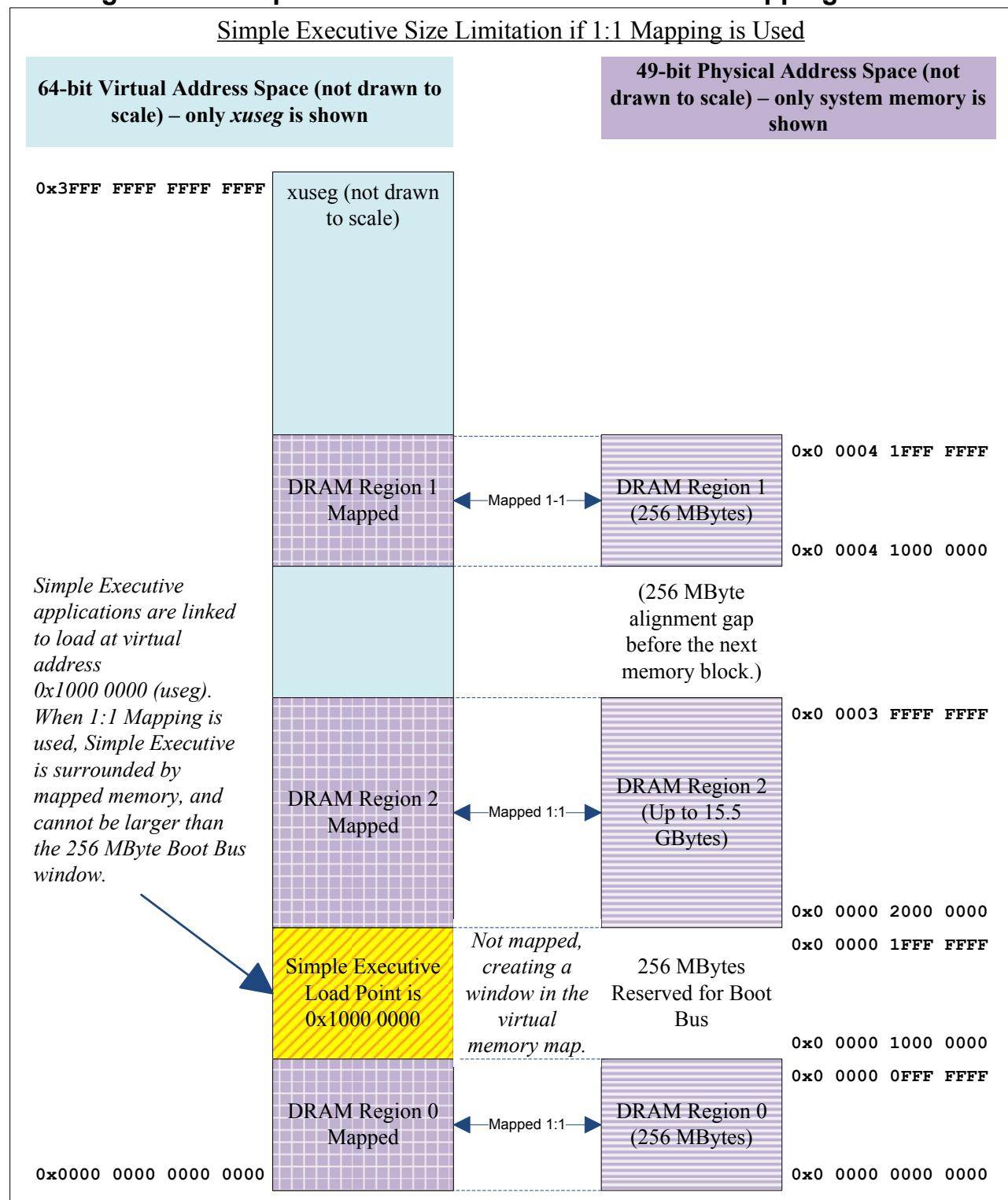
The 1:1 mapping allowed for “lazy” address translation, but causes two problems:

1. Porting problems occurred when code was not written to use the cvmx_ptr_to_phys() and cvmx_phys_to_ptr() functions. These functions hide pointer / address conversions, creating highly portable code.
2. The size of the Simple Executive application’s runtime size was limited to 256 MBytes. (Note the application’s in-memory image size is larger than the ELF file size because it includes memory allocated for the stack and heap.)

The reason the in-memory image size is limited to 256 MBytes is because the Simple Executive application will be loaded into the virtual memory map, squeezed between two blocks of system memory (see figure below). If 1:1 mapping is not used, Simple Executive applications load at 0x1000 0000, but memory can be mapped anywhere instead of immediately above and below the application, so the application can be larger than 256 MBytes.

The following figure is shown using the 64-bit virtual address space for simplicity. A similar problem exists in the 32-bit virtual address space.

If CVMX_USE_1_TO_1_TLB_MAPPINGS is defined to 1, then the application must fit inside of 256 MBytes (0x2000 0000).

Figure 54: Simple Executive Size Limitation if 1:1 Mapping is Used

The Simple Executive application's virtual memory map is shown in the figure below. If 1:1 mapping is not used, memory is not mapped into the user segment. Instead it is accessed either via *xkphys* (for 64-bit applications), or via inline assembly code.

Figure 55: SE-S 64-Bit Virtual Memory Map

<u>SE-S 64-Bit Virtual Memory Map</u>	
SE-S 64-Bit Virtual Address Space (not drawn to scale) (Only the relevant subset of the virtual memory map is shown.)	
0xFFFF FFFF FFFF BFFF	CVMSEG – IO (only valid address = 0xFFFF FFFF FFFF A200)
0xFFFF FFFF FFFF A000	
0xFFFF FFFF FFFF 9FFF	CVMSEG – LM (part of DCACHE)
kseg3	
0xFFFF FFFF FFFF 8000	
0x8001 6700 0000 03FF	Unmapped and uncached I/O Space (accessed through <i>xkphys</i>) (discontiguous where there is no matching I/O device)
0x8001 0000 0000 0000	
0x8000 0004 1FFF FFFF	
xkphys	Unmapped and uncached system memory (accessed through <i>xkphys</i>) (discontiguous where system memory is not present)
0x8000 0000 0000 0000	
0x0000 0004 1FFF FFFF	Mapped and cached system memory: Second 256 MBytes of DRAM
0x0000 0004 1000 0000	
0x0000 0003 FFFF FFFF	Mapped and cached system memory: Upper 15.5 GBytes of DRAM (as much as is present)
0x0000 0000 2000 0000	
0x0000 0000 1FFF FFFF	
xuseg	Application Space (256 Mbytes) The size of the loaded application must not exceed 256 MBytes.
0x0000 0000 1000 0000	
0x0000 0000 0FFF FFFF	Mapped and cached system memory: First 256 MBytes of DRAM (the first MByte is unmapped)
0x0000 0000 0000 0000	

Virtual address “0” (the first 1 MByte) is usually unmapped. If virtual address “0” is unmapped, a NULL pointer access will cause that core to crash because there is no TLB exception handler.

Memory may be accessed through *xuseg* (for example, 0x0000 0000 0020 0000) or through *xkphys* (0x8000 0000 0020 0000). Accesses through *xuseg* are mapped. Accesses through *xkphys* are unmapped. System memory is always cached, whether it is accessed through *xuseg* or *xkphys*.

Note that although all of system memory is mapped to each application, it does not necessarily belong to that application. It is possible to overwrite memory belonging to another application. Careful coding is needed.

14.3 Simple Executive I/O Space Access

The hardware IO Space is accessed only via *xkphys*. The IO space is unmapped and uncached. This IO space includes the configuration and status registers for the various hardware units.

14.4 Simple Executive Virtual Memory Configuration Options

Note that in the figure above, there are two compile-time defines:

`CVMX_USE_1_TO_1_TLB_MAPPINGS` and `CVMX_NULL_POINTER_PROTECT`.

General information on configuring Simple Executive may be found in the SDK document “*OCTEON SDK config and build system*”.

14.4.1 CVMX_USE_1_TO_1_TLB_MAPPINGS

The value of `CVMX_USE_1_TO_1_TLB_MAPPINGS` is set to 1 by default.

The use of 1:1 TLB mappings is discouraged because it leads to many time-consuming bugs to solve when porting code. SE-S code which uses 1:1 TLB mappings will function without use of `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()`. When run as SE-UM, the code breaks.

The 1:1 TLB mappings value *must* be changed to 0 if the application exceeds 256 MBytes.

Note that fewer TLB entries are needed if 1:1 mapping is not used (except when `CVMX_NULL_POINTER_PROTECT` is 1): each double TLB entry will map 512 MBytes of memory.

In all cases, memory access should go through `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()` to safely convert between virtual and physical addresses. By using this access routine, the address translation will occur as needed, transparent to the user.

14.4.1.1 Changing the Value of CVMX_USE_1_TO_1_TLB_MAPPINGS

To change the value of `CVMX_USE_1_TO_1_TLB_MAPPINGS`, configure `CVMX_CPPFLAGS_GLOBAL_ADD` to contain the string “`-DCVMX_USE_1_TO_1_TLB_MAPPINGS=0`”, as shown in the following bash shell script (named “`doit.sh`”):

```
#!/bin/bash

source env-setup OCTEON_CN38XX # change this to the correct OCTEON_MODEL

export OCTEON_CPPFLAGS_GLOBAL_ADD="$OCTEON_CPPFLAGS_GLOBAL_ADD
-DCVMX_USE_1_TO_1_TLB_MAPPINGS=0" # all one line, not
two lines

echo $OCTEON_CPPFLAGS_GLOBAL_ADD
```

Then “source” doit.sh:

```
host$ source ./doit.sh # Correct!
host$ echo $OCTEON_CPPFLAGS_GLOBAL_ADD
-DCVMX_USE_1_TO_1_TLB_MAPPINGS=0
```

This will cause the application initialization code to not setup 1:1 mappings, and also will direct cvmx_phys_to_ptr() and cvmx_ptr_to_phys() to do the proper conversions.

Note: If you do not “source” doit.sh after the script exits, the values set when it was run will no longer be set:

```
host$ ./doit.sh # Wrong!!! The file must be "sourced"
host$ echo $OCTEON_CPPFLAGS_GLOBAL_ADD
host$ # the variable is not set when doit.sh exits
```

14.4.2 CVMX_NULL_POINTER_PROTECT

CVMX_NULL_POINTER_PROTECT is also set to 1 by default. This setting causes an extra 12 TLB entries to be consumed. To recover the TLB entries, you can set this define to 0. If that happens, NULL pointer accesses will not be rejected by the system. Since this space is reserved for use by the bootloader, even after it exits, an accidental store to this area may create problems.

14.4.2.1 Changing the Value of CVMX_NULL_POINTER_PROTECT

This value can be changed by editing cvmx-config.h.

```
***** Config Specific Defines
*****
#define CVMX_LLM_NUM_PORTS 1
#define CVMX_NULL_POINTER_PROTECT 0 // 0 = FALSE
```

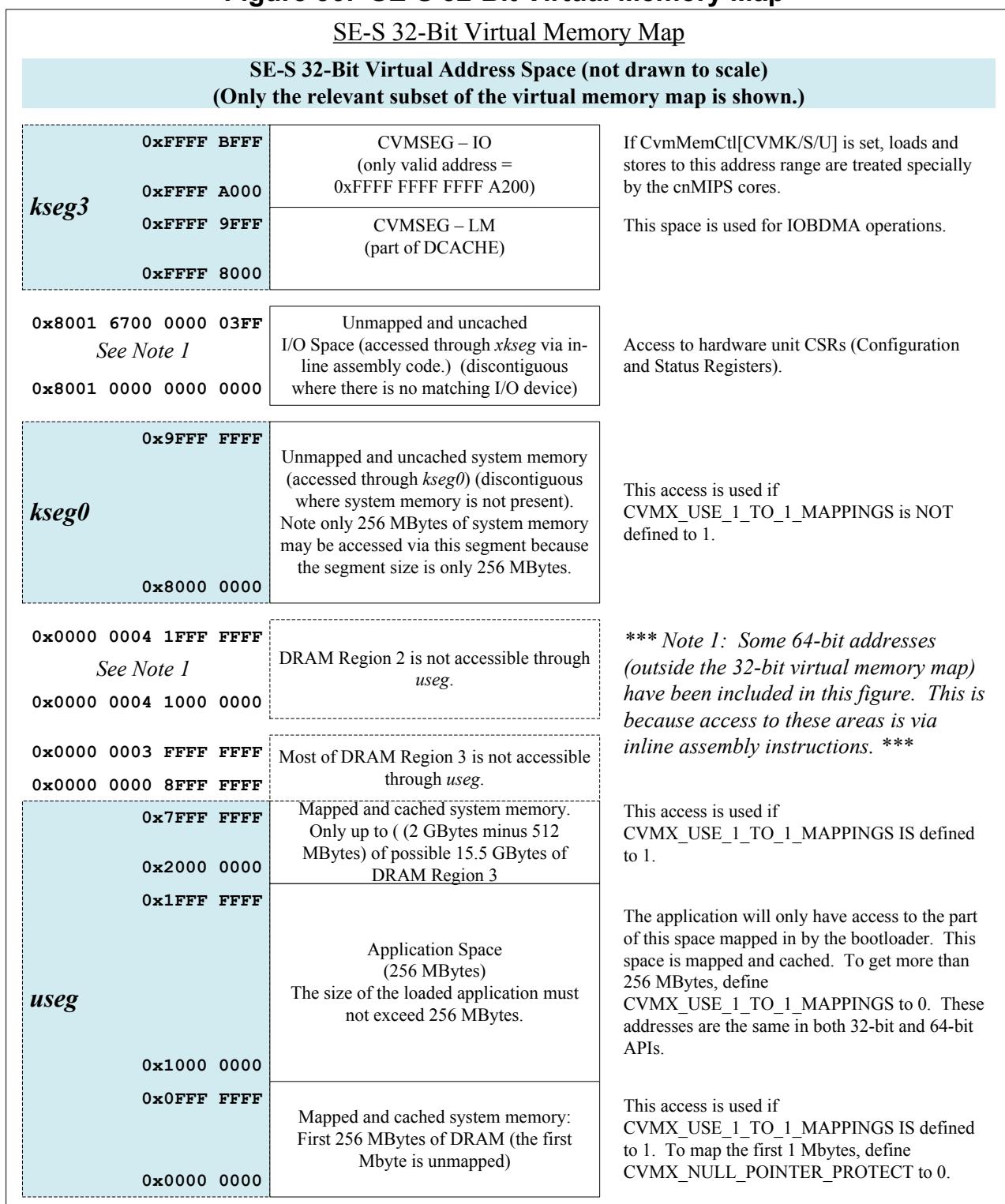
Note this file is local to the application’s config directory. It will be automatically read when the application is rebuilt.

14.5 SE-S 32-Bit Applications

Simple Executive Standalone applications do nothing special about memory allocation. They assume the results from cvmx_bootmem_alloc() will be in the lower 2 GBytes of memory. Most of the time that assumption is true because the boot memory allocator returns the

low addresses first, so higher addresses will not be returned for the first allocations (unless the allocations are huge.) 32-bit applications currently must use range limits (`cvmx_bootmem_alloc_address()` or `cvmx_bootmem_alloc_named_address()`) for allocations if they require 32-bit addressable memory. This requirement is expected to change in a future SDK. At that time, the boommem allocation functions will not return memory which is out of the process address range. Note that the range of addresses allocated will be from physical address `<address>` to `<address+size>`.

Note that the 32-bit *useg* virtual address space is only 2 GBytes.

Figure 56: SE-S 32-Bit Virtual Memory Map


15 Linux Memory Model

The Linux kernel is always 64 bits. The Cavium Networks Ethernet driver runs in kernel mode. User-Mode applications run on the kernel may be either 64-bit or 32-bit applications.

When Simple Executive User-Mode (SE-UM) applications are run on Linux, access to system memory is different than for SE-S applications.

Unless configured otherwise, 64-bit applications can only access system memory and I/O space through mapped *xuseg* addresses, not *xkphys* addresses. This would require an extra *mmap()* step before using allocated addresses, so the kernel should be configured to support *xkphys* access.

32-bit applications can never use *xkphys* addresses because they are outside the 32-bit virtual address space. Unless configured otherwise, 32-bit applications would have to map system memory before using it, which would require conditionalized code and would also hurt runtime performance. Instead the kernel can be configured to support a *reserve32* area of memory at addresses accessible to 32-bit SE-UM applications.

Whenever possible, use 64-bit SE-UM applications. This will result in improved performance, and simpler code: they can access the physical address space through *xkphys* addresses, allowing access to I/O space and all of system memory.

When the application is compiled, the file `$OCTEON_ROOT/executive/cmvx.mk` will include `cvmx-app-init-linux.c` instead of `cvmx-app-init.c` when a SE-UM target is specified on the `make` command line. This will cause `main()` to be run for SE-UM. The equivalent function for SE-S applications is `cvmx_user_app_init()`. The Linux-specific `main()` will initialize the SE-UM applications.

15.1 Configuring Linux and the Effect on the Memory Model

Cavium Networks-specific options may be configured when the kernel is built. The exact details of the Linux memory model is controlled by several configuration parameters which are set by running “`make menuconfig`” in the `$OCTEON_ROOT/linux/kernel_2.6/linux` directory.

There are five configuration options which affect the virtual memory map:

1. The size of *cvmseg*
2. Whether 64-bit applications can use *xkphys* addresses to access I/O space.
3. Whether 64-bit applications can use *xkphys* addresses to access system memory.
4. How much free memory should be reserved for 32-bit applications (*reserve32*).
5. Whether the *reserve32* memory should be wired so all applications can access it.

15.1.1 Linux *cvmseg* (IOBDMA and Scratchpad) Size

The configuration variable `CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE` is used to set the number of Dcache lines to reserve for scratchpad and IOBDMA use.

Note that specifying a large *cvmseg* will reduce the number of Dcache blocks available for process use, which can degrade performance.

15.1.2 SE-UM 64-Bit: Direct Access to I/O Space Via *xkphys*

If the configuration variable `CONFIG_CAVIUM_OCTEON_USER_IO` is set to “1” (true), then 64-bit applications may access I/O space thorough *xkphys* without switching to kernel mode. Although these processes run in user mode, special access is allowed via bits in the COP2 *cvmctl* register. This is a kernel configuration option.

This option is used to allow SE-S applications to be compiled as SE-UM applications without changing the code.

15.1.3 SE-UM 64-Bit: Direct Access to System Memory Via *xkphys*

If the configuration variable `CONFIG_CAVIUM_OCTEON_USER_MEM` is set to “1” (true), then 64-bit applications may access system memory through *xkphys* without switching to kernel mode. Note that *xkphys* memory accesses are not mapped, unlike *xuseg* accesses. Although these processes run in user mode, special access is allowed via bits in the COP2 *cvmctl* register. This is a kernel configuration option.

This option is used to allow SE-S applications to be compiled as SE-UM applications without changing the code.

15.1.4 SE-UM 32-bit: Reserving a Pool of Free Memory

In Section 12.3.2 – “SE-UM 32-Bit Bootmem Access”, the problem of how a 32-bit SE-UM application accesses system memory was introduced. Because a 32-bit SE-UM application can not access system memory via *xkphys* addresses, it accesses system memory via a continuous block of virtual addresses within the *useg* address range: *reserve32*. The *reserve32* block is reserved by the kernel, but this memory does not “belong” to the SE-UM application. When the SE-UM application starts up, `main()` will `mmap()` *reserve32* into the virtual address space of the process. The application must use `cvmx_bootmem_alloc()` to allocate the memory.

Note that the application may access system memory which it does not “own” because the entire reserve32 region is mapped to its address space. This can create a security problem because protection from writing into un-owned system memory are absent.

If the configuration variable `CONFIG_CAVIUM_RESERVE32` is set to a legal value, then the *reserve32* region will be set up by the kernel. This region is shared by all SE-UM applications, both 32-bit and 64-bit.

The *reserved32* region is needed to allow SE-S applications to be compiled as 32-bit SE-UM applications without changing the code.

Note: This option is configured-in so that the kernel will reserve a contiguous block of system memory for the reserve32 region. When using reserve32 in a hybrid system, boot Linux first to make sure enough low memory is available for reserve32. The 32-bit application is then responsible for hardware initialization (such as initializing the FPA).

This is necessary so that buffer pointers, such as Packet Data Buffers, are created from memory in the reserve32 region which is already mapped into the 32-bit address space.

After allocating memory, use the `cvmx_phys_to_ptr()` and `cvmx_ptr_to_phys()` functions to convert between physical and virtual addresses as needed.

SE-UM 32-bit applications will allocate memory *only* from `reserve32`. SE-UM 64-bit applications and 32-bit and 64-bit SE-S applications will have all of memory to allocate from, and may or may not allocate memory from `reserve32`.

If the memory will be “wired” (described in the next section), then only a limited amount of memory is available (512 MBytes, 1024 MBytes, or 1536 MBytes). Otherwise, the only restriction is that the memory be a power of 2.

Note: Because 32-bit application space is limited to 2 GBytes, if 1.5 GBytes are set up in reserve32, only 512 MBytes are left for the rest of the application.

The file `/proc/octeon_info` contains the physical address of the `reserve32` region after the kernel is booted if the memory was successfully allocated.

If there not enough memory for the `reserve32` region, an error message is printed at boot time and the physical addresses of the `reserve32` region in `/proc/octeon_info` are set to zero, as if the `reserve32` region was not configured.

15.1.4.1 Using Wired TLB Entries for `reserve32`

`CONFIG_CAVIUM_RESERVE32_USE_WIRED`: map the free memory into every process (32-bit and 64-bit) (including Linux binaries like `bash`).

Specifying wired TLB means that the mapping will stay resident in the TLB (cannot be evicted and replaced by a different mapping).

When using this option, the amount of `reserve32` is limited to the following choices: 512 MBytes, 1024 MBytes, or 1536 MBytes.

When using wired TLB entries, the entire `reserve32` region is mapped into the address space of every 32-bit and 64-bit application (including Linux binaries like `bash`) on all cores running the same SMP Linux image (started from the same boot command).

Warning: Wired `reserve32` presents a huge security risk for the system. Allowing applications to access system memory or I/O space without switching to Kernel Mode will allow one rogue application to corrupt system memory, which can result in difficult-to-debug errors in unrelated applications.

For some applications, this option can result in a significant improvement in performance (up to 3 times faster). For example, mapping 512 MBytes using 4 KByte pages takes 131,072 entries (the TLB has 128 entries (64 double entries)). When using wired TLB, 512 MByte pages are mapped, resulting in only 1-3 TLB entries consumed, depending on the size of `reserve32`.

Warning: *For some applications, this option can degrade performance because TLB entries are consumed, causing more TLB misses as processes contend for fewer remaining entries. The impact of this option on performance is application-dependent.*

Use of this option should be delayed until the performance tuning phase of product development.

15.2 Linux Kernel Space and Simple Executive API Calls

The OCTEON Ethernet driver is an example of a kernel-space use of Simple Executive API calls.

The kernel may use the `cvmx` functions, but they are used differently than for Simple Executive applications:

- There is no equivalent of `appmain()` (*The main() function (for instance in linux-filter.c) is aliased to appmain(), so the function actually running instead of main() is appmain().*)
- Each SE-UM instance is a single-threaded process.
- Global variables are shared
- The `cvmx_shared` section has no meaning (there is no other process to share memory with)

15.3 Linux Memory Configuration Steps

The following options are relevant to the userspace memory map and are all set via `menuconfig`. There are more `menuconfig` options than are mentioned in this section. Only the options affecting the memory map are mentioned here.

Table 17: Cavium Networks-Specific Linux `menuconfig` Options

Option	Variable in <code>autoconfig.h</code>	Default Value	Brief Description
Number of L1 cache lines reserved for CVMSEG memory.	<code>CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE</code>	2	This memory is reserved for CVMSEG LM, the dcache lines set aside for IOBDMA operations.
Allow User space to access hardware IO directly.	<code>CONFIG_CAVIUM_OCTEON_USER_IO</code>	1 (yes)	64-bit applications can access the OCTEON I/O registers without switching to kernel mode.
Allow User space to access memory directly	<code>CONFIG_CAVIUM_OCTEON_USER_MEM</code>	1 (yes)	64-bit applications can access hardware buffers (such as FPA buffers) without switching to kernel mode.

Option	Variable in <code>autoconfig.h</code>	Default Value	Brief Description
Memory to reserve for user processes shared region (MB).	<code>CONFIG_CAVIUM_RESERVE32</code>	0 Mb	The number of MBytes to reserve so that 32-bit applications can use <code>cvmx_bootmem_alloc()</code> functions. Required for 32-bit applications to send and receive packets directly.
Use wired TLB entries to access the reserved memory region.	<code>CONFIG_CAVIUM_RESERVE32_USE_WIRED_TLB</code>	#undef	When this option is set, the <code>reserve32</code> region is globally mapped to all userspace programs using wired TLB entries. If <code>CONFIG_CAVIUM_RESERVED32</code> is NOT 0, then this value will be automatically defined.

When running “make menuconfig”, the memory configuration options are accessed via the “Machine selection” sub-menu.

The first menuconfig screen looks similar to this:

```

Machine selection --->
  Endianess selection (Big endian)      --->
  CPU selection --->
  Kernel type --->
  Code maturity level options      --->
  General setup --->
  Loadable module support      --->
  Block layer --->
  Bus options (PCI, PCMCIA,      EISA, ISA, TC)      --->
  Executable file formats      --->
  Networking --->
  Device Drivers      --->
  File systems      --->
  Profiling support      --->
  Kernel hacking      --->
  Security options      --->
  Cryptographic options      --->
  Library routines      --->

```

To navigate this screen, use the arrow keys on the keyboard. The bottom of the screen provides some options that can be selected with the TAB key. In the first screen, these options are “Select,

Exit, and Help". To select a highlighted option, press **Enter** when the option "Select" (at the bottom of the screen) is highlighted.

To configure the Cavium Networks-specific options, select "Machine selection". The next screen will look similar to this (options discussed in this chapter are shown in bold red):

```
System type (Support for the Cavium Networks OCTEON reference board) --->
[*] Enable OCTEON specific options
[*] Build the kernel to be used as a 2nd kernel on the same chip
[*] Enable support for Compact flash hooked to the OCTEON Boot Bus
[*] Enable hardware fixups of unaligned loads and stores
[*] Enable fast access to the thread pointer
[*] Support dynamically replacing emulated thread pointer accesses
(2) Number of L1 cache lines reserved for CVMSEG memory
[*] Lock often used kernel code in the L2
[*] Lock the TLB handler in L2
[*] Lock the exception handler in L2
[*] Lock the interrupt handler in L2
[*] Lock the 2nd level interrupt handler in L2
[*] Lock memcpy() in L2
[*] Allow User space to access hardware IO directly
[*] Allow User space to access memory directly
(0) Memory to reserve for user processes shared region (MB)
[*] Use wired TLB entries to access the reserved memory region
(5000) Number of packet buffers (and work queue entries) for the
Ethernet
    driver
<M> POW based internal only Ethernet driver
<*> OCTEON watchdog driver
[ ] Enable enhancements to the IPsec stack to allow protocol offload.
```

When using menuconfig:

- Type "?" for help with a highlighted option.
- Items marked with [*] are "on". To turn them to off, change the star to a space ("*" becomes " ").

To see the configured values, look in the file

`linux/kernel_2.6linux/include/linux/autoconf.h`. This file is created during the build.

```
#define CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE 2
#define CONFIG_CAVIUM_OCTEON_USER_IO 1
#define CONFIG_CAVIUM_OCTEON_USER_MEM 1
#define CONFIG_CAVIUM_RESERVE32 0
#undef CONFIG_CAVIUM_RESERVE32_USE_WIRED_TLB
```

Items marked with [*] are "on". To turn them to off, change the star to a space ("*" becomes " ").

Example: Change the amount of memory to reserve for user processes shared region, highlight the line, and then select it using the choices at the bottom of the screen. The number is in MBytes, and

should be a power of 2 for optimal performance (if the memory is wired, then only the values 512 MBytes, 1024 MBytes, or 1536 MBytes are legal.) In this example, the value is changed from 0 to 512.

Note: if there isn't sufficient memory for the *reserve32*, the kernel fails the bootmem allocate step during boot. It prints a message and the entries in */proc/octeon_info* will be zero (as if *reserve32* was not configured).

After changing any needed items and exiting menuconfig, remake the kernel in the *linux/kernel_2.6/linux* directory:

```
host$ sudo make kernel
```

(This build takes about 20 minutes.)

The file *autoconf.h* now has the new values, and *CONFIG_CAVIUM_RESERVE32_USE_WIRED_TLB* is now defined:

```
host$ grep RESERVE32 autoconf.h
#define CONFIG_CAVIUM_RESERVE32_USE_WIRED_TLB 1
#define CONFIG_CAVIUM_RESERVE32 512
```

Note that this build is not the same as the *make kernel* command typed in the *\$OCTEON_ROOT/linux* directory. The top-level kernel build, which is run after this step, will create a bootable ELF file. This process will be discussed later in this chapter.

Before this change, the file */proc/octeon_info* contains:

```
host# cat /proc/octeon_info
32bit_shared_mem_base: 0x0
32bit_shared_mem_size: 0x0
32bit_shared_mem_wired: 0
```

When the new kernel is booted with the configuration change, the file */proc/octeon_info* contains:

```
host# cat /proc/octeon_info
32bit_shared_mem_base: 0x20000000
32bit_shared_mem_size: 0x20000000
32bit_shared_mem_wired: 1
```

An Example Linux Memory Configuration Error:

If *reserve32* is being used, and the memory is “wired”, but the configured memory is *not* a legal value or there is not enough free memory to fill the request, after the kernel is booted, the file */proc/octeon_info* will not show the configured shared memory. The incorrect values will fail on boot and the configured size is set to 0 on failure:

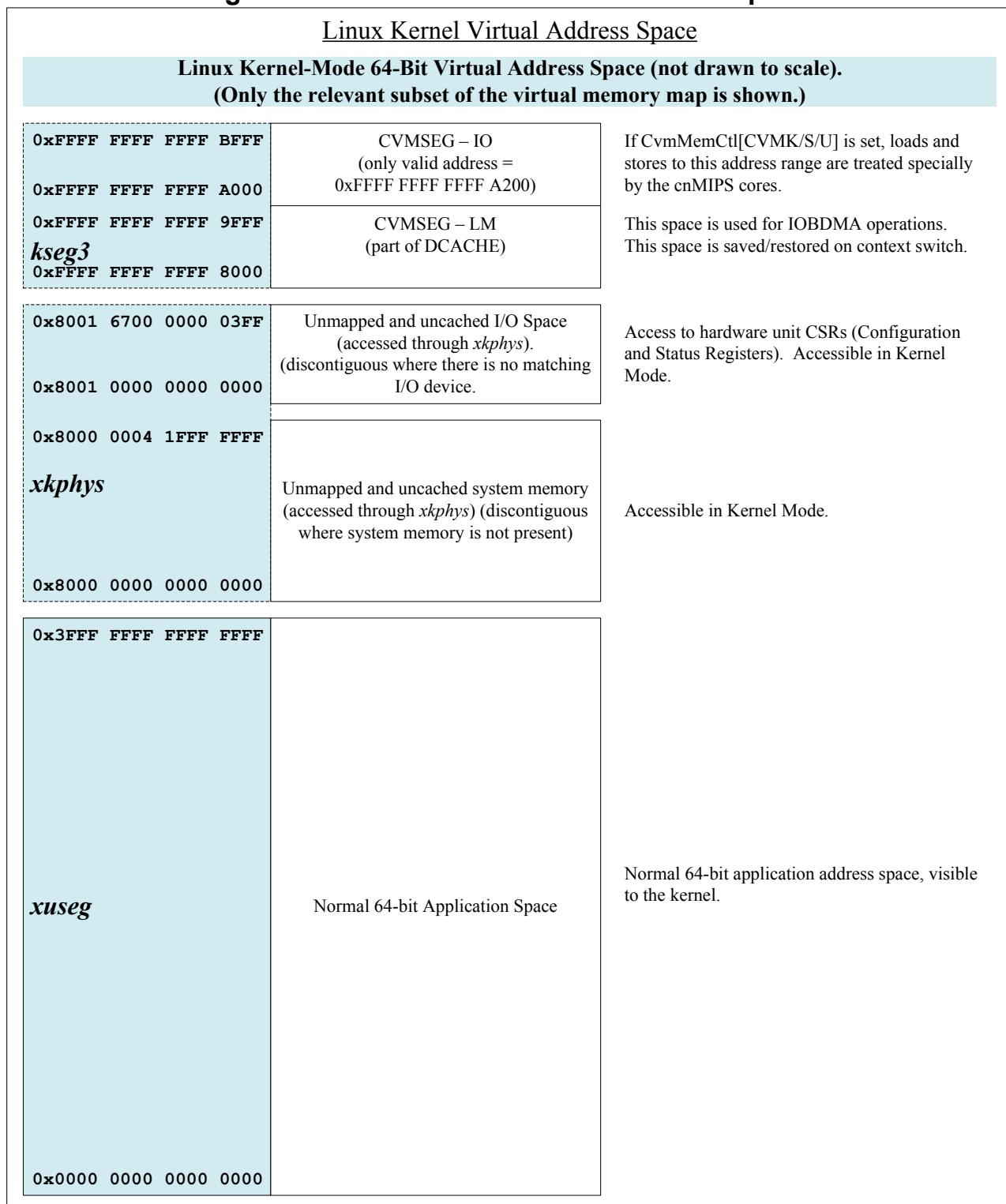
```
# cat /proc/octeon_info
32bit_shared_mem_base: 0x0
32bit_shared_mem_size: 0x0
32bit_shared_mem_wired: 1
```

For a detailed discussion, see the SDK document “*Linux Userspace on the OCTEON*”.

15.4 Linux Kernel-Mode Virtual Address Space on the OCTEON Processor

The following figure shows the Linux Kernel-Mode 64-bit Virtual Address Space for the OCTEON processor. Processes running in kernel mode may access all segments.

The size of *cvmseg* is set during kernel configuration.

Figure 57: Linux Kernel Virtual Address Space

15.5 Linux 64-bit User-Mode Virtual Address Space for OCTEON

The following figure shows the Linux User-Mode 64-bit Virtual Address Space for the OCTEON processor. 64-bit applications may optionally access *xkphys* addresses. The size of *cvmseg* is set during kernel configuration.

Figure 58: Linux 64-Bit SE-UM Virtual Address Space for OCTEON

<u>Linux 64-Bit SE-UM Virtual Address Space for OCTEON</u>	
Linux SE-UM 64-Bit Virtual Address Space (not drawn to scale) (Only the relevant subset of the virtual memory map is shown.)	
0xFFFFF FFFF FFFF BFFF	CVMSEG – IO (only valid address = 0xFFFF FFFF FFFF A200)
0xFFFFF FFFF FFFF A000	
0xFFFFF FFFF FFFF 9FFF	CVMSEG – LM (part of DCACHE)
kseg3	
0xFFFFF FFFF FFFF 8000	
0x8001 6700 0000 03FF	Unmapped and uncached I/O Space (accessed through <i>xkphys</i>). (discontiguous where there is no matching I/O device.)
0x8001 0000 0000 0000	
0x8000 0004 1FFF FFFF	
xkphys	Unmapped and uncached system memory (accessed through <i>xkphys</i>) (discontiguous where system memory is not present)
0x8000 0000 0000 0000	
0x3FFF FFFF FFFF FFFF	
xuseg	Normal 64-bit Application Space
0x0000 0000 0000 0000	Normal 64-bit application address space. Physical memory may be accessed from this space if it is allocated via the bootmem functions, and mapped with <i>mmap()</i> . CVMSEG (in <i>kseg3</i>) can be accessed from user mode.

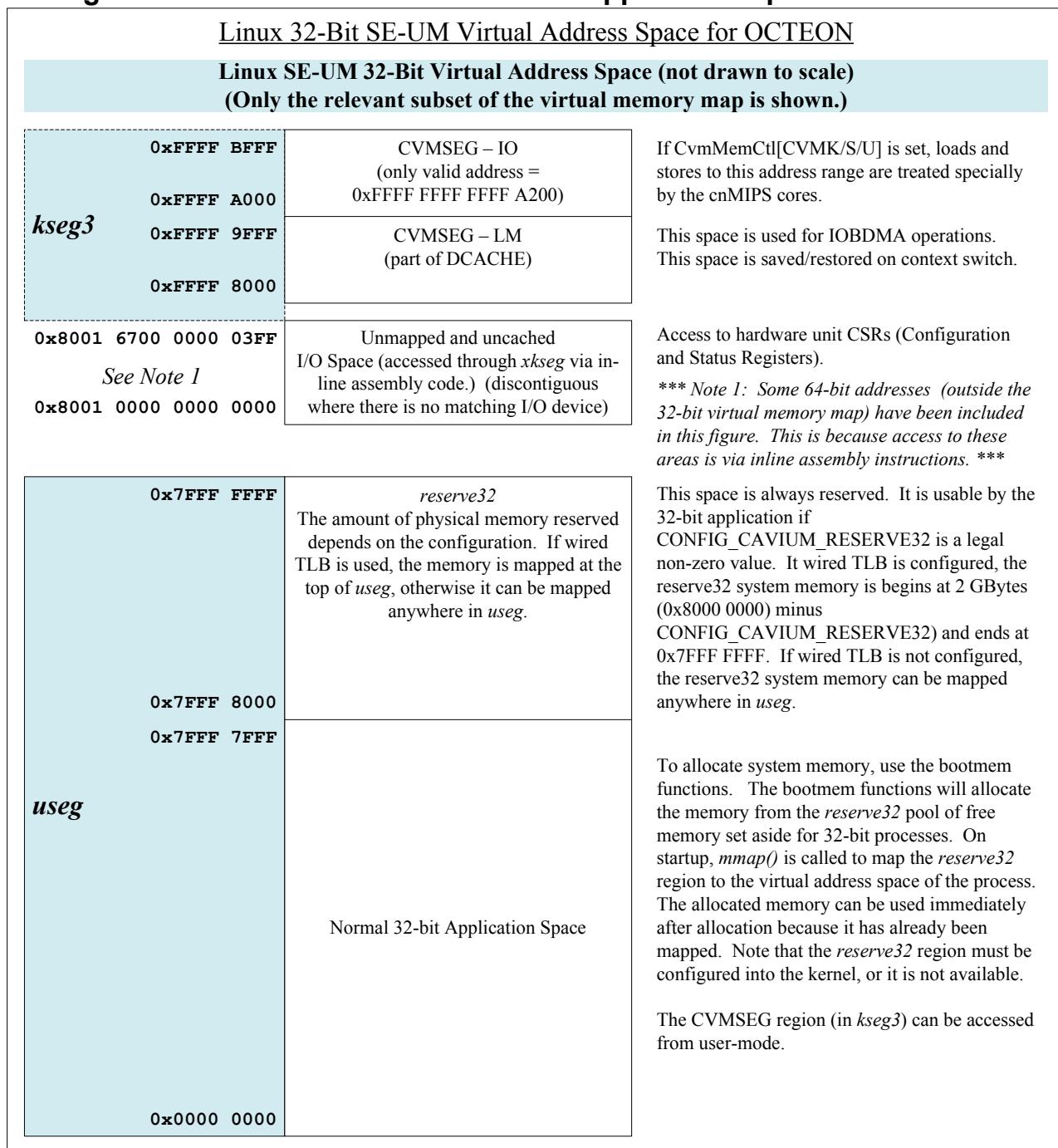
15.6 Linux 32-Bit Virtual Address Space for OCTEON

The following figure shows the Linux 32-bit Virtual Address Space for the OCTEON processor. The kernel always runs in 64-bit mode. The size of *cvmseg* is set during kernel configuration.

Note that the Shared Memory Reserved Region is always reserved. If the kernel was configured with `CONFIG_CAVIUM_RESERVE32` set to a legal value, then the amount of memory specified will be allocated and mapped into the user's application space. If `reserve32` is wired, the memory is mapped to `0x8000 0000` (2 GB) minus the size of the memory region requested. If `reserve32` is not wired, the memory may be mapped anywhere in the processes address space.

Only trusted user applications should be allowed to access system memory without going through the Kernel.

If `CONFIG_CAVIUM_USE_WIRED_TLB` is specified, then this memory is mapped to every process running on the system. This may cause problems if a rogue process writes to this address, corrupting memory. It also consumes TLB entries. If all processes do not need to access shared memory, this option should not be used.

Figure 59: Linux 32-Bit SE-UM Virtual Application Space on OCTEON

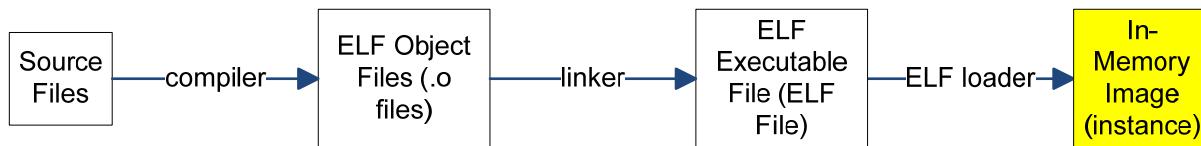
16 Downloading and Booting the ELF File

After the ELF file (either Linux or a SE-S program), it will need to be downloaded to the board and booted. In this section, a quick overview of downloading and booting an ELF file is provided. The *SDK Tutorial* chapter provides more detailed instructions.

Figure 60: Creating an In-Memory Image

Creating an In-Memory Image

When run as a SE-UM application, the in-memory image file is called a process or instance of the program. When run as a SE-S application, the in-memory image file is called an instance of the program.



For SE-S applications, the ELF loader reads the ELF File and creates an in-memory image (instance). The in-memory image is larger than the ELF File: it contains system memory allocated for the stack and heap.

The instance of the program may share .text and .rodata with another instance of the program in the same load set, but will always have some private (not shared) regions which make it unique.

ELF File

- SE-S ELF files may be stored in flash, or downloaded to the board.
- SE-UM files may be stored in the embedded_rootfs of the Linux ELF file, or downloaded after Linux is booted.

ELF Loader

- For SE-S processes, the ELF loader is `bootoctl`.
- For Linux, the ELF loader is `bootoctllinux`.
- For SE-UM instances, the kernel handles ELF file loading automatically. The `oncpu` utility may be used to start SE-UM applications on a subset of cores.

The in-memory image (instance of the program) may be an SE-S, SE-UM, or other Linux process.



Note: SE-S and SE-UM applications must be statically linked.

16.1 Bootloader Memory Model

Two memory areas are reserved by the bootloader: the Reserved Download Block, which is used to download the application, and the Reserved Linux Block. These two areas may be seen with the bootloader command `namedprint`.

Beginning with bootloader 1.7, the bootloader sets the location and size of the Reserved Download Block based on available memory. For information on bootloaders prior to SDK 1.7, see Section 18 – “Bootloader Historical Information”.

The following output is from a 1.7 bootloader. The exact configuration selected by the bootloader will vary depending on how much memory is installed in the target.

```
target# namedprint
List of currently allocated named bootmem blocks:
Name: __tmp_load, address: 0x0000000020000000, size: 0x0000000006000000,
index: 0
Name: __tmp_reserved_linux, address: 0x000000000001000000, size:
0x0000000008000000, index: 1
Name: __tmp_fpa_alloc_0, address: 0x00000000ffde800, size:
0x000000000001f400, index: 2
Name: __tmp_fpa_alloc_1, address: 0x00000000ffbe800, size:
0x0000000000020000, index: 3
Name: __tmp_fpa_alloc_2, address: 0x000000000fdca800, size:
0x000000000001f4000, index: 4
Name: cvmx_cmd_queues, address: 0x0000000008100000, size:
0x0000000000007800, index: 5
```

16.1.1 The Reserved Download Block

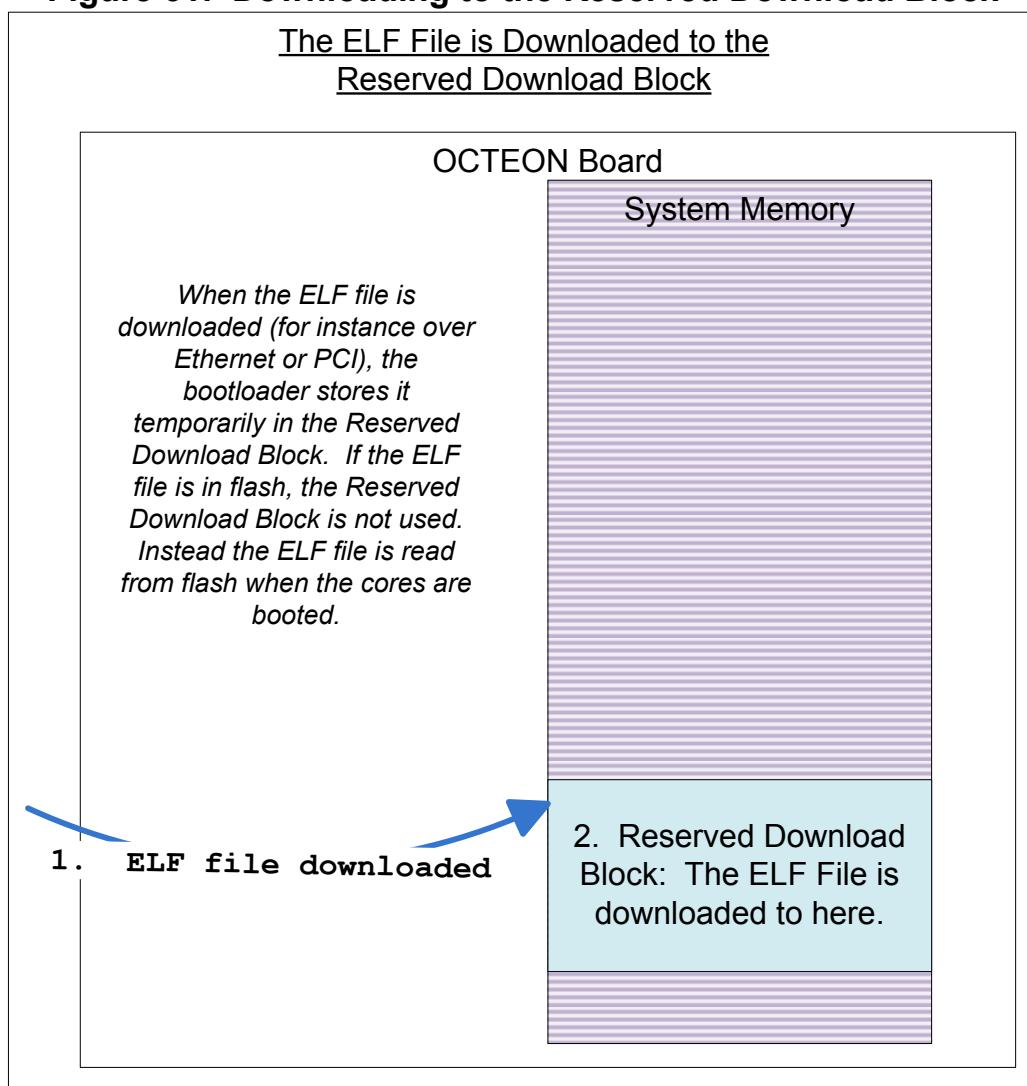
When downloading an ELF file, for instance over PCI, the ELF file is downloaded from the host and stored in memory in a temporary location: the Reserved Download Block.

Note: If the ELF file is in on-board flash, this step is not needed. In that case, the bootloader will read the ELF file from the on-board flash.

16.1.2 ELF File Maximum Download Size

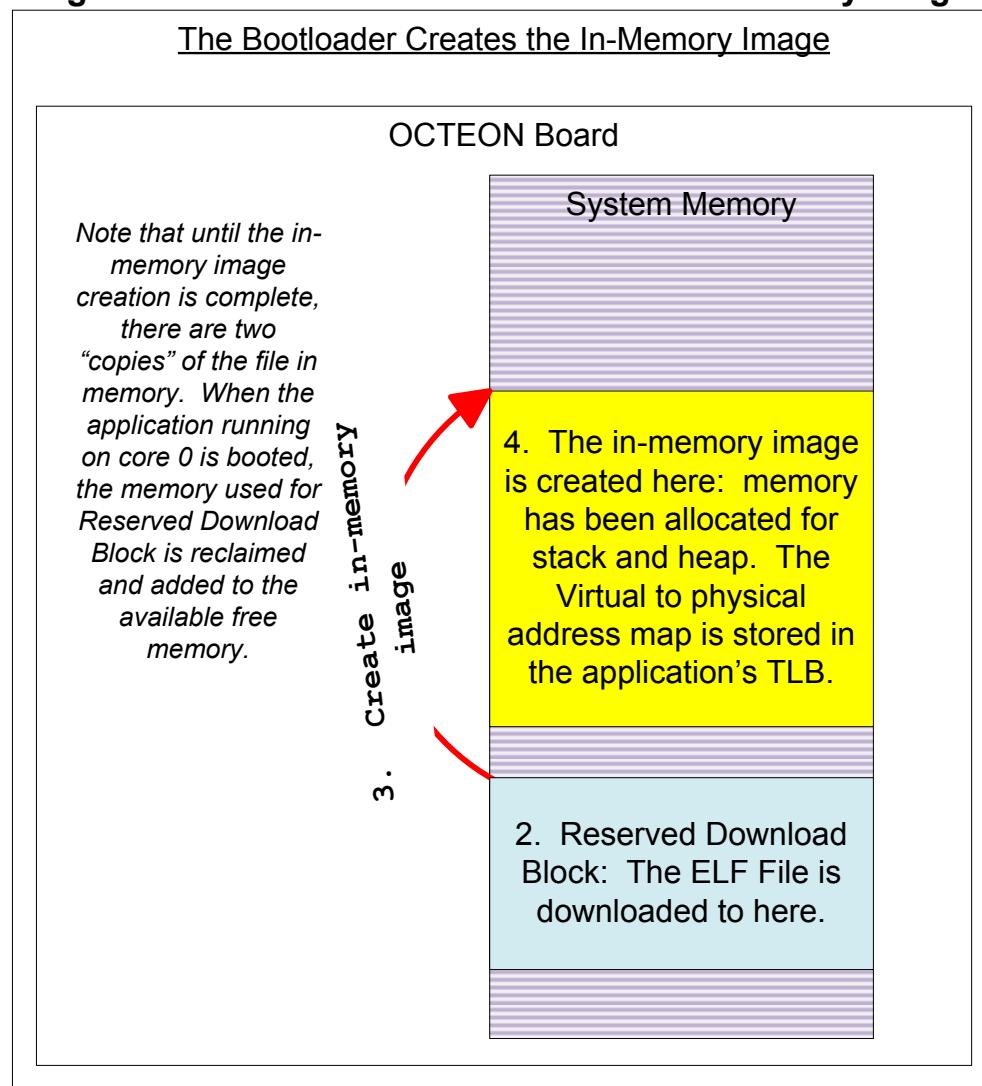
In all ABIs, the created file is in ELF format. The current (SDK 1.8) maximum downloadable ELF file size is 256 MBytes.

Figure 61: Downloading to the Reserved Download Block



After storing the ELF file in the Reserved Download Block, the bootloader reads the ELF file, parses it, allocates system memory for the in-memory image, and creates the in-memory image(s) in different system memory location(s). All of this processing is part of the boot command.

The bootloader creates the needed TLB entries to map the virtual to physical addresses for the in-memory image. Note that the in-memory image is larger than the ELF file: memory is allocated for the stack and heap.

Figure 62: The Bootloader Creates the In-memory Image

After the in-memory image is created, the Reserved Download Block memory may be reused to download another ELF file. For instance, if the system will run both Linux and Simple Executive, then first Simple Executive may be downloaded, and then the Linux (note that whichever is running on core 0 should be loaded last). Both load commands may use the same Reserved Download Block address.

If the ELF file is in flash, then the reserved downloading memory location is not used. The bootloader will read the file from flash instead of the Reserved Download Block.

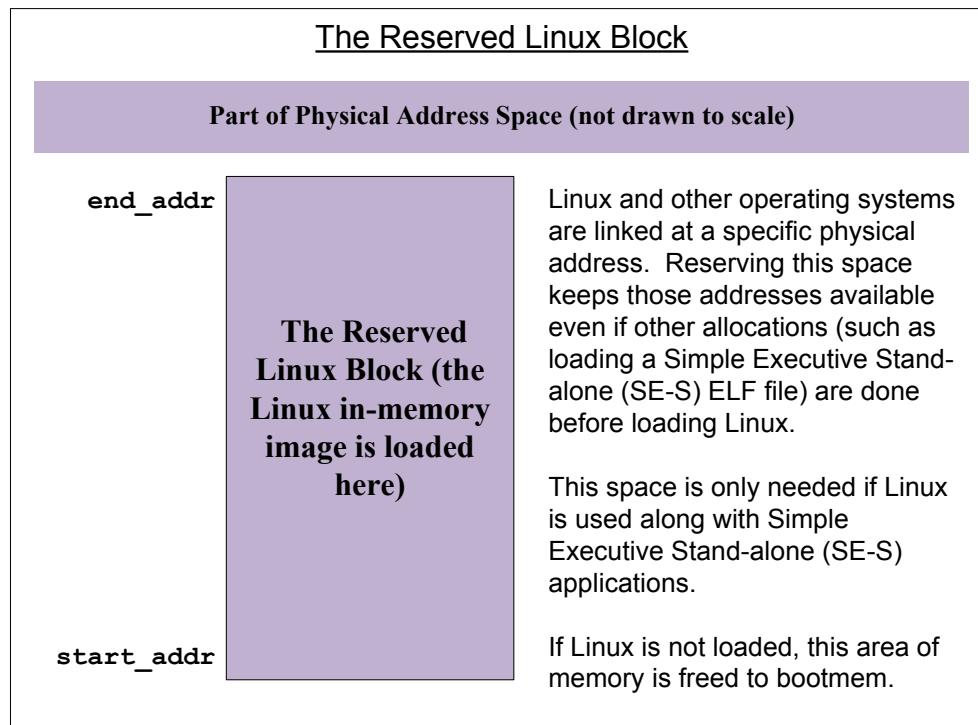
Once the application begins running (when the application running on core 0 is booted), the memory used for Reserved Download Block is reclaimed and added to the available free memory.

16.1.3 The Reserved Linux Block

In addition to the Reserved Download Block, a block of memory is reserved for Linux: the Reserved Linux Block. Unlike Simple Executive applications, which can be loaded anywhere in

memory, Linux is linked to run at specific physical addresses. A block of memory is reserved so that when the Simple Executive application's in-memory image is created, the bootloader will not locate it in the area of memory Linux requires. If Linux is not loaded, this area of memory is reclaimed. If Linux will not be run on the system, then the Linux reserved area size can be set to zero. The only advantage to doing this is to eliminate the memory fragmentation caused when the block is freed.

Figure 63: The Reserved Linux Block



The values of *start_addr* and *end_addr* will depend on the amount of system memory is installed in the target.

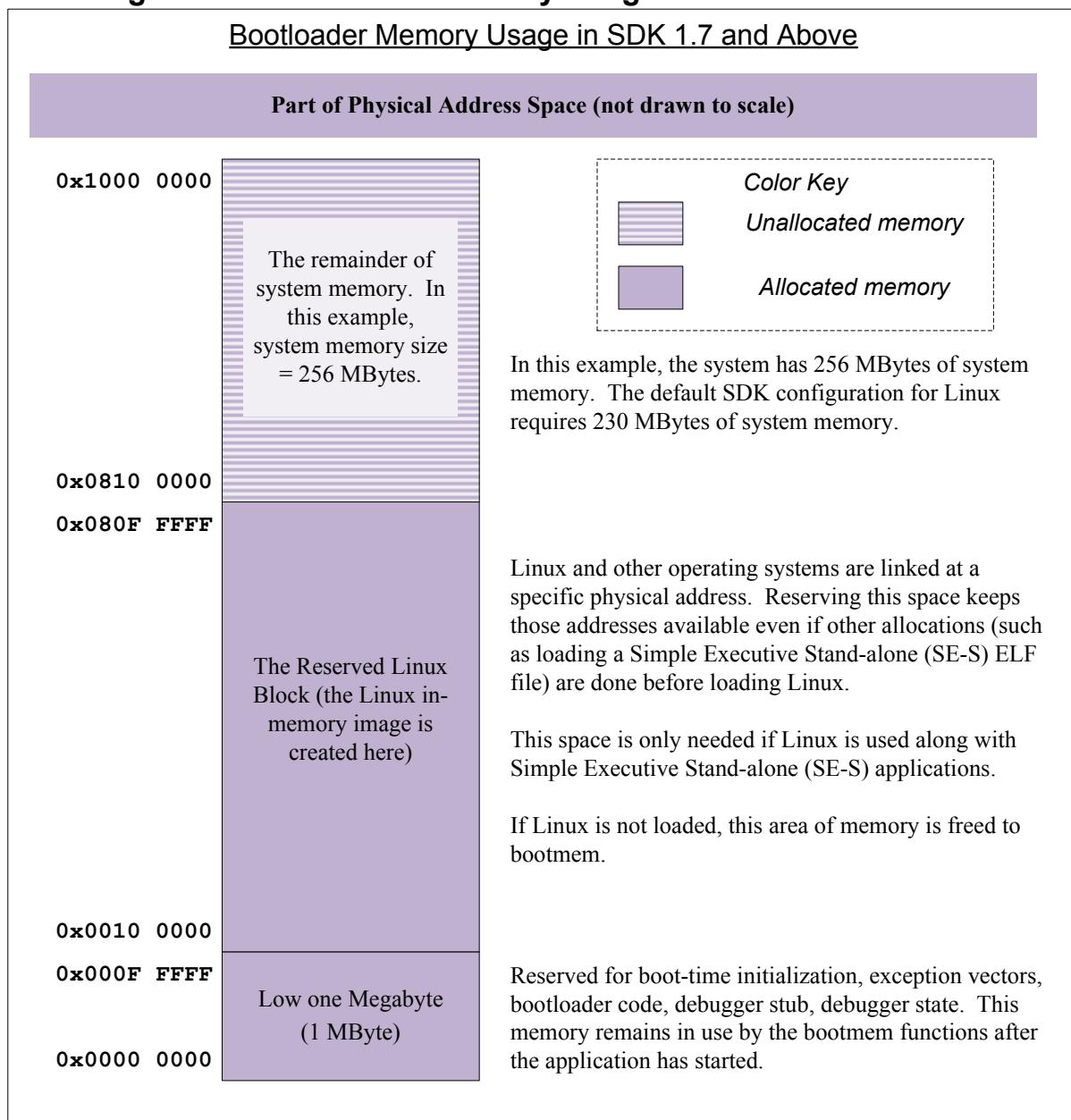
For example, if the target's boot command `namedprint` shows the Reserved Linux Block is:

```
Name: tmp_reserved_linux, address: 0x00000000000100000, size:  
0x0000000008000000, index: 1
```

Then the start address is 0x100000 (a 1 MByte offset), the size is 128 MBytes and:

```
end_addr = 0x80F FFFF  
start_addr = 0x10 0000
```

In the following figure, the Reserved Download Block is not shown: the specific address and size is configuration dependent. The size of the Reserved Linux Block is adjusted based on how much memory is on the board, and the user can also configure it manually using bootloader environment variables.

Figure 64: Bootloader Memory Usage in SDK 1.7 and Above

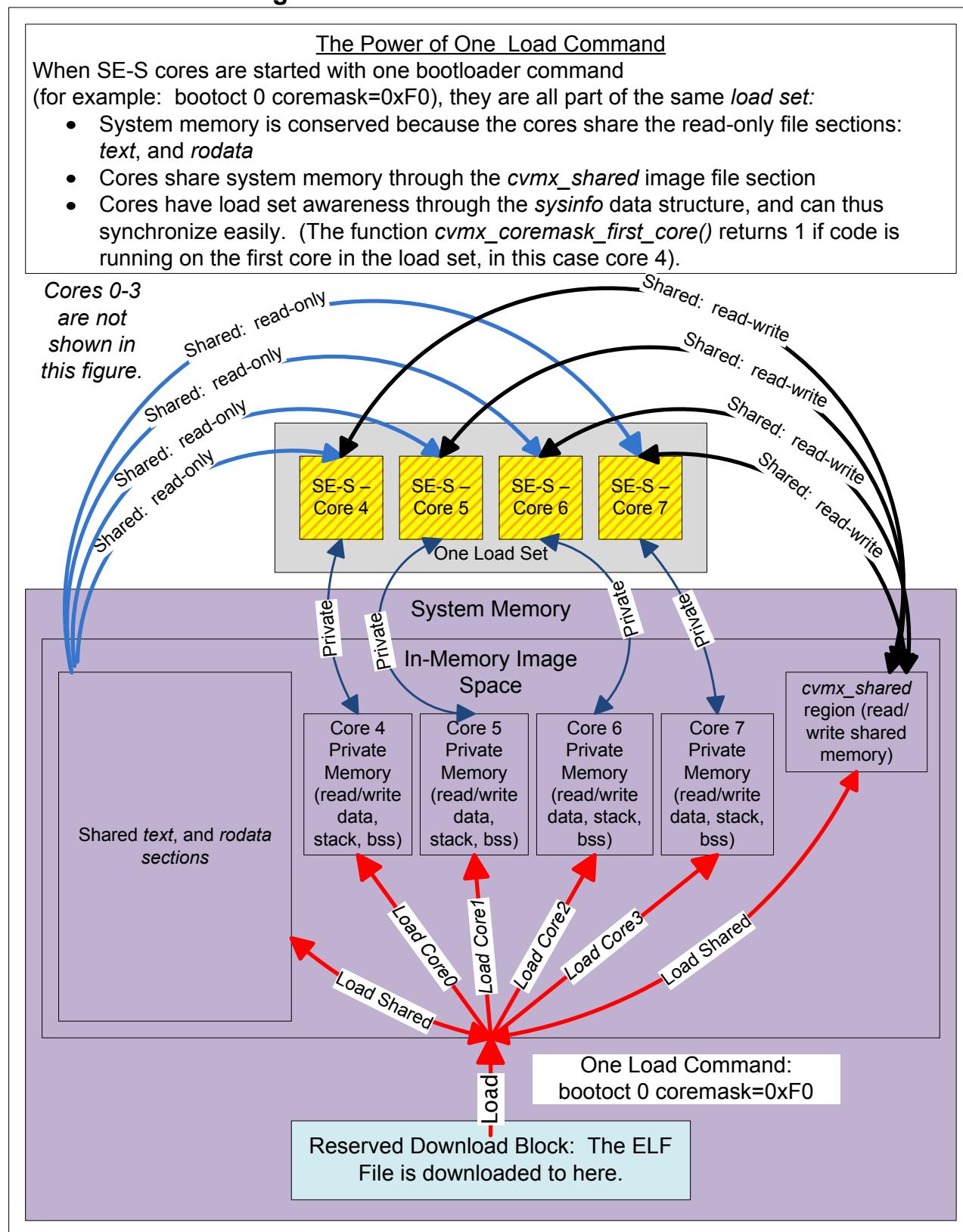
SW OVERVIEW

16.2 Booting the Same SE-S ELF File on Multiple Cores

Typically, one SE-S application is booted on multiple cores. The cores share the read-only parts of the in-memory image: the *text* and *read only data*. They also share *cvmx_shared* variables. These cores are all booted from the same boot command by specifying all the cores in the *coremask* argument to the boot command. Because of this, they are in the same load set.

To download the same application on multiple cores, specify the cores it should run on as an argument to the boot command (*bootoct*, *bootoctlinux*, or *bootelf*): *-coremask=<hex value>*.

Figure 65: The Power of One Load Set



16.3 Downloading and Booting Multiple ELF Files

When downloading multiple ELF files it is important to be aware of what the command is doing. If both downloads go to the same Reserved Download Block, one ELF file will over-write the other unless the first ELF file has been booted before the second is downloaded. An alternative is to use two different Reserved Download Block addresses, but they both have to be within the bootloader's Reserved Download Block. Using two separate Reserved Download Block addresses is not recommended.

16.3.1 Downloading by Re-using One Reserved Download Block

To download both Linux and a Simple Executive application, the following commands might be used (the example is for an 8 core system). Note that the bootloader for SDK 1.7 and higher, the address argument should be "0" to take the default Reserved Download Block address.

Note: If the PCI target commands are in a script, add "sleep 1" between the first boot command and the second download command. The bootloader needs some time to finish booting the first application before the second ELF file is downloaded to the same space.

For example:

On a PCI target:

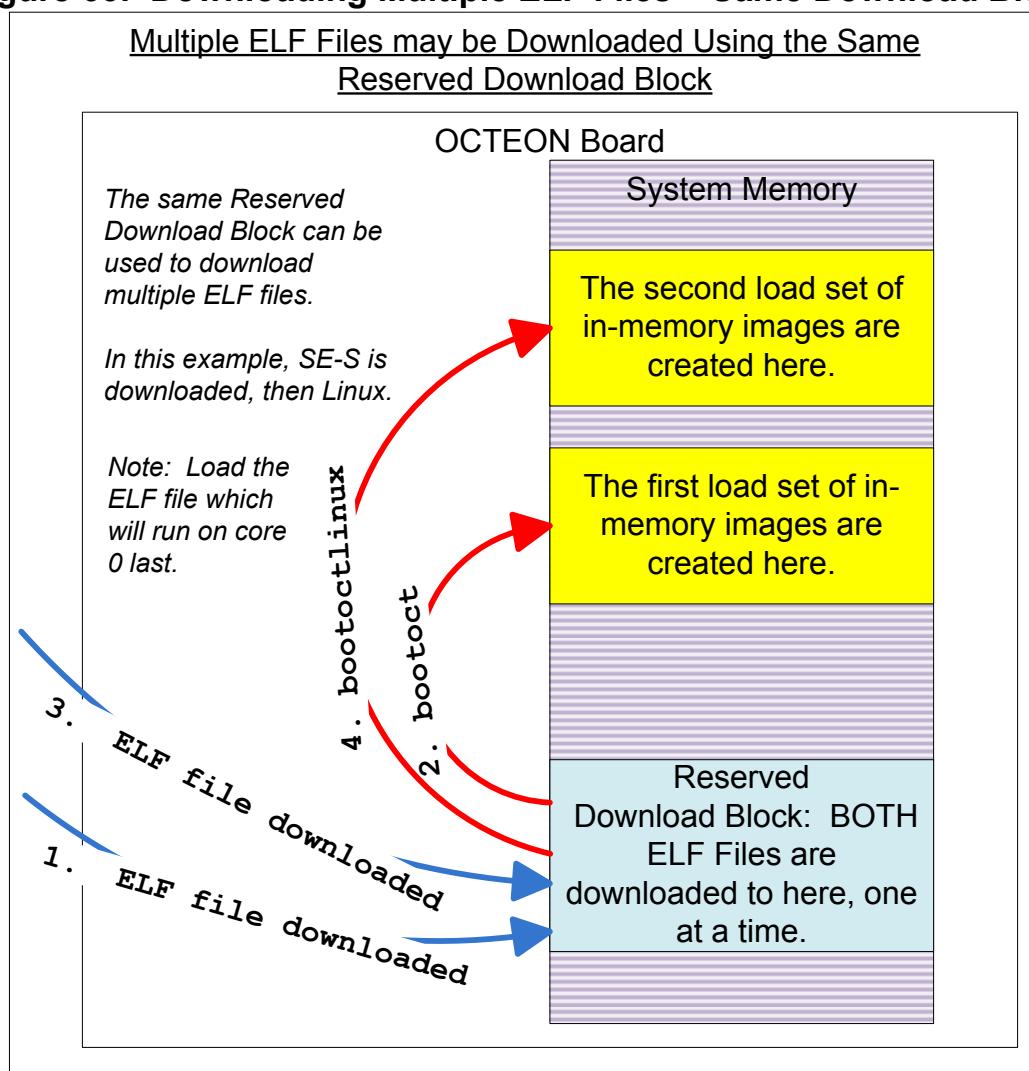
```
host$ oct-pci-load 0 testname/dl/vmlinu.x.64
host$ oct-pci-bootcmd "bootoctlinux 0 coremask=0xF0"
host$ oct-pci-load 0 testname/dl/linux-filter
host$ oct-pci-bootcmd "bootoct 0 coremask=0x0F"
```

On a Standalone Board:

```
target# dhcp
target# tftpboot 0 testname/dl/vmlinu.x.64
target# bootoctlinux 0 coremask=0xF0
target# tftpboot 0 testname/dl/linux-filter
target# bootoct 0 coremask=0x0F
```

NOTE: Notice that the application which will run on core 0 is booted last. Once this core is booted, the other cores are taken out of reset and their applications run.

Figure 66: Downloading Multiple ELF Files – Same Download Block



16.3.2 Downloading Using Two Different Reserved Download Blocks

As an alternative, two separate Reserved Download Block addresses may be used to download both Linux and a Simple Executive application. This is not recommended unless both addresses are within the bootloader's Reserved Download Block. It is far simpler to re-use the download block. The following commands might be used (the example is for an 8 core system). Note that in this case, the "boot" does not have to happen before the second download.

First, get the start address of the Reserved Download Block. In the Minicom session to the bootloader, type

```
target# namedprint
```

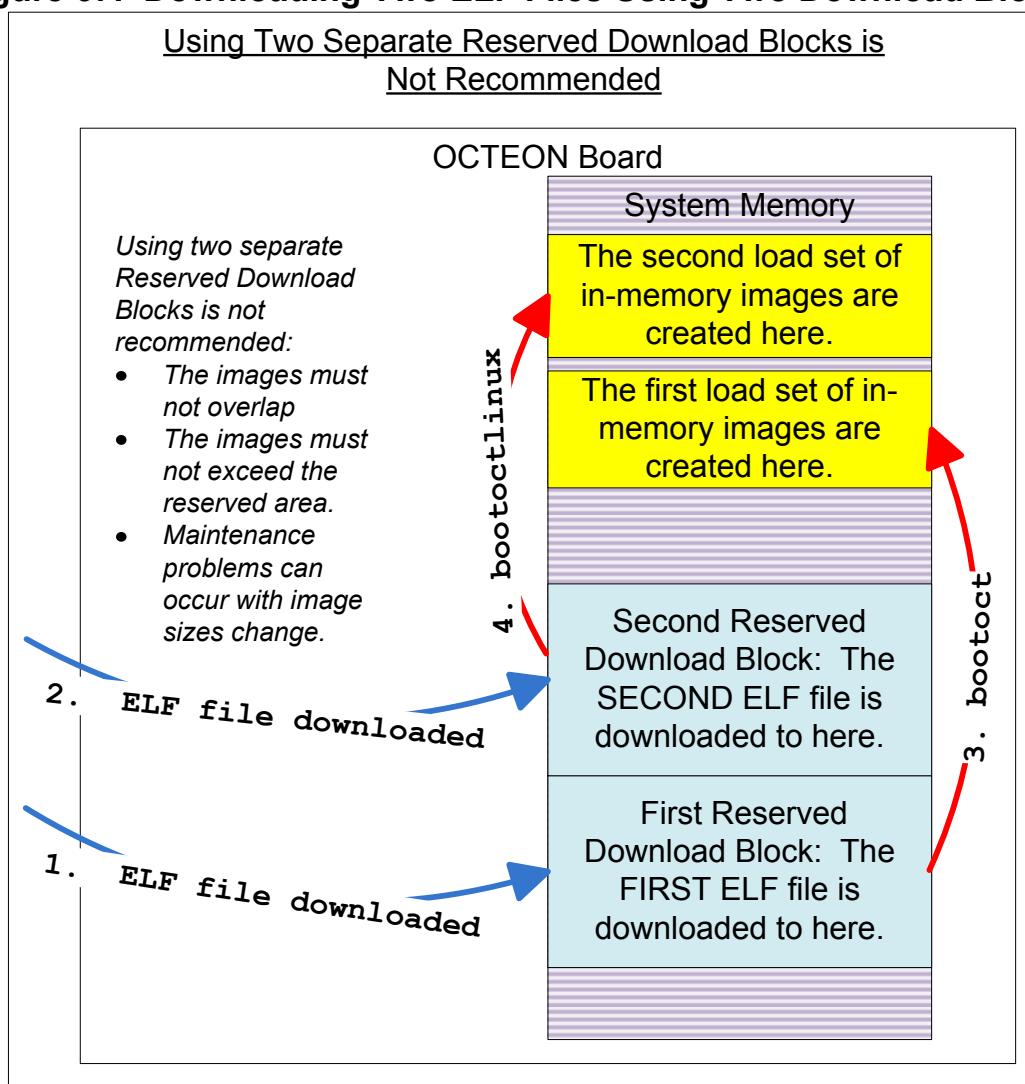
This will show you the Reserved Download Block: “__tmp_load”. In this example, the start address is 0x20000000, and the length is 96 MBytes (0x6000000).

The exact start address and size will depend on your configuration. The bootloader detects the amount of memory you have and sets the values appropriately.

```
target# namedprint
List of currently allocated named bootmem blocks:
Name: __tmp_load, address: 0x0000000020000000, size: 0x000000006000000,
index: 0
Name: __tmp_reserved_linux, address: 0x0000000000100000, size:
0x00000000800000
0, index: 1
```

Since both ELF files must fit in the same Reserved Download Block, care must be taken. Using two separate Reserved Download Blocks is not recommended due to the effort involved in ensuring the two loaded ELF files do not overlap, and do not exceed the area reserved by the bootloader. Maintenance problems can occur as the ELF file sizes change and the size of the load addresses need to also change. Errors can occur from miscalculation or a change in the ELF file size when the space allocated for each download is not changed.

Figure 67: Downloading Two ELF Files Using Two Download Blocks



16.4 Protection from Booting Multiple Applications on the Same Core

The bootloader will issue a warning if the user tries to boot multiple applications on the same core:

```
target# bootoct 0x20000000 coremask=0xD0
ERROR: Can't load code on core twice! (provided coremask overlaps
previously loaded coremask)
```

17 SDK Code Conventions

17.1 Register Definitions and Accessing Registers

17.1.1 Register Definitions

Registers are defined in *cvmx-csr-addresses.h*. Each one is assigned the appropriate virtual address in the include file. The register name in the *Hardware Reference Manual* will become a

name (in all upper case) which is used to access the register. This name is pre-pended with “CVMX” (for example: CVMX_FPA_CTL_STATUS).

Where more than one pool has a register with a similar name, the API convention is to use a macro with an *X* in the name. The argument to the macro is the pool number. The macro will use the pool number to calculate the address of the matching register. This allows the code to easily access the matching register for different pools. The macro will take a pool number as an argument. For example: CVMX_FPA_QUEX_PAGE_INDEX(2) will access the same register as CVMX_FPA_QUE2_PAGE_INDEX.

Some of the FPA registers defined in cvmx-csr-addresses.h are:

```
CVMX_FPA_CTL_STATUS
CVMX_FPA_INT_ENB
CVMX_FPA_INT_SUM
CVMX_FPA_QUE_ACT
CVMX_FPA_QUE_EXP
CVMX_FPA_QUEX_AVAILABLE(offset)
CVMX_FPA_QUEX_PAGE_INDEX(offset)
CVMX_FPA_FPFX_MARKS(offset)
CVMX_FPA_FPFX_SIZE(offset)
```

17.1.2 Register Typedefs

To access a *field* inside the register, instead of the *entire* register, read the register into a data structure, then access the field. The data structures are defined in cvmx-csr-typedefs.h. The register data structures are given the same name as the register, except they are all lower case. The typedefs end in the characters “_t”. The register data structure fields will also have names matching the *Hardware Reference Manual* names (see Table 18: “Accessing Register Fields”).

Here is an example register data structure typedef for the FPA_CTL_STATUS register (known to the Simple Executive as CVMX_FPA_CTL_STATUS, with the register data structure typedef cvmx_fpa_ctl_status_t).

```

/**
 * cvmx_fpa_ctl_status
 *
 * FPA_CTL_STATUS = FPA's Control/Status Register
 *
 * The FPA's interrupt enable register.
 */
typedef union
{
    uint64_t u64;
    struct cvmx_fpa_ctl_status_s
    {
#if __BYTE_ORDER == __BIG_ENDIAN
        uint64_t reserved_18_63      : 46;
        uint64_t reset               : 1;
        uint64_t use_ldt             : 1;
        uint64_t use_stt             : 1;
        uint64_t enb                 : 1;
        uint64_t mem1_err            : 7;
        uint64_t mem0_err            : 7;
#else
        uint64_t mem0_err            : 7;
        uint64_t mem1_err            : 7;
        uint64_t enb                 : 1;
        uint64_t use_stt             : 1;
        uint64_t use_ldt             : 1;
        uint64_t reset               : 1;
        uint64_t reserved_18_63      : 46;
#endif
    } s;
    struct cvmx_fpa_ctl_status_s
    {
        struct cvmx_fpa_ctl_status_s cn3020;
        struct cvmx_fpa_ctl_status_s cn30xx;
        struct cvmx_fpa_ctl_status_s cn31xx;
        struct cvmx_fpa_ctl_status_s cn36xx;
        struct cvmx_fpa_ctl_status_s cn38xx;
        struct cvmx_fpa_ctl_status_s cn38xxp2;
        struct cvmx_fpa_ctl_status_s cn56xx;
        struct cvmx_fpa_ctl_status_s cn58xx;
    } cvmx_fpa_ctl_status_s;
} cvmx_fpa_ctl_status_t;

```

17.1.3 Accessing Registers Using Register Definitions and Data Structures

To read a register, call the function `cvmx_read_csr()`. Give this function the name of the register or register macro (such as `CVMX_FPA_QUEX_PAGE_INDEX(pool)`). Use `cvmx_write_csr()` to write the register.

To access a field inside the register, not the entire register, read the register into a data structure, then access the field. The data structures are defined `cvmx-csr-typedefs.h`.

Example 1: Read from a register, modify a field, and then write to the register:

In this example, we read the CVMX_FPA_CTL_STATUS register, write a “1” to the Enable field (setting the Enable bit), and then write the new value to the register. This enables the FPA.

```
cvmx_fpa_ctl_status_t status;

status.u64 = cvmx_read_csr(CVMX_FPA_CTL_STATUS);
status.s.enb = 1;
cvmx_write_csr(CVMX_FPA_CTL_STATUS, status.u64);
```

Example 2: Read from a register using the register macro, which requires a pool number:

In the case of CVMX_FPA_QUEX_AVAILABLE, the pool number is provided as an argument. This number is then used in calculating the FPA_QUEn_AVAILABLE address for this pool. For example:

```
cvmx_fpa_quex_available_t queue_size_register;

// Ask FPA the number of buffers available
// using the data structure defined in cvmx-csr-typedefs.h

printf("\nReading the FPA register to see how many buffers"
      " are available.\n");
queue_size_register.u64 =

cvmx_read_csr(CVMX_FPA_QUEX_AVAILABLE(CVMX_MY_POOL));

// que_siz, a bit field, is declared uint64_t, but is modified by the
// compiler to be a unsigned int, thus is printed %u instead of %lu
printf("The number of buffers available in MY POOL = %u\n",
      queue_size_register.s.que_siz);
```

Table 18: Accessing Register Fields

Register	Field Name	Access from SDK: typedef (union)For example: <code>cvmx_fpa_available_t avail;</code>	Field (N is one of pool 0-7) For example: <code>avail.s.que_siz</code>
FPA_CTL_STATUS	ENB	<code>cvmx_fpa_ctl_status_t</code>	<code>s.enb</code>
FPA_FPFn_SIZE	FPF_SIZ	<code>cvmx_fpa_fpf0_size_t</code>	<code>s.fpf_siz</code>
FPA_FPFn_MARKS	FPF_RD	<code>cvmx_fpa_fpf_marks_t</code>	<code>s.fpf_rd</code>
FPA_FPFn_MARKS	FPF_WR	<code>cvmx_fpa_fpf_marks_t</code>	<code>s.fpf_wr</code>
FPA_INT_ENB	FED0_SBE	<code>cvmx_fpa_int_enb_t</code>	<code>s.fed0_sbe</code>
FPA_INT_ENB	FED0_DBE	<code>cvmx_fpa_int_enb_t</code>	<code>s.fed0_dbe</code>
FPA_INT_ENB	FED1_SBE	<code>cvmx_fpa_int_enb_t</code>	<code>s.fed1_sbe</code>

Register	Field Name	Access from SDK: <code>typedef (union)For example: cvmx_fpa_available_t avail;</code>	Field (N is one of pool 0-7) For example: <code>avail.s.que_siz</code>
FPA_INT_ENB	FED1_DBE	<code>cvmx_fpa_int_enb_t</code>	<code>s.fed1_dbe</code>
FPA_INT_ENB	Q_n _UND	<code>cvmx_fpa_int_enb_t</code>	<code>s.qN_und</code>
FPA_INT_ENB	Q_n _COFF	<code>cvmx_fpa_int_enb_t</code>	<code>s.qN_coff</code>
FPA_INT_ENB	Q_n _PERR	<code>cvmx_fpa_int_enb_t</code>	<code>s.qN_perr</code>
FPA_INT_SUM	FED0_SBE	<code>cvmx_fpa_int_sum_t</code>	<code>s.fed0_sbe</code>
FPA_INT_SUM	FED0_DBE	<code>cvmx_fpa_int_sum_t</code>	<code>s.fed0_dbe</code>
FPA_INT_SUM	FED1_SBE	<code>cvmx_fpa_int_sum_t</code>	<code>s.fed1_sbe</code>
FPA_INT_SUM	FED1_DBE	<code>cvmx_fpa_int_sum_t</code>	<code>s.fed1_dbe</code>
FPA_INT_SUM	Q_n _UND	<code>cvmx_fpa_int_sum_t</code>	<code>s.qN_und</code>
FPA_INT_SUM	Q_n _COFF	<code>cvmx_fpa_int_sum_t</code>	<code>s.qN_coff</code>
FPA_INT_SUM	Q_n _PERR	<code>cvmx_fpa_int_sum_t</code>	<code>s.qN_perr</code>
FPA_QUE n _PAGES_AVAILABLE	QUE_SIZ	<code>cvmx_fpa_quex_available_t</code>	<code>s.que_siz</code>
FPA_QUE n _PAGE_INDEX	PG_NUM	<code>cvmx_fpa_quex_page_index_t</code>	<code>s.pg_num</code>
FPA_QUE_EXP	EXP_INDX	<code>cvmx_fpa_que_exp_t</code>	<code>s.exp_idx</code>
FPA_QUE_EXP	EXP_QUE	<code>cvmx_fpa_que_exp_t</code>	<code>s.exp_que</code>
FPA_QUE_ACT	ACT_INDX	<code>cvmx_fpa_que_act_t</code>	<code>s.act_idx</code>
FPA_QUE_ACT	ACT_QUE	<code>cvmx_fpa_que_act_t</code>	<code>s.act_que</code>

17.2 The `cvmx_sysinfo_t` `TypeDef`

The `cvmx_sysinfo_t` data structure is private to each process. The data in it was copied from the global info from the bootloader.

This data structure is accessed by the Simple Executive API function `cvmx_sysinfo_get()`.

The `cvmx_sysinfo_get()` function is used in the `passthrough` example to determine whether the application is running on the simulator:

```
if (cvmx_sysinfo_get() ->board_type == CVMX_BOARD_TYPE_SIM)
{
```

The `cvmx_sysinfo_get()` function is used in the `linux-filter` example to determine whether the application is running on the first core in the load set:

```
cvmx_sysinfo_t *sysinfo = cvmx_sysinfo_get();  
  
/* Have one core do the hardware initialization */  
if (cvmx_get_core_num() == sysinfo->init_core)  
{
```

Information in this structure is also used to synchronize cores in the same load set. See Section 5.7 – “Synchronizing Multiple Cores”.

17.3 OCTEON Models

OCTEON models are defined in \$OCTEON_ROOT/executive/octeon-model.h. The choices for the env_setup command line are in \$OCTEON_MODEL/octeon-model.txt.

18 Bootloader Historical Information

The bootloader's memory map changed with SDK 1.7. If the bootloader on the board is older than 1.7, it should be upgraded.

Along with this upgrade, commands used to boot the board have changed. In particular, instead of specifying a specific download address such as 0x21000000, the value “0” is used, allowing the bootloader to select the download address.

This historical information is provided for persons who need to make these modifications to a previously developed product, and need to understand the technical differences between pre 1.7 bootloaders and post 1.7 bootloaders.

The version command is used to find out whether the bootloader was compiled by SDK 1.6 or newer. After SDK 1.6, a change was made to how the bootloader loads ELF files in memory, affecting the commands used to load the ELF files.

To find out if your board was built with SDK 1.7 or higher, use the bootloader command **version**, typed in the Minicom session to the bootloader.

The following bootloader command shows a bootloader built with SDK 1.7.3:

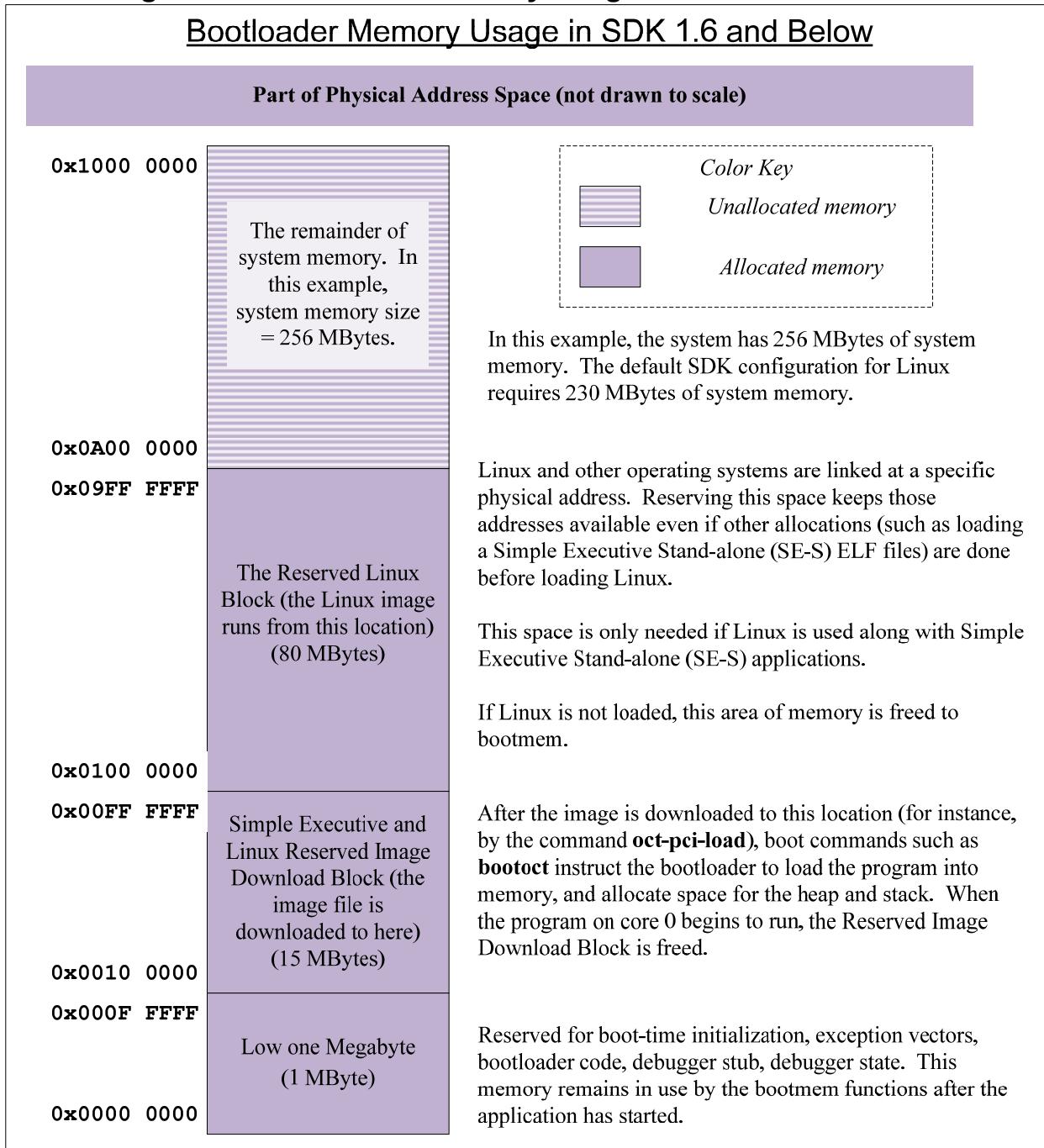
```
target# version  
U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time:  
Jun 13)
```

As shown when discussing bootloader 1.7 and above, there are two memory areas which are reserved by the bootloader: the Reserved Download Block, and the Reserved Linux Block. These two areas may be seen with the bootloader command **namedprint**.

On bootloader 1.6 and lower, the size and location of both the Reserved Download Block (15 MBytes) and the Reserved Linux Block (80 MBytes) are fixed. (On bootloader 1.7 and higher, the bootloader sets the location and size of the Reserved Download Block based on available memory.)

In SDK 1.6 and lower, the Reserved Download Block and Reserved Linux Block were located in the bottom 256 MBytes of memory (DDR0) as shown in the following figure:

Figure 68: Bootloader Memory Usage in SDK 1.6 and Below



ELF files larger than 15 MBytes would not fit into the Reserved Download Block.

To solve this problem a different physical address, usually 0x21000000 was specified on the “**oct-pci-load**” command line. However, this address is out of the memory range for systems with only 256 MBytes (0x10000000).

For example the ELF file for Linux, vmlinux.64, is about 69 MBytes, not 15 MBytes, so it will not fit in the SDK 1.6 default Reserved Download Block.

```
host$ ls -l vmlinux.64
-rwxr-xr-x 1 testname software 71500064 Jul 14 16:00 vmlinux.64
```

Note that 15 MBytes, shown by “ls -l” is 15,728,640. Clearly vmlinux.64 is too big to fit.

To solve these problems, SDK 1.7 and higher allow the bootloader to evaluate the amount of memory available on the system and select suitable addresses.

18.1 Backward Compatibility for Linux ELF Files Built Under SDK 1.6

Linux compiled under SDK 1.6 and lower will load and run on a 1.7 bootloader because the Reserved Linux Block includes the SDK 1.6 Linux link addresses if the board has at least 256 MBytes of memory.

Software Development Kit (SDK)

Tutorial

TABLE OF CONTENTS

TABLE OF CONTENTS	1
LIST OF TABLES.....	5
LIST OF FIGURES	7
1 Introduction.....	8
1.1 Where to Get More Information	11
2 Overview.....	11
3 Hardware and Software Requirements	11
3.1 Development Target Requirements	12
3.2 Development Host Requirements	12
3.3 PCI Host, TFTP Server, and Test System Requirements	13
3.4 DHCP Server	13
3.5 Traffic Generator	14
4 Hands-on: System Administration Tasks	14
4.1 User Account Configuration.....	14
4.2 Multiple Users on the Same Development Host.....	14
5 Hands-on: Connect the Development Target	15
5.1 PCI Development Target	15
5.2 Standalone Development Target.....	17
6 Hands-on: Viewing the Target Board Console Output.....	19
6.1 Starting Minicom	19
6.2 Configuring Minicom	19
6.3 Minicom Basics	21
6.4 Verify Connection to Target Console Works	21
6.5 Minicom Line Wrap and Viewing the Bootloader Help Menu	22
6.6 Scrolling Up and Down	22
6.7 A Typical Minicom Error	23
6.8 Troubleshooting a Missing Bootloader Prompt.....	23
6.9 Determining the Number of Cores on the OCTEON Processor.....	24
7 Hands-on: Gather Key Hardware Information	24
7.1 Determining the OCTEON Model on the Development Target.....	24
7.2 Determining the Number of Cores on the OCTEON Processor.....	24
8 Hands-on: Install the SDK.....	25
8.1 Mounting the CD	25
8.2 Using the RPM Utility to Install the Packages	26

8.3	Making a Copy of the Installed SDK	27
8.4	The OCTEON_ROOT Environment Variable	28
8.5	Setting Environment Variables on the Development Host	28
8.6	Adding <code>env-setup</code> to Your Profile	31
8.7	Viewing the Installed SDK Version	32
9	Hands-on: Tour the Installed SDK	32
9.1	Key Information	32
9.2	Looking at the Installed Directories	32
9.3	Documentation Provided with the SDK	35
9.4	Development Tools	37
9.5	Oprofile Profiling Tools	43
9.6	Native Tools (Run on the Development Target)	43
9.7	Example Applications	47
10	About Building Example Applications	49
10.1	Makefiles	49
10.2	Makefile Targets for Example Code	50
10.3	Building SE-S Examples	51
10.4	Building SE-UM Examples	52
10.5	Saving <code>make</code> Output	52
10.6	Other Makefile Targets	52
10.7	Using the <code>strip</code> Utility	53
11	Hands-on: Build and Run a SE-S Application (<code>hello</code>)	54
11.1	Run <code>hello</code> on a PCI Target Board	55
11.2	Run <code>hello</code> on a Standalone Target Board	59
12	Hands-on: Run <code>hello</code> on Multiple Cores	67
13	About the Bootloader	69
13.1	Booting an OCTEON Board	69
13.2	Review of Bootloader Memory Use	71
13.3	The Failsafe Bootloader	71
13.4	Bootloader Commands	71
13.5	Bootloader Environment Variables	73
13.6	Upgrading the Bootloader	73
14	About Downloading the Application	73
15	About Booting SE-S Applications	76
15.1	The Coremask	76
15.2	The Boot Command	76
16	About Building Linux	77
16.1	The Root Filesystem	78
16.2	Linux Makefiles and Makefile Targets	80
16.3	Configuring Linux	81
16.4	Building Linux	82
16.5	About the <code>make clean</code> Command	83
16.6	The Kernel File Name: <code>vmlinux</code> vs <code>vmlinux.64</code>	83
16.7	The <code>strip</code> Utility and the <code>vmlinux.64</code> ELF File	84
17	Hands-on: Build and Run Linux	84
17.1	Build the Kernel and Embedded Root Filesystem	86

17.2	Download <code>vmlinux.64</code> to the Development Target.....	87
17.3	Boot Linux on the Development Target	87
18	Hands-on: Run a SE-UM Example (<code>named-block</code>)	88
19	About the <code>linux-filter</code> Example	88
20	Hands-on: Run <code>linux-filter</code> as a SE-S Application (Hybrid System)	92
21	Hands-on: Run <code>linux-filter</code> as a Linux SE-UM Application	98
22	Hands-on: Run <code>linux-filter</code> as a SE-UM Application on Multiple Cores.....	103
23	Hands-on: Creating a Custom Application.....	104
23.1	Adding Applications to the Embedded Root Filesystem.....	104
23.2	Example Application Which Breaks Ethernet Driver	107
24	The Hardware Simulator.....	108
24.1	Simulator Documentation.....	108
24.2	Run SE-S Applications on the Simulator	108
24.3	Specifying <code>-noperf</code> and <code>-quiet</code> to Speed Up Processing.....	110
24.4	Running Linux on the Simulator	110
24.5	Simulator: Download and Run Bootloader.....	113
24.6	Using the Simulator to Optimize Performance.....	114
25	Appendix A: Introduction to Available Products	115
26	Appendix B: Linux Basics.....	120
26.1	Linux Commands.....	120
26.2	Shell Scripts	124
26.3	Aliases.....	124
26.4	Linux File Information and the Set User ID Bit	125
26.5	Killing a Process	126
27	Appendix C: About the RPM Utility	126
27.1	Installing from the Support Site Instead of a CD	126
27.2	Useful RPM Commands	127
27.3	RPM Commands Quick Reference Guide	129
28	Appendix D: Other Useful Tools.....	130
28.1	Cscope.....	130
28.2	Ctags	130
28.3	Tera Term, Putty, VNC	131
29	Appendix E: U-Boot Commands Quick Reference Guide	131
30	Appendix F: ELF File Boot Commands Quick Reference	133
31	Appendix G: Null Modem Serial Cable Information	135
32	Appendix H: Query EEPROM to get Board Information	135
32.1	Detecting a Problem with the EEPROM	137
33	Appendix I: Updating U-Boot on a Standalone Board.....	137
33.1	Locating the Correct Bootloader	137
33.2	Save the old Bootloader Environment.....	139
33.3	Updating the Bootloader on the Board	140
34	Appendix J: TFTP Boot Assistance (<code>tftpboot</code>).....	144
34.1	TFTP Server Firewall	144
34.2	Verify that the TFTP Server RPM is Installed on the TFTP Server.....	144
34.3	Verify the TFTP Server is Currently Enabled	144
34.4	About the TFTP Download Directory on the TFTP Server	146

34.5	Verify serverip is set Correctly on the OCTEON Target Board	146
34.6	Test tftpboot: Boot hello on the OCTEON Target Board.....	147
34.7	Further Information	148
35	Appendix K: Downloading Using the Serial Connection.....	148
35.1	Kermit	148
35.2	Copy hello to /tmp	148
35.3	Set up the .kermrc File	149
35.4	Start Minicom	149
36	Appendix L: Simple Executive Configuration	149
37	Appendix M: Changing the ABI Used for Linux	150
38	Appendix N: Contents of the Embedded Root Filesystem.....	150
39	Appendix O: Getting Ready to Use a Flash Card.....	152
39.1	System Administration Steps.....	152
40	Appendix P: Booting an ELF File From a Flash Card	154
40.1	System Administration Steps.....	154
40.2	Copying the ELF File to the Flash Card	154
40.3	Moving the Flash Card to the Target.....	154
40.4	Loading the ELF File From the Flash Card into Memory.....	155
40.5	Booting the ELF File From the Flash Card	155
41	Appendix Q: Using the Debian Root Filesystem	155
41.1	System Administration Steps.....	155
41.2	About the Debian Root Filesystem.....	155
41.3	Install Kernel Plus Debian Onto the Flash card.....	156
41.4	Moving the Flash Card to the Target.....	156
41.5	Load the Kernel from the Flash Card into Memory	156
41.6	Boot the Kernel.....	156
41.7	Upgrading the Kernel on the Flash Card	157
42	Appendix R: About oct-pci-console	157
43	Appendix S: About oct-pci-reset and oct-pci-csr	158
43.1	Reset: oct-pci-reset	158
43.2	Access Control and Status Registers (CSRs): oct-pci-csr	158
44	Appendix T: Multiple Embedded Root Filesystem Builds.....	159
45	Appendix U: How to Find the Process's Core Number	161

LIST OF TABLES

Table 1: Options to the <code>env-setup</code> Script	30
Table 2: Key SDK Files.....	33
Table 3: Key SDK Directories.....	34
Table 4: Documentation Provided via <code>doc/html/index.html</code>	36
Table 5: GNU Tool Chain	40
Table 6: Host Tools	41
Table 7: Host Tools, continued.....	42
Table 8: Profiling Tools (Oprofile).....	43
Table 9: Hardware Diagnostic Tools.....	43
Table 10: Special Cavium Networks Native Tools, Part 1	45
Table 11: Special Cavium Networks Native Tools, Part 2	46
Table 12: Examples Provided with SDK 1.8.0.....	48
Table 13: Different Makefile Targets, Different Target Names.....	51
Table 14: Key <code>oct-pci-*</code> Commands	55
Table 15: Run <code>hello</code> on a PCI Development Target, Part 1	56
Table 16: Run <code>hello</code> on a PCI Development Target, Part 2	57
Table 17: Run <code>hello</code> on a Standalone Development Target, Part 1	60
Table 18: Run <code>hello</code> on a Standalone Development Target, Part 2	61
Table 19: Run <code>hello</code> on a Standalone Development Target, Part 3	62
Table 20: Key Bootloader Commands	72
Table 21: Commands to Download ELF File for Different Configurations	75
Table 22: ELF File Download Command Details	76
Table 23: Linux Top-Level Makefile Targets	81
Table 24: Build and Run Linux, Part 1	85
Table 25: Build and Run Linux, Part 2.....	86
Table 26: Build and Run Linux and <code>linux-filter</code> (SE-S), Part 1	93
Table 27: Build and Run Linux and <code>linux-filter</code> (SE-S), Part 2	94
Table 28: Build and Run Linux and <code>linux-filter</code> (SE-S), Part 3	95
Table 29: Build and Run Linux and <code>linux-filter</code> (SE-S), Part 4	96
Table 30: Build and Run Linux and <code>linux-filter</code> (SE-S), Part 5.....	97
Table 31: Build and Run Linux and <code>linux-filter</code> (SE-UM), Part 1.....	99
Table 32: Build and Run Linux and <code>linux-filter</code> (SE-UM), Part 2.....	100
Table 33: Build and Run Linux and <code>linux-filter</code> (SE-UM), Part 3.....	101
Table 34: Build and Run Linux and <code>linux-filter</code> (SE-UM), Part 4.....	102
Table 35: Simulator Documentation.....	108
Table 36: Run Linux on the Hardware Simulator, Part 1	112
Table 37: Run Linux on the Hardware Simulator, Part 2	113
Table 38: Packages Provided on the Installation CD, Part 1	116
Table 39: Packages Provided on the Installation CD, Part 2	117
Table 40: Example RPMs on the SDK Installation CD	118
Table 41: Package Dependencies	118
Table 42: Optional Toolkits.....	119
Table 43: Application Development Kits (ADKs)	119

Table 44: Linux Commands Quick Reference, Part 1	121
Table 45: Linux Commands Quick Reference, Part 2	122
Table 46: Linux Commands Quick Reference, Part 3	123
Table 47: RPM Commands Quick Reference	129
Table 48: U-Boot Commands Quick Reference, Part 1	132
Table 49: U-Boot Commands Quick Reference, Part 2	133
Table 50: ELF File Download and Boot Commands Quick Reference, Part 1	134
Table 51: ELF File Download and Boot Commands Quick Reference, Part 2	135
Table 52: SDK 1.7.3 Boards and Bootloader File Names, Part 1	138
Table 53: SDK 1.7.3 Boards and Bootloader File Names, Part 2	139
Table 54: Example Bootloader Environment File Changes due to Upgrade.....	143

LIST OF FIGURES

Figure 1: Different Runtime Modes Covered in This Tutorial	10
Figure 2: Development Host and Target	12
Figure 3: Hardware Configuration: PCI Development Target	17
Figure 4: Hardware Configuration: Standalone Development Target	18
Figure 5: Minicom Configuration Menu	20
Figure 6: Example Minicom Serial Port Configuration	20
Figure 7: Saving the Minicom Configuration to a File	21
Figure 8: Run <code>hello</code> on 1 out of 8 Cores	54
Figure 9: Run <code>hello</code> on 4 out of 8 Cores	67
Figure 10: Core 0 Runs Bootloader While the Other Cores Stay in Reset	69
Figure 11: Creating an In-Memory Image	74
Figure 12: Root Filesystem Locations	78
Figure 13: Example: <code>linux-filter</code> Forwards an IP Non-broadcast Packet	90
Figure 14: Example: <code>linux-filter</code> Does Not Forward an IP Broadcast Packet	91
Figure 15: Example of <code>linux-filter</code> (SE-S) and Linux Run on 2 out of 8 Cores	92
Figure 16: Example of <code>linux-filter</code> run as a SE-UM Application on Linux	98
Figure 17: Run <code>linux-filter</code> as a SE-UM Load Set	103
Figure 18: Running Linux on Simulated Hardware	110

1 Introduction

This chapter introduces the Software Development Kit (SDK) from a hands-on perspective, provides a tour of the installed SDK, and also contains useful information for users new to embedded software development or new to Linux.

This chapter assumes the reader will use either an evaluation or a reference board to follow the hands-on steps. To maximize understanding of this chapter, the reader will need:

- An i386 or x86_64 development host, running Linux
- An OCTEONprocessor reference or evaluation board
- The Cavium Networks OCTEON SDK, either on CD or downloaded from the support site at <http://www.caviumnetworks.com/>
- *root* privilege on the development host

Before reading this chapter, please read the *Packet Flow* and the *Software Overview* chapters. The *Packet Flow* chapter will provide background information on the basic hardware units and how they interact. This information is necessary to understand the Simple Executive API and the examples. The *Software Overview* chapter introduces the cnMIPS cores, runtime environment choices, high-level system design, memory map basics, Simple Executive API (Application Programming Interface), and the ABI (Application Binary Interface) choices available.

In addition to the chapters in this book, the *Quick Start Guide* provided with the evaluation or reference board is essential to correctly configuring and powering on the evaluation or reference board.

This chapter is designed to augment the SDK documentation by providing a high-level view of the SDK and step-by-step instructions from installing the SDK to running example code on an evaluation board.

The examples covered include running a SE-S application, booting Linux, running a SE-UM application, and running a hybrid system with both SE-S and SE-UM applications.

A large reference section begins at Section 25 – “Appendix A: Introduction to Available Products”. This section will assist readers who are new to the OCTEON processor and to Linux.

Because this chapter is a tutorial, hands-on sections are labeled “Hands-on”. Discussion sections are labeled “About”. The chapter should be read in order, and the hands-on steps taken in order (except for the Appendices). For example, before running Linux or a SE-UM application, be sure to run the SE-S application `hello`. Later steps in this document depend on successful execution of the earlier steps.

The details often follow the hands-on directions. This is done to keep the reader from being distracted by technical details. The goal of the tutorial is to get up and running as rapidly as possible.

If you are only interested in running Linux or SE-UM applications you may be tempted to skip the step of booting the example application `hello`. Running `hello` is actually an important part of verifying that:

- the SDK is properly installed
- the environment variables are properly configured
- the development target board has booted properly
- the development target board is connected properly to the development host
- the development target is working

After all of these things have been verified by running `hello`, it is a simple step to run Linux. Once Linux is running, it is a simple step to run a SE-UM application. From there, the step to running a hybrid system is also simple. Most of the new material to learn is presented in the simplest of all examples: `hello`.

Readers new to the OCTEON processor will benefit from the details in this chapter, especially the extra information in the Reference Section.

Note: In the text below,

- `host$` means execute the command as a normal user on the development host. Commands which require `root` privilege will be preceded by `sudo`, which is used to obtain the `root` privilege. When typing in the command as shown in the tutorial, omit this text.
- `target#` means execute the command as `root` on the development target. When typing in the command as shown in the tutorial, omit this text.

The hands-on sections in this tutorial are:

- Section 4 – “Hands-on: System Administration Tasks”
- Section 5 – “Hands-on: Connect the Development Target”
- Section 6 – “Hands-on: Viewing the Target Board Console Output”
- Section 7 – “Hands-on: Gather Key Hardware Information”
- Section 8 – “Hands-on: Install the SDK”
- Section 9 – “Hands-on: Tour the Installed SDK”
- Section 11 – “Hands-on: Build and Run a SE-S Application (`hello`)”
- Section 12 – “Hands-on: Run `hello` on Multiple Cores”
- Section 17 – “Hands-on: Build and Run Linux”
- Section 18 – “Hands-on: Run a SE-UM Example (`named-block`)”
- Section 20 – “Hands-on: Run `linux-filter` as a SE-S Application (Hybrid System)”
- Section 21 – “Hands-on: Run `linux-filter` as a Linux SE-UM Application”
- Section 22 – “Hands-on: Run `linux-filter` as a SE-UM Application on Multiple Cores”
- Section 23 – “Hands-on: Creating a Custom Application”

Figure 1: Different Runtime Modes Covered in This Tutorial

Different Runtime Modes Covered in This Tutorial

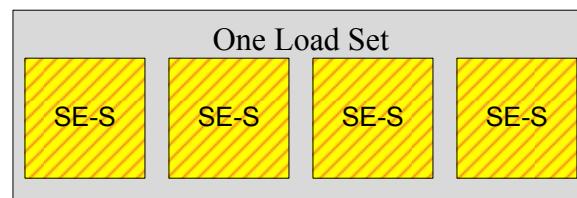
SE-S/Single Core

Run **hello** on one core as a Simple Executive Standalone (SE-S) application. This is helpful for debugging the hardware setup.



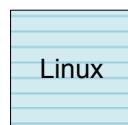
SE-S/Multicore

Run **hello** on four cores as a Simple Executive Standalone (SE-S) application in one load set.



Linux/Single Core

Run Linux on one core.



Linux/SE-UM/Single Core

Run **named-block** as a Simple Executive User-Mode (SE-UM) application on Linux.



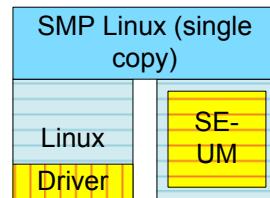
Hybrid: Linux/Ethernet Driver/SE-S

Run Linux and the Cavium Networks Ethernet Driver on one core and run **linux-filter** as a SE-S application on another core.



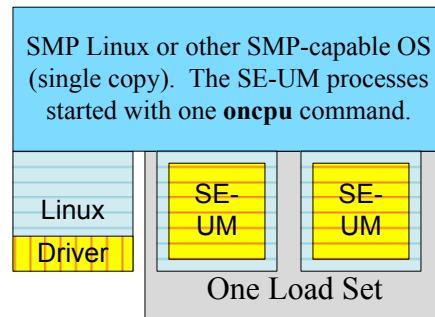
Hybrid: SMP Linux/Single Core SE-UM

Run Linux and the Cavium Networks Ethernet Driver on two cores and run **linux-filter** as a SE-UM application on one of the two cores.



SMP Linux/Multicore SE-UM

Run Linux and the Cavium Networks Ethernet Driver on three cores and run **linux-filter** as a SE-UM application on two of the three cores.



1.1 Where to Get More Information

Extensive documentation is supplied with the SDK. This documentation is outlined in Section 9.3 “Documentation Provided with the SDK”.

In addition to the documentation supplied with the SDK, the Hardware Reference Manual (HRM) for the specific OCTEON model used in your application will provide more information. The HRM is available from the support site at <http://www.caviumnetworks.com/>.

The *Quick Start Guide* (OCTEON-SDK-QSG.pdf) is needed to install the evaluation board. The *Quick Start Guide* included in hard-copy with the evaluation board, and is also available as a PDF file on the SDK CD, or in the `docs` directory of the installed base SDK.

The source for the tool chain and additional information is available from <http://www.cnusers.org/>.

2 Overview

The CD included with the evaluation or reference board contains the SDK. The SDK will be installed on a development host: a host PC running Linux.

The SDK is defined to be two packages: the base SDK, and OCTEON Linux. All other RPM packages are not included in the definition of the SDK.

The base SDK package includes:

- The complete GNU-based tool chain including the compiler, linker, and generic libraries, optimized to take advantage of the cnMIPS cores contained within the OCTEON processor.
- The OCTEON software simulator, which includes performance measuring tools.
- Cavium Networks Simple Executive: software that enables quick application development. This software provides a C or C++ API to the underlying hardware.
- Several example applications.

The OCTEON Linux package contains OCTEON Linux (ported to the OCTEON processor from the Linux source at <http://www.linux-mips.org/>).

Simple Executive applications may run on OCTEON Linux as user-mode applications (SE-UM), or may run as Simple Executive standalone application (SE-S). Because SE-S applications have no operating system overhead, they are typically used in the highest-performance designs.

This tutorial provides step-by-step instructions for running both SE-S and SE-UM applications. SE-S applications require only the base SDK. SE-UM applications require the OCTEON Linux package.

3 Hardware and Software Requirements

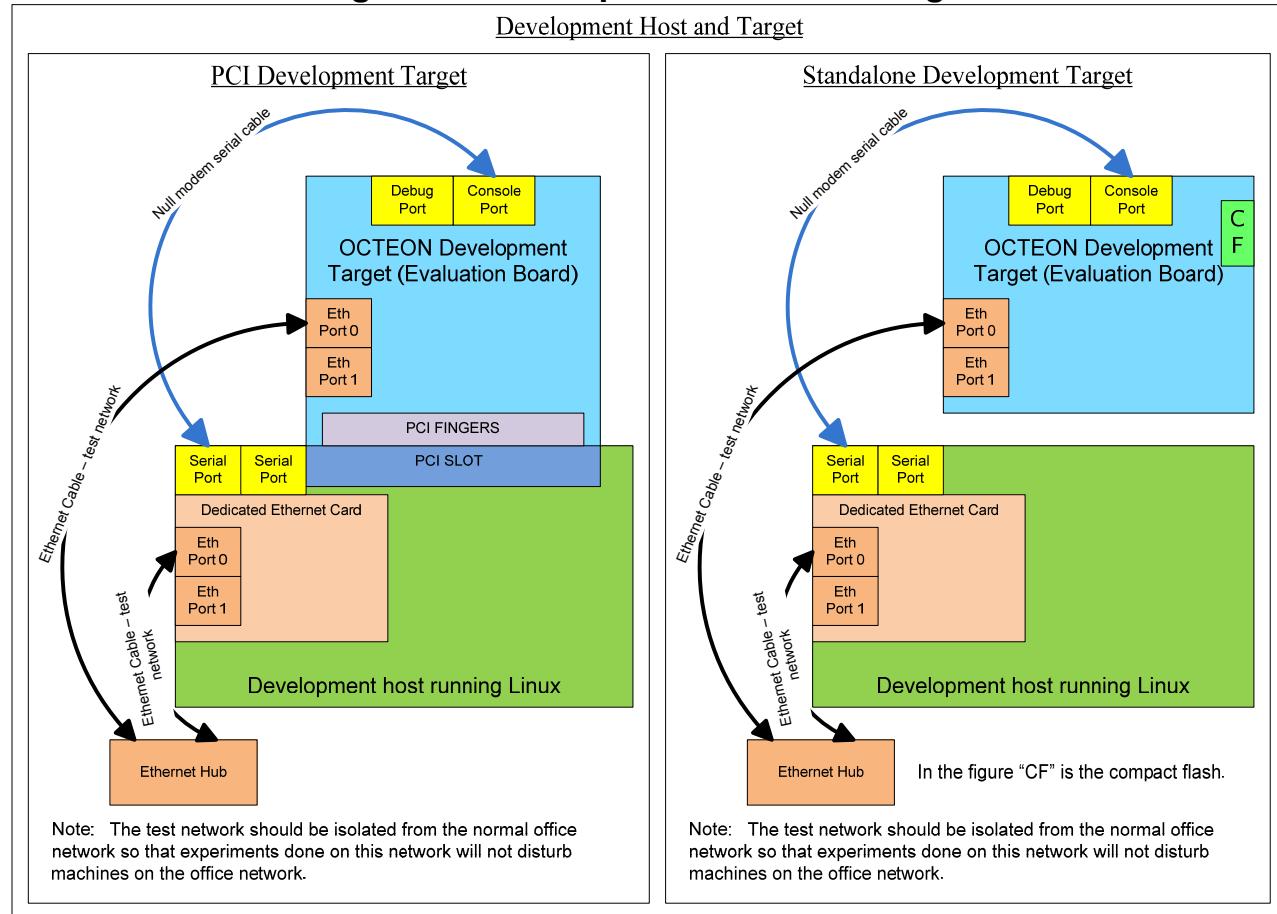
To avoid confusion with the term *PCI host*, the term *development host* will be used to describe the i386 or x86_64 machine which is used as a cross-development platform. The term *development target* refers to the OCTEON evaluation board connected to the development host.

In this chapter, the development host is configured to handle multiple roles. It is the:

- Development host used in cross-compilation
- PCI host when the development target is a PCI board
- TFTP server used to tftpboot the ELF file
- Test System which has a dedicated Ethernet connection to the development target, and/or can be used to run the hardware simulator

The application (or example program) is built on the development host. Once the application is built, it will be downloaded to either a PCI development target (inserted inside the development host), or to a standalone development target board connected to the development host. The two different configurations are shown in the following figure.

Figure 2: Development Host and Target



3.1 Development Target Requirements

The development target (OCTEON evaluation or reference board) may be either a PCI or standalone board.

3.2 Development Host Requirements

The application will be built on the development host using a cross-compiler.

Development host platform:

- i386 or
- x86_64

Development host operating system:

- Red Hat Linux, such as Fedora. Cavium Networks has tested the SDK on Fedora Core 4 and Red Hat Enterprise Linux (RHEL5-X, scientific build).
- Other configurations may work, but they have not been tested.

3.3 PCI Host, TFTP Server, and Test System Requirements

In this chapter, it is assumed that the development host, the PCI host, the TFTP Server, and the Test System are one PC used for multiple purposes. Since they are not required to be the same system, the separate requirements are listed below:

- PCI Host: If the OCTEON development target is a PCI board, then the most convenient configuration is to have the development host be the PCI host. The PCI host must have a 3.3 volt 64-bit PCI slot. Note: PCI Express (PCIe) does not have multiple voltage levels available, so all PCIe boards will work in all PCIe systems. In this chapter, *PCI host* means either a PCI or PCIe host, and *PCI target* means either a PCI or PCIe target.
- TFTP Server: If the OCTEON development target is a standalone board, then the most convenient configuration is to have the development host be the TFTP server. Note if TFTP is not available, a flash card may be used to download the application to the development target.
- Test System:
 - Test System Memory: If the hardware simulator will be run, and multiple cores will be simulated, the test system will need a minimum of 512 MBytes; 1 GByte or more recommended.
 - Test System Ethernet: The test machine either runs the hardware simulator or is connected to the target board and used to run tests on it. For instance, Ethernet ICMP echo requests (ping) may be sent from the test system to the development target.

Important: Configure the test system to have an isolated Ethernet connection to the Target board (separate from the office Ethernet). Tests done between the development host and development target over Ethernet can interfere with normal uses for Ethernet, for instance by flooding the Ethernet with packets.

3.4 DHCP Server

A Dynamic Host Configuration Protocol (DHCP) server can be used to assign the IP address to the development target. Otherwise, simply use a static IP. The directions in this chapter include both choices.

3.5 Traffic Generator

A traffic generator can be helpful in running examples and in testing your application. If a traffic generator is not available, run the example program example `traffic-gen` on the hardware simulator to generate traffic. This solution is adequate for simple tests.

Note that a traffic generator is not needed to run the examples in this chapter.

4 Hands-on: System Administration Tasks

In this chapter, the assumption is made that each user has their own development host. If the development host is shared among several users, see Section 4.2 – “Multiple Users on the Same Development Host”.

Before continuing, the following system administration tasks should be done on the development host:

- Create a personal account on the development host (shown as *testname* in this document). The shell used in these directions is bash. Login as this user and do all work as this user, except when directed to use the `sudo` command.
- Obtain *root* (`sudo`) permission on the development host. This is needed to install the SDK. To test if this permission has been correctly set up, type: `sudo ls`. If this command works, `sudo` has been correctly set up. Sudo permission is also needed to build the Linux kernel, and to use the PCI tools. Note that when using `sudo` you do not enter the *root* password, you enter your own password.

4.1 User Account Configuration

When configuring the user’s account, two items are important:

1. The user’s account should *not* be on a NFS-mounted drive:

Note: If the user’s home directory is in a NFS-mounted drive, and the user’s SDK workspace is in their home directory, then some steps in this tutorial will fail. This failure is because some steps require root privilege. NFS will, by default, prevent root on client machines from having access to the files on the NFS-mounted drive. Usually root is mapped to nobody. This is known as root squashing. If the user’s home directory must be on an NFS-mounted drive, then the user must also have a separate non-NFS-mounted workspace. The SDK may be copied to that workspace.

2. As root, use the `sudo vi sudoers` command to edit the `/etc/sudoers` file. See `man sudo` and `man sudoers` for assistance. Only *root* can run `vi sudo`. An example entry looks like:

```
testname ALL=(ALL) ALL
```

4.2 Multiple Users on the Same Development Host

The following items should be considered if the development host is shared by multiple users:

1. If users will share the same copy of the SDK (not recommended), then the users should all be in the same access group. The directory permissions can then grant read/write access for the entire group.

2. If multiple users will need to build the embedded root filesystem, see Section 44 – “Appendix T: Multiple Embedded Root Filesystem Builds” for directions on how to modify the `sudoers` entry.
3. If multiple users will use `tftpboot` to download to a board, see Section 34.4 – “About the TFTP Download Directory on the TFTP Server”.

5 Hands-on: Connect the Development Target

A copy of the *Quick Start Guide* is needed to correctly configure, connect, and power-on the development target board.

There are critical board-specific directions in the Quick Start Guide. In some cases, failing to read and follow the directions can irreparably harm both the development target and the development host.

The *Quick Start Guide* contains essential hardware information, including jumper information. Note that software issues including SDK installation, Minicom connection to the target console, and loading and booting are discussed in greater detail in this chapter than in the *Quick Start Guide*. After configuring and powering on the hardware, return to this chapter.

A hard-copy of the *Quick Start Guide* is provided with the evaluation or reference board. A copy of the *Quick Start Guide* is also on the CD supplied with the OCTEON evaluation board, and on the support site which is accessible from <http://www.caviumnetworks.com/>. If the hard-copy of the *Quick Start Guide* is missing, simply skip to Section 8 – “Hands-on: Install the SDK”, and use the directions there to locate the *Quick Start Guide* provided with the SDK.

5.1 PCI Development Target

To install a PCI development target into the development host (PCI host), first power off the development host using the command:

```
host$ sudo poweroff
```

Warning: Power off and unplug the host before inserting the PCI development target board. Most modern PCs still have power auxiliary on the PCI bus when powered off. Unplug the machine to make sure all power is off before plugging in the development target board.

PCI development target boards have special configuration options controlled by jumpers on the board (see the *Quick Start Guide* for details):

1. PCI target mode versus PCI host mode: This chapter assumes the PCI board is configured to be a PCI *target* board. To set to PCI target mode: PCI HOST jumper = OFF (not installed).
2. Boot from flash or boot over PCI bus from PCI host. Booting from flash is easiest. PCI boot is used when the bootloader itself is under development, and the board is configured as a PCI *target* board. To use flash boot: PCI BOOT jumper = OFF and PCI HOST jumper = OFF (not installed).

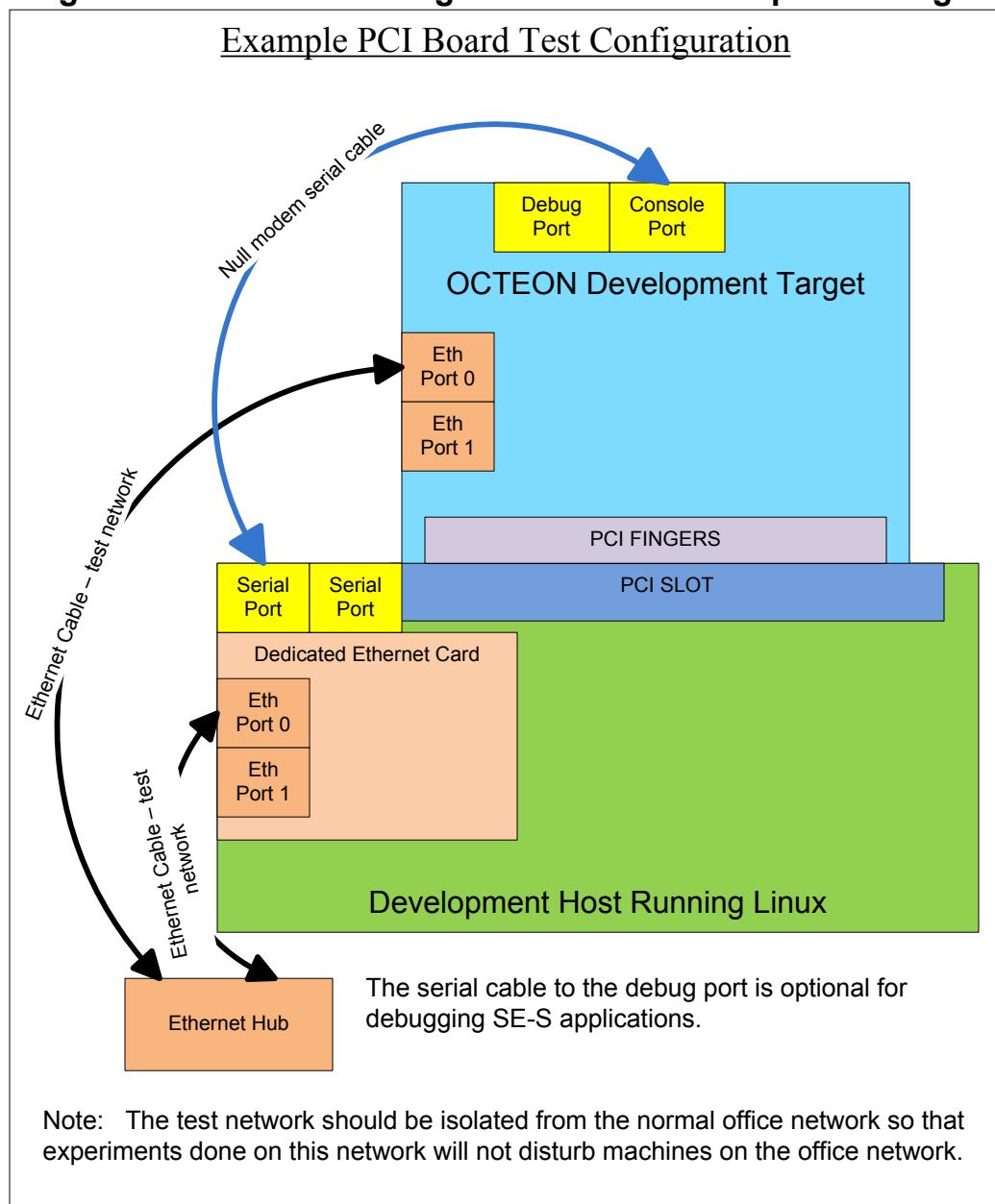
Connect the supplied null modem serial cable between UART0 on the board, and COM1 on the development host. This serial connection may be used to connect to the target console.

Note: A null modem serial cable is different from an ordinary serial cable. Null modem cables swap the transmit and receive lines. See Section 31 – “Appendix G: Null Modem Serial Cable Information” for details.

See Section 9.4.4 – “PCI Host Tools” for a list of `oct-pci-*` tools which can be used to control the PCI target from the PCI host. Using these convenient commands on the host, the target may be easily reset and an application downloaded and booted over the PCI bus. The target console may either be viewed over the PCI bus or via a serial cable attached to the target console port.

Debugging connections are discussed in the *Software Debugging Tutorial* chapter.

In the figure below, the PCI host is the same as the development host.

Figure 3: Hardware Configuration: PCI Development Target

SDK TUTORIAL

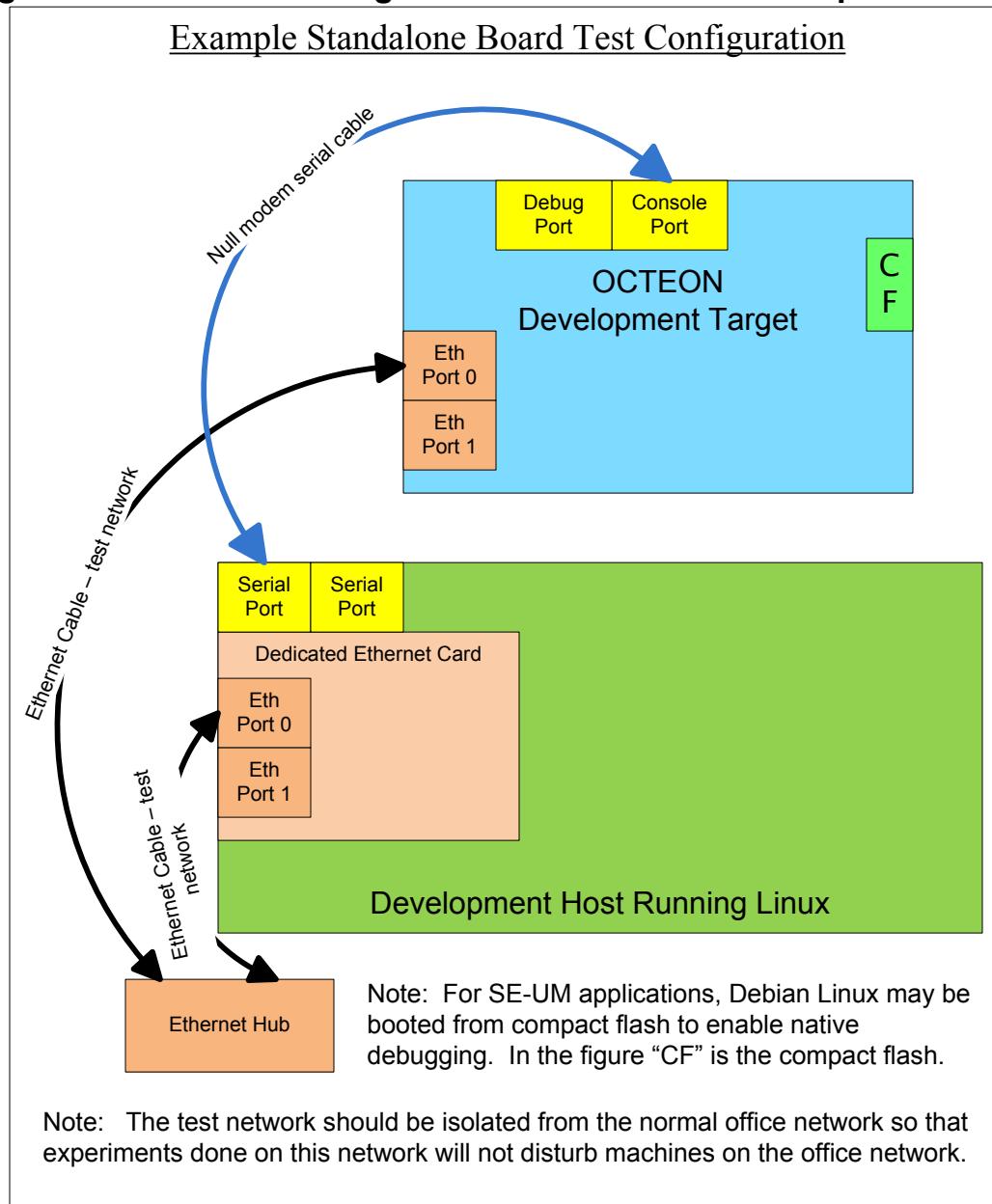
5.2 Standalone Development Target

If the development target is a standalone board, the board is booted from onboard flash. The application is typically downloaded via `tftpboot`. The target console is viewed over a serial cable attached to the console port on the target board.

Debugging connections are discussed in the *Software Debugging Tutorial* chapter.

In the figure below, a standalone OCTEON board is connected via Ethernet to the TFTP Server. In this case, the TFTP server is the same as the development host, which is running Linux. The TFTP server is used to `tftpboot` the application.

Figure 4: Hardware Configuration: Standalone Development Target



6 Hands-on: Viewing the Target Board Console Output

The easiest way to view the target board console output is by running the Minicom utility on the host, and connecting to the target board via a null-modem serial cable.

The console output for the target board is directed to UART0 on the target board. In this example, UART0 is connected via a serial cable to the first serial port on the development host. Linux connects to the first serial port on the device /dev/ttys0.

To connect to the console, use a terminal emulator such as minicom. The minicom utility is usually provided in a default Linux installation.

6.1 Starting Minicom

To start minicom, type

```
host$ minicom ttys0
```

If the following error occurs:

```
minicom: there is no global configuration file /etc/minirc.ttys0
Ask your sysadm to create one (with minicom -s).
```

then go to Section 6.2 – “Configuring Minicom”.

If the following error occurs:

```
Device /dev/ttys0 access failed: Permission denied.
```

then modify the group permission on the minicom utility (set group ID):

```
host$ sudo chmod g+s /usr/bin/minicom
```

(This will give anyone executing minicom its group ID, which is *uucp*, providing access to the /dev/ttys0 file.) For more about set group ID, see Section 26.4.2 – “The Set User ID Bit”.

If minicom is not in /usr/bin, then use the following command to locate the full pathname:

```
host$ which minicom
```

6.2 Configuring Minicom

This step is only needed if minicom has not already been configured correctly. By convention, the configuration file for ttys0 is /etc/minirc.ttys0.

If the file /etc/minirc.ttys0 already exists, and has correct contents, then this step may be skipped.

The `minirc.ttyS0` file will only contain changed settings relative to the `minirc.dfl` file, so the exact contents will vary. The key information is:

```
# Machine-generated file - use "minicom -s" to change parameters.
pr port      /dev/ttys0
pu baudrate 115200
pu bits     8
pu parity   N
pu stopbits 1
```

If the `/etc/minirc.ttyS0` file does not exist, or needs to be changed, type:

```
host$ sudo minicom -s
```

Figure 5: Minicom Configuration Menu

Filenames and paths
File transfer protocols
Serial port setup
Modem and dialing
Screen and keyboard
Save setup as dfl
Save setup as..
Exit
Exit from Minicom

Use the arrow keys to move down to “Serial port setup”, and press Enter.

Figure 6: Example Minicom Serial Port Configuration

A - Serial Device	:	/dev/ttys0
B - Lockfile Location	:	/var/lock
C - Callin Program	:	
D - Callout Program	:	
E - Bps/Par/Bits	:	115200 8N1
F - Hardware Flow Control	:	Yes
G - Software Flow Control	:	No

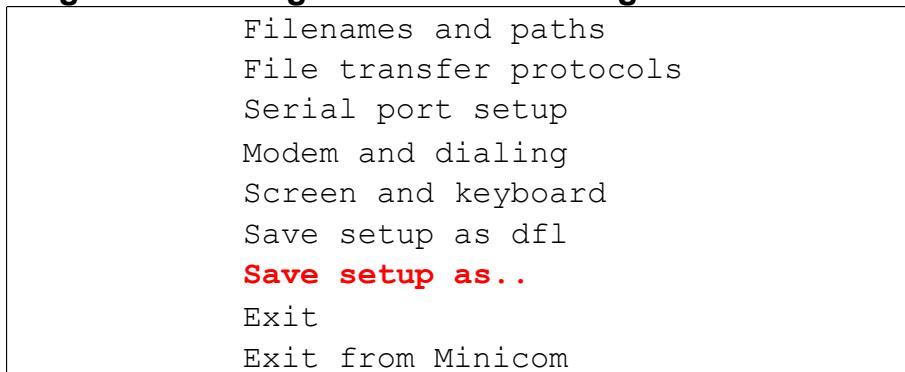
Change which setting?

Use letters to select any item which needs to be changed (such as “E”), and follow the prompts on the screen.

The required Minicom settings are: `/dev/ttys0`, `115200` (`115K`), `8-N-1` (8 bits, no parity, 1 stop bit). (Typically the first serial port is `/dev/ttys0`, and the second serial port is `/dev/ttys1`). Hardware control may be set to “Yes”. Not all boards support hardware control, but the “Yes” value will not cause problems on boards which do not support it.

Press **Enter** to return to the previous menu.

Figure 7: Saving the Minicom Configuration to a File



Use the arrow keys to move down to “Save setup as..” to save the new settings, and press **Enter**. When prompted for a name, enter the string “ttyS0”, then select “Exit from Minicom” to exit Minicom.

The file /etc/minirc.ttyS0 has now been created.

6.3 Minicom Basics

After the configuration file is created, to start Minicom use the command:

```
host$ minicom -w ttyS0 #substitute your tty file for ttyS0
```

Once in Minicom, type Ctrl-A Z to see the help menu (hold down the Ctrl key and the A key at the same time, and then let them go and press Z. The letters you type are really lower case A and Z).

Use Ctrl-A X to exit Minicom.

Note that Minicom offers session logging, to allow you to save the console session to a file.

Refer to the Minicom documentation for more information.

6.4 Verify Connection to Target Console Works

Power on or reset the development target board. If the target board is a PCI board inside the development host, it is already powered on.

On the target console, press **Enter** a few times. You should see text similar to the following text:

```
U-Boot 1.1.1 (U-boot build #: 160) (SDK version: 1.6.0-221) (Build time: Aug 31)
```

```
EBT3000 board revision major:3, minor:1, serial #: 2005-00087-3.1
OCTEON CN38XX-NSP revision: 3, Core clock: 500 MHz, DDR clock: 266 MHz (532
Mhz)
DRAM: 2048 MB
```

```
Flash: 8 MB
Using default environment

IPD backpressure workaround verified, took 36 loops
Clearing DRAM..... done
BIST check passed.
Net: octeth0, octeth1, octeth2, octeth3
Octeon ebt3000(ram) #
```

If there is no output, or if the output is odd looking, then recheck that cables are plugged into the correct ports, and recheck the Minicom configuration. (Note: There is a Microcontroller (MCU) port on the evaluation board which is only for factory use. Verify the serial cable is not accidentally connected to this port. There is only one MCU port, but there are two UART ports. Look for the double ports on the board.) If there is still a problem, see Section 6.8 – “Troubleshooting a Missing Bootloader Prompt”.

6.5 Minicom Line Wrap and Viewing the Bootloader Help Menu

The Minicom screen is as wide as the window it was run in. When attempting to view the bootloader help menu by using the command `help bootoct`, the help line is usually longer than the screen. When this happens, the `help` output is difficult to read because it flows off the screen, printing each character in the last column, so only the first screen-width characters in the line can be read.

For example, without line wrap:

```
target# help
bootoctelf - Boot a generic ELF image in memory. NOTE: This command does not
support
simple executive applications, use bootoct for those.
```

With line wrap on, the “ppor” in “support” is missing:

```
target# help
bootoctelf - Boot a generic ELF image in memory. NOTE: This command does not
support
simple executive applications, use bootoct for those.
```

To see the entire help, either start Minicom with the `-w` option, or use the Minicom help menu to find the *line wrap* option, and turn it on. Then the line will wrap and the entire line may be read.

Another option is to resize the minicom window. Note that resizing the minicom window will clear the screen. The screen will be blank until more characters are received.

6.6 Scrolling Up and Down

To scroll up and down use the arrow keys. To page up and down, use either the `PageUp` and `PageDown` keys, or `Ctrl-F` (forward) or `Ctrl-B` (backward).

6.7 A Typical Minicom Error

It is not unusual to get the following error when starting Minicom:

```
host$ minicom -w
Device /dev/ttyS0 lock failed: Operation not permitted.
```

The error can occur because Minicom creates a lock file that prevents more than one user from using the same device. This protection is useful when sharing a target board. To identify the person using the device, look at the owner of the lock file. The lock file is usually located in /var/lock. In the case of /dev/ttyS0, the lock file is named LCK..ttyS0.

```
host$ ls -l /var/lock
-rw-rw-r-- 3 testname lock 11 Aug 5 13:01 LCK..ttyS0
```

This problem may occur if the terminal session is closed before exiting Minicom.

Check the Minicom instructions on the development host for details.

6.8 Troubleshooting a Missing Bootloader Prompt

The word `Boot` should appear on the red diagnostic LEDs on the board indicating that the bootloader is running (if the board has the diagnostic LEDs). If the LEDs are present, but the word `Boot` is not, something is wrong with either the bootloader or the development target board.

The following steps will usually resolve the problem:

1. Verify the board is configured to boot from flash.
2. Verify the board is powered on.
3. Examine the board's LEDs for the word `Boot`. If the word `Boot` appears, but there is no prompt on the Minicom screen (even after the `Enter` key is pressed), then check the cables and port names carefully. Common errors include:
 - Plugging into the wrong UART port on the target board
 - Specifying the wrong serial port in the Minicom command line
 - Incorrect Minicom serial port settings
 - Failing to use a null-modem serial cable
4. If the word `Boot` does not appear in the board's LEDs, and the board is not a new one, the problem may be the bootloader code. In this case, boot from the failsafe bootloader, then restore the main bootloader. See Section 13.3 – “The Failsafe Bootloader”.
5. If there is still no bootloader prompt in the Minicom screen, re-check the hardware connection to the target console.

Note: If the board is not new, there is sometimes a problem if the EEPROM is corrupted. The bootloader will come up, but will also issue a warning. See Section 32.1 – “Detecting a Problem with the EEPROM” for more information.

6.9 Determining the Number of Cores on the OCTEON Processor

Once the bootloader is up, the `printenv` command can be used to determine the number of cores running on the OCTEON processor. The number of available cores is defined in the `coremask_override` environment variable:

```
target# printenv
<text omitted>
coremask_override=0xffff <>< 12 cores available
<text omitted>
```

7 Hands-on: Gather Key Hardware Information

The following information is needed later in this chapter:

1. The OCTEON model on the development target. If the OCTEON model is not obvious, see the next section.
2. The number of cnMIPS cores provided by the OCTEON model on the development target. This information is needed for application design, and to download the application to the board.

The following information is not needed for this chapter, but is useful for software development:

1. CPU Frequency
2. DDR Frequency
3. The board type

To locate CPU frequency, DDR frequency, and board type see Section 32 – “Appendix H: Query EEPROM to get Board Information”.

7.1 Determining the OCTEON Model on the Development Target

Note that when the board boots up, board information is printed on the target console, including the OCTEON model.

The bootloader will output text similar to the following text:

```
EBT3000 board revision major:4, minor:0, serial #: 2006-00257-4.0
OCTEON CN3860-NSP pass 2.X, Core clock: 500 MHz, DDR clock: 266 MHz (532
Mhz data rate)
DRAM: 2048 MB
Flash: 8 MB
```

7.2 Determining the Number of Cores on the OCTEON Processor

The number of cores can be determined by using the information in Section 6.9 – “Determining the Number of Cores on the OCTEON Processor”.

8 Hands-on: Install the SDK

The installation CD which accompanies the evaluation board contains the SDK (the base SDK and OCTEON Linux) and other product files which may or may not be needed depending on your application and OCTEON model.

The optional products on the CD include:

1. Linux on the OCTEON processor: If you plan to use Linux on any OCTEON cores, install the Linux package after installing the base SDK. Linux is needed to run some of the examples in this chapter.
2. Support for special hardware capabilities on the OCTEON processor: Crypto, ZIP, and DFA. The OCTEON model must support these optional hardware capabilities. The software support for these capabilities is provided by special RPMs.
3. Software to implement packet send/receive over the PCI bus. A PCI host can be configured to offload work to the OCTEON PCI target by passing packets over the PCI bus.

These other product files are discussed briefly in Section 25.1.1 – “Product Files on the Installation CD”.

The product files may also be downloaded from the support site. See instructions in Section 27.1 – “Installing from the Support Site Instead of a CD”.

Note that the other packages on the CD are not included in the SDK or in the SDK documentation.

Note that only two SDK files need to be installed to follow the directions in this tutorial: the base SDK (OCTEON-SDK-* .i386 .rpm) and OCTEON Linux (OCTEON-LINUX-* .i386 .rpm). (The “*” character means the exact text varies with SDK version number.)

In addition to the SDK and optional product files, there are optional toolkits and Application Development Kits (ADKs) which can accelerate product development. The toolkits are introduced in Section 25.1.2 – “Toolkits”. The ADKs are introduced in Section 25.1.3 – “Optional Application Development Kits (ADKs) “.

At a minimum, install the base SDK and the *Quick Start Guide*. These are needed by everyone.

8.1 Mounting the CD

Insert the CD which came with the Evaluation or Production Board into your development host’s CDROM drive. Usually the system will automatically mount the CD. In this example, the mount point is /media/cdrom.

Each file name which ends in .rpm is referred to as an RPM (Red Hat Package Manager) file, *product*, or *package*.

The following files will be installed:

OCTEON-SDK-* .i386.rpm
OCTEON-LINUX-* .i386.rpm

Note that no technical problems will occur if unneeded RPMs are installed.

8.2 Using the RPM Utility to Install the Packages

Unless instructed otherwise, the `rpm` utility will install the SDK packages in the default directory, `/usr/local/Cavium_Networks/OCTEON-SDK`. During installation, this directory will be created.

We recommend installing the SDK to a custom directory instead of the default directory. The advantage in using a custom directory is that you can specify the SDK version number as part of the directory path. This is helpful when a new SDK is later installed: it can be installed to a different path. For example, `/opt/173` and `/opt/181` would contain SDK 1.7.3 and 1.8.1, respectively. If the default installation directory is used to install two different SDKs, the new SDK will over-write the previously installed version.

Note: When using RPM, the directory name cannot be simply changed after installation. Instead, to change the directory path name, the package must be uninstalled and reinstalled.

After the SDK is installed, the installation directory can only be changed by un-installing the SDK and re-installing it.

In this chapter, the custom SDK installation directory is `/opt/181`. When following the instructions in this chapter, specify your custom installation directory instead of `/opt/181`.

To install to a custom directory, add the `--prefix <directory_name>` option to the `rpm` command. Note: `<directory_name>` is the directory pathname, including the leading `/`.

To install the base SDK and OCTEON Linux to a custom directory, run the command:

```
# substitute /media/cdrom for your mount directory if needed
host$ sudo rpm -i --prefix /opt/181 /media/cdrom/OCTEON-SDK*.rpm
host$ sudo rpm -i --prefix /opt/181 /media/cdrom/OCTEON-LINUX*.rpm
```

As an alternative, to install all the RPMs:

```
# substitute /media/cdrom for your mount directory if needed
host$ sudo rpm -i --prefix /opt/181 /media/cdrom/*.rpm
```

Note that each RPM can be installed only once. Attempting to re-install the RPM without first removing it will result in an error message.

Note: When installing other software packages, use the same prefix so the packages install in the same directory.

For more information about the `rpm` utility, see Section 27 – “Appendix C: About the RPM”.

After installing the SDK, unmount the CD:

```
host$ umount /media/cdrom
```

8.3 Making a Copy of the Installed SDK

The directions in this chapter assume the SDK is copied from the original installation directory (`/opt/181/OCTEON-SDK`) to a working directory, usually located in the user’s home directory. The user edits the copy and original installation is preserved untouched for reference.

In addition to preserving a reference copy, creating a working copy allows the user to work at a lower privilege level. The original installation can only be used by a user with *root* privilege; the copy can be edited by a normal (non-*root*-privilege) user. Using a normal login instead of *root* helps protect the system from accidental damage.

The copy of the SDK must be done using the `sudo` command because there are some file permissions which are damaged if the copy is not made using *root* (super user) privilege. See Section 26.4 – “Linux File Information and the Set User ID Bit” for more information.

Additionally, be sure to copy the *entire* SDK.

Note: The GNU tool chain provided by Cavium Networks is included in the SDK. This tool chain contains special Cavium Networks-specific instructions which take advantage of the OCTEON hardware acceleration. Details of the special instructions are provided in the Software Overview chapter, and in the Hardware Reference Manual. Using this tool chain is highly recommended because it will result in the highest performance applications. This tool chain is automatically provided when the entire SDK is copied.

For help with Linux commands, see Section 26 – “Appendix B: Linux Basics”.

A good private directory name will include the SDK version, such as `/home/testname/sdk181`. In this document, the private copy of the SDK is in `/home/testname/sdk`. This name was chosen to provide directions which do not depend on the SDK version.

For example:

```
host$ mkdir ~/sdk # create the sub-directory "sdk" in the user's home
directory
host$ cd /opt/181/OCTEON-SDK
host$ sudo cp -r . ~/sdk # recursively copy files to the new directory
```

This copy can take a few minutes. The word *recursive* means that the command operates on all the sub-directories, not only the top-level directory.

To see the files being copied, add the verbose option to cp, type cp -rv . ~/sdk. (Note that using verbose option will slow down the command.) A copy of the SDK should now be located in the user's home directory, in the sub-directory sdk. In this document, the copy is located in /home/testname/sdk.

The new sub-directories will be owned by *root*.

```
host$ cd /home/testname/sdk
host$ ls -ld docs
drwxr-xr-x 4 root root 4096 Jan 23 12:14 docs
```

If the owner is not root, then the sudo cp -x step was not executed correctly. Double check that the sudo command was used. Also double check that the destination directory was not on an NFS-mounted drive. To see if the destination is on an NFS-mounted drive, cd ~/sdk and type the df . command. If the output is of the form hostname:mountname, then the directory is NFS-mounted. See Section 26.4.2 – “The Set User ID Bit” for information on why the cp -x command requires root privilege.

Change the owner of all the directories to your user name before continuing:

```
# use your user name instead of "testname"
host$ sudo find . -type d -exec chown testname {} \;
```

Note: If this step is omitted, then examples cannot be built due to a permission error.

8.4 The OCTEON_ROOT Environment Variable

The working directory (in this example /home/testname/sdk) is referred to as \$OCTEON_ROOT in the rest of this chapter. OCTEON_ROOT is an environment variable. \$OCTEON_ROOT refers to the value of the environment variable.

8.5 Setting Environment Variables on the Development Host

A script in the \$OCTEON_ROOT directory, env-setup, will set essential environment variables. Before executing the script, cd to the \$OCTEON_ROOT directory. The env-setup script will be located there. This script must be executed in the \$OCTEON_ROOT directory because the value of OCTEON_ROOT will be set by the script to the current working directory. An example of the error which will occur if the script is not executed in the correct directory is shown at the end of this section.

The following variables are set by the env-setup script:

1. OCTEON_ROOT – this variable will be set to the directory the env-setup script is executed in (typically, the user's copy of the SDK).
2. The PATH environment variable is modified to add the directories containing the tool chain binaries and development host utilities.
3. OCTEON_MODEL
4. OCTEON_CPPFLAGS_GLOBAL_ADD

The script has only one required argument: OCTEON_MODEL. The file \$OCTEON_ROOT/octeon-models.txt contains a list of legal values for OCTEON_MODEL. Unless you have a specific reason to specify a model containing the suffix *pass1* or *pass2*, select a

model without a *pass* suffix, such as `OCTEON_CN58XX`. The `env-setup` script will set the environment variable `OCTEON_MODEL` to the specified value.

In addition to the required argument, the optional arguments shown in the table below can be specified.

Table 1: Options to the env-setup Script
In the table below, the default values are highlighted.

Option	Description	Value added to OCTEON_CPPFLAGS_GLOBAL_ADD
--noverbose (default)	The script will not provide verbose output.	
--verbose	The script will provide verbose output.	
--runtime-model (default)	Runtime model checking is useful if software can run on different OCTEON models. If this value is set to 1, then the binary will run on all OCTEON models. The value shown in the column to the right is added to the environment variable OCTEON_CPPFLAGS_GLOBAL_ADD.	-DUSE_RUNTIME_MODEL_CHECKS=1
--noruntime-model	If the value is set to 0, the binary will only run on the OCTEON model specified during the env-setup step. The value shown in the column to the right is added to the environment variable OCTEON_CPPFLAGS_GLOBAL_ADD.	-DUSE_RUNTIME_MODEL_CHECKS=0
--nochecks (default)	Default value. Disables various consistency and programming checks. The values shown in the column to the right are added to the environment variable OCTEON_CPPFLAGS_GLOBAL_ADD.	-DCVMX_ENABLE_PARAMETER_CHECKING=0 -DCVMX_ENABLE_CSR_ADDRESS_CHECKING=0 -DCVMX_ENABLE_POW_CHECKS=0
--checks	Enables runtime checking of SSO consistency, CSR addresses, and other parameters. This option strongly recommended to identify coding errors which are otherwise difficult to find. The values shown in the column to the right are added to the environment variable OCTEON_CPPFLAGS_GLOBAL_ADD. See Note 1.	-DCVMX_ENABLE_PARAMETER_CHECKING=1 -DCVMX_ENABLE_CSR_ADDRESS_CHECKING=1 -DCVMX_ENABLE_POW_CHECKS=1
Note 1: Do not use --checks when doing performance testing: The --checks option adds code which will slow performance.		

Note: More information on `OCTEON_CFLAGS_GLOBAL_ADD` may be found in the SDK document “*OCTEON SDK Config and Build System*”.

The `env-setup` script must be *sourced* to modify the environment variables of the current shell (usually `bash`). When the script is sourced (`source env-setup`), all shells started from this shell will inherit the shell’s environment variables. If the script is simply executed using the command `./env-setup`, then the environment variables are no longer set when the script exits. (The command `./env-setup` will result in the message: “`bash: ./env-setup: Permission denied`” because the file does not have execute permission: it is meant to be sourced, not simply executed.)

Note: The source command shown in this tutorial is for the shell `bash`. Different shells source files differently. For example, other shells often use the command `. env-setup` (dot space `env-setup`) instead of `source env-setup`.

For example, after using the `--runtime-model` argument, the change to `OCTEON_CPPFLAGS_GLOBAL_ADD` can be seen using the `echo` command:

```
host$ source env-setup OCTEON_CN38XX --runtime-model
host$ echo $OCTEON_CPPFLAGS_GLOBAL_ADD
-DUSE_RUNTIME_MODEL_CHECKS=1 -DCVMX_ENABLE_PARAMETER_CHECKING=0
-DCVMX_ENABLE_CSR_ADDRESS_CHECKING=0 -DCVMX_ENABLE_POW_CHECKS=0
```

If the script is not executed in `$OCTEON_ROOT`, the following error will occur:

```
host$ source /home/testname/sdk/env-setup OCTEON_CN38XX
bash: ./env-setup.pl: No such file or directory
```

8.6 Adding `env-setup` to Your Profile

To add the `env-setup` step to your user profile (so it will be performed each time you login), login using your non-root user name, and add the three lines shown below your `~/.bash_profile`, then either start a new shell (`host$ exec bash -l`), or log out and back in. Either of these actions will cause the `.bash_profile` file to be executed. (Note that simply exiting a terminal session and starting a new one will not cause `.bash_profile` to be executed. It is necessary to log out of the development host and log back in.)

In the example shown below, substitute your desired values for `OCTEON_MODEL`, `--runtime-model`, and `--verbose`:

For example, the three lines to add to the `.bash_profile` might be:

```
pushd /home/testname/sdk # substitute your path to the working copy
# substitute your OCTEON_MODEL for OCTEON_CN38XX and your options
source env-setup OCTEON_CN38XX --noruntime-model --verbose --checks
popd
```

8.7 Viewing the Installed SDK Version

A good test of correct installation and environment setup is to call the tool `oct-version` from the command line.

For example:

```
host$ oct-version
Cavium Networks Octeon SDK version 1.8.0, build 275
```

If the `env-setup` step is not right, then the following error will occur:

```
host$ oct-version
oct-version: Command not found.
```

This command can optionally be added after the `popd` in `.bash_profile`.

The next section introduces the files, directories, tools, and example code provided with the SDK.

9 Hands-on: Tour the Installed SDK

The SDK includes documentation, tools, and example code. In this section, a quick overview of these items is presented. The reader can follow along by locating the items discussed in this chapter.

9.1 Key Information

The three most important directories are:

- `executive/` – where the Simple Executive code is located
- `examples/` – where the example applications are located
- `linux/` – where the Linux files are located

The most important documents are:

- `$OCTEON_ROOT/docs/OCTEON-SDK-QSG.pdf` – the *Quick Start Guide*
- `$OCTEON_ROOT/docs/html/index.html` – the index to the SDK documentation

9.2 Looking at the Installed Directories

The following two tables introduce some of the key files and directories in the SDK.

Table 2: Key SDK Files

Files	Description
application.mk	One of three Makefiles included by all of the example Makefiles.
common.mk	One of three makefiles included by all example Makefiles, such as examples/hello/Makefile. It includes common-config.mk.
common-config.mk	Common configured variables needed by Makefiles which refer to all or a portion of the simple executive. This Makefile is included by common.mk.
env-setup	Setup script needed to set environment variables such as OCTEON_ROOT before using the tools to download or compile.
env-setup.pl	Supporting script for env-setup: this script is not called directly by the user
executive/cvmx.mk	One of three Makefiles included by all of the example Makefiles. Contains the list of Simple Executive source files. Customize this to fit the application as needed.
octeon-models.txt	A list of OCTEON models which may be used as input to the env-setup script
README.txt	Contains information on the operation system needed for the host, on memory requirements, environment variables, how to build and run hello (on the simulator), how to determine which version of the SDK you are running.
release-notes.txt	Contains an introduction to the directories, and a list of changes made with each SDK release.

Table 3: Key SDK Directories

Directories	Description
bootloader/	Contains source for the bootloader (U-Boot).
components/	Non-SDK packages install here (this directory is not present until other packages are installed).
diagnostic/	Contains the utilities for testing and diagnosing OCTEON boards (hardware diagnostics). Includes tests for RGMII, UART, memory, etc.
docs/	Contains the html documentation for the SDK. Also contains the original documentation and copyright information for the tool chain, debugger, and bootloader.
examples/	Contains the example applications. See the README files inside for details. Many examples can be compiled and run under both Simple Executive and Linux.
executive/	Contains the source for the simple executive. Customize this to fit your application as needed.
gpl-executive/	A symbolic link to the executive directory (newer SDKs may not have this link).
host/	Directory containing links to all of the development tools for x86. Includes host/bin.
licenses/	Contains Cavium Networks License agreements.
linux/	Contains the version of Linux designed to run with the OCTEON.
simulator/	Contains the OCTEON simulator and related utilities such as oct-debug, oct-profile, viewzilla, perfzilla.
target/	Contains links to all of the target (OCTEON) files. Has three sub-directories: bin, include, and lib. The bin directory contains downloadable bootloader files.
tools/	Symbolic link to one of the two tools directories, for example: tools -> tools-gcc-4.1. By default, this will be set to the most current set of tools. Use the command ls -l tools to see what directory tools is linked to.
tools-gcc-*/	The exact tool path will vary. Two tools directories are provided. The higher number corresponds to the newest and best tools. The path with the lower number corresponds to the next-oldest set of tools provided by Cavium Networks. When starting new development, use the newest tools (the higher number directory). The tools-gcc-* directory contains two different tool chains: mipsisa64-octeon-elf target tools based on newlib C library and mips64-octeon-linux target tools based on the glibc or the uClibc libraries.

9.3 Documentation Provided with the SDK

The documentation provided with the SDK provides detailed information. At the top level, there is a `README.txt` and `release-notes.txt` file. Under the `docs` directory, there is another `README.txt` file. Most of the documentation is in html format and can be accessed through `docs/html/index.html`. This information can be displayed in your web browser.

For GNU *cross* tool chain documentation, see:

1. `$OCTEON_ROOT/tools/man`
2. `$OCTEON_ROOT/tools/info` directories.

For GNU *native* tool chain documentation, see:

1. `$OCTEON_ROOT/tools/mips64-octeon-linux-gnu/sys-root/usr/man`
2. `$OCTEON_ROOT/tools/mips64-octeon-linux-gnu/sys-root/usr/info`

Other documentation can be found most easily with the Linux command:

```
# find all files in the current directory ending with the string ".txt".  
host$ find . -name "*.txt"
```

(There is a huge amount of documentation provided, especially with the GNU tool chain.)

The documentation referenced in the table below may be viewed from your browser after the SDK is installed:

```
host$ firefox file:///$/OCTEON_ROOT/docs/html/index.html &
```

Table 4: Documentation Provided via doc/html/index.html

Getting Started	
<i>OCTEON SDK Documentation Table of Contents</i>	A brief list of documents accessible from index.html.
<i>OCTEON SDK Quick Start Guide</i>	Basic information needed to get an evaluation board up and running. This is also available as a separate PDF file in the <code>docs</code> directory.
<i>OCTEON SDK Release Notes</i>	Release notes.
<i>OCTEON SDK Overview</i>	A brief summary of what is included with the SDK.
<i>OCTEON SDK Examples Overview</i>	Brief descriptions of example code provided with the SDK.
Hardware Configuration	
<i>Developing with OCTEON as a PCI target</i>	Introduces the tools used when OCTEON is a PCI target. This is the most common configuration.
<i>OCTEON as a PCI host</i>	OCTEON as a PCI host.
Simulator	
<i>OCTEON Simulator</i>	Details about the OCTEON hardware simulator.
Simple Executive	
<i>OCTEON Simple Executive Overview</i>	Briefly introduces the Simple Executive: API, memory map. Introduces the hardware units.
<i>OCTEON SDK Config and Build System</i>	Details on how to configure and build Simple Executive applications
Linux	
<i>Linux Userspace on the OCTEON</i>	Read first when using Linux. Contains information also useful for Simple Executive users. ABIs, User Space Memory Map, Building Applications, Detailed Simple Executive Port, Startup, Booting, Core Affinity, and other OCTEON-specific features.
<i>Linux on the OCTEON</i>	Read second when using Linux.
<i>Running Debian GNU/Linux on OCTEON</i>	How to run the Debian root filesystem from compact flash. The Debian root filesystem can be used to provide native build and debug of Linux applications).
<i>Linux on Small OCTEON Systems</i>	Configuring Linux to run on less than 256 Mbytes of system memory.
Bootloader Details	
<i>OCTEON Bootloader</i>	Details of the bootloader
Debugging	
<i>Simple Executive Debugger</i>	How to debug SE-S applications. The same debugger is used to debug the Linux kernel.
<i>Linux Userspace Debugging</i>	How to debug Linux user-mode applications.
Performance Tuning	
<i>Performance Profiling using Oprofile</i>	Performance tuning on Linux
<i>Performance Profiling using Viewzilla</i>	Performance tuning on Simulator
<i>Profile-feedback optimization</i>	Performance tuning on Simple Executive
Tools	
<i>GCC 4.1 Upgrade Guide</i>	Notes on how to upgrade to the newer GNU tool chain.
Hardware Diagnostics	
<i>OCTEON Diagnostics</i>	Hardware diagnostics.
Additional Material	
<i>File List</i>	Contains documentation about the contents of Simple Executive files. Once in the file, click on "more" to get more information.
<i>Data Structures</i>	Documentation about the Simple Executive data structures.

9.4 Development Tools

There are three tools directories commonly used in building, running, and debugging applications:

```
$OCTEON_ROOT/tools/bin
$OCTEON_ROOT/host/bin
$OCTEON_ROOT/linux/kernel_2.6
```

The tools directory is a symbolic link to the actual version of the GNU tool chain being used, which should be the newest version available. In this example, the tools directory is linked to tools-gcc-4.1.

This symbolic link can be seen with the Linux command ls:

```
host$ ls -ld tools
lrwxrwxrwx 1 root root 13 Oct 10 09:55 tools -> tools-gcc-4.1
```

The tools/bin directory contains the GNU compiler and associated programs which are run on the development host used in cross-development to the target. The tools/man directory contains manual pages documenting these tools.

If there are two tools-gcc directories (directory names are similar to: tools-gcc-4.1/bin), then these are two different versions of the GNU tool chain. The higher number is the most recent version (preferred).

The host/bin directory contains other tools which are run on the development host, including the simulator, and performance analysis tools such as perfzilla.

9.4.1 Accessing the Tools from the Command Line

The script env-setup will add the tools/bin and host/bin directories to the PATH, relative to \$OCTEON_ROOT.

In this example, the working copy of the SDK is in /home/testname/sdk. After the env-setup step, the two directories tools/bin and host/bin are pre-pended to the PATH variable.

```
host$ echo $PATH
/home/testname/sdk/tools/bin:/home/testname/sdk/host/bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
```

9.4.2 Tools Documentation

9.4.2.1 Tools Manual Pages

The manual (man) pages for the GNU tools man are accessed via the man utility. The tools manual pages are located in \$OCTEON_ROOT/tools/man. To access the manual pages, first add this directory to the beginning of the MANPATH environment variable. The MANPATH variable

contains a list of directories to be searched (in left to right order) by the `man` utility when looking for manual pages. In this example `$OCTEON_ROOT` is `/home/testname/sdk`.

Note: To be extra-sure you are reading the correct manual pages, consider moving the host system's manual directory:

```
host$ sudo mv /usr/share/man /usr/share/man.save
host$ man ar
No manual entry for ar
```

(Move `/usr/share/man.save` back to `/usr/share/man` when done with this experiment.)

In bash, using the directory `/home/testname/sdk` as an example:

```
host$ cd /home/testname/sdk/tools/man
host$ pwd
/home/testname/sdk/tools/man
# be careful to use the back quote ( ` ) character on the keyboard.
# It is NOT the quote ( " ) or single quote ( ' ) character.
# To verify the back quote is used, try the command: echo `pwd`
host$ echo `pwd`
/home/testname/sdk/tools/man
host$ export MANPATH=`pwd`:$MANPATH
host$ echo $MANPATH
/home/testname/sdk/tools/man:/usr/share/man
```

9.4.2.2 Tools Info Files

The information (`info`) files for the GNU tools are accessed via the `info` utility. The tools information pages are located in `$OCTEON_ROOT/tools/info`. To access the information pages, `cd` to the `$OCTEON_ROOT/tools/info` directory, use `ls` to locate the file of interest, such as `gdb.info`, and use the `info -f` option to open the file.

Note: To be extra-sure you are reading the correct information pages, consider moving the host system's information directory:

```
host$ sudo mv /usr/share/info /usr/share/info.save
host$ info gdb
info: dir: No such file or directory
```

In bash, using the directory `/home/testname/sdk` as an example:

```
host$ cd /home/testname/sdk/tools/info
host$ pwd
/home/testname/sdk/tools/info
host$ info -f ./gdb.info
```

9.4.3 GNU Cross-development Tool Chain

In the `tools/bin` directory, there are two sets of GNU tools. These tools include the cross compiler, linker, and libraries which are needed to build software to run on the OCTEON processor. Both sets of tools have been modified to support OCTEON-specific instructions to achieve maximum runtime performance. The two sets of tools are:

1. Simple Executive Development Tools: The `mipsisa64-octeon-elf-*` tools are used to build Simple Executive applications.
2. Linux Development Tools: The `mips64-octeon-linux-gnu-*` tools are used to build the Linux kernel and Linux User-mode applications.

Note: The GNU tool chain which is provided by Cavium Networks contains special OCTEON-specific instructions which take advantage of the OCTEON hardware acceleration. Using this tool chain is highly recommended because it will result in the highest performance applications.

The GNU cross tool chain includes the following utilities (shown without their `mipsisa64-octeon-elf-` or `mips64-octeon-linux-gnu-` prefix):

Table 5: GNU Tool Chain

Select the correct prefix when using these tools:

1. The `mipsisa64-octeon-elf-*` tools are used to build Simple Executive applications.
2. The `mips64-octeon-linux-gnu-*` tools are used to build the Linux kernel and Linux User-Mode applications.

Tool	Description
addr2line	Convert addresses into file names and line numbers. If code is compiled with <code>-g</code> , the pc values may be traced back to the source line numbers.
ar	Archiver
as	Assembler
c++	Same as <code>g++</code>
cpp	C preprocessor
ddd	GUI Debugger (Data Display Debugger).
g++	GNU C++ compiler
gcc	GNU C compiler - same as "gcc-*", below
gcc-*	GNU C Compiler version *. (* means the exact version number can vary.)
gcov	Coverage testing tool. Used to display some aspects of profiling data. See Note 1.
gdb	Debugger
ld	Linker
merge-gcdata	Merge profiling data. See Note 1.
nm	List symbols from object files.
objcopy	Copies a binary file, possibly transforming it in the process.
objdump	Display information from object files.
ranlib	Generate and index to archive
readelf	Display information about the object file
size	Display size of segments in object file
strings	Display strings in object file
strip	Strip object file to reduce size.

Note 1: For more information, see the SDK Document *Profile-feedback Optimization*.

9.4.4 PCI Host Tools

When the OCTEON processor is configured as a PCI target, `oct-pci-*` commands may be executed on the PCI host to control the target board. This is very convenient during development and debugging. Note that for the `oct-pci-boot` command to work, the board must be configured as a PCI target and also configured for PCI booting instead of flash booting.

There are no man pages for the PCI host tools. For more information on `oct-pci-*` utilities, see the SDK document “Developing with OCTEON as a PCI Target”. Also see Section 42 –

“Appendix R: About `oct-pci-console`” and Section 43 – “Appendix S: About `oct-pci-reset` and `oct-pci-csr`”.

Fred

Table 6: Host Tools

Tool	Description (<i>Commonly used commands are highlighted.</i>)
cvmx-config	Configure Simple Executive resources such as FPA, FAU, Scratch pad, Timers. Called only by the Makefiles to process files in the config directory.
dwarfdump	Run dwarfdump on an ELF file to print DWARF object file information in human-readable form. This is used internally by <code>perfzilla</code> and <code>viewzilla</code> to get debugging information.
oct-debug	Invokes the GUI debugger ddd (Data Display Debugger). This is for use with Simple Executive applications only. This debugger will connect to either the simulator, or over to the hardware target. Will connect to hardware over serial connection or PCI. If <code>mipsisa64-octeon-elf-ddd</code> is not in the PATH, then the command line debugger will be invoked instead.
oct-debuginfo	This utility is used when analyzing a crash dump. If code which crashed was compiled with <code>-g</code> , then the PC (program counter) values obtained from a crash dump may be traced back to the source line numbers in the code. This utility works with both Linux and Simple Executive applications.
oct-linux	Used to simplify execution of Linux on the simulator. See Note 5.
oct-packet-io	Used to pass packet data to and from the simulator. Use is demonstrated in the passthrough example.
oct-pci-boot	Boot the OCTEON PCI target board. This program configures the DDR controller on the OCTEON, then loads and runs the U-Boot bootloader on it. See Note 1.
oct-pci-bootcmd	Used to run bootloader commands over PCI to an OCTEON PCI target board. Used to run image files on the cores. This command may be used to start the application which was downloaded via the <code>oct-pci-load</code> command. See Note 1.
Notes	
Each note references an SDK Document where more information can be found. (view \$OCTEON_ROOT/docs/html/index.html in a web browser)	
Note 1: See the SDK document <i>Developing with OCTEON as a PCI Target</i>	
<i>Continued in the next table...</i>	

Table 7: Host Tools, continued

Tool	Description (<i>Commonly used commands are highlighted.</i>)
oct-pci-console	Used to allow the bootloader, simple executive applications, and Linux to redirect their console output over the PCI bus to the <code>oct-pci-console</code> utility. This allows running interactive programs on OCTEON without a serial connection. See Note 1.
oct-pci-csr	OCTEON CSR (Control and Status Registers) may be read and written directly over PCI using this utility. See Note 1.
oct-pci-ddr	Tests memory after it is set up. This is a quick "alive" test. Use <code>hw-ddr2</code> for more extensive memory testing.
oct-pci-load	Load a file into OCTEON memory over PCI from the host. See Note 1.
oct-pci-memory	Read or write DDR memory on the OCTEON target over PCI.
oct-pci-pow	Display OCTEON POW (SSO) state. A debugging tool which should not be used unless requested by a Cavium Networks representative. The output is not easy to read.
oct-pci-profile	Profiling of any code running on OCTEON: both Simple Executive and Linux userspace applications. This profiler runs over PCI.
oct-pci-reset	Reset the OCTEON PCI target board. See Note 1.
oct-pci-tra	Access the trace buffer. See the Hardware Reference Manual for more information on trace functions. This is somewhat complicated to use.
oct-profile	Performance tuning on Linux. See Note 1.
oct-sim	Invoke the simulator. See Note 4.
oct-uart-io	A script to help when running the simulator. The script telnets to localhost to get the simulator output. Not used for any other purpose.
oct-uudecode	Used with profile-feedback optimization. See Note 2.
oct-version	Show the version of the SDK installed on the development machine.
perfzilla	Graphical profiling tool. Used to view the output of the <code>oct-pci-profile</code> utility. See Note 1.
viewzilla	Graphical performance tuning on the simulator. See Note 3.
vz-cut	Copy a section out of a <code>viewzilla</code> file into a smaller file. See Note 3.
Notes	
Each note references an SDK Document where more information can be found.	
Note 1: <i>Developing with OCTEON as a PCI Target</i>	
Note 2: <i>Profile-feedback Optimization</i>	
Note 3: <i>Performance Profiling with Viewzilla</i>	
Note 4: <i>OCTEON Simulator</i>	
Note 5: <i>Linux on the OCTEON</i>	

9.5 Oprofile Profiling Tools

Profiling tools are built with the Linux build (`make kernel`). After the build, the following tools are in `$OCTEON_ROOT/linux/embedded_rootfs/build/oprofile-0.9.2:`

Table 8: Profiling Tools (Oprofile)

Tool	Description	SDK Document (<i>view docs/html/index.html in a browser</i>)
utils/opcontrol	Profiling a Linux application on <code>embedded_rootfs</code>	<i>Performance Profiling using Oprofile</i>
pp/opreport	Display the Oprofile profiling data.	<i>Performance Profiling using Oprofile</i>

9.5.1 Hardware Diagnostic Tools

The following tools are in the `$OCTEON_ROOT/diagnostic` directory. The list of tools available will change over time, so check the SDK document “*OCTEON Diagnostics*” for the most current list.

Table 9: Hardware Diagnostic Tools

Tool	Description
hw-ddr2	DDR2 memory diagnostics
hw-gpio	General Purpose I/O (GPIO) diagnostics
hw-llm	Low-Latency Memory (LLM) diagnostics
hw-pcm	PCM (Pulse Code Modulation) telephony interface diagnostics
hw-rc	cnMIPS RISC Core diagnostics
hw-rgmii	RGMII communications interface diagnostics
hw-spi	SPI-4.2 communication interface diagnostics
hw-uart	UART (RS-232 serial port) diagnostics
hw-usb	USB device interface diagnostics
support	Miscellaneous diagnostic utility routines

Note: In the SDK document *OCTEON Diagnostics*, click on "Notes" for more information on each tool.

9.6 Native Tools (Run on the Development Target)

Native tools are tools which run on the development target, not the development host. Native tools are only available when the development target is running Linux. Cores which are running SE-S applications do not have native tools because SE-S applications run standalone (without an operating system), so there is only one process.

Native tools may be accessed via:

1. The in-memory embedded root filesystem: contains a *subset* of the native tools.
2. Debian root filesystem located on a flash card: contains *all* the native tools.
3. A NFS-mounted root filesystem: contains *all* the native tools. This option is not covered in this tutorial. Note that using a NFS-mounted filesystem might not be the best choice for your application: NFS depends on the Cavium Networks Ethernet driver, and applications which re-configure the OCTEON hardware should not be run when the Ethernet driver is running.
4. An NFS-mounted filesystem (not the root filesystem): contains the tool chain. This option is sometimes used when running Linux on the embedded root filesystem. The tool chain is located in a directory on the development host, \$OCTEON_ROOT/tools/mips64-octeon-linux-gnu/sys-root. This directory is NFS mounted on the target. Directions may be found in the SDK document “*Linux Userspace on OCTEON*”. Note that using a NFS-mounted filesystem might not be the best choice for your application: NFS depends on the Cavium Networks Ethernet driver, and applications which re-configure the OCTEON hardware should not be run when the Ethernet driver is running.

Directions for building and running Linux on the embedded root filesystem and the Debian root filesystem are in this tutorial.

See Figure 12 – “Root Filesystem Locations”.

Once Linux is booted on the target, the native utilities and tools are usually located in the /bin, /sbin, and /usr/bin directories in the root filesystem.

The Cavium Networks native tool chain (gcc, etc) is supplied to build user-mode applications on the OCTEON processor. Three tools are supplied in the embedded root filesystem: gdb, gdbserver, and gprof. (Note that these commands are supplied without the mips64-octeon-linux-gnu-* prefix.) To access the other tools, use the Debian root filesystem instead of the embedded root filesystem.

In addition to the Cavium Networks native tool chain, the following Cavium Networks-specific tools are supplied:

Table 10: Special Cavium Networks Native Tools, Part 1

The following native tools are available when Linux is running on OCTEON.

Tool	Description
oct-linux-csr	OCTEON Control and Status Registers (CSRs) may be read and written using this utility.
oct-linux-identify	Query Board about model, size of L2 cache, hardware units supported, etc.
oct-linux-jtg	This debugging tool should not be used unless requested by a Cavium Networks representative. The output is not easy to read. This tool is not usually used.
oct-linux-mdio	Read/write on the MDIO bus (typically to Ethernet PHYs and switches).
oct-linux-memory	Read or write DDR memory on the OCTEON.
oct-linux-pow	Display OCTEON POW (SSO) state. This debugging tool should not be used unless requested by a Cavium Networks representative. The output is not easy to read. This tool is not usually used.
oct-linux-profile	Profiling of any code running on OCTEON: both Simple Executive and Linux userspace applications.

Continued in the next table...

Table 11: Special Cavium Networks Native Tools, Part 2

The following native tools are available when Linux is running on OCTEON and OCTEON is a PCI host, connected to an OCTEON PCI target.

Tool	Description
oct-pci-boot	Boot the OCTEON PCI target board. This program configures the DDR controller on the OCTEON, then loads and runs the u-boot bootloader on it. See Note 1.
oct-pci-bootcmd	Used to run bootloader commands over PCI to an OCTEON PCI target board. Used to run image files on the cores. This command may be used to start the application which was downloaded via the oct-pci-load command. See Note 1.
oct-pci-csr	OCTEON CSR (Control and Status Registers) may be read and written directly over PCI using this utility. See Note 1.
oct-pci-ddr	Tests memory after it is set up. This is a quick "alive" test. Use hw-ddr2 for more extensive memory testing.
oct-pci-load	Load a file into OCTEON memory over PCI from the host. See Note 1.
oct-pci-memory	Read or write DDR memory on the OCTEON target over PCI.
oct-pci-profile	Profiling of any code running on OCTEON: both Simple Executive and Linux userspace applications. This profiler runs over PCI.
oct-pci-reset	Reset the OCTEON PCI target board. See Note 1.
oct-pci-tra	Access the trace buffer over Poise the Hardware Reference Manual for more information on trace functions. This is somewhat complicated to use.
Notes	
<p>Each note references an SDK Document where more information can be found. (view \$OCTEON_ROOT/docs/html/index.html in a browser)</p> <p>Note 1: <i>Developing with OCTEON as a PCI Target</i></p>	

Note: The *oct-pci-** tools are used when the OCTEON processor is a PCI host (instead of a PCI target). Not all boards support this configuration.

An example of tools which run on the target is `oct-linux-identify`. Once Linux has been booted on the target, in the target console, type `oct-linux-identify`:

```
target# oct-linux-identify
Model: CN3860p2.X-500-NSP
Level 2 cache: 1024 KB
Number of cores: 16
Crypto: Yes
Extended Multiplier: Yes
Low Latency Memory: Yes
Kasumi: No
RAID: No
DFA: Yes
ZIP: Yes
PCI BAR2: No
```

9.6.1 Linux Tools: Debian Filesystem Native Tools

Both the Debian compiler and the Cavium Networks compiler are supplied with Debian, however only the Cavium Networks compiler should be used to build SE-UML applications on the target.

When running the Debian filesystem on an OCTEON core, the Cavium Networks tool chain is located in `/usr/local/Cavium_Networks/OCTEON-SDK/tools/usr/bin` on the development target. The compiler is named `gcc` (without the `mips64-octeon-linux-gnu-*` prefix). The `PATH` environment variable needs to be modified to use these tools before those supplied with Debian:

```
target# PATH=/usr/local/Cavium_Networks/OCTEON-SDK/tools/usr/bin:$PATH
```

Use the command `gcc -v` to see if the `PATH` variable was set up correctly. If it is correct, the `gcc` version will include the string “Cavium Networks Version”.

The Debian tool chain may be used on the target to compile o32 applications. Note that these applications cannot use the Cavium Networks-specific instructions or the Simple Executive API functions.

Note: The Debian binaries are standard Debian files. Cavium Networks does not alter these files, or rebuild them. Cavium Networks is not responsible for these files.

See the SDK document “*Running Debian GNU/Linux on OCTEON*” for more information.

9.7 Example Applications

Example applications are provided in the `examples` directory. Each of these is a Simple Executive application.

Simple Executive standalone (SE-S) applications are self-contained: they do not require an operating system to run. The ELF file may be booted on either one core or on a load set.

Simple Executive User-Mode (SE-UM) applications are compiled to run under Linux. Usually these files are executable files in the Linux filesystem. After Linux is booted, the SE-UM application is started using the `oncpu` command. Linux can be booted from an embedded root filesystem ELF image (`vmlinux.64`) or Linux can be booted from a flash card running Debian Linux. The `vmlinux.64` ELF image already includes the example programs, compiled to run under Linux. Once this image is booted, the examples are ready to use. If Debian is used, the examples need to be copied to the Debian filesystem. In this tutorial, the simpler embedded root filesystem is used.

The following example programs are included with SDK 1.8.0. Each example is described in its `README.txt` or `README-Linux.txt` file. All examples can be run as SE-S applications, and most can be run as SE-UM applications under Linux. If the example application cannot be run under Linux, the Makefile will issue the message: "This example doesn't support Linux". In this chapter, the examples `hello` and `linux-filter` will be built and run (see table, below).

Table 12: Examples Provided with SDK 1.8.0

Example	Can Run as Linux SE-UM	Hardware Support Required	Multicore Example?	Brief Description
application-args	NO	-	NO	How to pass arguments to Simple Executive applications.
crypto	YES	Crypto	YES	Crypto hardware acceleration example.
debugger	NO	-	YES	Sample program for evaluating the debugger.
hello	NO	-	YES	Simple "hello world" application.
linux-filter	YES	-	YES	Shows how to pass packets between cores running Linux and cores running a Stand-alone Simple Executive application.
low-latency-mem	YES	LLM, DFA	YES	Low Latency Memory (LLM) usage example.
mailbox	NO	-	YES	Shows how to use mailbox interrupts.
named-block	YES	-	NO	Shows how to use bootmem allocation with named blocks.
passthrough	YES	-	YES	Packet passthrough example using packet I/O.
queue	YES	-	YES	Implements a memory-access-efficient message queue.
traffic-gen	NO	-	YES	A network traffic generator.
uart	NO	UART	YES	Simple example usage of the UART.
zip	YES	ZIP	YES	Example showing how to use the ZIP Unit.

10 About Building Example Applications

This section is an overview of building example applications. Building Linux will be explained in Section 16 – “About Building Linux”.

Even if the reader is only interested in running applications under Linux, it is best to follow the steps in the order presented here. Each section depends on material presented in the preceding section. As an example, running the SE-S application `hello` is a perfect way to test the hardware configuration without the added complexity of Linux, and introduces the reader to downloading and running software on the OCTEON processor.

The steps taken will include:

- Section 11 – “Hands-on: Build and Run a SE-S Application (`hello`)”, a SE-S application will be built and run.
- Section 17 – “Hands-on: Build and Run Linux”, Linux will be built and run.
- Section 18 – “Hands-on: Run a SE-UM Example”, a SE-UM application will be built and run.
- Section 20 – “Hands-on: Run `linux-filter` as a SE-S Application (Hybrid System)”, Linux will be run as a hybrid system with an SE-S application
- Section 21 – “Hands-on: Run `linux-filter` as a Linux SE-UM Application”, Linux will be run with a SE-UM application.

Each step builds on knowledge gained in the previous step. Explanations are provided before and after the steps.

10.1 Makefiles

Both applications and Linux are created by using Makefiles. These are typically files named `makefile`, `Makefile`, or `*.mk` (as in `cvmx.mk`, or `application.mk`).

A Makefile can include other Makefiles. In the examples directory, each Makefile automatically include several other files: `common.mk`, `application.mk`, and `cvmx.mk`:

```
include $(OCTEON_ROOT)/common.mk
include $(dir)/cvmx.mk
include $(OCTEON_ROOT)/application.mk
```

(The value of `$(dir)` is `$(OCTEON_ROOT)/executive`.)

And these Makefiles may include other Makefiles: `$(OCTEON_ROOT)/common.mk` includes `common-config.mk`:

```
include $(OCTEON_ROOT)/common-config.mk
```

When `linux-filter` is built, first the Simple Executive code located in the `$OCTEON_ROOT/executive` directory is built, and the archive program (`ar`) packages the Simple Executive object files into one library: `libcvmx.a`. The object files and library are put in the `linux-filter/obj` directory:

```
mipsisa64-octeon-elf-ar -cr obj/libcvmx.a <more text omitted>
```

The `linux-filter` is then linked with `libcvmx.a`:

```
mipsisa64-octeon-elf-gcc obj/linux-filter.o -L/home/testname/sdk/target/lib  
obj/libcvmx.a -o linux-filter
```

10.2 Makefile Targets for Example Code

When looking at the example code, there are two Makefile targets possible for SE-S applications, and four possible for SE-UM applications. The target is selected by setting `OCTEON_TARGET` equal to the desired target on the `make` command line.

SE-S Targets:

- `OCTEON_TARGET=cvmx_64` – the default Simple Executive target (standalone)
- `OCTEON_TARGET=cvmx_n32` – the Simple Executive n32 target (not discussed in this chapter)

SE-UM Targets:

- `OCTEON_TARGET=linux_64` – the default Linux target
- `OCTEON_TARGET=linux_n32` – the Linux n32 target (not discussed in this chapter)
- `OCTEON_TARGET=linux_uclibc` – the application is built with the 32-bit uclibc library (saves memory). This is not discussed further in this chapter.
- `OCTEON_TARGET=linux_o32` – not recommended. This API will not allow the user to use Cavium Networks-specific instructions or the Simple Executive API. For that reason, it will not be discussed further in this chapter.

Each built target ELF file has a unique name, as shown in the following table.

Table 13: Different Makefile Targets, Different Target Names

OCTEON_TARGET	Built Example Target Name	Description
cvmx_64	linux-filter	the default Simple Executive target EABI (SE-S)
cvmx_n32	linux-filter-cvmx_n32	the Simple Executive N32 target (not discussed in this chapter) (SE-S)
linux_64	linux-filter-linux_64	the default Linux target: Linux N64 (SE-UM)
linux_n32	linux-filter-linux_n32	Linux N32 application (not discussed in this chapter) (SE-UM)
linux_uclibc	linux-filter-linux_uclibc	Linux N32 application built with smaller uclibc to save memory (SE-UM)

Each target is built with the object files in a separate directory, so multiple targets can share the same source directory:

```
host$ cd examples/linux-filter
host$ ls -CF
config/           linux-filter-linux_n32      obj-linux_n32/
linux-filter       linux-filter-linux_uclibc   obj-linux_uclibc/
linux-filter.c     Makefile                   README.txt
linux-filter-cvmx_n32 obj/                     u-boot-env
linux-filter.input obj-cvmx_n32/
linux-filter-linux_64 obj-linux_64/
```

10.3 Building SE-S Examples

An example of how to build an SE-S example may be seen by going to the examples/linux-filter directory, and typing one of:

- make
- make OCTEON_TARGET=cvmx_64
- make OCTEON_TARGET=cvmx_n32

See Section 8.3 – “Making a Copy of the Installed SDK” if the following error occurs:

```
<text omitted> mipsisa64-octeon-elf/bin/ld: cannot open output file
hello: Permission denied
collect2: ld returned 1 exit status
make: *** [hello] Error 1
```

Notice that running the command `make` with no arguments builds the application identically to the option `cvmx_64`. The target is named `linux-filter`. When building with the option `OCTEON_TARGET=cvmx_n32`, the target file is `linux-filter-cvmx_n32`. The object files are put in separate directories: `obj`, and `obj-cvmx_n32`.

In this tutorial, the SE-S target will be `cvmx_64`.

10.4 Building SE-UM Examples

An example of how to build a SE-UM application may be seen by going to the \$OCTEON_ROOT/examples/linux-filter directory, and typing one of:

- make OCTEON_TARGET=linux_64
- make OCTEON_TARGET=linux_n32

The target is named either linux-filter-linux_64 or linux-filter-linux_n32, depending on which target was specified on the make command line. Object files are put in separate directories: obj_linux_64, and obj_linux_n32.

In this tutorial, the SE-UM target will be linux_64.

The next step is to put the object file into a filesystem, accessible once the Linux kernel is booted. The details are discussed in Section 23.1 – “Adding Applications to the Embedded Root Filesystem”.

10.5 Saving make Output

To execute the command, saving the output for later reference (in bash), type:

```
host$ make OCTEON_TARGET=cvmx_64 > make.out 2>&1 &
```

This will redirect the standard output (stdout) and standard error output (stderr) to the file named make.out. The “&” symbol at the end of the command line puts the job into the background (a prompt will occur on the screen, allowing you to type the next command).

The contents of the make.out file can be viewed by using the commands cat, head, tail, or vi. During a long make, it is customary to use the tail -f command which will display information as it is added to the file.

To move the make command from the background to the foreground, type fg.

10.6 Other Makefile Targets

Most Makefiles also support the clean and clobber targets.

To remove the executive .o files and target file to start fresh, type:

```
host$ make clean OCTEON_TARGET=<your target> # specify OCTEON target
```

For the default (cvmx_64), simply type:

```
host$ make clean
```

The `clean` target is used to remove all output of the build, and allow a fresh beginning.

```
clean:
    rm -f $(TARGET) config/cvmx-config.h
    rm -fr $(OBJ_DIR)
```

The `clean` command and the `clobber` command are not always used in the same way. For example, in the passthrough example, `clean` is used to clean up after a test run, and `clobber` is used to remove the `.o` files created in the prior build. Check the Makefile to be sure what exact action will be performed.

From passthrough/Makefile:

```
clean:
    rm -f $(TARGET) output.log input-*.*.data output-*.*.data run-all.log
    rm -f $(CLEAN_LIST)
    rm -f $(CVMX_CONFIG)

clobber: clean
    rm -rf $(OBJ_DIR)
```

10.7 Using the `strip` Utility

The `strip` utility is used to remove debugging information from the ELF file. This step is usually done after debugging is complete. The result is a smaller file, which downloads and boots faster. `Strip` is used before putting the file into the onboard flash.

An example of the `strip` utility is seen in the following command line:

```
host$ mv hello hello_unstripped # save the unstripped version of hello
host$ mipsisa64-octeon-elf-strip -o hello hello_unstripped
```

Verify the new executable is smaller:

```
host$ ls -l hello hello_unstripped
-rwxr-xr-x 1 testname software 64164 Jan 26 16:00 hello
-rwxr-xr-x 1 testname software 315199 Jan  9 18:26 hello_unstripped
```

In the SE-S examples shown in this chapter, `strip` will not be used to simplify the instructions.

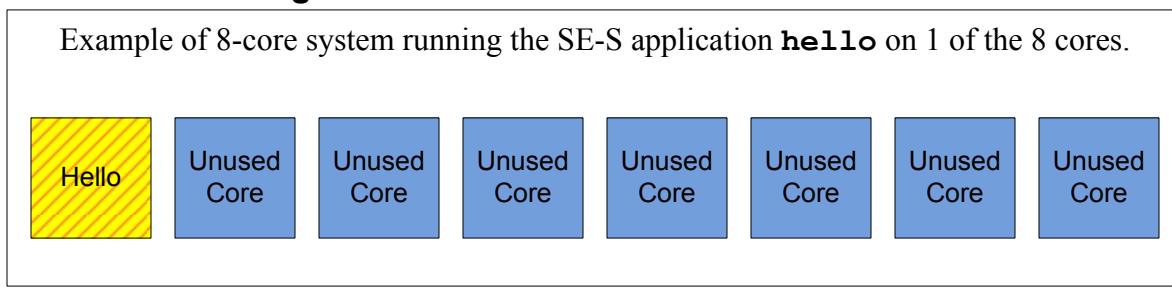
11 Hands-on: Build and Run a SE-S Application (`hello`)

In this section a simple SE-S application will be built, downloaded, and booted on the OCTEON processor. Although the specific directions are for an SE-S application, only the build and boot commands are different between SE-S and SE-UM applications.

It is important to read this section and follow the steps before going on to build, download, and boot Linux and run a SE-UM example. The directions which are common to both types of application will not be repeated.

The simplest application to build, download, and run is `hello`. The `hello` example is a Simple Executive 64-bit standalone application (SE-S). This example does not support Linux (there is no SE-UM `hello` application available).

Figure 8: Run `hello` on 1 out of 8 Cores



There are several key steps from power-on or reset to running the application:

1. Download and run the bootloader (if it is not already in flash)
2. Download the application ELF file (if it is not already in flash)
3. Boot the application: The bootloader reads the ELF file, loads the application into memory, and runs it.

The board must be reset manually after the application has run to completion.

Currently, there is no support for returning to the bootloader once the program exits, so the board must be reset after the application has completed. The user must manually reset the board:

- For standalone development target boards, use either the manual push-button reset or power cycling the board.
- For PCI development target boards, the development target must be reset from the development host, using either `oct-pciboot` or `oct-pci-reset`. For PCI target boards, do not use the manual push-button reset.

As an alternative to manually resetting the board, the application can call `cvmx_reset_octeon()` to reboot the board. Note that this will reboot all cores on the board.

Note that the ELF file is only stored in memory, so after a reset it will no longer be in memory and must be downloaded again.

The directions for PCI development targets are more streamlined than for standalone development target because all the commands may be typed on the host, `tftpboot` is not needed, and the board may be reset from the host.

The following tables summarize the directions for a first a PCI development target, then a standalone development target. Select the directions which match your board type:

- To run `hello` on a *standalone* development target, go to Section 11.2 – “Run `hello` on a Standalone Target Board”.
- To run `hello` on a *PCI* development target, see the next section.

Detailed directions follow the tables.

11.1 Run `hello` on a PCI Target Board

When using a PCI development target, special `oct-pci-*` commands are provided to use the PCI bus to simplify application development.

Table 14: Key `oct-pci-*` Commands

Tool	Description
<code>oct-pci-boot</code>	Reset the OCTEON PCI target, download the bootloader file to memory, and boot the OCTEON PCI target board. The board must be configured for PCI boot, not flash boot.
<code>oct-pci-bootcmd</code>	Used to run bootloader commands over PCI to an OCTEON PCI target board. This command may be used to start the application which was downloaded via the <code>oct-pci-load</code> command.
<code>oct-pci-console</code>	Used to allow the bootloader, Simple Executive applications, and Linux to redirect their console output over the PCI bus to the <code>oct-pci-console</code> utility. This allows running interactive programs on OCTEON without a serial connection.
<code>oct-pci-load</code>	Download a file to OCTEON memory over the PCI bus.
<code>oct-pci-reset</code>	Reset the OCTEON PCI target board. The board should be configured for flash boot, not PCI boot.

For more `oct-pci-*` commands, see Table 6 – “Host Tools” and Table 7 – “Host Tools, continued”.

See the SDK document “*Developing with OCTEON as a PCI Target*” for more information.

The following tables summarize the directions for a PCI development target. Detailed instructions follow the tables. Steps 1-5 are identical for PCI and standalone target boards.

Warning: If an OCTEON PCI development target board is powered off or reset via the manual reset switch, the host/target communication may become disrupted. The result will be that configuration cycles and CSR access will continue to work, but bulk data transfer will fail in strange and generally unpredictable ways. This problem occurs

because the development target has lost essential data in the PCI block. Instead of powering off or using the manual reset switch, use either `oct-pci-reset` or `oct-pci-boot` to reset an OCTEON PCI development target. These commands do not disturb the data in the PCI block.

Table 15: Run `hello` on a PCI Development Target, Part 1

Steps	Note
1. Connect the Hardware	
Connect serial cable to target console. Connect the Ethernet cable.	Follow directions in the <i>Quick Start Guide</i> . Note: The Ethernet cable is not needed to run <code>hello</code> on a PCI target board, but will be needed later in the <i>SDK Tutorial</i> . Be careful to isolate the test network from the office network so that experiments will not disturb the office network.
2. Reset the board	
<code>host\$ oct-pci-reset</code>	The board should have been configured to boot from flash, so <code>oct-pci-boot</code> is not needed. The word Boot should appear on the red LEDs on the board. If not, the board is not configured to boot from flash, or something is wrong with the board.
3. Connect to the Target Console	
<code>host\$ minicom -w ttys0</code>	Substitute the serial port actually used on the host to connect to the OCTEON target board if it is not <code>ttys0</code> . Minicom will provide a connection to the target console. You should see the bootloader prompt.
4. Verify Bootloader Prompt is Visible	
<code>target# version</code>	The bootloader should reply with text similar to: U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun 13)
5. Verify Bootloader Version is at Least SDK 1.7	
If the bootloader's SDK version is not at least 1.7, then before continuing, upgrade the bootloader to a newer version.	Directions for upgrading the bootloader are included in the <i>SDK Tutorial</i> .
6. Build the Application	
<code>host\$ cd \$OCTEON_ROOT/examples/hello</code> <code>host\$ make clean</code> <code>host\$ make</code>	The <code>make</code> command will create the executable file <code>hello</code> .
<i>Continued in the next table...</i>	

Table 16: Run hello on a PCI Development Target, Part 2

Steps	Note
7. Download the Application to the Development Target	
host\$ oct-pci-load 0 hello	Expect to see text similar to: Found Octeon on bus 3 in slot 13. BAR0=0xd8000000, BAR1=0xd0000000
8. Boot the Application	# type the following command on one line host\$ oct-pci-bootcmd "bootoct 0 coremask=0x1" The bootoct command will run hello on core 0. Expect to see (on target console): PP0:~CONSOLE-> Hello world! PP0:~CONSOLE-> Hello example run successfully. Note: PP0 is "Packet Processor 0" (core 0). The coremask must include core 0 (bit 0x1 in the coremask). All cores are held in reset until an application is loaded on core 0. Setting the coremask to 0xF will run hello on 4 cores.
9. Reset the Target Board	host\$ oct-pci-reset Note: To run hello a second time, the board must be reset because there is no other way to return to the bootloader prompt. Use oct-pci-reset to reset the board, not the manual reset switch. Then download the application again. The application cannot be re-run without downloading it again because it is no longer in memory on the development target.

11.1.1 Connect the Hardware

This step is covered in Section 5 – “Hands-on: Connect the Development Target”.

11.1.2 Reset the Target Board

Use the command `oct-pci-reset` to reset the board. The board should have been configured to boot from flash. Note that this step is not necessary if the board was just powered on or reset, but the command only takes a second to complete. Do not reset the board using the push-button reset.

11.1.3 Connect to the Target Console

This step is covered in Section 6 – “Hands-on: Viewing the Target Board Console Output”.

11.1.4 Verify Bootloader Prompt is Visible

This step is covered in Section 6 – “Hands-on: Viewing the Target Board Console Output”.

If the bootloader is located in flash, and the hardware is configured to boot from flash, then the bootloader will run when the development target is powered on.

If the bootloader is not in flash then the bootloader must be downloaded before it can be run. In order to simplify the directions in this chapter, the development target should be configured to boot from flash. If the main bootloader is not usable, it can be restored from the failsafe bootloader.

See Section 13.3 – “The Failsafe Bootloader” for more information. If necessary, the board may be booted over PCI. See Section 11.1.9.2 – “Reset if booting over PCI”.

11.1.5 Verify Bootloader is at Least SDK 1.7

This step is covered in Section 6 – “Hands-on: Viewing the Target Board Console Output”.

11.1.6 Build hello

To build the example hello:

```
host$ cd $OCTEON_ROOT/examples/hello
host$ make OCTEON_TARGET=cvmx_64 > make.out 2>&1 &
host$ cat make.out
```

You should see output similar to:

```
mipsisa64-octeon-elf-gcc -o hello -g -O2 -W -Wall -Wno-unused-parameter
-I/home/testname/sdk/target/include -I/home/testname/sdk/target/include
-Iconfig -DUSE_RUNTIME_MODEL_CHECKS=1 -DCVMX_ENABLE_PARAMETER_CHECKING=0
-DCVMX_ENABLE_CSR_ADDRESS_CHECKING=0 -DCVMX_ENABLE_POW_CHECKS=0
-DOCTEON_MODEL=OCTEON_CN38XX -DOCTEON_TARGET=cvmx_64 hello.c
```

The file hello should now be in the examples/hello directory.

11.1.7 Download the Application to the Development Target

Use the command oct-pci-load to download the application to the Reserved Download Block.

For SDK 1.7 and higher:

```
host$ oct-pci-load 0 hello
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000, BAR1=0xd0000000
```

11.1.8 Boot the Application

Use the oct-pci-bootcmd to boot hello:

```
host$ oct-pci-bootcmd "bootoct 0 coremask=0x1"
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000, BAR1=0xd0000000
```

You should see:

```
bootloader: Booting Octeon Executive application at 0x20000000,
core mask: 0x1,
stack size: 0x100000, heap size: 0x300000
Bootloader: Done loading app on coremask: 0x1
PP0:~CONSOLE->
PP0:~CONSOLE->
PP0:~CONSOLE-> Hello world!
PP0:~CONSOLE-> Hello example run successfully.
```

11.1.9 Reset the Target Board

The board must be reset manually after the application has completed.

Currently, there is no support for returning to the bootloader once the program exits. The user must manually reset the board using either the oct-pci-reset command or the oct-pci-

boot command. Do not use the manual push-button reset. (The `oct-pci-boot` command is used if the board is not configured to boot from flash.)

Note that the ELF file is only stored in memory, so after a reset it will no longer be in memory and must be downloaded again.

11.1.9.1 Reset if booting from flash

A PCI target board may be reset using the command `oct-pci-reset` if it is configured to boot from flash.

11.1.9.2 Reset if booting over PCI

If booting over PCI from the PCI host, `oct-pci-boot` will reset the board, download the bootloader from the PCI host and boot the bootloader. Note that the command will automatically figure out the correct bootloader file to use.

```
host$ oct-pci-boot
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000, BAR1=0xd0000000
Using bootloader image:
/home/testname/sdk/target/bin/u-boot-octeon_ebt3000_pciboot.bin
```

11.1.10 Multiple OCTEON PCI Target Boards

By default, utilities operate on the first OCTEON target board found on the PCI bus. For example, `oct-pci-reset` resets the first OCTEON target board found on the PCI bus.

If there are multiple OCTEON PCI targets attached to the same PCI host, the environment variable `OCTEON_PCI_DEVICE` can be set to select which OCTEON target should be accessed. For example, if `OCTEON_PCI_DEVICE=1`, the second OCTEON target is reset. The command `/sbin/lspci` enumerates the PCI targets.

Skip Section 11.2 – “Run `hello` on a Standalone Target Board” and resume reading at Section 6.9 – “Determining the Number of Cores on the OCTEON”.

11.2 Run `hello` on a Standalone Target Board

The following tables summarize the directions for a standalone development target. Detailed instructions follow the tables. Steps 1-5 (shown in the table containing the instructions) are identical for PCI and standalone target boards.

The key differences between running an application on a standalone versus PCI board are in the following steps:

1. The application will be downloaded over Ethernet from the development host to the development target using the `tftpboot` utility. Note: `tftpboot` details are located in Section 34 – “Appendix J: TFTP Boot Assistance (`tftpboot`)”.
2. After it is downloaded, the application is booted by typing the `bootoct` command in the target console.
3. To reboot the board, push the reset switch on the board.

Table 17: Run hello on a Standalone Development Target, Part 1

Steps	Note
1. Connect the Hardware	
Connect serial cable to target console. Connect the Ethernet cable.	Follow directions in the <i>Quick Start Guide</i> . Note: The Ethernet cable is not needed to run <code>hello</code> on a PCI target board, but will be needed later in the <i>SDK Tutorial</i> . Be careful to isolate the test network from the office network so that experiments will not disturb the office network.
2. Reset the board	
Power on or reset the target board.	The word <code>Boot</code> should appear on the red LEDs on the board. If not, the board is not configured to boot from flash, or something is wrong with the board.
3. Connect to the Target Console	
host\$ <code>minicom -w ttys0</code>	Substitute the serial port actually used on the host to connect to the OCTEON target board if it is not <code>ttys0</code> . Minicom will provide a connection to the target console. You should see the bootloader prompt.
4. Verify Bootloader Prompt is Visible	
target# <code>version</code>	The bootloader should reply with text similar to: U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun 13)
5. Verify Bootloader Version is at Least SDK 1.7	
If the bootloader's SDK version is not at least 1.7, then before continuing, upgrade the bootloader to a newer version.	Directions for upgrading the bootloader are included in the <i>SDK Tutorial</i> .
6. Build the Application	
host\$ <code>cd \$OCTEON_ROOT/examples/hello</code> host\$ <code>make clean</code> host\$ <code>make</code>	The <code>make</code> command will create the executable file <code>hello</code> .
<i>Continued in the next table...</i>	

Table 18: Run hello on a Standalone Development Target, Part 2

Steps	Note
7. Copy the ELF file to the tftpboot Directory	
host\$ sudo cp hello /tftpboot	See <i>SDK Tutorial</i> directions for tftpboot.
8. Select Target IP Address, if Needed	
If a DHCP server is available, then selecting the target IP address is handled by the server. Otherwise, select a target IP address.	In this example, the target IP address is 192.168.51.159 .
9. Set the Development Target's IP Address	
9a. No DHCP Server	
First, set the IP address of the target (replace items in italic with your IP addresses).	Note: serverip is the IP address of the TFTP server.
<i># Use your IP addresses instead of the example values!</i> target# setenv gatewayip 192.168.51.254 target# setenv netmask 255.255.255.0 target# setenv ipaddr 192.168.51.159 target# setenv serverip 192.168.51.1 <i># save the values so they will still be set after a reset</i> target# saveenv	
9b. DHCP Server Available	
If a DHCP server is available substitute the following step:	
target# dhcp	
10. Test the Ethernet Connection to the Host	
<i># use your host IP address instead of the example value!</i> target# ping 192.168.51.254	Expect to see: Using octeth0 device host 192.168.51.254 is alive Note that the development target will <i>not</i> reply to a ping from the development host. Note: to see the development host's IP address, use the /sbin/ifconfig command on the development host.
<i>Continued in the next table...</i>	

Table 19: Run hello on a Standalone Development Target, Part 3

Steps	Note
11. Download the Application to the Development Target	
# Use tftpboot to download the application. target# tftpboot 0 hello	See <i>SDK Tutorial</i> directions for tftpboot. If this step does not work, check the /etc/xinetd.d/tftp file on the host to verify that server_args = -s /tftpboot.
12. Boot the Application	
target# bootoct 0 coremask=0x1	This command will run hello on core 0. Expect to see: PP0:~CONSOLE-> Hello world! PP0:~CONSOLE-> Hello example run successfully. Note: PP0 is "Packet Processor 0" (core 0). The coremask must include core 0 (bit 0x1 in the coremask). All cores are held in reset until an application is loaded on core 0. Setting the coremask to 0xF will run hello on 4 cores.
13. Reset the Target Board	
To reset the development target, use the the target's reset switch.	To run hello a second time, the board must be reset or power cycled because there is no other way to return to the bootloader prompt. Then download the application again. The application cannot be re-run without downloading it again because it is no longer in memory on the development target.

11.2.1 Connect the Hardware

This step is covered in Section 5 – “Hands-on: Connect the Development Target”.

11.2.2 Reset the Development Target

A standalone target board may be either reset using the reset button, or power cycled. In most cases, the manual push-button should be adequate to reset the board.

11.2.3 Connect to the Target Console

This step is covered in Section 6 – “Hands-on: Viewing the Target Board Console Output”.

11.2.4 Verify Bootloader Prompt is Visible

This step is covered in Section 6 – “Hands-on: Viewing the Target Board Console Output”.

If the bootloader is located in flash, and the hardware is configured to boot from flash, then the bootloader will run when the development target is powered on.

If the bootloader is not in flash then the bootloader must be downloaded before it can be run. If it is a standalone board, then the bootloader must be present in flash. If the main bootloader is not

usable, it can be restored from the failsafe bootloader. See Section 13.3 – “The Failsafe Bootloader” for more information.

11.2.5 Verify Bootloader is at Least SDK 1.7

This step is covered in Section 6 – “Hands-on: Viewing the Target Board Console Output”.

11.2.6 Build hello

To build the example `hello`:

```
host$ cd $OCTEON_ROOT/examples/hello
host$ make OCTEON_TARGET=cvmx_64 > make.out 2>&1 &
host$ cat make.out
```

You should see output similar to:

```
mipsisa64-octeon-elf-gcc -o hello -g -O2 -W -Wall -Wno-unused-parameter
-I/home/testname/sdk/target/include -I/home/testname/sdk/target/include
-Iconfig -DUSE_RUNTIME_MODEL_CHECKS=1 -DCVMX_ENABLE_PARAMETER_CHECKING=0
-DCVMX_ENABLE_CSR_ADDRESS_CHECKING=0 -DCVMX_ENABLE_POW_CHECKS=0
-DOCTEON_MODEL=OCTEON_CN38XX -DOCTEON_TARGET=cvmx_64 hello.c
```

The file `hello` should now be in the `examples/hello` directory.

11.2.7 Copy Application to the /tftpboot directory

Copy the example application `hello` to the `/tftpboot` folder. This folder should have been created when the `tftp-server` RPM was installed. (Note that on some systems the default `tftpboot` directory is `/var/lib/tftpboot`. The exact directory can be determined by looking at the `server_args` value in the TFTP startup file.)

```
host$ cd $OCTEON_ROOT/examples/hello
host$ sudo cp hello /tftpboot
```

Note that the folder is only writable by the owner, `root`:

```
host$ cd /tftpboot
host$ ls -ld .
drwxr-xr-x 2 root root 4096 Feb 17 2004 .
```

11.2.8 Select Target IP Address, if Needed

If a DHCP server is used, the target IP address will be automatically assigned when the `dhcpc` command is executed. Otherwise, select an IP address for use in the next step.

11.2.9 Set the Development Target's IP Address

Before starting `tftpboot` to download the application, set the IP address of the target, and the IP address of the TFTP server (`serverip`).

11.2.9.1 Using `dhcp` to Set the Target IP Address

If possible, use `dhcp` (Dynamic Host Configuration Protocol) to configure the IP address of the OCTEON target.

In the target console, connected to the serial port on the OCTEON target, type `dhcp`. The following text shows an example of the expected reply from the bootloader:

```
target# dhcp
BOOTP broadcast 1
octeth0: Up 1000 Mbps Full duplex (port 16)
DHCP client bound to address 192.168.51.193
```

11.2.9.2 Setting the Target IP Address without a `dhcp` server

If a DHCP Server is not available, then set a static IP address using bootloader commands:

```
target# setenv gatewayip 192.168.51.254      # use your gateway IP address!
target# setenv netmask 255.255.255.0
target# setenv ipaddr 192.168.51.159          # use your target IP address!
target# setenv serverip 192.168.51.1           # use your server IP address!
```

11.2.9.3 Confirm the IP Addresses are Correct

Use `printenv` to confirm:

```
target# printenv
gatewayip=192.168.51.254
netmask=255.255.255.0
ipaddr=192.168.51.159
serverip=192.168.51.1
```

11.2.10 Test the Ethernet Connection to the Development Host

Note that the development target will *not* reply to a ping from the development host. To test the connection, a ping can be sent from the development target to the development host (in this example, the development host's IP address is *192.168.51.254*). Note: to see the development host's IP address, use the `/sbin/ifconfig` command on the development host.

```
target# ping 192.168.51.254      # use your host IP address HERE!
Using octeth0 device
host 192.168.51.254 is alive
```

11.2.11 Download the Application to the Development Target

Before beginning this step, you will need a bootloader prompt, and a physical Ethernet connection to the development host. The application to be downloaded should be available on the development host in the directory configured for `tftpboot` to use, such as `/tftpboot`. Then use the `tftpboot` command in the target console to download the application.

On a bootloader from SDK 1.7 or higher:

```
target# tftpboot 0 hello

For example:
target# tftpboot 0 hello
Using octeth0 device
TFTP from server 192.168.51.254; our IP address is 192.168.51.186
Filename 'testname/dl/hello'.
Load address: 0x20000000
Loading: #### <<< success! # characters will appear showing downloading
done
Bytes transferred = 317852 (4d99c hex), 8868 Kbytes/sec
```

Note that the bootloader will provide basic help on the tftpboot command:

```
target# help tftpboot
tftpboot [loadAddress] [bootfilename]
If loadAddress is 0, then file will be loaded to the default load address
```

Note: The name of the application to run, bootfilename, can be an absolute path name, or a pathname relative to the tftpboot directory. For example, if the tftpboot directory is /home and file is in /home/testname/dl/hello, the file may be booted by using either the full or relative path name. For example, either testname/dl/hello or /home/testname/dl/hello can be used to tftpboot the file.

11.2.11.1 Common tftpboot Errors

11.2.11.1.1 dhcp Step Forgotten

Example output if dhcp step has not yet been done:

```
target# tftpboot testname/dl/hello
Interface 1 has 4 ports (RGMII)
*** ERROR: `serverip' not set
```

If this output occurs, then the dhcp step was omitted.

11.2.11.1.2 ELF Image File Not Found on TFTP Server

When the ELF file is not found on the TFTP server, tftpboot will continue to try to download the file, printing the character “T” on the target console, until the user types reset or Ctrl-C. Also, when the file is not found, tftpboot prints a “File not found” error. This error is hidden in the output, as shown in the following example:

```

target# tftpboot 0 badfilename
Using octeth0 device
TFTP from server 192.168.51.254; our IP address is 192.168.51.184
Filename 'badfilename'.
Load address: 0x20000000
Loading: *
TFTP error: 'File not found' (1) <<< ** difficult to see the error! **
Starting again

Using octeth1 device
TFTP from server 192.168.51.254; our IP address is 192.168.51.184
Filename 'badfilename'.
Load address: 0x20000000
Loading: octeth1: Down (port 17)
T T T T

```

Try again with the correct file name:

```

target# tftpboot 0 hello
Using octeth0 device
TFTP from server 192.168.16.41; our IP address is 192.168.16.61
Filename 'hello'.
Load address: 0x20000000
Loading: ### <<< success! # characters will appear showing downloading
done
Bytes transferred = 315148 (4cf0c hex), 12823 Kbytes/sec

```

11.2.11.2 Ethernet Cable Plugged into Wrong Ethernet Port on Target

The default Ethernet port used for tftpboot on the target can be seen with the `printenv` bootloader command. The variable `ethact=octeth0`. If the Ethernet cable is not plugged into Port 0, then either change the cable to Port 0 or change the value of environment variable to refer to the correct Ethernet port.

11.2.12 Boot the Application

From the target console, type `bootoctl` for Simple Executive Applications. For example, to run Simple Executive on the first core:

```

target# bootoctl 0 coremask=0x1
Bootloader: Booting Octeon Executive application at 0x20000000, core
mask: 0x1,0
Bootloader: Done loading app on coremask: 0x1
PP0:~CONSOLE->
PP0:~CONSOLE->
PP0:~CONSOLE-> Hello world!
PP0:~CONSOLE-> Hello example run successfully.

```

Note that after running the application, the program halts, and does not return to the bootloader. There is currently no support for returning to the bootloader.

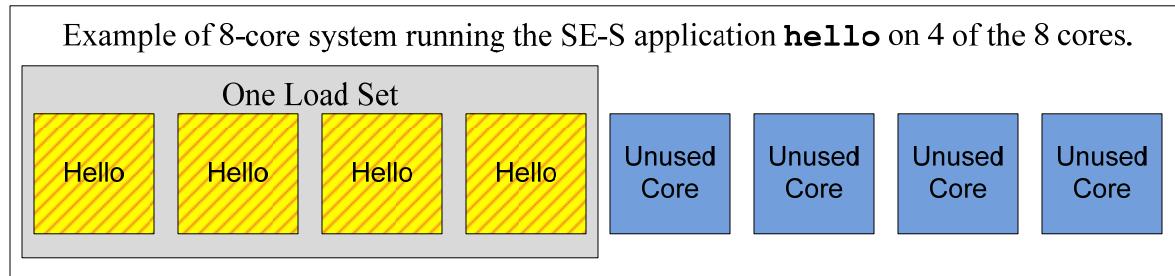
11.2.13 Reset the Development Target

A standalone target board may be either reset using the reset button, or power cycled. In most cases, the manual push-button should be adequate to reset the board.

12 Hands-on: Run `hello` on Multiple Cores

The following figure shows `hello` running on a load set of 4 cores out of a total of 8 cores.

Figure 9: Run `hello` on 4 out of 8 Cores



To boot `hello` on multiple cores:

- **PCI Development Target Command:**

```
# Adjust the coremask to not exceed the number of
# cores available on your OCTEON model.
host$ oct-pci-bootcmd "bootoct 0 coremask=0xF"
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000, BAR1=0xd0000000
```

- **Standalone Development Target Command:**

```
# Adjust the coremask to not exceed the number of
# cores available on your OCTEON model.
target# bootoct 0 coremask=0xF
```

Note: *Coremask needs to be an odd number, so that core 0 is always booted.*

What you should see on the target console:

In this example, the program is running on 4 out of 8 cores, specified by the “coremask=0xF” argument to the bootloader. Output from the cores is interleaved (whole lines, not partial lines) as shown on the target console because they are running simultaneously (PP1 is core1).

You should see:

```
Bootloader: Booting Octeon Executive application at 0x20000000, core  
mask: 0xf,  
stack size: 0x100000, heap size: 0x300000  
Bootloader: Done loading app on coremask: 0xf  
PP3:~CONSOLE->  
PP1:~CONSOLE->  
PP3:~CONSOLE->  
PP0:~CONSOLE->  
PP1:~CONSOLE->  
PP2:~CONSOLE->  
PP1:~CONSOLE-> Hello world!  
PP3:~CONSOLE-> Hello world!  
PP2:~CONSOLE->  
PP0:~CONSOLE->  
PP1:~CONSOLE-> Hello example run successfully.  
PP2:~CONSOLE-> Hello world!  
PP3:~CONSOLE-> Hello example run successfully.  
PP0:~CONSOLE-> Hello world!  
PP2:~CONSOLE-> Hello example run successfully.  
PP0:~CONSOLE-> Hello example run successfully.
```

Note: The exact order of the output lines will vary because the different processors are competing for access to the UART. This is a simple example of a *race condition*.

Note: If more cores are specified in the coremask than are available on your OCTEON model, you will see an error message. In this example, only 12 cores are available, but 16 cores are requested (0xFFFF). The bootloader replies that only 0xFFF is available (12 cores).

```
target# bootoct 0 coremask=0xFFFF  
ERROR: Can't boot cores that don't exist! (available coremask: 0xffff)  
Invalid coremask.
```

13 About the Bootloader

This section provides a quick introduction to the bootloader.

Note: When following the steps in this tutorial, is important to run a bootloader which was provided with SDK 1.7 or higher.

13.1 Booting an OCTEON Board

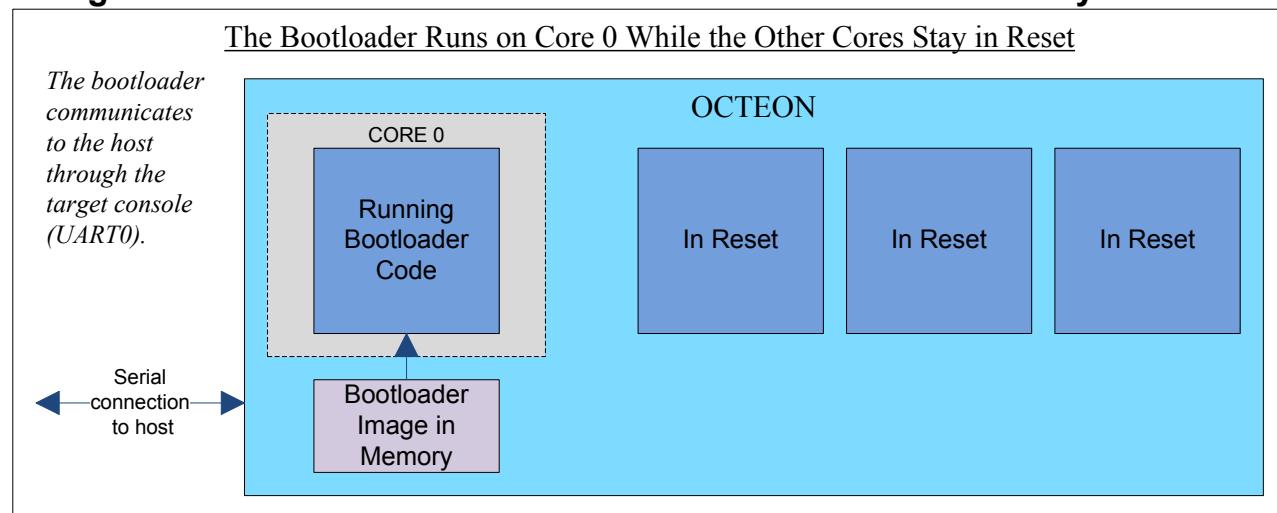
If the bootloader is located in flash, and the hardware is configured to boot from flash, then the bootloader will run when the development target is powered on.

If the development target is a standalone board, then the bootloader must be present in flash. If the main bootloader is not usable, it can be restored from the failsafe bootloader.

If the development target is a PCI board, then it can be configured to download the bootloader from the PCI host over PCI. The `oct-pci-boot` command will reset the board and download and run the bootloader.

In the following figure, the bootloader image is being run on core 0. The other cores stay in reset until an application is loaded onto core 0 and begins to run.

Figure 10: Core 0 Runs Bootloader While the Other Cores Stay in Reset



13.1.1 Booting from Onboard Flash

When the board is booted and is configured to boot from onboard flash:

- Core 0 starts execution at the reset vector 0xBFC00000 (the location of the bootloader code in onboard flash).

The bootloader (U-Boot) then:

1. Initializes the UART
2. Configures the DRAM controller to allow physical memory to be used. (The UART provides the target console and debug console.)
3. Relocates itself from the onboard flash to DRAM, and continues executing from DRAM.
4. Executes the default command, if present.

13.1.2 Booting an OCTEON Board as a PCI Target

PCI development targets may be configured to boot from flash (described in the prior paragraph), or to boot over the PCI bus. This option is selected via a jumper on the board. See the *Quick Start Guide* for directions.

When the board is configured to boot over the PCI bus, after the board is powered on or after `oct-pci-reset`, the bootloader will not run from the onboard flash, so there will not be a prompt on the target console. The cores will stay in reset. The following is the output shown if the board is reset, and it is not configured to boot from onboard flash.

```
host$ oct-pci-reset
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x80000000]
Warning: Timed-out waiting for bootloader (-1)
```

To boot the board over PCI, use the command `oct-pci-boot`, typed on the PCI host. This command uses a version of the bootloader which has been modified to support booting over PCI. If a bootloader is not specified on the command line, the default bootloader will be used. The `oct-pci-boot` command:

1. Configures the DRAM controller on the OCTEON processor
2. Copies the bootloader code to DRAM.
3. Configures the development target so that the bootloader code will start when core 0 jumps to the reset vector.
4. Takes core 0 out of reset.
5. Core 0 starts execution at the reset vector, which is now jumps to the bootloader code.

The bootloader (U-Boot) then:

1. Initializes the UART. (The UART provides the target console and debug console.)
2. Skips the DRAM controller setup.
3. Then U-Boot executes the default command, if present. When U-Boot is run on hardware instead of the simulator, it displays a prompt on UART0 (the serial console):

13.1.3 Verifying the Bootloader is Up and Running

When U-Boot is run on hardware, it displays the bootloader prompt on UART0 (the target console):

```
Octeon ebt3000(ram) #
```

If this prompt is not visible, typing **Enter** in the Minicom screen will result in a response from the bootloader, ending with the bootloader prompt.

The board is now up and ready to download the application. The Minicom screen is now referred to as the target console. Typing **help** on the Minicom screen will result in a reply from the bootloader: a menu of possible commands.

13.2 Review of Bootloader Memory Use

As discussed in the *Software Overview* chapter, there are two memory areas reserved by the bootloader: the Reserved Download Block, which is used to download the application, and the Reserved Linux Block. These two areas may be seen with the bootloader command **namedprint**.

The exact configuration selected by the bootloader will vary. The following output is from a 1.8.0 bootloader.

```
target# namedprint
List of currently allocated named bootmem blocks:
Name: __tmp_load, address: 0x0000000020000000, size: 0x000000006000000, index:0
Name: __tmp_reserved_linux, address: 0x000000000001000000, size:
0x000000008000000, index: 1
```

Bootloader memory usage changed between SDK 1.6 and lower and SDK 1.7 and higher. Be sure to find out which bootloader is running on the board before continuing with the tutorial. Upgrade if needed. The new bootloader is simpler to use than the old one.

13.3 The Failsafe Bootloader

Note: There are two bootloaders in the onboard flash: a failsafe image and the main bootloader image. If the main image is damaged, the board may be booted from the failsafe bootloader image. A new bootloader image may be downloaded to the onboard flash by copying it from the flash card to the onboard flash, using commands in the failsafe bootloader image.

The failsafe bootloader does not have the full features of the main bootloader: it only contains commands needed to read and write the flash, and a few configuration and diagnostic commands.

Note: The failsafe bootloader may ONLY be used to restore the main bootloader. It is not safe to use the failsafe bootloader to download and run an application.

More information about the bootloader can be found in the SDK document “*OCTEON Bootloader*”.

13.4 Bootloader Commands

Once the bootloader is running, type **help** in the target console to see a list of bootloader commands. The command **help <cmd>** will show more details about the individual command.

A very important command is **version**. This command is used to find out whether the bootloader was compiled by SDK 1.6 or newer. After SDK 1.6, a change was made to how the bootloader loads ELF files in memory, changing the address used in the load commands.

The following bootloader command shows a bootloader built with SDK 1.7.3:

```
target# version
U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun
13)
```

Cavium Networks has added the following commands to U-Boot:

1. **bootoct**: Boot from an OCTEON Simple Executive ELF file in memory
2. **bootoctlinux**: Boot from a Linux ELF file in memory
3. **bootoctelf**: Added to support booting other ELF files such as non-Linux Operating Systems. The OCTEON SDK does NOT create any ELF files suitable for use with this command.

Bootloader commands which are used in this tutorial or in the *Software Overview* chapter are listed in the following table. A complete list of bootloader commands is available in Table 48 – "U-Boot Commands Quick Reference, Part 1".

Table 20: Key Bootloader Commands

<i>Commands are shown without arguments.</i>	
Command	Description
?	An alias for help.
bootoct	Boot from an OCTEON Executive ELF image in memory. This is used to boot Simple Executive Stand-alone applications.
bootoctlinux	Boot from a Linux ELF image in memory. This is used to boot Linux.
dhcp	Invoke DHCP client to obtain IP/boot params (get IP address from DHCP server).
fatload	Load ELF file from a DOS file system (boot from compact flash).
help	Print bootloader help menu or help with a specific command if <code>help <cmd></code> is typed.
namedprint	Print list of named bootmem blocks.
ping	Send ICMP ECHO_REQUEST to network host.
printenv	Print environment variables. Some of the environment variables are defined to be multiple commands, (for example <code>nuke_env</code>).
saveenv	Save environment variables to persistent storage.
setenv	Set environment variables such as the IP address.
tftpboot	Boot image via network using TFTP protocol.
version	Print the bootloader version information. This command will display the matching SDK version, for example: <code>SDK version: 1.7.3-264</code> .

The `oct-pci-bootcmd` is used to run bootloader commands over PCI instead of the from the target console. For example, after downloading a Simple Executive ELF file using the

oct-pci-load utility (creating an in-memory image); the oct-pci-bootcmd utility can be used to boot the in-memory image.

Note that multiple applications cannot be accidentally booted on the same core because U-Boot will issue a warning:

```
target# bootoct 0 coremask=0x2
Bootloader: Booting Octeon Executive application at 0x20000000, core
mask: 0x2,
stack size: 0x100000, heap size: 0x300000
Bootloader: Done loading app on coremask: 0x2
# boot on the same core again by accident
target# bootoct 0 coremask=0x2
ERROR: Can't load code on core twice! (provided coremask(0x2) overlaps
previously loaded coremask(0x2))
```

13.5 Bootloader Environment Variables

Bootloader environment variables may be seen by typing printenv in the target console when the bootloader is running.

Note that some of these, such as nuke_env are compound commands. Details on environment variables may be found in the SDK document “*OCTEON Bootloader*”.

To prevent the bootloader from trying to automatically execute an application, set the bootloader environment variable autoload to n in the Minicom session (autoload = n is the default when the bootloader is upgraded):

```
target # setenv autoload n
# save the value so it will still be set after a reset
target # saveenv
```

13.6 Upgrading the Bootloader

To view the bootloader version number, type version in the target console. If the bootloader version number is lower than 1.7, then it should be upgraded. See Section 33 – “Appendix I: Updating U-Boot on a Standalone Board” for directions.

14 About Downloading the Application

To run the application image in memory, it must first have been downloaded to memory. In a production system, the image file is usually in onboard flash. During development, the image is not in onboard flash, and must be downloaded to memory after each time the board is powered off or reset.

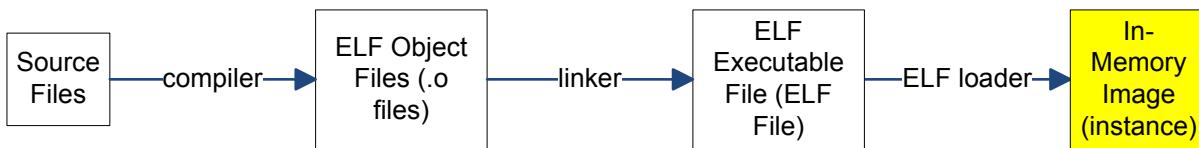
When running a hybrid system (more than one load set), be sure to start the application running on core 0 last. Once the application runs on core 0, the other cores come out of reset and begin to run their applications. Core 0 may be one of many cores in the load set, there is no need to start an application only on core 0.

As discussed above, this download requires temporary memory to store the image. At the next step, the ELF file boot step, the bootloader reads the ELF image from the Reserved Download Block, parses it, and loads the application into memory (creating the in-memory image). After the core 0 image starts to run, the memory used for the image download is freed and added to the pool of free memory.

Figure 11: Creating an In-Memory Image

Creating an In-Memory Image

When run as a SE-UM application, the in-memory image file is called a process or instance of the program. When run as a SE-S application, the in-memory image file is called an instance of the program.



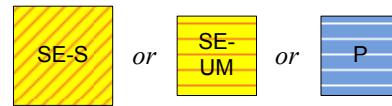
For SE-S applications, the ELF loader reads the ELF File and creates an in-memory image (instance). The in-memory image is larger than the ELF File: it contains system memory allocated for the stack and heap.

The instance of the program may share .text and .rodata with another instance of the program in the same load set, but will always have some private (not shared) regions which make it unique.

ELF File

- SE-S ELF files may be stored in flash, or downloaded to the board.
- SE-UM files may be stored in the embedded_rootfs of the Linux ELF file, or downloaded after Linux is booted.

The in-memory image (instance of the program) may be an SE-S, SE-UM, or other Linux process.



ELF Loader

- For SE-S processes, the ELF loader is `bootoctl`.
- For Linux, the ELF loader is `bootoctllinux`.
- For SE-UM instances, the kernel handles ELF file loading automatically. The `oncpu` utility may be used to start SE-UM applications on a subset of cores.

Note: SE-S and SE-UM applications must be statically linked.

Downloading the application ELF file to the development target can be accomplished via:

1. PCI (`oct-pci-load`): The board is a PCI target (preferred).
2. Flash card (`fatload`): The flash card writer and cable are supplied with most evaluation boards.
3. Ethernet via TFTP (`tftpboot`): The board is stand alone with an Ethernet connection to the development host.
4. Serial Port: the slowest and least preferred, with a serial connection to the development host. For information on how to download the application to the board over a serial connection, see Section 35 – “Appendix K: Downloading Using the Serial Connection”. For small applications such as `hello` the serial connection is fast enough, but for larger

applications it can be very slow. When possible, `strip` the ELF file to download the file faster.

The ELF file is downloaded to the specified Reserved Download Block address. Bootloaders built with SDK 1.7 and higher allow the specified address to be 0. When the address is 0, the default Reserved Download Block address is selected by the bootloader. This simpler address (0) is used in examples in this chapter. After the ELF file is downloaded, the bootloader relocates it to a physical location of its choice (creating the in-memory image). The Reserved Download Block is now available for the next download.

For shell script users: note that after a `oct-pci-load` command completes, about one second (`sleep 1`) is needed for the bootloader to complete the relocation before the next download command to the same location. Otherwise, the second downloaded image will overwrite the first downloaded image before the bootloader is finished reading and relocating it.

The following table introduces the different ELF file download commands available. The choice of download command depends on the runtime environment (PCI, standalone, or simulator), and where the ELF file is located (on a PCI host, a flash card, onboard flash, etc.).

Table 21: Commands to Download ELF File for Different Configurations

Runtime Environment	ELF File Location	Command to Load ELF File into Memory (create in-memory image)
OCTEON PCI Target Board	On the PCI Host	<code>oct-pci-load</code> (fastest for PCI Target Board)
	File is on a Compact Flash Card	<code>fatload</code>
	File is burned into on-board flash	No download needed.
Stand-alone OCTEON Board	File is on another system on the network	<code>tftpboot</code> (fastest for Stand-alone boards)
	File is on a Compact Flash Card	<code>fatload</code>
	File is burned into on-board flash	No download needed.
OCTEON Simulator	On the same computer	<code>oct-sim</code> or <code>oct-linux</code>

The following table includes some notes on the different commands used to download an ELF file to the board.

Table 22: ELF File Download Command Details

Command	Notes
# type the following command on one line oct-pci-load <tmp_download_address> <filename>	In SDK 1.7 and above, <i>tmp_download_address</i> should be 0 to let the bootloader choose the address. When using a shell script, sleep 1 second between oct-pci-load commands to allow the relocation from the image loading block to the permanent address to complete.
tftpboot <tmp_download_address> <filename>	In SDK 1.7 and above, specify 0 for the <i>tmp_download_address</i> and let the bootloader choose the address.
# type the following command on one line fatload <dev> <num> <tmp_download_address> <filename>	In SDK 1.7 and above, specify 0 for the <i>tmp_download_address</i> and let the bootloader choose the address.

Directions on how to boot using a flash card are in Section 40 – “Appendix P: Booting an ELF File From a Flash Card”.

15 About Booting SE-S Applications

15.1 The Coremask

The coremask is a bit mask where core 0 is represented by bit 0; core 1 is represented by bit 1, etc.

The value 0x0F (binary 0000 1111) would start an application on cores 0, 1, 2, 3.

The value 0xF0 (binary 1111 0000) would start an application on cores 4, 5, 6, 7.

The application running on core 0 should be started last. Once the application runs on core 0, the other cores come out of reset and begin to run their applications.

15.2 The Boot Command

The arguments to the boot command are used to specify the stack and heap sizes, and the coremask. In example programs such as `hello`, the default stack and heap size are adequate, so they are not specified on the command line.

Note that the boot command will load the applications onto the cores, but all cores except core 0 are held in reset (the applications do not run) until the application running on core 0 is booted. For this reason, core 0 should be booted last.

For SDK 1.7 and higher, the bootoct usage is:

```
target# help bootoct
bootoct - Boot from an Octeon Executive ELF image in memory
          [elf_address [stack=stack_size] [heap=heap_size]
           [coremask=mask_to_run | numcores=core_cnt_to_run]
           [forceboot] [debug] [break] [endbootargs] [app_args...]
elf_address - address of ELF image to load. defaults to $(loadaddr).
              If 0, default load address used.
stack      - size of stack in bytes. Defaults to 1 megabyte
heap       - size of heap in bytes. Defaults to 3 megabytes
coremask   - mask of cores to run on. Anded with coremask_override
              environment variable to ensure only working cores are used
numcores   - number of cores to run on. Runs on specified number of
cores,
              taking into account the coremask_override.
skipcores  - only meaningful with numcores. Skips this many
              cores (starting from 0) when
              loading the numcores cores. For example, setting
              skipcores to 1 will skip core 0
              and load the application starting at the next available core
debug      - if present, bootloader passes debug flag to application
              which will cause the application to stop at a
              breakpoint on startup
break      - if present, exit program when control-c is received on
              the console
forceboot  - if set, boots application even if core 0 is not in mask
endbootargs - if set, bootloader does not process any further arguments
              and only passes the arguments that follow to the
              application. If not set, the application
              gets the entire command line as arguments.
```

Note: If you do not see the entire help text on the screen, start Minicom with the **-w** (line wrap) option.

To boot an SE-S application, use the bootloader command **bootoct**. If the OCTEON board is a PCI target, the utility **oct-pci-bootcmd** may be used instead of typing into the target console.

For example:

```
host$ oct-pci-bootcmd "bootoct 0 coremask=0x1"
```

The output of the **hello** application will appear on the target console.

To boot Linux, use the bootloader command **bootoctlinux**.

16 About Building Linux

In order to build Linux, some knowledge about the kernel and the root filesystem, Makefiles and Makefile targets, and kernel and embedded root filesystem configuration are essential. This basic information is provided in this section.

16.1 The Root Filesystem

The Linux kernel requires access to a root filesystem. There are three possible locations for the root filesystem:

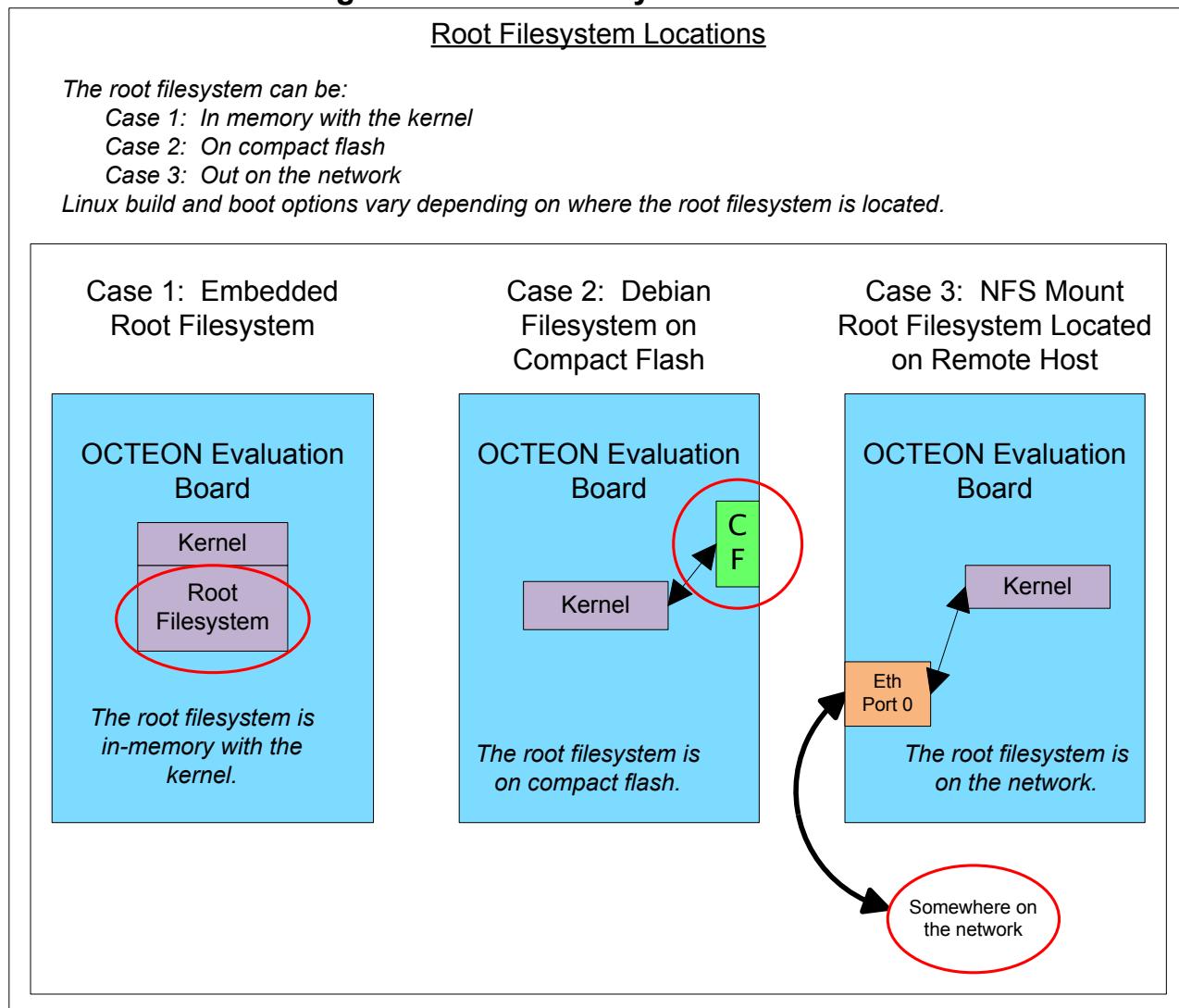
In the simplest case, the root filesystem is located right next to the kernel in memory.

In a slightly more complex case, the root filesystem is on a flash card, or on another device which is local to the target board, such as a USB device.

In a very complex case, the root filesystem is out on the network somewhere. This case is not discussed in this tutorial.

The following figure is a simplified view of the Linux kernel running on the OCTEON processor, with three different locations for the root filesystem.

Figure 12: Root Filesystem Locations



The information provided in this section is a brief overview. Detailed instructions may be found in the SDK documents “*Linux on the OCTEON*” and “*Running Debian GNU/Linux on OCTEON*”.

For directions on using a root filesystem located on a USB device, see the SDK document “*Linux on Small OCTEON Systems*”.

16.1.1 The Embedded Root Filesystem

Usually embedded systems do not have disk drives, so Linux is booted from an ELF file (`vmlinux.64`) stored in onboard flash. The ELF file contains the Linux kernel and a filesystem which runs in memory only. This filesystem is the embedded root filesystem (*embedded_rootfs*). When the system is powered off, the in-memory copy of the filesystem is gone. The system must be booted from the `vmlinux.64` file stored in onboard flash again. Because the onboard flash is small, the utilities in the root filesystem are the compact set supplied by busybox. This set is smaller than the full Linux utilities in two ways: fewer utilities are present, and each utility supports only essential options.

During development, `vmlinux.64` may be downloaded from the host instead of stored in onboard flash. The download process is identical to the one used for the `hello` application.

The `vmlinux.64` file may also be copied to a flash card and booted from there. Directions are in Section 40 – “Appendix P: Booting an ELF File From a Flash Card”.

The contents of the root filesystem can be seen on the host before Linux is booted. See Section 38 – “Appendix N: Contents of the Embedded Root Filesystem”.

16.1.2 The Debian Root Filesystem

The Debian filesystem is a non-embedded root filesystem. Typically, it is located on a flash card. When the kernel is booted, it is directed to use the root filesystem located on the flash card.

The native tools, including the Cavium Networks GNU tool chain, may be used from the Debian root filesystem to simplify development and debugging. See Section 9.6 – “Native Tools (Run on the Development Target)” for more information.

See Section 41 – “Appendix Q: Using the Debian Root Filesystem” for more information.

16.2 Linux Makefiles and Makefile Targets

Linux consists of the kernel and the root filesystem. There are three key Makefiles involved in the Linux build:

1. \$OCTEON_ROOT/linux/Makefile: This is the top-level Linux Makefile.
2. \$OCTEON_ROOT/linux/kernel_2.6/Makefile: This Makefile controls the kernel and modules build: it is called by the top-level Makefile.
3. \$OCTEON_ROOT/linux/kernel_2.6/linux/Makefile: This Makefile controls the kernel build – it is called by the \$OCTEON_ROOT/linux/kernel_2.6 Makefile.
4. \$OCTEON_ROOT/linux/embedded_rootfs: This Makefile controls the embedded root filesystem build.
5. \$OCTEON_ROOT/linux/Debian: This Makefile includes the target compact-flash which is used to put the Debian root filesystem onto the flash card.

The top-level Makefile has several key targets:

1. kernel – build the kernel and the embedded root filesystem
2. kernel-deb – builds the kernel only. The kernel will use the Debian root file system.
3. sim – build the kernel and the embedded root filesystem, designed to run on the hardware simulator.
4. strip – strip the symbols out of vmlinux.64, creating a smaller vmlinux.64 file.
5. clean – remove all files created by the prior build

The kernel, kernel-deb, and the sim targets will build the ELF file

\$OCTEON_ROOT/linux/kernel_2.6/linux/vmlinux.64. The vmlinux.64 file may or may not contain a root filesystem, depending on the target selected.

To see the other make targets, cd \$OCTEON_ROOT/linux and type make without a target. The following table contains some of the commonly-used targets.

Table 23: Linux Top-Level Makefile Targets

Target	Description
kernel	The command <code>make kernel</code> will build the Linux kernel and the embedded root filesystem, and create the <code>vmlinux.64</code> ELF file. The <code>vmlinux.64</code> file contains the Linux kernel and the embedded root filesystem.
kernel-deb	The command <code>make kernel-deb</code> will build the Linux kernel, and create the <code>vmlinux.64</code> ELF file. The <code>vmlinux.64</code> file contains <i>only</i> the Linux kernel. This target is used when the kernel uses the separate Debian root filesystem. The Debian root filesystem is typically located on compact flash.
sim	The command <code>make sim</code> will build the Linux kernel and the embedded root filesystem, and create the <code>vmlinux.64</code> ELF file. The <code>vmlinux.64</code> file contains the Linux kernel and the embedded root filesystem. The <code>vmlinux.64</code> file is built to run on the hardware simulator.
strip	Strip symbols out of <code>vmlinux.64</code> . This option will strip debugging information, creating a smaller <code>vmlinux.64</code> file.
tftp	Copy <code>vmlinux.64</code> to <code>/tftpboot</code> , then strip it.
flash	Copy <code>vmlinux.64</code> onto compact flash. The Makefile expects that the compact flash was already mounted at <code>/mnt/cf1</code> .
clean	Remove all generated files and the KERNEL CONFIG.

16.2.1 The sudo Command Needed to Configure and Build Linux

The `sudo` command is required to configure and build the Linux filesystem on the development host. Note that once the `sudo` command is entered, and the password accepted, the system will remember the password for awhile. The default amount of time is 5 minutes.

To make executing Linux build commands simpler, type `sudo ls` and enter the password immediately before building Linux targets. This will facilitate putting the `make` command into the background, and saving the output of the command, as discussed in Section 10.5 – “Saving make Output”.

Note: If you omit the `sudo` before the `make`, you will be prompted for your `sudo` password several times during the build. The first kernel is built takes about 20 minutes. It is bothersome to have to watch for the multiple password prompts during the entire build. It is also not possible to simply check on the build from time to time: the `sudo` password prompt will time out after 5 minutes. If multiple users are sharing the same system, see Section 44 – “Appendix T: Multiple Embedded Root Filesystem Builds”.

16.3 Configuring Linux

Both the kernel and the embedded root filesystem builds may be separately configured.

16.3.1 Configuring the Linux Kernel

To configure the Linux kernel, go to \$OCTEON_ROOT/linux/kernel_2.6/linux and type:
host\$ **make menuconfig**

16.3.2 Configuring the Embedded Root Filesystem

To configure the embedded root filesystem, go to
\$OCTEON_ROOT/linux/embedded_rootfs and type:
host\$ **sudo make menuconfig**

While following the steps in this tutorial, do not change the kernel or embedded root filesystem configuration unless specified in the instructions.

16.4 Building Linux

There are 3 separate ways to build Linux from the top-level Makefile:

1. The kernel with the embedded root filesystem
2. The kernel only
3. The kernel with embedded root filesystem which is built to run on the hardware simulator.

16.4.1 Build Linux with the Embedded Root Filesystem

Here are the steps for building a bootable embedded version of Linux which includes the embedded root filesystem:

```
host$ cd $OCTEON_ROOT/linux
# type sudo ls to set the root password.  The host will remember it for
# about 5 minutes.
host$ sudo make clean
host$ sudo ls
host$ sudo make kernel >make.out 2>&1 &
host$ tail -f make.out
```

This build takes about 20 minutes. (Note that some warnings are normal with a Linux build.)

The Makefile will create the \$OCTEON_ROOT/linux/kernel_2.6/linux/vmlinux.64 ELF file, which can be run on the OCTEON processor. (Note that newer SDKs may have a newer version of the kernel, so that the kernel_2.6 directory may need to be updated to a higher version number.)

Note: If multiple users are sharing a development host, and will need to build the embedded root filesystem, see Section 44 – “Appendix T: Multiple Embedded Root Filesystem Builds”.

16.4.2 Build the Linux Kernel Only

To build the Linux kernel only, use the instructions for building Linux with the embedded root filesystem, but substitute `kernel-deb` instead of `kernel` as the make target:

```
host$ sudo make kernel-deb >make.out 2>&1 &
```

The Makefile will create the \$OCTEON_ROOT/linux/kernel_2.6/linux/vmlinux.64 ELF file, which can be run on the OCTEON processor. (Note that newer SDKs may have a newer version of the kernel, so that the kernel_2.6 directory may need to be updated to a higher version number.)

16.4.3 Build Linux to Run on the Hardware Simulator

To build the Linux kernel only, use the instructions for building Linux with the embedded root filesystem, but substitute sim instead of kernel as the make target:

```
host$ sudo make sim >make.out 2>&1 &
```

The Makefile will create the \$OCTEON_ROOT/linux/kernel_2.6/linux/vmlinux.64 ELF file, which can be run on the hardware simulator. (Note that newer SDKs may have a newer version of the kernel, so that the kernel_2.6 directory may need to be updated to a higher version number.)

16.5 About the `make clean` Command

If the `make clean` command is executed in the \$OCTEON_ROOT/linux directory, then the kernel's \$OCTEON_ROOT/linux/kernel_2.6/linux/.config file will be deleted. On the next build, if the \$OCTEON_ROOT/linux/kernel_2.6/linux/.config file is missing, then the pristine (original kernel.config supplied with the release) is copied to .config. This is done to allow the user to restore the original working kernel configuration.

This can be inconvenient if the user would like to save the .config file. The .config file can be saved to another name, such as .config.save. After the clean step, this file can be copied to .config.

To clean the kernel directory without removing the .config file, cd to \$OCTEON_ROOT/linux/kernel_2.6 before executing `make clean`, or use the following command in the \$OCTEON_ROOT/linux directory:

```
host$ make -s -C kernel_2.6 clean
```

16.6 The Kernel File Name: vmlinu \times VS vmlinu \times .64

In the SDK documentation, the name vmlinu \times .64 is used instead of vmlinu \times . Both files have the same content, so when the directions specify vmlinu \times .64, it is okay to use vmlinu \times instead.

The presence of two files is for historical reasons. At one point in the past there was a difference between vmlinu \times and vmlinu \times .64. The kernel build process made a 32-bit binary and then changed it into a 64-bit binary. This is no longer the case, but the vmlinu \times .64 name was kept for backwards compatibility.

16.7 The *strip* Utility and the *vmlinux.64* ELF File

The strip utility is used to remove debugging information from the ELF file. This step is usually done after debugging is complete. The result is a smaller file, which downloads and boots faster. Strip is used before putting the file into the onboard flash.

To strip the kernel:

```
host$ cd $OCTEON_ROOT/linux/kernel_2.6/linux
# save the unstripped version
host$ cp vmlinux.64 vmlinux.64_unstripped
# go to the $OCTEON_ROOT/linux directory
host$ cd ../..
# strip the kernel
host$ sudo make strip
```

Verify the new executable is smaller:

```
-rwxr-xr-x 1 testname software 20097552 Feb 22 14:54 vmlinux.64
-rwxr-xr-x 1 testname software 81380505 Feb 22 14:54 vmlinux.64.unstripped
```

17 Hands-on: Build and Run Linux

In this step, the *vmlinux.64* file will be build, downloaded and run. Linux will then use the embedded root filesystem, running in memory.

Once the *vmlinux.64* ELF file has been downloaded and booted, Linux will communicate via the target console. Once the Linux prompt is up, type `ls` to see the busybox tools in action.

Table 24: Build and Run Linux, Part 1

Steps	Note
1. Connect the Hardware	
Connect serial cable to target console. Connect the Ethernet cable.	Follow directions in the <i>Quick Start Guide</i> . Note: The Ethernet cable is not needed to run Linux on a PCI target board, but will be needed later in the <i>SDK Tutorial</i> . Be careful to isolate the test network from the office network so that experiments will not disturb the office network.
2. Reset the board	
For <i>stand-alone</i> boards: power on or reset the board. For <i>PCI</i> boards: host\$ oct-pci-reset	The word Boot should appear on the red LEDs on the board. If not, something is wrong with the board.
3. Connect to the Target Console	
host\$ minicom -w ttys0	Substitute <i>ttys0</i> for the serial port on the host connected to the OCTEON target board. This will provide a connection to the target console. You should see the bootloader prompt.
4. Verify Bootloader Prompt is Visible	
target# version	The bootloader should reply with text similar to: U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun 13)
5. Verify Bootloader Version is at Least SDK 1.7	
If the bootloader's SDK version is not at least 1.7, then before continuing, upgrade the bootloader to a newer version.	Directions for upgrading the bootloader are included in the <i>SDK Tutorial</i> .
6. Build Linux	If this step was already done, no need to repeat it!
<pre>host\$ cd \$OCTEON_ROOT/linux host\$ sudo ls host\$ sudo make kernel >make.out 2>&1 & host\$ make strip host\$ cd kernel_2.6/linux</pre>	Set the sudo password for 5 minutes Build the kernel and embedded root filesystem in the background. Use tail -f to see the output. The make command will create the executable file vmlinux.64 . Strip the vmlinux.64 file to make it smaller. The vmlinux.64 file is located in the kernel_2.6/linux directory.
7. Copy the ELF file to the tftpboot Directory	
<i># verify the build is complete before continuing</i> host\$ wait <i># go to the created vmlinux file</i> host\$ cd kernel_2.6/linux host\$ sudo cp vmlinux /tftpboot	See <i>SDK Tutorial</i> directions for tftpboot .
<i>Continued in the next table...</i>	

Table 25: Build and Run Linux, Part 2

Steps	Note
8. Select Target IP Address, if Needed	
If a DHCP server is available, then selecting the target IP address is handled by the server. Otherwise, select a target IP address.	In this example, the target IP address is <i>192.168.51.159</i> .
9. Set the Development Target's IP Address	
9a. No DHCP Server	
First, set the IP address of the target (replace items in italic with your IP addresses). <i># Use your IP addresses instead of the example values!</i> target# setenv gatewayip 192.168.51.254 target# setenv netmask 255.255.255.0 target# setenv ipaddr 192.168.51.159 target# setenv serverip 192.168.51.1 <i># save the values so they will still be set after a reset</i> target# saveenv	Note: <code>serverip</code> is the IP address of the TFTP server.
9b. DHCP Server Available	
If a DHCP server is available substitute the following step: target# dhcp	
10. Test the Ethernet Connection to the Host	
<i># use your host IP address instead of the example value!</i> target# ping 192.168.51.254	Expect to see: Using <code>octeth0</code> device host <i>192.168.51.254</i> is alive Note that the development target will <i>not</i> reply to a ping from the development host. Note: to see the development host's IP address, use the <code>/sbin/ifconfig</code> command on the development host.
11. Download Linux to the Development Target	
<i># Use tftpboot to download the application.</i> target# tftpboot 0 vmlinuz.64	See SDK Tutorial directions for <code>tftpboot</code> . If this step does not work, check the <code>/etc/xinetd.d/tftp</code> file on the host to verify that <code>server_args = -s /tftpboot</code> .
12. Boot Linux	
target# bootoctlinux 0 coremask=0x1	This command will run <code>vmlinuz.64</code> on core 0. Expect to see the Linux prompt. Type <code>ls</code> to verify Linux is up and running.
13. Reset the Target Board	
To reset the development target, <code>reboot</code> Linux, or For stand-alone boards: power on or reset the board. For PCI boards: host\$ oct-pci-reset	Note: To run <code>vmlinuz.64</code> a second time, the board must be reset or power cycled because there is no other way to return to the bootloader prompt.

17.1 Build the Kernel and Embedded Root Filesystem

First, build `vmlinuz.64`:

```
host$ cd OCTEON_ROOT/linux
host$ sudo make clean
```

```
host$ sudo ls
host$ sudo make kernel >make.out 2>&1 &
```

The result of this build (the default ABI is EABI64 ABI) is an ELF file with the kernel and filesystem packaged as a 64-bit ELF binary. On SDK 1.8.0, this file is \$OCTEON_ROOT/linux/kernel_2.6/linux/vmlinux.64.

17.2 Download vmlinu.x.64 to the Development Target

Using the steps shown in the hello example, download vmlinu.x.64 to the target board.

17.3 Boot Linux on the Development Target

Use the bootloader command bootoctlinux to boot Linux. For example, to run Linux on the first core:

```
target# bootoctlinux 0 coremask=0x1
argv[2]: coremask=0x1
ELF file is 64 bit
Attempting to allocate memory for ELF segment: addr: 0xffffffff81100000
(adjust8
Allocated memory for ELF segment: addr: 0xffffffff81100000, size 0x1303988
Loading .text @ 0xffffffff81100000 (0x43e8bc bytes)
Loading __ex_table @ 0xffffffff8153e8c0 (0x6f80 bytes)
Loading .rodata @ 0xffffffff81546000 (0xb42a4 bytes)
Loading .pci_fixup @ 0xffffffff815fa2a8 (0xbe0 bytes)
Loading __ksymtab @ 0xffffffff815fae88 (0xa0a0 bytes)
Loading __ksymtab_gpl @ 0xffffffff81604f28 (0x2c40 bytes)
Loading __ksymtab_gpl_future @ 0xffffffff81607b68 (0x30 bytes)
Loading __ksymtab_strings @ 0xffffffff81607b98 (0x109c8 bytes)
Loading __param @ 0xffffffff81618560 (0x1ba8 bytes)
Loading .data @ 0xffffffff8161c000 (0x40eb0 bytes)
Loading .data.cacheline_aligned @ 0xffffffff8165d000 (0x8b80 bytes)
Loading .init.text @ 0xffffffff81666000 (0x38270 bytes)
Loading .init.data @ 0xffffffff8169e270 (0x6190 bytes)
Loading .init.setup @ 0xffffffff816a4400 (0x6d8 bytes)
Loading .initcall.init @ 0xffffffff816a4ad8 (0x5c8 bytes)
Loading .con_initcall.init @ 0xffffffff816a50a0 (0x18 bytes)
Loading .exit.text @ 0xffffffff816a50b8 (0x2278 bytes)
Loading .init.ramfs @ 0xffffffff816a8000 (0xccd997 bytes)
Loading .data_percpu @ 0xffffffff82375a00 (0x36c8 bytes)
Clearing .bss @ 0xffffffff8237a000 (0x443a0 bytes)
Clearing .cvmx_shared_bss @ 0xffffffff823be3a0 (0x310 bytes)
## Loading Linux kernel with entry point: 0xffffffff81666000 ...
Bootloader: Done loading app on coremask: 0x1
```

For an example boot output, see the SDK Document “*Linux on the OCTEON*”.

Note that arguments may be passed to the kernel by using `bootoctlinux`:

```
target# help bootoctlinux
bootoctlinux elf_address [coremask=mask_to_run | numcores=core_cnt_to_run]
              [forceboot] [skipcores=core_cnt_to_skip] [endbootargs]
              [app_args ...]
elf_address - address of ELF image to load. If 0, default load address is used.
coremask     - mask of cores to run on. Anded with coremask_override
environment   variable to ensure only working cores are used
numcores      - number of cores to run on. Runs on specified number of cores,
                 taking into account the coremask_override.
skipcores    - only meaningful with numcores. Skips this many cores (starting
                 from 0) when loading the numcores cores. For example,
                 setting skipcores to 1 will skip core 0
                 and load the application starting at the next available core.
forceboot     - if set, boots application even if core 0 is not in mask
endbootargs   - if set, bootloader does not process any further arguments and
                 only passes the arguments that follow to the kernel. If not
                 set, the kernel gets the entire command line as arguments.
```

18 Hands-on: Run a SE-UM Example (`named-block`)

When the embedded root filesystem was built, the Linux examples were automatically built and included in the root filesystem. Once Linux is up and running, a SE-UM example can be run.

If you type `cd /examples`, you will see some of the example programs there. These have been compiled to run on Linux. The program `named-block` is a good example to run. This example is very simple and does not require any special knowledge or optional hardware units.

Note: to execute an example, precede the file name with the characters `./` to provide the pathname to the file (because `/examples` is not in the PATH list). Without the `./`, the file will not be found.

```
target# ./named-block
CVMX_SHARED: 0x1201a0000-0x1201b0000
Active coremask = 0x2
INFO: Size of pointer is 8 bytes
PASS: All tests passed
```

Note that this example may be run over and over without rebooting the system because, after the application exits, the system should display the Linux prompt.

In the next section, a much more complex example will be run.

19 About the `linux-filter` Example

An example which shows downloading and booting multiple ELF files on multiple cores is `linux-filter`. A Simple executive application, `linux-filter`, is downloaded to one set of cores, and Linux is downloaded to another set of cores. In the steps shown here, to simplify the example, only 2 cores are used, one for each application. In this example, core 0 will run

`linux-filter` and core 1 will run Linux. They can run on any core: there is no requirement that `linux-filter` is run on core 0, nor that `linux-filter` and Linux be assigned to cores in any particular order. Additionally, Linux and `linux-filter` may both be run on multiple cores to add more processing power. The only requirement is that the application which will run on core 0 is loaded last.

In this example, the Simple Executive application `linux-filter` receives all incoming packets. If the packet is not an IP broadcast packet (such as `ping -b`), then `linux-filter` forwards the packet to Linux, which is running on a different core.

Because cores receive packets to process via the `get_work()` operation, it is not necessary to forward the packet to a specific core. All the Linux cores are set to receive packets destined for their group, while the Simple Executive cores receive packets destined for their group. (See the *Software Overview* chapter for a discussion of processing groups.) This allows cores to combine their processing power.

This example requires an OCTEON processor with at least 2 cores. Although it can be run on more than 2 cores, only 2 cores will be shown here. The directions below are for a 1.7 and higher bootloader.

Figure 13: Example: linux-filter Forwards an IP Non-broadcast Packet

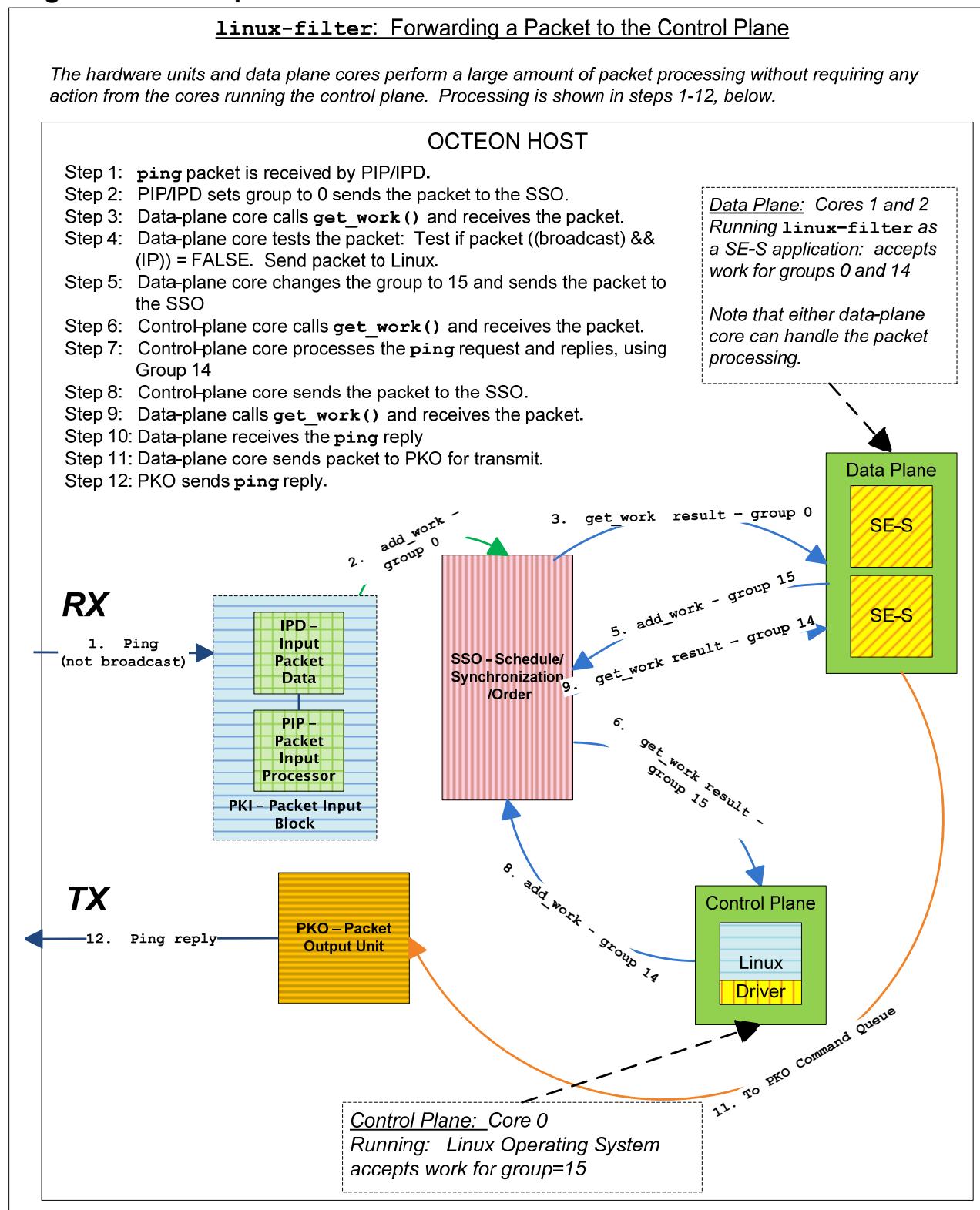
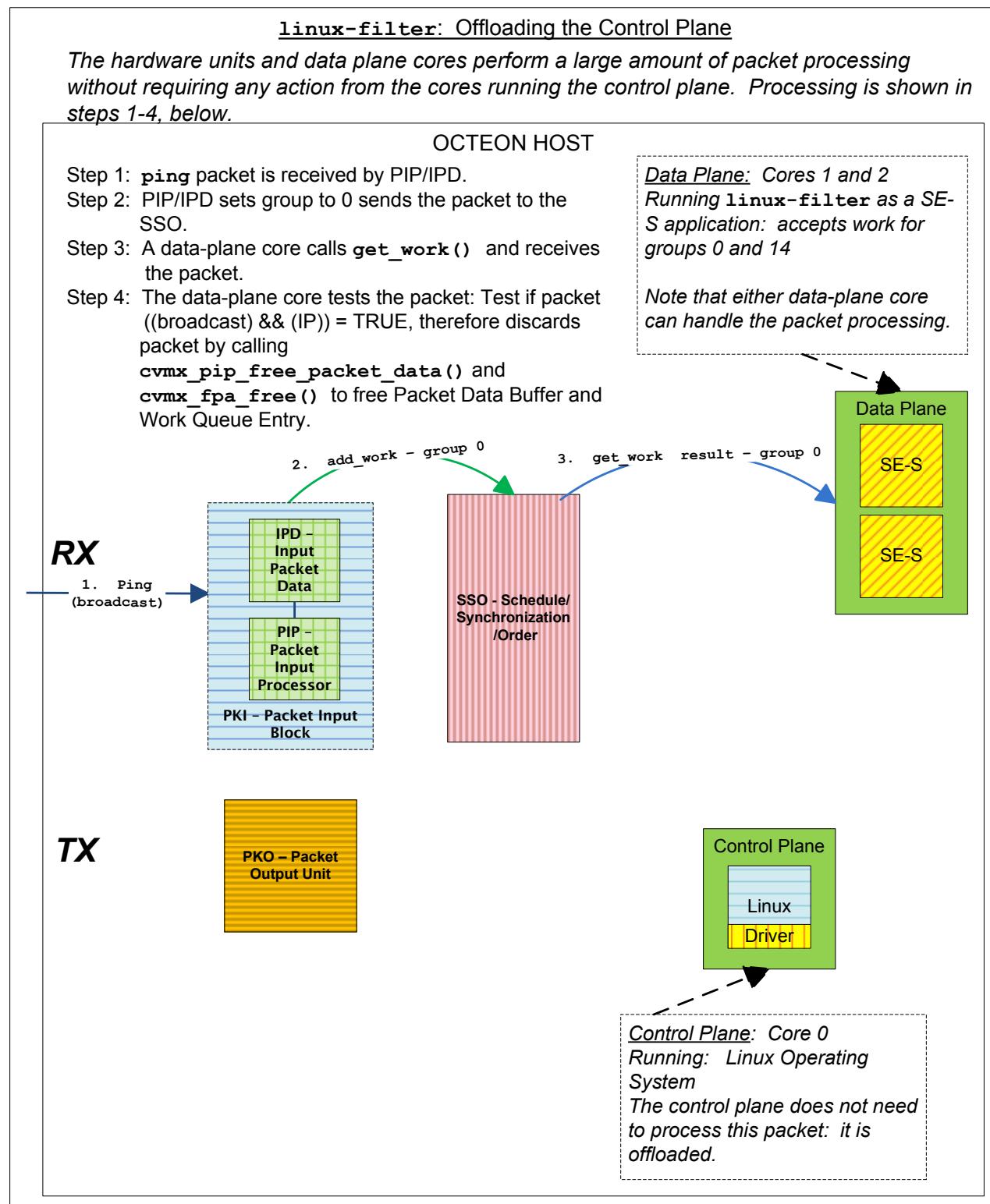


Figure 14: Example: linux-filter Does Not Forward an IP Broadcast Packet



Test configuration: In the test configuration used in the examples shown in this chapter, the development host IP address (`eth0`) is `192.168.51.254`. The development host and development target are on an isolated Ethernet. There are no other devices on the isolated Ethernet, so the IP address used for the development target will be `192.168.51.159`.

This is a summary of what will happen when the program is run. Note that in this example there is only one core running Linux and one core running `linux-filter`. Each application may be run on many cores.

1. The PIP/IPD receives the packet and set the group to the `FROM_INPUT_PORT_GROUP` (group 0).
2. The core running `linux-filter` receives packets for group `FROM_INPUT_PORT_GROUP`.
3. The core running `linux-filter` drops packets which are IP broadcast. It forwards the others to the Linux cores by setting the group to the `TO_LINUX_GROUP` (group 15).
4. The Linux core replies to the packet, setting the group for the reply to `FROM_LINUX_GROUP` (group 14).
5. The core running `linux-filter` also receives packets for group `FROM_LINUX_GROUP`.
6. The core running `linux-filter` sends the packet to the PKO for transmission.

20 Hands-on: Run `linux-filter` as a SE-S Application (Hybrid System)

These are the directions for running `linux-filter` as a Simple Executive Standalone application (SE-S). This is an example of a hybrid system: Linux will run on one core, and the SE-S application will run on another core.

Figure 15: Example of `linux-filter` (SE-S) and Linux Run on 2 out of 8 Cores

Example of 8-core system running the SE-S application `linux-filter` on core 0, and Linux on core 1.



The following tables show the steps to run `linux-filter` as a SE-S application on one, and Linux on another core. In this example, `linux-filter` is run on core 0 and Linux is run on core 1, but there is no reason not to run both on more cores. This example shows two different load sets, creating a hybrid system.

To test `linux-filter`, first a ping is sent from the development host to the development target, then a broadcast ping is sent from the development host to the development target. The first ping should be forwarded, the broadcast ping should be filtered, offloading the core running Linux. This example shows the data-plane offloading the control-plane.

Table 26: Build and Run Linux and `linux-filter` (SE-S), Part 1

Steps	Note
1. Connect the Hardware	
Connect serial cable to target console. Connect the Ethernet cable.	Follow directions in the <i>Quick Start Guide</i> . Note: The Ethernet cable is not needed to run Linux on a PCI target board, but will be needed later in the <i>SDK Tutorial</i> . Be careful to isolate the test network from the office network so that experiments will not disturb the office network.
2. Reset the board	
For <i>stand-alone</i> boards: power on or reset the board. For <i>PCI</i> boards: <code>host\$ oct-pci-reset</code>	The word <code>Boot</code> should appear on the red LEDs on the board. If not, something is wrong with the board.
3. Connect to the Target Console	
<code>host\$ minicom -w ttys0</code>	Substitute the serial port actually used on the host to connect to the OCTEON target board if it is not <code>ttys0</code> . Minicom will provide a connection to the target console. You should see the bootloader prompt.
4. Verify Bootloader Prompt is Visible	
<code>target# version</code>	The bootloader should reply with text similar to: <code>U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun 13)</code>
5. Verify Bootloader Version is at Least SDK 1.7	
If the bootloader's SDK version is not at least 1.7, then before continuing, upgrade the bootloader to a newer version.	Directions for upgrading the bootloader are included in the <i>SDK Tutorial</i> .
6. Build the Application	
# type the following command on one line <code>host\$ cd \$OCTEON_ROOT/examples/linux-filter</code>	
<code>host\$ make clean</code>	
<code>host\$ make</code>	Build the 64-bit SE-S application. The output of the <code>make</code> command is the file <code>linux-filter</code> . The SE-UM version is automatically built by the Linux build, and installed in the embedded root filesystem.
7. Copy the ELF file to the <code>tftpboot</code> Directory	
<code>host\$ sudo cp linux-filter /tftpboot</code>	See the <i>SDK Tutorial</i> directions for <code>tftpboot</code> .
<i>Continued in the next table...</i>	

Table 27: Build and Run Linux and linux-filter (SE-S), Part 2

Steps	Note
8. Build Linux and Copy the ELF File	If this step was already done, no need to repeat it!
<pre>host\$ cd \$OCTEON_ROOT/linux host\$ sudo ls host\$ sudo make kernel >make.out 2>&1 &</pre>	Set the <code>sudo</code> password for 5 minutes Build the kernel and embedded root filesystem in the background. Use <code>tail -f</code> to see the output. The <code>make</code> command will create the executable file <code>vmlinux.64</code> .
<i># verify the build is complete before continuing</i> <pre>host\$ wait host\$ make strip # go to the created vmlinux file host\$ cd kernel_2.6/linux host\$ sudo cp vmlinux.64 /tftpboot</pre>	See <i>SDK Tutorial</i> directions for <code>tftpboot</code> . Strip the <code>vmlinux.64</code> file to make it smaller. The <code>vmlinux.64</code> file is located in the <code>kernel_2.6/linux</code> directory.
9. Select Target IP Address, if Needed	
If a DHCP server is available, then selecting the target IP address is handled by the server. Otherwise, select a target IP address.	In this example, the target IP address is <code>192.168.51.159</code> .
10. Set the Development Target's IP Address	
10a. No DHCP Server	
First, set the IP address of the target (replace items in italic with your IP addresses). <i># Use your IP addresses instead of the example values!</i> <pre>target# setenv gatewayip 192.168.51.254 target# setenv netmask 255.255.255.0 target# setenv ipaddr 192.168.51.159 target# setenv serverip 192.168.51.1 # save the values so they will still be set after a reset target# saveenv</pre>	Note: <code>serverip</code> is the IP address of the TFTP server.
10b. DHCP Server Available	
If a DHCP server is available substitute the following step: <pre>target# dhcp</pre>	
11. Test the Ethernet Connection to the Host	
<i># use your host IP address instead of the example value!</i> <pre>target# ping 192.168.51.254</pre>	Expect to see: Using <code>octeth0</code> device host <code>192.168.51.254</code> is alive Note that the development target will <i>not</i> reply to a <code>ping</code> from the development host. Note: to see the development host's IP address, use the <code>/sbin/ifconfig</code> command on the development host.
<i>Continued in the next table...</i>	

Table 28: Build and Run Linux and linux-filter (SE-S), Part 3

Steps	Note
12. Download Linux to the Development Target	
# Use tftpboot to download the application. target# tftpboot 0 vmlinuz.64	See the <i>SDK Tutorial</i> directions for tftpboot. If this step does not work, check the /etc/xinetd.d/tftp file on the host to verify that server_args = -s /tftpboot.
13. Boot Linux	
target# bootoctlinux 0 coremask=0x2	This command will run vmlinuz.64 on core 1. Note: Linux will not run until core 0 comes out of reset (when the linux-filter SE-S application is downloaded).
14. Download the Application to the Development Target	
# Use tftpboot to download the application. target# tftpboot 0 linux-filter	See <i>SDK Tutorial</i> directions for tftpboot. If this step does not work, check the /etc/xinetd.d/tftp file on the host to verify that server_args = -s /tftpboot.
15. Boot the Application	
target# bootoct 0 coremask=0x1	This command will run linux-filter on core 0. Note: Linux will now run, and output a prompt on the target console. Type ls to verify Linux is up and running.
16. Load the Ethernet Driver	
# type the following command on one line target# modprobe cavium-ethernet pow_send_group=14 pow_receive_group=15	Type into the target console to load the Cavium Networks Ethernet Driver. (Type all the text on the left as one command line, omitting the target#.)
17. Configure the Interfaces	
# use your target IP address instead of the example value! target# ifconfig pow0 192.168.51.159 target# ifconfig eth0 promisc up	Substitute your IP address for 192.168.51.159.
<i>Continued in the next table...</i>	

Table 29: Build and Run Linux and linux-filter (SE-S), Part 4

Steps	Note
18. Check the Ethernet Configuration	<pre>target# ifconfig</pre> <p>Expect to see:</p> <pre>eth0 Link encap:Ethernet HWaddr 00:0F:B7:10:0C:16 inet6 addr: fe80::20f:b7ff:fe10:c16/64 <text omitted> lo Link encap:Local Loopback inet addr:127.0.0.1 Mask:255.0.0.0 inet6 addr: ::1/128 Scope:Host <text omitted> pow0 Link encap:Ethernet HWaddr 00:00:00:00:00:0E inet addr:192.168.51.159 Bcast:192.168.51.255 Mask:255.255.255.0 <text omitted></pre>
19. Ping the Target from the Development Host	<p># use your target IP address instead of the example value!</p> <pre>target# ping 192.168.51.159</pre> <p>Substitute your IP address for 192.168.51.159. This ping should succeed. The linux-filter application is forwarding the ping to Linux. In the Minicom session the following text will be visible:</p> <pre>PP0:~CONSOLE-> Received 98 byte packet. Sending to Linux. PP0:~CONSOLE-> Received 98 byte packet from Linux. Sending to PKO. PP0:~CONSOLE-> Received 98 byte packet. Sending to Linux. PP0:~CONSOLE-> Received 98 byte packet from Linux. Sending to PKO. <text omitted></pre>

Continued in the next table...

Table 30: Build and Run Linux and linux-filter (SE-S), Part 5

Steps	Note
20. Ping the Target from the Development Host, Using a Broadcast Ping <i># use your target IP address instead of the example value!</i> target# ping -b 192.168.51.159	The development target should not reply to the ping, only the development host will reply to the broadcast ping. The following text will be visible in the target console: PP0:~CONSOLE-> Received 92 byte packet. Filtered . PP0:~CONSOLE-> Received 92 byte packet. Filtered. PP0:~CONSOLE-> Received 92 byte packet. Filtered. (etc) Success! Linux filter has not passed the packet on to the slow (control) path Linux core.
21. Reset the Target Board To reset the development target, reboot Linux, or For <i>stand-alone</i> boards: power on or reset the board. For <i>PCI</i> boards: host\$ oct-pci-reset	Note: To run vmlinux.64 a second time, the board must be reset or power cycled because there is no other way to return to the bootloader prompt.

Note: After the **ifconfig** command, because there are normal Ethernet packets being transmitted, in addition to ping requests typed in by the user, expect to see some output of this form in addition to any generated by the ping commands. These printouts are normal.

```

target# ifconfig pow0 192.168.16.61
PP0:~CONSOLE-> Received 90 byte packet from Linux. Sending to PKO.
PP0:~CONSOLE-> Received 78 byte packet from Linux. Sending to PKO.
PP0:~CONSOLE-> Received 70 byte packet from Linux. Sending to PKO.
PP0:~CONSOLE-> Received 90 byte packet from Linux. Sending to PKO.
PP0:~CONSOLE-> Received 70 byte packet from Linux. Sending to PKO.
PP0:~CONSOLE-> Received 70 byte packet from Linux. Sending to PKO.

target# ifconfig eth0 promisc up device eth0 entered promiscuous mode
PP0:~CONSOLE-> Received 390 byte packet. Sending to Linux.
PP0:~CONSOLE-> Received 60 byte packet. Sending to Linux.
PP0:~CONSOLE-> Received 60 byte packet. Sending to Linux.
PP0:~CONSOLE-> Received 106 byte packet. Sending to Linux.

```

21 Hands-on: Run `linux-filter` as a Linux SE-UM Application

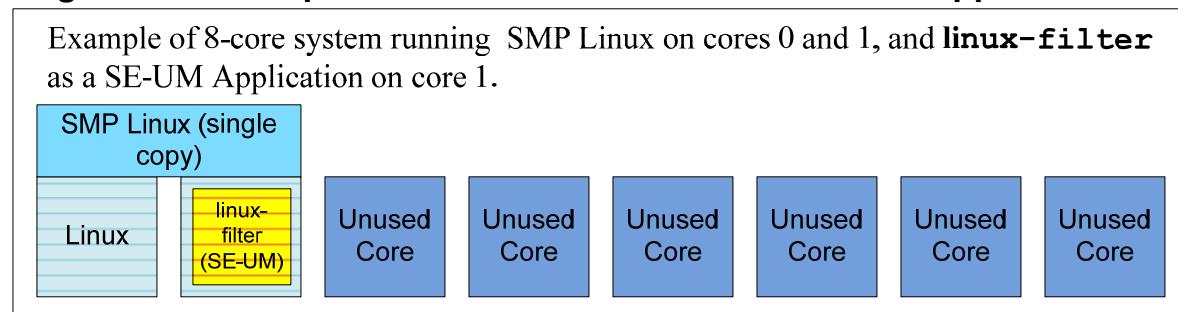
Previously, `linux-filter` was compiled to run as a Simple Executive Standalone application (SE-S). Now it will be run as a Simple Executive User-Mode Application (SE-UM) on Linux.

In this example, Linux is booted on cores 0 and 1, and then `linux-filter` is run on core 1 as a SE-UM application. The `linux-filter` application is started using the `oncpu` command. Details about the `oncpu` command are located in the *Software Overview* chapter.

To test `linux-filter`, first a ping is sent from the development host to the development target, and then a broadcast ping is sent from the development host to the development target. The first ping should be forwarded to Linux, the broadcast ping should be filtered.

Note that `linux-filter` cannot be run on the same core running the Cavium Networks Ethernet driver, so this exercise cannot be done if the OCTEON model has only 1 core.

Figure 16: Example of `linux-filter` run as a SE-UM Application on Linux



The `linux-filter` application was automatically compiled when the Linux kernel target was build. It has already been added to the embedded root filesystem. It can be found, after Linux is booted, in the `/examples` directory on the target. Note that when it is installed in the embedded root filesystem, the program name is changed to from `linux-filter-linux_64` to `linux-filter`.

On the target:

```

target# cd /examples
target# ls
busybox-testsuite  low-latency-mem      passthrough
crypto            named-block          testsuite
linux-filter       openssl-testsuite   zip

```

Table 31: Build and Run Linux and linux-filter (SE-UM), Part 1

Steps	Note
1. Connect the Hardware	
Connect serial cable to target console. Connect the Ethernet cable.	Follow directions in the <i>Quick Start Guide</i> . Note: The Ethernet cable is not needed to run Linux on a PCI target board, but will be needed later in the <i>SDK Tutorial</i> . Be careful to isolate the test network from the office network so that experiments will not disturb the office network.
2. Reset the board	
For <i>stand-alone</i> boards: power on or reset the board. For <i>PCI</i> boards: host\$ oct-pci-reset	The word Boot should appear on the red LEDs on the board. If not, something is wrong with the board.
3. Connect to the Target Console	
host\$ minicom -w ttys0	Substitute the serial port actually used on the host to connect to the OCTEON target board if it is not <i>ttys0</i> . Minicom will provide a connection to the target console. You should see the bootloader prompt.
4. Verify Bootloader Prompt is Visible	
target# version	The bootloader should reply with text similar to: U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun 13)
5. Verify Bootloader Version is at Least SDK 1.7	
If the bootloader's SDK version is not at least 1.7, then before continuing, upgrade the bootloader to a newer version.	Directions for upgrading the bootloader are included in the <i>SDK Tutorial</i> .
6. Build Linux	If this step was already done, no need to repeat it!
host\$ cd \$OCTEON_ROOT/linux	
host\$ sudo ls	Set the sudo password for 5 minutes
host\$ sudo make kernel >make.out 2>&1 &	Build the kernel and embedded root filesystem in the background. Use tail -f to see the output. The make command will create the executable file <i>vmlinu.x.64</i> .
7. Copy the ELF file to the tftpboot Directory	
# verify the build is complete before continuing host\$ wait host\$ make strip # go to the created vmlinu.x file host\$ cd kernel_2.6/linux host\$ sudo cp vmlinu.x /tftpboot	See <i>SDK Tutorial</i> directions for tftpboot . Strip the <i>vmlinu.x.64</i> file to make it smaller. The <i>vmlinu.x.64</i> file is located in the <i>kernel_2.6/linux</i> directory.
<i>Continued in the next table...</i>	

Table 32: Build and Run Linux and linux-filter (SE-UM), Part 2

Steps	Note
8. Select Target IP Address, if Needed	
If a DHCP server is available, then selecting the target IP address is handled by the server. Otherwise, select a target IP address.	In this example, the target IP address is 192.168.51.159 .
9. Set the Development Target's IP Address	
9a. No DHCP Server	
First, set the IP address of the target (replace items in italic with your IP addresses). <i># Use your IP addresses instead of the example values!</i> target# setenv gatewayip 192.168.51.254 target# setenv netmask 255.255.255.0 target# setenv ipaddr 192.168.51.159 target# setenv serverip 192.168.51.1 <i># save the values so they will still be set after a reset</i> target# saveenv	Note: <code>serverip</code> is the IP address of the TFTP server.
9b. DHCP Server Available	
If a DHCP server is available substitute the following step: target# dhcp	
10. Test the Ethernet Connection to the Host	
<i># use your host IP address instead of the example value!</i> target# ping 192.168.51.254	Expect to see: Using <code>octeth0</code> device host 192.168.51.254 is alive Note that the development target will <i>not</i> reply to a ping from the development host. Note: to see the development host's IP address, use the <code>/sbin/ifconfig</code> command on the development host.
11. Download Linux to the Development Target	
Use <code>tftpboot</code> to download the application. target# tftpboot 0 vmlinuz.64	See the <i>SDK Tutorial</i> directions for <code>tftpboot</code> . If this step does not work, check the <code>/etc/xinetd.d/tftp</code> file on the host to verify that <code>server_args = -s /tftpboot</code> .
12. Boot Linux	
target# bootoctlinux 0 coremask=0x3	This command will run <code>vmlinuz.64</code> on cores 0 and 1. Note: Linux will now run, and output a prompt on the target console. Type <code>ls</code> to verify Linux is up and running.
13. Load the Ethernet Driver	
<i># type the following command on one line</i> target# modprobe cavium-ethernet pow_send_group=14 pow_receive_group=15	Type into the target console to load the Cavium Networks Ethernet Driver. (Type all the text on the left as one command line, omitting the <code>target#</code> .)
<i>Continued in the next table...</i>	

Table 33: Build and Run Linux and linux-filter (SE-UM), Part 3

Steps	Note
14. Configure the Interfaces	
# use your target IP address instead of the example value! target# ifconfig pow0 192.168.51.159 target# ifconfig eth0 promisc up	Substitute your IP address for 192.168.51.159
15. Check the Ethernet Configuration	
target# ifconfig	<p>Expect to see:</p> <pre> eth0 Link encap:Ethernet HWaddr 00:0F:B7:10:0C:16 inet6 addr: fe80::20f:b7ff:fe10:c16/64 <text omitted> lo Link encap:Local Loopback inet addr:127.0.0.1 Mask:255.0.0.0 inet6 addr: ::1/128 Scope:Host <text omitted> pow0 Link encap:Ethernet HWaddr 00:00:00:00:00:0E inet addr:192.168.51.159 Bcast:192.168.51.255 Mask:255.255.255.0 <text omitted> </pre>
16. Start the SE-UM Application	
# Be sure to start linux-filter in the background by using the & character. # type the following command on one line target# oncpu 0x2 /examples/linux-filter &	<p>Expect to see:</p> <pre> Setting affinity to 0x2 CVMX_SHARED: 0x1201a0000-0x1201b0000 Active coremask = 0x2 </pre>
Continued in the next table.	

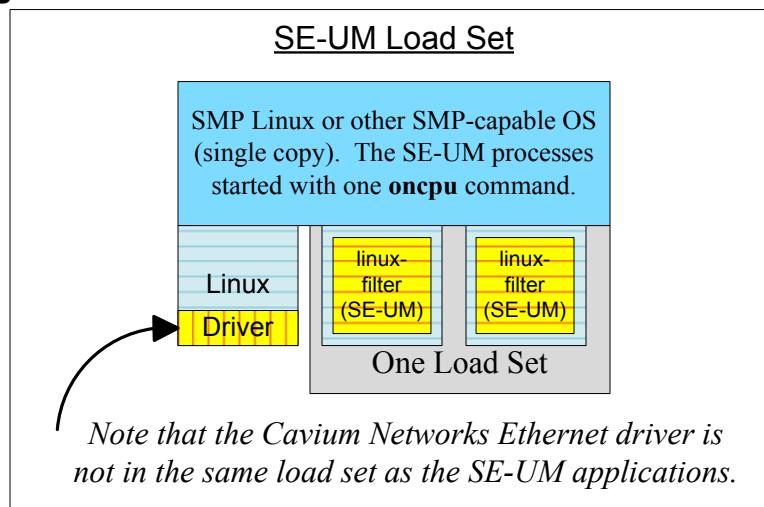
Table 34: Build and Run Linux and linux-filter (SE-UM), Part 4

Steps	Note
17. Ping the Target from the Development Host	<pre># use your target IP address instead of the example value! target# ping 192.168.51.159</pre> <p>Substitute your IP address for 192.168.51.159. This ping should succeed. The linux-filter application is forwarding the ping to Linux. In the Minicom session the following text will be visible:</p> <pre>PP0:~CONSOLE-> Received 98 byte packet. Sending to Linux. PP0:~CONSOLE-> Received 98 byte packet from Linux. Sending to PKO. PP0:~CONSOLE-> Received 98 byte packet. Sending to Linux. PP0:~CONSOLE-> Received 98 byte packet from Linux. Sending to PKO. <text omitted></pre>
18. Ping the Target from the Development Host, Using a Broadcast Ping	<pre># use your target IP address instead of the example value! target# ping -b 192.168.51.159</pre> <p>The development target should not reply to the ping, only the development host will reply to the broadcast ping. The following text will be visible in the target console:</p> <pre>PP0:~CONSOLE-> Received 92 byte packet. Filtered. PP0:~CONSOLE-> Received 92 byte packet. Filtered. PP0:~CONSOLE-> Received 92 byte packet. Filtered. (etc) Success! Linux filter has not passed the packet on to the slow (control) path Linux core.</pre>
19. Reset the Target Board	<p>To reset the development target, reboot Linux, or For stand-alone boards: power on or reset the board. For PCI boards: host\$ oct-pci-reset</p> <p>Note: To run vmlinux.64 a second time, the board must be reset or power cycled because there is no other way to return to the bootloader prompt.</p>

22 Hands-on: Run linux-filter as a SE-UM Application on Multiple Cores

The following figure shows `linux-filter` running on a load set of 2 cores of the 3 cores which are running Linux. This figure does not show the unused cores.

Figure 17: Run `linux-filter` as a SE-UM Load Set



In the steps shown in Section 21 – “Hands-on: Run `linux-filter` as a Linux SE-UM Application”, make the following changes:

Boot Linux on 3 cores (cores 0-2):

```
target# bootoctlinux 0 coremask=0x7
```

Note that `linux-filter` should not be run on the same core running the Cavium Networks Ethernet driver, so this exercise cannot be done if the OCTEON model has only 2 cores.

Start the Cavium Networks Ethernet Driver

```
target# modprobe cavium-ethernet pow_send_group=14 pow_receive_group=15
target# ifconfig pow0 192.168.51.159
target# ifconfig eth0 promisc up
```

Start `linux-filter` on cores 1 and 2:

```
# be sure to start linux-filter in the background using the & character
target# oncpu 0x6 /examples/linux-filter &
Setting affinity to 0x6
CVMX_SHARED: 0x1201a0000-0x1201b0000
Active coremask = 0x6
```

Load the Cavium Networks Ethernet driver using the modprobe command:

```
target# modprobe cavium-ethernet pow_send_group=14 pow_receive_group=15
Waiting for ethernet module to complete initialization...
Waiting for packets from port 16...
```

Type ps to verify linux-filter is running on two cores:

```
target# ps
<text omitted>
784 root      324 S  /examples/linux-filter
785 root      184 S  /examples/linux-filter
<text omitted>
```

Then perform the ping and ping -b commands from the host to test the example.

See Section 45 – “Appendix U: How to Find the Process’s Core Number” for an advanced hands-on step which can be taken at this point. This hands-on step is extra: it is not required for basic understanding.

23 Hands-on: Creating a Custom Application

The next step is to decide on the runtime model and software architecture for the custom application being developed.

Then select an example which does processing similar to the desired processing, such as linux-filter or passthrough, and copy that example to a new sub-directory. Use this as a base for the custom application to save time.

After copying the example, modify the Makefile as needed (for example, change the name of the source .c file to a new name). The custom application can then be built in the same way the examples are built.

Note that the hello example is too simple to use as a starting point for your own application. Choose a more complex example such as linux-filter or passthrough.

The Free Pool Allocator Chapter provides guidelines on configuring Simple Executive including creating custom FPA pools. Other API chapters will provide guidelines on configuring individual hardware units.

A small piece of code is provided with the on-line *OCTEON Programmer’s Guide* at the support site. This tiny application is used to demonstrate configuring the FPA. It is a simple example of creating a custom application by copying an existing example and modifying it. This example is too simple to use to develop a network application, but may be helpful in showing how to modify the Simple Executive configuration.

23.1 Adding Applications to the Embedded Root Filesystem

The examples are already added to the embedded root filesystem. To add a new application, copy the information for a different application and change the names to the new application. A

simplified FPA example is available at the Cavium Networks support site. That application can be installed in the `examples` directory on the development host. To add it to the Linux embedded root filesystem build, edit three files as show below. In the text below, the data for the `zip` example is copied and modified to specify the `fpa` example. The file text shown in this example is from SDK 1.8.

1. `$OCTEON_ROOT/examples/Makefile`

```
examples = application-args crypto debugger hello linux-filter \
low-latency-mem mailbox passthrough queue traffic-gen uart zip \
named-block fpa_simplified
```
2. `$OCTEON_ROOT/linux/embedded_rootfs/pkg_makefiles/sdk-examples.mk`

```
menu "SDK Examples"

config CONFIG_sdk-examples
    bool "Include SDK Examples"
    default y
    help
        Include the SDK examples

<text omitted>
ifdef SDK_EXAMPLES_ZIP
install: build_SDK_EXAMPLES_ZIP
.PHONY: build_SDK_EXAMPLES_ZIP
build_SDK_EXAMPLES_ZIP:
    ${MAKE} -C ${OCTEON_ROOT}/examples/zip OCTEON_TARGET=${ABI}
config/cvmx-
config.h
    ${MAKE} -C ${OCTEON_ROOT}/examples/zip OCTEON_TARGET=${ABI}
endif
ifdef SDK_EXAMPLES_FPA
install: build_SDK_EXAMPLES_FPA
.PHONY: build_SDK_EXAMPLES_FPA
build_SDK_EXAMPLES_FPA:
    ${MAKE} -C ${OCTEON_ROOT}/examples/fpa_simplified \
OCTEON_TARGET=${ABI} config/cvmx-
config.h
    ${MAKE} -C ${OCTEON_ROOT}/examples/fpa_simplified \
OCTEON_TARGET=${ABI}
Endif
<text omitted>
.PHONY: install
install:
    mkdir -p ${ROOT}/examples
<text omitted>
ifdef SDK_EXAMPLES_ZIP
    cp ${OCTEON_ROOT}/examples/zip/zip-${ABI} ${ROOT}/examples/zip
endif
ifdef SDK_EXAMPLES_FPA
    cp ${OCTEON_ROOT}/examples/fpa_simplified/fpa-${ABI} \
${ROOT}/examples/fpa
endif
```

3. \$OCTEON_ROOT/linux/embedded_rootfs/pkg_kconfig/ 74-sdk-examples.kconfig:

```

<text omitted>
    bool "zip"
    depends CONFIG_sdk-examples
    default y

    config SDK_EXAMPLES_FPA
    bool "fpa"
    depends CONFIG_sdk-examples
    default y

```

The new item should appear when using the make menuconfig command in \$OCTEON_ROOT/linux/embedded_rootfs:

```

host$ cd $OCTEON_ROOT/linux/embedded_rootfs
host$ sudo make menuconfig
    Global Options --->
        [*] device-files
        [*] busybox
        [*]   Include the Busybox testsuite
        [*] init-scripts
            NFS Root filesystem --->
<text omitted>
        [*] mtd-tools
            SDK Examples --->
        [*] lockstat
<text omitted>

```

When SDK Examples is selected, the next menu shows the new item:

```

[*] Include SDK Examples
    [*] intercept-example
    [*] crypto
    [*] named-block
    [*] passthrough
    [*] linux-filter
    [*] low-latency-mem
    [*] zip
    [*] fpa_simplified (NEW)

```

Then rebuild the embedded root filesystem, by invoking the make kernel target in the \$OCTEON_ROOT/linux directory.

To verify that the new application is built correctly:

```

host$ grep examples/fpa make.out
make -C /home/testname/sdk18/examples/fpa OCTEON_TARGET=linux_64
config/cvmx-config.h
make[4]: Entering directory `/home/testname/sdk18/examples/fpa'
make[4]: Leaving directory `/home/testname/sdk18/examples/fpa'
make -C /home/testname/sdk18/examples/fpa OCTEON_TARGET=linux_64
make[4]: Entering directory `/home/testname/sdk18/examples/fpa'
make[4]: Leaving directory `/home/testname/sdk18/examples/fpa'
cp /home/testname/sdk18/examples/fpa/fpa-linux_64
/tmp/root-rootfs/examples/fpa

```

Verify that it is installed in the build system correctly:

```
host$ cd /tmp/root-rootfs/examples
host$ ls
busybox-testsuite  fpa          low-latency-mem  openssl-testsuite  testsuite
crypto              linux-filter    named-block      passthrough     zip
```

To verify the new program is in the embedded root filesystem without taking the time to boot Linux, see Section 38 – “Appendix N: Contents of the Embedded Root Filesystem”.

Then boot Linux. After Linux is booted, run the **fpa** example.:

```
target# cd /examples
target# ls
busybox-testsuite  linux-filter      openssl-testsuite  zip
crypto              low-latency-mem  passthrough
fpa               named-block      testsuite
target# oncpu 0 ./fpa
<The example should run correctly.>
```

Detailed instructions may be found in the SDK Document “*Linux on the OCTEON*”, in the section “*How to Add a Package*”. This document includes information on how to add custom libraries as well as a custom application.

23.2 Example Application Which Breaks Ethernet Driver

The Cavium Networks Ethernet driver configures the SSO, FPA, CIU, PIP, IPD, PKO, and FAU. Applications which reconfigure these units will cause the Ethernet driver to stop working.

The FPA example is a perfect example of an application which reconfigures hardware already configured by the Cavium Networks Ethernet driver (in this case, only the FPA). This reconfiguration will cause the Ethernet driver to stop working, as shown in the following steps.

After booting Linux (with the FPA example included), run the FPA example:

```
target# cd /examples
target# oncpu 0 ./fpa
<the example should run correctly>
```

Then configure the Ethernet. In this example, a DHCP server is available. The IP address returned by the DHCP server is shown in bold print in the example below:

```
target# modprobe cavium-ethernet
target# udhcpc -i eth0
udhcpc (v1.2.1) started
Jan 1 00:05:51 (none) local0.info udhcpc[753]: udhcpc (v1.2.1) started
Sending discover...
Jan 1 00:05:51 (none) local0.debug udhcpc[753]: Sending discover...
Sending select for 192.168.51.173...
Jan 1 00:05:51 (none) local0.debug udhcpc[753]: Sending select for
192.168.51.1
73...
Lease of 192.168.51.173 obtained, lease time 86400
Jan 1 00:05:51 (none) local0.info udhcpc[753]: Lease of 192.168.51.173
obtained
, lease time 86400
```

```

deleting routers
SIOCDELRT: No such process
adding dns 192.168.51.254

```

Ping the development target from the development host:

```

# this ping should succeed
host$ ping 192.168.51.173

```

Run the FPA example – there should be a lot of errors, for example:

```

ERROR: cvmx_fpa_shutdown_pool: Illegal address 0x2e1f8880 in pool
(null) (2)

```

Ping the development target from the development host again:

```

# this ping should fail
host$ ping 192.168.51.173

```

24 The Hardware Simulator

A hardware simulator can be run on an x86 Linux host. The simulator is most useful in running SE-S applications. The Linux kernel may also be run on the simulator, but it is often impractical to run SE-UM applications on the simulator because the simulator is too slow. The simulator is an essential tool for SE-S application performance optimization. See Section 24.6 – “Using the Simulator to Optimize Performance” for more information. See also the *Software Debugging Tutorial* chapter, in the section on debugging on the hardware simulator for more information.

There are separate simulator binaries for different OCTEON models, but this information is hidden when using the script `oct-sim` or `oct-linux`. These scripts get the OCTEON model number from the `OCTEON_MODEL` environment variable (set by `env-setup`), then call the appropriate simulator binary.

When the simulator is started, it will run the bootloader. The bootloader commands are located in a file, and the filename is specified on the simulator command line. Once the bootloader is running, the simulator will use the commands in the file to boot the application.

24.1 Simulator Documentation

More information about the simulator can be found in the following SDK documents:

Table 35: Simulator Documentation

SDK Document
<i>Simple Executive Debugger</i>
<i>Linux on the OCTEON - Running Linux on the Simulator</i>
<i>OCTEON Bootloader - Simulator Specific Usage</i>

24.2 Run SE-S Applications on the Simulator

When using the simulator, start the simulator using the `oct-sim` command to create an easier debugging environment. This command will open a console session with scroll bars to allow the user to review session activity which has scrolled off the screen.

The example `hello` is a simple way to see a SE-S application run on the simulator.

The `hello` Makefile has two targets which will run `hello` on the simulator. After the application is built, the command `make run` will run `hello` on the simulator, simulating 1 core. The command `make run4` will run `hello` on the simulator, simulating 4 cores.

```
host$ make run4
```

The last part of the output should be:

```
<text omitted>
PP1:~CONSOLE-> Hello world!
PP3:~CONSOLE->
PP2:~CONSOLE-> Hello world!
PP3:~CONSOLE->
PP1:~CONSOLE-> Hello example run successfully.
PP3:~CONSOLE-> Hello world!
PP2:~CONSOLE-> Hello example run successfully.
PP0:~CONSOLE-> Hello world!
PP3:~CONSOLE-> Hello example run successfully.
PP0:~CONSOLE-> Hello example run successfully.
PP0: stopping due to BREAK instruction

SIMULATION COMPLETE at cycle (approximate instruction) 753958 (0 global
stop phases)
```

Note that PP0, PP1... stand for “Packet Processor” 0, etc. These are the printouts from the different cores on the OCTEON processor.

The `$OCTEON_ROOT/examples/hello/Makefile` contains the `run` and `run4` targets. The value of `$TARGET` is `hello`:

```
run: $(TARGET)
      oct-sim $(TARGET) -quiet -noperf -numcores=1

run4: $(TARGET)
      oct-sim $(TARGET) -quiet -noperf -numcores=4
```

Note: Not all example Makefiles support these targets.

Try this from the command line:

```
host$ cd $OCTEON_ROOT/examples/hello
host$ oct-sim hello -quiet -noperf -numcores=4
```

(At the end of the simulator output, type `Enter` a couple of times to see the bash prompt on the screen.)

Note that the output of hello is seen in the same terminal session used to start the simulator. This is not true for Linux.

24.3 Specifying `-noperf` and `-quiet` to Speed Up Processing

When using the simulator, `-noperf` simulator option tells the simulator to not perform cycle-accurate timing. This option is not required, but is especially useful when simulating a large number of cores.

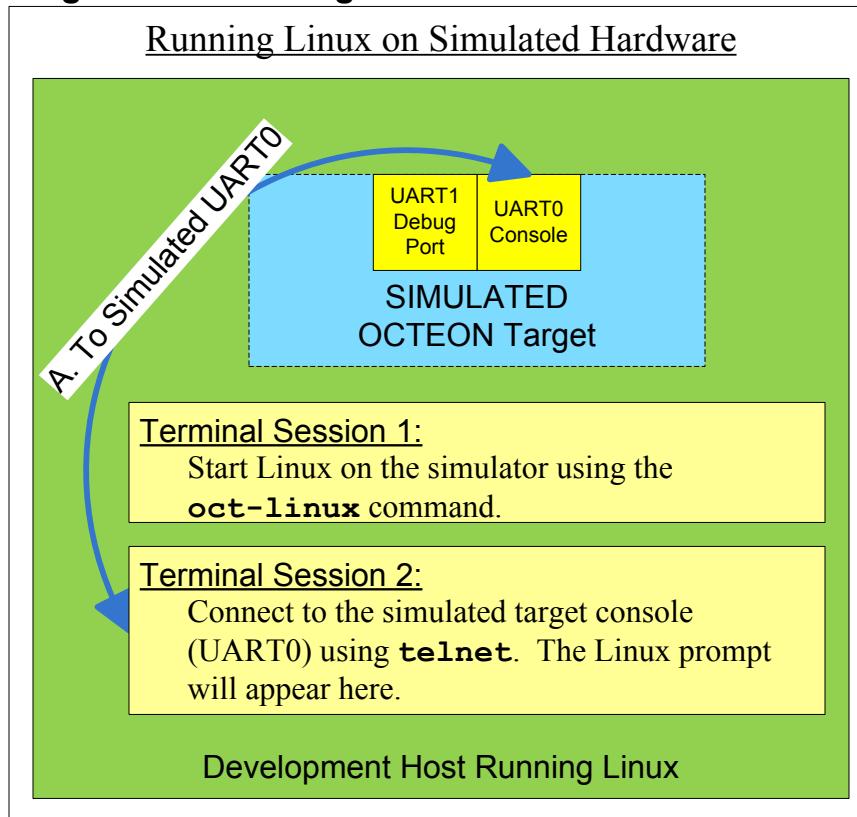
Similarly, the `-quiet` option will reduce the amount of verbose output and turn off tracing if tracing is not specifically requested.

The `oct-sim` commands shown in `examples/hello/Makefile` both specify the `-noperf` option.

24.4 Running Linux on the Simulator

It is easiest to use the `oct-linux` command to run Linux on the simulator. This command will open a console session with scroll bars to allow the user to review session activity which has scrolled off the screen.

Figure 18: Running Linux on Simulated Hardware



24.4.1 Build Linux to Run on the Simulator

Build Linux to run on the simulator, sending the `make` output to the file `make.out` and examining `make.out` using the command `tail` (type `Ctrl-C` to exit the `tail` utility).

```
host$ cd $OCTEON_ROOT/linux
```

```
host$ sudo make clean
host$ sudo ls
host$ sudo make sim >make.out 2>&1
```

To see the output of the make command, in a different terminal session, type

```
host$ cd $OCTEON_ROOT/linux
host$ tail -f make.out
```

24.4.2 Run Linux and SE-UM Applications on the Simulator

To start Linux on the simulator, first open two terminal sessions. One will be used to run the simulator and the other will connect to the simulated target console.

The script used to start the simulator, oct-linux, is located in the linux/kernel_2.6 directory. This script calls oct-sim with the arguments needed by Linux:

```
#!/bin/bash

memory=384
uart=2020
packet_port=2000

oct-sim linux/vmlinuz.64 -envfile=u-boot-env -memsize=${memory} -
uart0=${uart} -
serve=${packet_port} -ld0x40000000:../embedded_rootfs/rootfs.ext3 $*
```

Note that oct-linux is not in the usual PATH:

```
host$ pwd
/home/testname/sdk
host$ oct-linux
bash: oct-linux: command not found
host$ find . -name oct-linux -print
./linux/kernel_2.6/oct-linux
```

Table 36: Run Linux on the Hardware Simulator, Part 1

Steps	Note
1. Build the Kernel with Embedded Root Filesystem	If this step was already done, no need to repeat it!
<pre>host\$ cd \$OCTEON_ROOT/linux # enter password now so it is stored for the next command host\$ sudo ls # type the following text on one line host\$ sudo make -s sim >make.out 2>&1 &</pre> <p># Note: This step can take up to 20 minutes. When it has completed successfully, the kernel_2.6/linux/vmlinux file will have been created.</p>	Note: The sudo ls command is merely used to set the root password. The root password will be stored for about 5 minutes for use in the next command (sudo make....).
# verify the build is complete before continuing host\$ wait	
2. Create a Total of Two Terminal Sessions on the Development Host	
Create two terminal sessions on the host, for these three purposes: 1. run the simulator 2. connect to the target console	
3. Boot the Kernel on the OCTEON Simulator	
First Terminal Session (hostT1\$): <pre>host\$ cd \$OCTEON_ROOT/linux/kernel_2.6 # type the following text on one line hostT1\$ oct-linux -noperf -quiet -numcores=1</pre> <p>Note: The following error will occur if the <code>make sim</code> step was omitted:</p> <p>Error: Unable to open binary file: <code>./embedded_rootfs/rootfs.ext3</code></p>	This command will run Linux on the OCTEON simulator. Expect to see: Loading u-boot environment from file: u-boot-env mem size is 384 Megabytes Using simulator: cn38xx-simulator Loading linux/vmlinux.64 to boot bus address 0x1000000 Starting simulator.... <text omitted> Live Packet Listening at pak.caveonetworks.com, port 2000 (0x7d0) Uart Listening at pak.caveonetworks.com, port 2020 (0x7e4) waiting for a connection to uart <text omitted>
<i>Continued in the next table...</i>	

Table 37: Run Linux on the Hardware Simulator, Part 2

Steps	Note
4. Connect to the target console	
<pre>Second Terminal Session (hostT2\$): hostT2\$ telnet localhost 2020</pre> <p>Note: The following error will occur if the telnet to 2020 is started before the simulator "waiting for a connection to uart 0 1" message appears:</p> <pre>Trying 127.0.0.1... telnet: connect to address 127.0.0.1: Connection refused</pre>	<p>This command will connect to the simulated target console. Note: Wait until "waiting for a connection to uart 0 1" appears in the simulator output, but don't wait too long. The simulator will timeout after about 45 seconds if no console connection is made.</p> <p>Expect to see:</p> <pre>argv[2]: coremask=0x1 argv[3]: debug ELF file is 64 bit Attempting to allocate memory for ELF segment: addr: 0xfffffffff80100000 (adjusted to: 0x00000000000100000), size 0x1a99d58 <text omitted> Bootloader: Done loading app on coremask: 0x1 <Linux will stop here></pre>
The Linux prompt will now come up on the simulated target console.	

Eventually, an interactive shell will appear in the second terminal session. “Eventually” can be 5-20 minutes, depending on the speed of the host processor, the amount of memory installed, and the number of cores requested.

Once Linux boots to the interactive shell prompt, it can be useful to change `telnet` to character mode instead of line mode. In the `telnet` session, press `Ctrl-J` and enter mode `char` at the prompt. Then hit `Enter` a few times. Shell tab completion and other interactive aspects should now work.

Once the interactive shell prompt appears, a SE-UM application, such as `named-block`, can be run on the simulator.

24.5 Simulator: Download and Run Bootloader

When running an application on the simulator instead of hardware, the bootloader is automatically invoked by the `oct-sim` script which starts the simulator.

Boot commands and application arguments are passed to the bootloader via an environment file. The environment file is specified by the `-envfile` option to `oct-sim`.

See the SDK document “*OCTEON Bootloader*” for more information about booting, downloading, and running on the simulator.

24.6 Using the Simulator to Optimize Performance

Viewzilla is the best tool to analyze the code and optimize the performance, but this tool only runs along with the simulator (on the output of the simulator).

When writing code for the Simple Executive, the programmer should try to make sure the code also runs on the simulator through the entire development process. It is common to abandon the simulator when the actual hardware becomes available. Then, when performance testing becomes important near the end of the project, the code may no longer work on the simulator. This means the key performance analysis tool, Viewzilla, will not work.

There are a couple of reasons why code will not run on the simulator. Some of them are simple, while the others may be more complicated.

First, make sure the software can be configured to a minimal configuration. When doing performance testing with the simulator, use a minimal setup (limited number of ports, etc), which can pass packets. This minimal setup can then be analyzed with the simulator and Viewzilla.

The most common issue is a hardware setup which is difficult to replicate under the simulator (more ports, PCI etc). The simulator does not support PCI. To simulate PCI, modify the code to mimic PCI messages.

If PCI is used for initialization, provide some hard-coded initialization values which could be used under the simulator instead. This solution is very application specific.

The remainder of the information in this chapter is for reference only! It will be in the Appendix of the final book.

25 Appendix A: Introduction to Available Products

In this section the various products which can be installed are introduced.

If an installation CD is not available, the SDK may be downloaded from the support site. See instructions in Section 27.1 – “Installing from the Support Site Instead of a CD”. The information in the next section which introduces the various files available is useful even if the installation CD is not available.

Note that only 3 files need to be installed to follow the directions in this tutorial: the base SDK, OCTEON Linux, and the *Quick Start Guide*.

The RPM file naming convention is:

<Product>-<rel><build>.i386.rpm

Where *<rel>*= release number and *<build>* is the build number.

For instance, SDK 1.7.3 contains the base SDK (OCTEON-SDK) product file:

OCTEON-SDK-1.7.2-244.i386.rpm

Note: This file was first released with SDK 1.7.2, and not changed for the 1.7.3 release, so the file’s *Product* is 1.7.2.

25.1.1 Product Files on the Installation CD

The installation CD contains the SDK files: the base SDK, OCTEON Linux, the *Quick Start Guide*, and other product files which may or may not be needed depending on your application. The SDK files and optional product files may also be downloaded from the support site. Each file which ends in .rpm is referred to as an RPM (Red Hat Package Manager) file, *package*, or *product*.

At a minimum, install the base SDK and the *Quick Start Guide*. These are needed by everyone.

The following optional support is provided by RPMs on the CD:

- OCTEON Linux: To use Linux on any OCTEON cores, install the OCTEON Linux package after installing the base SDK. OCTEON Linux is needed to run some of the examples in this chapter.
- Software support for optional hardware capabilities on the OCTEON processor: Crypto, ZIP, DFA. The OCTEON model must provide these optional hardware capabilities.
- Software which enables a PCI host to communicate with OCTEON PCI targets by sending and receiving packets over the PCI bus.

In order to the examples in this tutorial, the base SDK and OCTEON Linux package are required. This tutorial only introduces these packages.

See the table below for the descriptions of the various packages available on the CD. The far right column specifies whether the development target cores will be running Simple Executive standalone or Linux.

Table 38: Packages Provided on the Installation CD, Part 1

Packages on the CD	Brief Description	SE-S or Linux
Basic Items		
SDK-QSG	The <i>Quick Start Guide</i> for the SDK Needed by everyone.	Both
SDK	The base SDK. Needed by everyone.	Both
LINUX	This is OCTEON Linux. This package is needed to run Linux on the OCTEON cores. This version of Linux has been ported to take advantage of OCTEON features. This package may be called "Linux Distro" on the Support Site.	Linux only
Special Hardware Capability Support		
CRYPTO-CORE	Requires an OCTEON model with cryptographic functions enabled. This package provides the software which runs on the core and performs hardware crypto functions.	Both
DFA	Requires an OCTEON model with a DFA hardware unit. This package includes a DFA compiler that runs on the i386 or x86_64 development host, and two examples of how to use Simple Executive functions to access the DFA Engine on OCTEON.	Both
ZLIB-CORE	Requires an OCTEON model with a ZIP hardware unit. The OCTEON must run a SE-S application on at least one core. This package provides the software which runs on the core and uses the ZIP hardware unit.	SE-S only
ZLIB-LINUX	Requires an OCTEON model with a ZIP hardware unit. The OCTEON must run OCTEON Linux on at least one core. This package provides the software which runs on a core and uses ZIP hardware unit.	Linux only.
Continued in the next table...		

Table 39: Packages Provided on the Installation CD, Part 2

Packages on the CD	Brief Description	SE-S or Linux
Support for Linux PCI Host passing packets to OCTEON PCI target over PCI bus		
COMPONENTS-COMMON	The header files needed by PCI RPMs and the toolkits.	Both
PCI-BASE	Requires a PCI host and an OCTEON PCI target. This software enables a PCI host to communicate with OCTEON PCI targets by sending and receiving packets over the PCI bus. The PCI host must be an i386, x86_64, PPC or OCTEON PCI host running Linux. This software includes a Linux driver which can be installed on the PCI host to implement the host-side of this capability. This is the base PCI driver which other PCI drivers depend upon.	Both
PCI-NIC	Requires a PCI host and an OCTEON PCI target. Includes an application which runs on the OCTEON target board, plus a driver to run on the Linux host to provide networking interfaces corresponding to the physical ports on the OCTEON NIC. When you install this, special devices are created on the host system which look like network interfaces to the host (Ethernet devices).	Both
Support for Linux PCI Host offloading crypto or ZIP work to OCTEON over PCI		
CRYPTO-HOST	Requires a PCI host and an OCTEON PCI target with crypto hardware capability. This software allows the PCI host to use the crypto hardware capabilities on the OCTEON PCI target. (The PCI host offloads the encrypt and decrypt (crypto) functions to the OCTEON PCI target by passing packets to OCTEON over the PCI bus.)	Both
PCI-CNTQ	Requires a PCI host and an OCTEON PCI target with either ZIP or crypto hardware capability. Install on a i386, x86_64, PPC or OCTEON PCI host. This software implements an enhancement to the base PCI packet interface: the Common Command Queue (CCQ). The ZLIB-HOST and CRYPTO-HOST packages depend on this package.	Both
ZLIB-HOST	Requires a PCI host and an OCTEON PCI target with a ZIP hardware unit. This software allows the PCI host to use the ZIP hardware unit on the OCTEON PCI target. (The PCI host offloads compression and decompression (ZIP) functions by passing packets to OCTEON over the PCI bus.)	Both

The files available on the installation CD are listed in the following table.

Table 40: Example RPMs on the SDK Installation CD

RPM Files
OCTEON-COMPONENTS-COMMON-* .i386.rpm
OCTEON-CRYPTO-CORE-* .i386.rpm
OCTEON-CRYPTO-HOST-* .i386.rpm
OCTEON-DFA-* .i386.rpm
OCTEON-LINUX-* .i386.rpm
OCTEON-PCI-BASE-* .i386.rpm
OCTEON-PCI-CNTQ-* .i386.rpm
OCTEON-PCI-NIC-* .i386.rpm
OCTEON-SDK-* .i386.rpm
OCTEON-SDK-QSG-* .pdf
OCTEON-ZLIB-CORE-* .i386.rpm
OCTEON-ZLIB-HOST-* .i386.rpm
OCTEON-ZLIB-LINUX-* .i386.rpm
(* is the release number for each package)

Some of these packages depend on others. These dependencies can change with newer releases. The dependencies for each package are always listed along with their descriptions on the support site.

The dependencies for SDK 1.7.3 are shown in the following table.

Table 41: Package Dependencies

Packages on CD	Prerequisites
Base SDK	None
COMPONENTS-COMMON	Base SDK
CRYPTO-CORE	Base SDK
CRYPTO-HOST	Base SDK, COMPONENTS-COMMON, PCI-BASE, CRYPTO-CORE
DFA	Base SDK
LINUX-DISTRO	Base SDK
PCI-BASE	Base SDK, COMPONENTS-COMMON
PCI-CNTQ	Base SDK, COMPONENTS-COMMON, PCI-BASE
PCI-NIC	Base SDK, COMPONENTS-COMMON, PCI-BASE
ZLIB-CORE	Base SDK
ZLIB-HOST	Base SDK, COMPONENTS-COMMON, PCI-BASE, ZLIB-CORE
ZLIB-LINUX	Base SDK, ZLIB-CORE, LINUX-DISTRO

25.1.2 Toolkits

In addition to these packages, various optional software toolkits are available. These toolkits are designed to simplify application development under Linux. To use them, first download the base SDK and the OCTEON Linux SDK. Note that if you plan to use a toolkit it is important to start with the correct SDK. The toolkits are not ported to all of the SDKs. The toolkits have all been ported to SDK 1.7.3. Contact your Cavium Networks sales representative for more information.

Table 42: Optional Toolkits

Toolkit	Description
OCTEON IPSEC stack	Provides hardware-accelerated IPSEC stack.
OCTEON L2 and IP stack	Provides Ethernet frame and IP processing. Also provides an IP forwarding example application.
OCTEON SSL stack	Provides hardware-accelerated SSL stack.
OCTEON TCP/IP stack	Provides TCP/IP stack for OCTEON Simple Executive applications.

25.1.3 Optional Application Development Kits (ADKs)

Application Development Kits (ADKs) are designed to speed product development under Linux by providing pre-ported software which runs on top of the SDK. The ADKs are targeted to specific types of applications, such as router-bridge, or gateway.

Note that if you plan to use an ADK it is important to start with the correct SDK. The ADKs are not ported to all of the SDKs. First determine the highest-numbered ADK available, and then download both the ADK and the matching base SDK and OCTEON Linux SDK from the support site at <http://www.caviumnetworks.com/>.

Table 43: Application Development Kits (ADKs)

ADK	Contents
Router-bridge binary package	IP forwarding, Network-Attached Storage (NAS)
Gateway binary package	Triple play, security, and media server
Source package	Open-source software for both binary packages. See Note 1.

Note 1: For source to third-party products, contact your Cavium Networks sales representative.

The ADK software is primarily open-source software, but some third-party software is also included in binary-only form. This third-party software is provided for evaluation purposes only; contact your Cavium Networks sales representative for third-party licensing information. The open-source part of the ADKs is included with the SDK license, and source code is provided in the source package.

25.1.4 Product Updates

Note that updates to these files may be available at the support site, accessible from <http://www.caviumnetworks.com/>. Registered support site users who have requested updates will receive email when updates are released.

26 Appendix B: Linux Basics

26.1 Linux Commands

The following is a list of very commonly used Linux commands. These commands are typed in at the shell prompt. This list is provided for persons new to Linux. If the man pages are installed on your development host, then documentation may be accessed via the `man` utility.

Linux files are divided into two basic types: ASCII (human readable text) and Binary (machine readable). Some file commands (such as `cat` or `diff`) are designed to work on ASCII; others (such as `cmp` (compare)) are designed to work on binary. If you are not sure of the file type, you can simply try it, or you can type `file <filename>` to determine the file type. Note that if you `cat` a binary file to the screen, you may need to log out and back in to fix the screen.

To stop a command, use `Ctrl-C` (hit the `Ctrl` key, and while holding it down, hit the “`C`” key. Note the key does not have be a capital letter `C`.

When using the `bash` shell, to recall a command typed before, use the arrow keys. For instance, use the up-arrow key to see prior commands. Type it several times: each time the key is hit, an older command is recalled. To edit the command, use the arrow keys to move left and right. Use the backspace key to delete while moving left. To add characters, go to the place where they should be added, using the arrow keys, then type in the new text.

To show the list of most recently executed commands, type `history`. In `bash`, you can use the up-arrow and down-arrow keys to recall prior commands so that the entire command doesn't have to be typed in again (this is only one of many speed techniques available).

The `\` (back slash) character is used to continue a line and is often used in Makefiles. This character tells the shell or Makefile to *escape* the next character. When the next character is a carriage return (end of line), the `\` will cause the carriage return to be ignored. Back slash can also be used in shell scripts or on the command line, for example:

```
# Just type "echo hello" then press Enter and type "world": bash
# will add the > character.
host$ echo hello \
> world
hello world
```

Note that the `\` character must be placed at the *end* of the line. It only "escapes" the *next* character.

See also: <http://www.unix-manuals.com/quicktips> (search for “`tail -f`” Linux command).

Table 44: Linux Commands Quick Reference, Part 1

Command	Brief Description
'	"Back quote". This character is used in commands such as <code>echo `pwd`</code> . When used this way, the value of the quoted command (<code>pwd</code>) is returned, such as <code>/home/testname/sdk</code> . Note this character is not the same as " ' " (the single quote).
#	Comment delimiter. Used in shell scripts.
	"Pipe". This will take the output of a prior command and redirect it to be the input to the following command. For example: <code>rpm -q --all more</code> .
*	"Star". This is a "wildcard" character. <code>ls *.txt</code> will show all files in the current directory which end with the string ".txt". The command <code>rpm -i *.rpm</code> will install all the files in the current directory which end with the string ".rpm".
~	"Tilde". Refers to the user's home directory, as in <code>cp filename ~/sdk18</code> .
alias	Used to set a string to a special meaning. For example, a series of commands may be aliased to a convenient short name. After the commands are aliased, only the short name is typed. This is very convenient for often-repeated commands. To see aliases currently on the computer, type <code>alias</code> .
bash	Call the <code>bash</code> shell. This is a fast way to change from using another shell to using <code>bash</code> . The correct way to change the login shell is to have it changed in the <code>passwd</code> file where the per-user login information is stored.
cat	Send the contents of the file to the screen, as in <code>cat octeon-models.txt</code> . Note if the file is not an ASCII file, the characters sent to the screen will not have meaning. Also, if the file is not an ASCII file, the terminal session's configuration may be set to new and incorrect values, causing the output on the screen to be unreadable. In that case, try the commands <code>clear</code> , <code>stty sane</code> , or simply exit the terminal session to recover.
cd	Change directory. Remember that, under Linux, directories are separated by "/" not "\ ", as in <code>cd /bin</code> . The command <code>cd ..</code> (cd dot dot) goes up one level. The command <code>cd</code> without any argument changes to the user's home directory.
chgrp	Used to change the group ownership of a file. For instance, <code>chgrp staff doit.sh</code> will change the group ownership of <code>doit.sh</code> to <code>staff</code> . The file attributes can be seen using the <code>ls -l</code> command.
chmod	Used to change the mode of a file. For instance, <code>chmod +x doit.sh</code> will make the file <code>doit.sh</code> executable. The file attributes can be seen using the <code>ls -l</code> command.
chown	Used to change the owner of a file. For instance, <code>chown testname doit.sh</code> will change the owner of <code>doit.sh</code> to <code>testname</code> . The file attributes can be seen using the <code>ls -l</code> command.
clear	Used to clear off the screen, especially before starting a new experiment.
cmp	Compare two binary files.
diff	Compare two ASCII files.
echo	Display a string to the screen. Examples: <code>echo "hello world"</code> to display the string <code>hello world</code> on the screen, or <code>echo \$PATH</code> to show the value of the <code>PATH</code> variable. Use <code>echo \$SHELL</code> to see which shell you are running (<code>bash</code> , <code>csh</code> , <code>ksh</code> , etc.).
<i>Continued in the next table...</i>	

Table 45: Linux Commands Quick Reference, Part 2

Command	Brief Description
export	Make an environment variable's definition accessible to all child processes. Without <code>export</code> , a child process will not have the variable's definition. For example <code>host\$ JUNK="my junk"; echo \$JUNK; bash</code> . Once the new shell is spawned, type <code>host\$ echo \$JUNK</code> . Then try the same, adding <code>export</code> as in <code>\$host export JUNK=my junk</code> .
fg	This command is used to bring a background process into the foreground, usually so that the process may be aborted using Ctrl-C. For example, the command <code>sudo make kernel >make.out 2>&1 &</code> puts the Linux build process into the background. Typing <code>fg</code> will bring the build process into the foreground.
file	Show the type of the file.
find	Locate items such as files and/or directories. For example <code>find . -name "* .txt"</code> will find all the files in the directory, recursively, which end with the suffix <code>.txt</code> . The command <code>find . -type d</code> will find all the directories in the directory, recursively.
gedit	This editor may be used by users not familiar with the <code>vi</code> or <code>emacs</code> editors. The <code>gedit</code> editor may be used from the host system console, or by VNCing (Virtual Network Computing) from another computer to the host system.
grep	Search in ASCII files for a particular string. As an alternative, the command <code>egrep</code> allows multiple strings and other special searches.
head	This command displays the text at the beginning of an ASCII file.
history	Show previously typed commands.
ifconfig	Configure Ethernet devices; show current configuration. Path = <code>/sbin/ifconfig</code> .
info	Display detailed information about some commands, such as <code>vi</code> .
kill	Used to kill a process. Locate the process to be killed by using the <code>ps</code> command.
less	Display to the screen, one page at a time, with controls to allow the user to scroll up as well as down. Use the up-arrow to go up, the down-arrow to go down. The <code>vi</code> command <code>1G</code> will go to the start of the file. The <code>vi</code> command <code>G</code> will go to the end of the file. Use the <code>vi</code> command <code>:q</code> (colon q) to quit. If the commands aren't working right, type <code>Esc</code> (Escape key) to exit input mode. After typing <code>Esc</code> the commands should work again. To search for a string in the file, type <code>:/</code> (colon forward-slash) followed by the search string, then <code>Enter</code> .
ln	The command <code>ln</code> (link) is used to create a link between two files. Linking allows the same file to have two different names or locations in the file system. The command <code>ln -s</code> can be used to <i>symbolically</i> link one file or directory to another. For example, the command <code>ln -s doit.sh newfile</code> will create a file named <code>newfile</code> which is linked to the original file <code>doit.sh</code> . The command <code>vi newfile</code> will then open the file <code>doit.sh</code> . The command <code>ls -l newfile</code> will show: <code>newfile -> doit.sh</code> . The <code>\$OCTEON_ROOT/host/bin</code> directory has many "files" which are symbolic links to the actual files. The <code>ln</code> command can be used without symbolic linking. See the <code>ln</code> man page for more information.

Continued in the next table...

Table 46: Linux Commands Quick Reference, Part 3

Command	Brief Description
ls	List the contents of a directory. The command <code>ls -l</code> (ls dash L) (long listing) provides more details). The command <code>ls -ld</code> shows the directory. The command <code>ls .*</code> (ls dot star) will show hidden files (hidden files begin with the character "." as in <code>.bash_profile</code>). The command <code>ls -CF</code> will show the difference between files and directories, and will mark executable files: executable files are marked with the * character; directory names will be followed by the / character. Note that the string "ls" is often set to and alias. Type alias to check.
make	Build software using the directions in a Makefile.
man	Display the manual pages for commands such as <code>cd</code> , <code>ls</code> , <code>vi</code> , or <code>man</code> . For example <code>man man</code> will display the manual page for the <code>man</code> utility. Note that the <code>MANPATH</code> environment variable may have to be set before this will work.
modprobe	Add or remove a module from the kernel, such as the Cavium Networks Ethernet driver.
more	Display to the screen, one page at a time. For example: <code>cat Makefile more</code> . See also <code>less</code> , which allows the use of <code>vi</code> commands to search and move to a prior screen.
mv	Move a file or directory to a new name.
poweroff	Shutdown Linux and power off the machine gracefully. Requires <code>root</code> permission.
pwd	Display the current directory.
rm	Remove a file. Note: on Linux, there is no chance to recover the file after typing this command unless there is a backup copy! The command <code>rm -rf</code> will remove all the contents of a directory, recursively.
rmdir	Remove directory. Note: on Linux, there is no chance to recover the file after typing this command unless there is a backup copy!
sleep	This command is used to introduce a delay (number of seconds) as in <code>sleep 1; echo "hello world"</code> . This command is often used in shell scripts, especially when downloading and booting multiple ELF files.
sudo	Obtain <code>root</code> permission. The normal <code>username</code> must be in the <code>/etc/sudoers</code> file.
tail	This command displays the text at the end of an ASCII file. The command <code>tail -f</code> will continue to display new lines as they are added to the file by another process. When using <code>tail -f</code> , use <code>Ctrl-C</code> to exit.
vi	The <code>vi</code> editor is a commonly used editor in Linux. Use <code>Esc :q</code> (Escape key, colon, Q) to exit. Use <code>Esc :q!</code> to get out without saving any changes to the file. For other commands, see the documentation for <code>vi</code> . See also <code>gedit</code> . If possible, download a <code>vi</code> cheat sheet from the internet (Google the string "vi cheat sheet").
which	This command is used to show the full path to a command, for example <code>which oct-pci-boot</code> . Note that the command must be in the search path defined by the <code>PATH</code> environment variable.
Notes	
<i>Note: An ASCII file is human readable text as opposed to binary, which is the output of a compiler and is machine readable.</i>	

26.2 Shell Scripts

Commands may be put into a *shell script* and executed. This is particularly useful when testing an application on a PCI target board. The shell script is located on the development host. The following is an example of a shell script.

The contents of the file `doit.sh` are:

```
oct-pci-boot
oct-pci-load 0 hello
# boot the application without going to the Minicom session
oct-pci-bootcmd "bootoct 0 coremask=0xff"
```

After creating the file in the `examples/hello` directory, make it executable:

```
host$ chmod +x doit.sh
```

Then execute it:

```
host$ ./doit.sh
```

The “`./`” means “find the file `doit.sh` in the current directory”.

Note that, in shell, “`#`” is a comment delimiter.

26.3 Aliases

Aliases are often used to provide a short form of a command line. To see the aliases already set on the system, type:

```
host$ alias
```

Here is an example of an alias. This alias is defined in `.bash_profile`. There are different “dot” files such as `.bash_profile` and `.bashrc`. Be careful to not use commands which output to the screen in `.bashrc`. The shells provided in different Linux distributions have different rules. Read the manual page for exact information.

```
# The following text is one line, no carriage returns in the middle
alias hello='(cd $OCTEON_ROOT/examples/hello; make clean; make; cp hello
/home/testname/dl; ls -l /home/testname/dl)'
```

Note that the semicolon character is used to separate commands (`make clean; make`).

Source `.bash_profile` to establish the alias, and run the aliased command:

```
host$ source .bash_profile
host$ hello
rm -f hello
rm -f output log.txt.gz mem?_*.*txt pctrace.txt hello-*
mipsisa64-octeon-elf-gcc -o hello -g -O2 -W -Wall -Wno-unused-parameter
-I/home/testname/sdk18/target/include
-I/home/testname/sdk18/target/include -Iconfig
-DUSE_RUNTIME_MODEL_CHECKS=1 -DCVMX_ENABLE_PARAMETER_CHECKING=0
-DCVMX_ENABLE_CSR_ADDRESS_CHECKING=0 -DCVMX_ENABLE_POW_CHECKS=0
```

```
-DOCTEON_MODEL=OCTEON_CN38XX -DOCTEON_TARGET=cvmx_64 hello.c
total 81800
-rwxr-xr-x 1 testname software 315128 Feb 20 11:33 hello
```

26.4 Linux File Information and the Set User ID Bit

26.4.1 File Basics

By typing `ls -l` you can see the permission bits of the file. There are different permission bits for each of *owner*, *group*, and *world* (everyone). The *owner* is shown first, then *group*, then *world* (from left to right). The file permissions are shown as “r” for readable, “w” for writeable, “x” for executable. The following file is “rwx” by *root*, “rx” by *group root*, and “rx” by everyone. Thus, only *root* may write to the file.

```
-rwxr-xr-x 1 root root 520060 Jun 13 15:50 mips64-octeon-linux-gnu-gcc
```

26.4.2 The Set User ID Bit and Set Group ID Bit

The `oct-pci-boot` file has the “s” (*set user ID* or *setuid*) permission bit set, as seen in the next line:

```
-rwsr-sr-x 1 root software 415120 Jul 18 10:14 oct-pci-boot
```

In this case, the set group ID (*setgid*) bit is also set:

```
-rwsr-sr-x 1 root software 415120 Jul 18 10:14 oct-pci-boot
```

(Note: All `oct-pci-*` tools have the set-user-ID bit and set-group-ID bit set. They are located in the `$OCTEON_ROOT/host/pci` directory.)

When a command has the set-user-ID bit set, anyone executing the file takes on the privilege of the file owner. In this case the owner is *root*.

When a command has the set-group-ID bit set, anyone executing the file takes on the privilege of the file group. In this case the group is *software*.

The setuid bit is important, because it means you do not need to have root privilege to run most commands shown in this chapter. One of the exceptions is you need root privilege to build the Linux filesystem. When the set-user-ID or set-group-ID bits are not set, root privilege is provided by the sudo command.

26.4.3 The Effect of Copying a File (`cp`)

This discussion is especially relevant when making a copy of the SDK, and explains why using *root* privilege to make the copy is critical.

26.4.3.1 Copy the file as *root*

When you copy the file as *root* (using `sudo`), then the file permissions are preserved, and the owner (*root*) is *not* changed. The set-user-ID bit remains set.

```
host$ sudo cp oct-pci-boot /tmp/testname
Password:
host$ ls -l /tmp/testname
-rwsr-sr-x 1 root      root      415120 Jul 30 15:59 oct-pci-boot
```

26.4.3.2 Copy as regular user

When you copy a file with the set user ID bit set, unless you have *root* permission when you copy the file, the set-user-ID bit is cleared, and the *owner* is changed to *your* login name.

```
host$ cp oct-pci-boot /tmp/testname
host$ ls -l /tmp/testname
-rw-r--r-- 1 testname software 415120 Jul 30 15:57 oct-pci-boot
```

The new file is now owned by *testname*, the owner when the copy was made.

26.5 Killing a Process

The following example shows how to locate and kill a process. In this example, the process to be killed is *oct-pci-console*.

```
host$ ps -a
   PID TTY          TIME CMD
 5319 pts/0        00:00:00 bash
 5369 pts/1        00:00:00 bash
 6157 pts/1        00:00:00 oct-pci-console
 6170 pts/0        00:00:00 ps
host$ kill 6157
```

27 Appendix C: About the RPM Utility

27.1 Installing from the Support Site Instead of a CD

To install ALL the RPMs downloaded from the support site at <http://www.caviumnetworks.com/> to a specific directory on the development host:

1. Create a new directory on the development host. The downloaded RPM files will be put in this directory.
2. Download any desired RPMs from the support site into the directory.
3. *cd* to the directory.
4. Verify that all the RPMs in the directory should be installed.
5. As a normal user, run the command:

```
host$ sudo rpm -i *.rpm
```

Note: If the RPM files are not visible at the support site, then contact your Cavium Networks sales representative to fix the access permissions. Cavium Networks customer support and Field Application Engineers (FAEs) do not have the authority to change access permissions.

27.2 Useful RPM Commands

This section introduces commonly used RPM commands.

27.2.1 Force the RPM to Install

The rpm installation may fail if the files were previously installed, but have since been deleted. In this case, rpm will incorrectly determine that the files are already installed. To force the rpm to reinstall the files, use the command:

```
host$ sudo rpm -i --prefix <PREFIX> --force OCTEON*.rpm
```

Where <PREFIX> is the desired directory pathname, including the leading /.

27.2.2 Determine Which SDK Packages are Installed

To query which packages are installed, you may query all of the installed packages with the following command:

```
host$ rpm -qa | more
```

For example, the end of the output on a development host where the 1.7.3 SDK was installed is:

```
CN3XXX-COMPONENTS-COMMON-1.3.3-30
OCTEON-LINUX-1.6.0-221
OCTEON-IPSEC-1.2.4-21
OCTEON-PCI-CNTQ-0.9.6-53
OCTEON-SDK-1.7.3-264
```

Note that the package name (for instance OCTEON-SDK-1.7.3-264) does *not* include the .386.rpm suffix.

27.2.3 Remove a SDK Package After Installation

To remove (erase) the package after installation, as *root*, run the following command ensuring that .rpm is not appended as this command uninstalls a previously installed *package*, not a RPM file:

```
host$ sudo rpm -e <PACKAGE_NAME>
```

For example:

```
host$ rpm -e OCTEON-SDK-1.7.3-264
host$ rpm -e OCTEON-LINUX-1.7.3-264
```

27.2.4 Check Whether the Installed Files Have Changed Since Installation

Use the rpm -V command to see any changes made to the installed files. (Note that this command only compares the files in the original installation directory against the system's master copy. It will not be useful to compare the master copy against a copy located in a different directory.)

It is helpful to use the output of rpm -qa to help determine the package name needed for this command:

```
host$ sudo rpm -V <PACKAGE_NAME>
```

The output was not free from errors, but these may be ignored:

```
host$ sudo rpm -V OCTEON-SDK-1.7.3-264
Unsatisfied dependencies for OCTEON-SDK-1.7.3-264: CN3_X-SDK < 1.6.0-204
.....T /usr/local/Cavium_Networks/OCTEON-SDK/bootloader/u-
boot/include/bmp_logo.h
SM5...GT /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-boot
.....GT /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-boot.c
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-
bootcmd
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-
console
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-csr
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-ddr
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-load
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-memory
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-pow
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-
profile
SM5...GT /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-reset
S.5....T /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-
reset.c
.M..... /usr/local/Cavium_Networks/OCTEON-SDK/host/pci/oct-pci-tra
```

27.2.5 More Information About RPM

Type `man rpm` on the development host to get more information about the `rpm` command.

27.3 RPM Commands Quick Reference Guide

Table 47: RPM Commands Quick Reference

Brief Description	Syntax
Install only one RPM package into the default destination directory.	<code>rpm -i <SRC_DIR> <PACKAGE_NAME>.rpm</code>
Install only one RPM package into a specific destination directory.	<code>rpm -i <SRC_DIR> --prefix <DEST_DIR> <PKG_NAME>.rpm</code>
Install all RPM packages in the current directory into the default directory.	<code>rpm -i *.rpm</code>
Install all RPM files in the current directory into a specific destination directory.	<code>rpm -i <SRC_DIR> --prefix <DEST_DIR> *.rpm</code>
Force the RPM package to install.	<code>rpm -i <SRC_DIR> --prefix <PREFIX> --force <PKG_NAME>.rpm</code>
Query which packages are installed on the system.	<code>rpm -qa</code>
Remove (erase) a RPM package after installation. Note only the package name is specified, not the .i386.rpm suffix.	<code>rpm -e <PKG_NAME></code>
Verify the installed files have not changed since installation.	<code>rpm -V <PKG_NAME></code>
Notes:	
<i>SRC_DIR</i> is the directory where the .rpm files are located.	
<i>PKG_NAME</i> is the name of the RPM file without the .i386.rpm suffix.	
<i>DEST_DIR</i> is the location where the files will be installed.	

28 Appendix D: Other Useful Tools

28.1 Cscope

The open-source tool `cscope` is used to search source code. The utility may be downloaded for free from: <http://cscope.sourceforge.net/>.

To build the `cscope` data base, go to the top-level source directory and type:

```
host$ cscope -R
```

To start `cscope` without building the data base, go to the top-level source directory and type:

```
host$ cscope -d
```

To exit `cscope`, use Ctrl-D (hold down control key (Ctrl) while pressing the letter “d”).

Use “tab” (the tab key) to toggle between the “found” half of the screen and the “menu” half of the screen.

The user interface provides the following different searches:

- Find this C symbol:
- Find this global definition:
- Find functions called by this function:
- Find functions calling this function:
- Find this text string:
- Change this text string:
- Find this egrep pattern:
- Find this file:
- Find files #including this file:

In the “found” half of the screen (the top half), `Enter` will take you to the line in the referenced file, using the editor `vi`. Quitting `vi` will bring you back to the `cscope` screen.

28.2 Ctags

The `ctags` utility is used to help `vi` users quickly move through the code. For example, go to the `$OCTEON_ROOT/executive` directory, and type:

```
host$ ctags *.[chS]
```

This will build a “tags” file.

The tags file will help `vi` users navigate quickly to a desired function:

For example:

```
host$ vi -t cvmx_fpa_alloc
```

Once inside vi, position the cursor on the start of the call to `cvmx_read_csr()`, and press **Ctrl-}** (**Ctrl** close-curley-bracket). This will take the editor to the code for `cvmx_read_csr()`. To return to the prior location, type “`:e#`” (colon e #).

For example, after arriving at `cvmx_fpa_alloc()` via the `vi -t cvmx_fpa_alloc` command, try using the **Ctrl-}** command to go to `cvmx_read_csr()`:

```
static inline void *cvmx_fpa_alloc(uint64_t pool)
{
    uint64_t address =
cvmx_read_csr(CVMX_ADDR_DID(CVMX_FULL_DID(CVMX_OCT_DID_FP
A, pool)));
    if (address)
        return cvmx_phys_to_ptr(address);
    else
        return NULL;
}
```

28.3 Tera Term, Putty, VNC

The tools Tera Term, Putty, and VNC are all useful when connecting from a remote system to a development host. These tools and information about them may be found on the Internet.

28.3.1 Putty Tip

When running Putty, sometimes weird symbols such as “**â**” instead of “**,**” appear when SSH'd into the Linux development host.

This is due to a character set mismatch. The following technique may fix the problem:

On the Linux box, use the command:

```
host$ locale charmap
```

In this example the response was:

```
UTF-8
```

In the Putty config for Window->Translation, the “Character Set Translation of Received Data”, select the value, which should match the `charmap` value. In this example, it had been set to a default of ISO-8859-1). Change the value in the Putty screen to the correct `charmap` type (in this example, to UTF-8).

29 Appendix E: U-Boot Commands Quick Reference Guide

Bootloader commands can be seen in the target console by typing `help`. If the entire help line cannot be seen because it is too wide for the screen, use the Minicom command **Ctrl-Alt W** to turn on line wrap. Next time, start `minicom` using the `-w` command-line option.

To see special aliases such as `bootloader_flash_update`, type `printenv`.

Table 48: U-Boot Commands Quick Reference, Part 1

Command	Description
?	alias for help
askenv	get environment variables from stdin (typed into the target console)
autoscr	run script from memory
base	print or set address offset
bootelf	Boot from an ELF image in memory (Note: This command does not support SE-S or Linux. Use bootoct to boot a SE-S application. Use bootoctlinux to boot Linux from an ELF image in memory.)
bootoct	Boot from a SE-S application from an ELF image in memory
bootoctelf	Boot a generic ELF image in memory. NOTE: This command does not support SE-S or Linux. Use bootoct to boot an SE-S application. Use bootoctlinux to boot Linux from an ELF image in memory.)
bootoctlinux	Boot from a Linux ELF image in memory
bootp	boot image via network using BootP/TFTP protocol
cmp	memory compare
coninfo	print console devices and informations
cp	memory copy
crc32	checksum calculation
dhcp	invoke DHCP client to obtain IP/boot params
echo	echo args to console
eeprom	EEPROM sub-system
erase	erase FLASH memory
ext2load	load binary file from a Ext2 filesystem
ext2ls	list files in a directory (default directory is /)
fatinfo	print information about filesystem
fatload	load binary file from a DOS filesystem
fatloadalloc	load binary file from a DOS filesystem, and allocate a named bootmem block for file data.
fatls	list files in a directory (default directory is /)
flinfo	print FLASH memory information
freeprint	Print list of free bootmem blocks
go	start application at address addr
gunzip	Uncompress an in-memory gzipped file
help	print online help

Continued in the next table...

Table 49: U-Boot Commands Quick Reference, Part 2

Command	Description
ide	IDE sub-system
itest	return true/false on integer compare
loadb	load binary file over serial line (kermit mode)
loop	infinite loop on address range
md	memory display
mii	MII utility commands
mm	memory modify (auto-incrementing)
mtest	simple RAM test
mw	memory write (fill)
namedalloc	Allocate a named bootmem block
namedfree	Free a named bootmem block
namedprint	Print list of named bootmem blocks
nm	memory modify (constant address)
ping	send ICMP ECHO REQUEST to network host
printenv	print environment variables
protect	enable or disable FLASH write protection
rarpboot	boot image via network using RARP/TFTP protocol
read64	read 64 bit word from 64 bit address
read64b	read 8 bit word from 64 bit address
read64l	read 32 bit word from 64 bit address
read64s	read 16 bit word from 64 bit address
read_cmp	read and compare memory to val
reset	Perform RESET of the CPU
run	run commands in an environment variable
saveenv	save environment variables to persistent storage
setenv	set environment variables
sleep	delay execution for some time
tftpboot	boot image via network using TFTP protocol
tlv_eeprom	EEPROM data parsing for ebt3000 board
version	print monitor version
write64	write 64 bit word to 64 bit address
write64b	write 8 bit word to 64 bit address
write64l	write 32 bit word to 64 bit address
write64s	write 16 bit word to 64 bit address

30 Appendix F: ELF File Boot Commands Quick Reference

The following table is a summary of commands needed to boot the board.

Table 50: ELF File Download and Boot Commands Quick Reference, Part 1

Elf File Download and Boot Commands Quick Reference, Part 1	
Command Cheat Sheet	Notes
Reset the Board	
PCI Target: <code>oct-pci-reset</code>	After reset, the bootloader will not come up if the hardware is configured for PCI boot. See Note 1.
Stand-alone: push the reset button on the board.	
Download the Bootloader and Run the Bootloader	
Stand-alone: Start from flash: no action needed	
PCI Target: <code>oct-pci-boot</code>	Usage: <code>oct-pci-boot [--memonly] [bootloader filename]</code> . With no arguments, this command will download and boot the default bootloader.
Download the ELF Image File	
PCI Target: <code>oct-pci-load <tmp_download_address> <filename></code>	For SDK 1.7+ use the address "0" to take the default. Addresses must be in hex.
Compact Flash: <code>fatload <dev> <num> <tmp_download_address> <filename></code>	
Over the Network: <code>dhcp; tftpboot <tmp_download_address> <filename></code>	Image file path is relative to /tftpboot. For SDK 1.7+, use the address "0" to take the default. When entering an address, the 0x is optional, but addresses must be in hex.
Run the ELF Image File	
Target Console: Simple Executive: <code>bootoctl</code>	See Note 2 and Note 3.
<code>bootoctl</code> - Boot from an OCTEON Executive ELF image in memory <code>bootoctl [elf_address [stack=stack_size] [heap=heap_size] [coremask=mask_to_run numcores=core_cnt_to_run] [forceboot] [debug] [break] [endbootargs] [app_args...]</code> elf_address - address of ELF image to load. Defaults to \$(loadaddr). If 0, default load address used. stack - size of stack in bytes. Defaults to 1 megabyte heap - size of heap in bytes. Defaults to 3 megabytes coremask - mask of cores to run on. Anded with coremask_override environment variable to ensure only working cores are used numcores - number of cores to run on. Runs on specified number of cores, taking into account the coremask_override. skipcores - only meaningful with numcores. Skips this many cores (starting from 0) when loading the numcores cores. For example, setting skipcores to 1 will skip core 0 and load the application starting at the next available core. <other options not shown> endbootargs - if set, bootloader does not process any further arguments and only passes the arguments that follow to the application. If not set, the application gets the entire command line as arguments.	
<i>Continued in the next table... Notes 1-4 are in the next table.</i>	

Table 51: ELF File Download and Boot Commands Quick Reference, Part 2

Elf File Download and Boot Commands Quick Reference, Part 2	
Target Console: Linux: bootoctlinux	
	<pre>bootoctllinux elf_address [coremask=mask_to_run numcores=core_cnt_to_run] [forceboot] [skipcores=core_cnt_to_skip] [endbootargs] [app_args...]</pre> <p>elf_address - address of ELF image to load. If 0, default load address is used.</p> <p>coremask - mask of cores to run on. Anded with coremask_override environment variable to ensure only working cores are used</p> <p>numcores - number of cores to run on. Runs on specified number of cores, taking into account the coremask_override.</p> <p>skipcores - only meaningful with numcores. Skips this many cores (starting from 0) when loading the numcores cores. For example, setting skipcores to 1 will skip core 0 and load the application starting at the next available core.</p> <p>forceboot - if set, boots application even if core 0 is not in mask</p> <p>endbootargs - if set, bootloader does not process any further arguments and only passes the arguments that follow to the kernel. If not set, the kernel gets the entire command line as arguments.</p>
Over PCI: oct-pci-bootcmd "<command>" # be sure to add quotes around the command	
Target Console: Used to boot some operating systems other than Linux.	
Notes	
<p>Note 1: For PCI commands, to specify the which OCTEON board attached to the same host the command is directed to, set the environment variable OCTEON_PCI_DEVICE=<number>. The number 0 is the first OCTEON PCI target. The command /sbin/lspci enumerates the PCI targets.</p> <p>Note 2: "Target Console" means: type the command at in the Minicom screen.</p> <p>Note 3: To test a with a simple file, use examples/hello. This is a SE-S application only, it cannot be compiled as a Linux SE-UUM application.</p> <p>Note 4: Whichever image is running on core 0 should be downloaded and booted last.</p>	

31 Appendix G: Null Modem Serial Cable Information

The evaluation board is shipped with null modem cables. The following technical information is included for reference in case the cables are missing.

The lines that need to be crossed are TXD <-> RXD, and RTS <-> CTS. The connections for DTR, RTS, and DSR are not used, so their specific connections are not important.

Most null-modem serial cables (which can be purchased off-the-shelf) are wired as follows:

```

TXD <-> RXD
RXD <-> TXD
RTS <-> CTS
CTS <-> RTS
DSR <-> DTR
DCD <-> DTR
DTR <-> DCD
DTR <-> DSR

```

32 Appendix H: Query EEPROM to get Board Information

Evaluation boards contain a serial EEPROM which is configured at the factory. This EEPROM contains the board type, clock rate information, the serial number, and the assigned MAC addresses.

Note: These values should not be altered without the help of a Cavium Networks representative. The board is only tested with the programmed values; other values are not guaranteed to work.

To query the EEPROM for board information, use the following bootloader command:

```
target# tlv_eeprom display
=====
CLOCK_DESC_TYPE (0x1) tuple found: at addr 0x0
type: 0x1, len: 0xe, csum: 0x197, maj_ver: 1, min_ver: 0
DDR clock: 266 Mhz (raw: 0x10a)
CPU ref clock: 50 Mhz (raw: 0x190)
SPI clock (deprecated): 1000 Mhz (raw: 0x3e8)
=====
CHIP_CAPABILITY_TYPE (0x3) tuple found: at addr 0xe
type: 0x3, len: 0xe, csum: 0x2fd, maj_ver: 1, min_ver: 0
Coremask: 0x7fff, voltage_x100: 120, cpu_freq_mhz: 500
=====
BOARD_DESC_TYPE (0x2) tuple found: at addr 0x1c
type: 0x2, len: 0x24, csum: 0x2e5, maj_ver: 1, min_ver: 0
Board type: EBT3000 (0x2)
Board revision major:3, minor:1
Chip type (deprecated): OCTEON_SAMPLE (0x2)
Chip revision (deprecated) major:1, minor:3
Board ser #: 2005-00087-3.1
=====
MAC_ADDR_TYPE (0x4) tuple found: at addr 0x40
type: 0x4, len: 0x10, csum: 0x1e1, maj_ver: 1, min_ver: 0
MAC base: 00:0f:b7:10:03:e2, count: 14
=====
```

In this example, the DDR frequency is 266 MHz, the CPU frequency is 500 MHz, and the board type is EBT3000.

Another example output, in slightly different form is:

```
=====
CLOCK_DESC_TYPE (0x1) tuple found: at addr 0x0
type: 0x1, len: 0x10, csum: 0xf2, maj_ver: 2, min_ver: 0
DDR clock: 333 Mhz (raw: 0x14d)
CPU ref clock: 50 Mhz (raw: 0x190)
DFA ref clock: 0 Mhz (raw: 0x0)
SPI clock (deprecated): 0 Mhz (raw: 0x0)
=====
VOLT_MULT_TYPE (0x5) tuple found: at addr 0x10
type: 0x5, len: 0xc, csum: 0x6e, maj_ver: 1, min_ver: 0
Voltage: 1100 millivolts
CPU multiplier: 12
=====
BOARD_DESC_TYPE (0x2) tuple found: at addr 0x1c
type: 0x2, len: 0x24, csum: 0x2ea, maj_ver: 1, min_ver: 0
Board type: EBH5200 (0x13)
Board revision major:1, minor:0
Chip type (deprecated): NULL (0x0)
Chip revision (deprecated) major:0, minor:0
=====
```

```

Board ser #: 2008-1.0-00084
=====
MAC_ADDR_TYPE (0x4) tuple found: at addr 0x40
type: 0x4, len: 0x10, csum: 0x1a4, maj_ver: 1, min_ver: 0
MAC base: 00:0f:b7:10:53:60, count: 6
=====
```

In the example above, the CPU frequency is (CPU Ref Clock Rate * CPU Multiplier) = (50 MHz * 12) = 600 MHz.

32.1 Detecting a Problem with the EEPROM

If nothing is returned from the U-Boot command, the EEPROM needs to be reprogrammed. U-boot will also display “Warning: Board descriptor tuple not found in eeprom, using defaults”. The customer should contact a Cavium Networks Field Application Engineer (FAE) for how to reprogram it. The following text shows an example where the EEPROM is corrupted.

U-Boot 1.1.1 (U-boot build #: 205) (SDK version: 1.8.0-275) (Build time: Aug 13 2008 - 20:11:19)

```

EBH5200 board revision major:1, minor:0, serial #: unknown
OCTEON CN5230-SCP pass 1.0, Core clock: 600 MHz, DDR clock: 199 MHz (398
Mhz data rate)
Warning: Board descriptor tuple not found in eeprom, using defaults
DRAM: 2048 MB
Flash: 8 MB
Clearing DRAM..... done
BIST check passed.
Net: octmgmt0, octmgmt1, octeth0, octeth1, octeth2, octeth3
Bus 0 (CF Card): OK

ide 0: Model: CF 1GB Firm: 20071116 Ser#: TSS20037080507104533
      Type: Hard Disk
      Capacity: 967.6 MB = 0.9 GB (1981728 x 512)
```

33 Appendix I: Updating U-Boot on a Standalone Board

These directions describe installing the pre-built newer version of U-Boot on an OCTEON board via tftpboot. More information may be found in the SDK document “*OCTEON Bootloader*”.

Note: If possible, contact a Cavium Networks FAE before doing this step. It is critical to identify the correct bootloader for your OCTEON board.

Note: Do NOT try to modify the failsafe bootloader on the board. If the failsafe needs to be modified, contact a Cavium Networks FAE.

33.1 Locating the Correct Bootloader

The various U-Boot files are located in \$OCTEON_ROOT/target/bin.

The following U-Boot files are available in SDK 1.7.3 (omitting failsafe ELF files). Items marked with a “*” are planned, but not yet released as of SDK 1.7.3. Note that this list will become out of date as new boards are released. It is not intended to be a complete list, merely a guideline.

Note that the board type can be determined from the EEPROM. See Section 32 – “Appendix H: Query EEPROM to get Board Information”.

Table 52: SDK 1.7.3 Boards and Bootloader File Names, Part 1

Accelerator Boards (PCI Target Only)			
Chip	Board	Stand-alone Bootloader Name	PCI Target Bootloader Name
CN38XX	OCTEON XL CPB Coprocessor Board	N/A	u-boot-octeon_thunder_pciboot.bin
CN38XX	OCTEON XL NIC Network Interface Card	N/A	u-boot-octeon_thunder_pciboot.bin
CN38XX	OCTEON XL NICPro Network Interface Card	N/A	u-boot-octeon_thunder_pciboot.bin
CN58XX	OCTEON XL NICPro2 - Network Interface Card	N/A	u-boot-octeon_nicpro2_pciboot.bin
CN58XX	OCTEON XL NICPro2 - Network Interface Card - Pro 2 version	N/A	u-boot-octeon_nicpro2_pciboot.bin
CN56XX	OCTEON XL NIC Express Network Interface Card - PCIe version *	N/A	u-boot-octeon_nic_xle_4g_pciboot.bin
CN54XX	OCTEON XL NIC Express Network Interface Card - PCIe version *	N/A	u-boot-octeon_nic_xle_4g_pciboot.bin
Notes			
Note: Items marked with a "*" were planned, but not yet released as of SDK 1.7.3.			
Note: N/A stands for "Not Applicable".			

Table 53: SDK 1.7.3 Boards and Bootloader File Names, Part 2

Evaluation Boards (unless otherwise noted, these are dual boards which can be configured as either Stand-alone or PCI target)			
Chip	Board	Stand-alone Bootloader Name	PCI Target Bootloader Name
CN3010	CN3010 Evaluation Board	<code>u-boot-octeon_cn3010_evb_hs5.bin</code>	N/A
CN31XX	CN3100 Evaluation Board	<code>u-boot-octeon_ebh3100.bin</code>	N/A
CN38XX	CN3800 Evaluation Board - host only	<code>u-boot-octeon_nac38.bin</code>	N/A
CN38XX	CN3800 Evaluation Board - target only	N/A	<code>u-boot-octeon_ebt3000_pciboot.bin</code>
CN50XX	CN5000 Evaluation Board	<code>u-boot-octeon_cn3010_evb_hs5.bin</code>	N/A
CN52XX	CN5200 Evaluation Board	<code>u-boot-octeon_ebh5200.bin</code>	N/A
CN54XX	CN5400 Evaluation Board	<code>u-boot-octeon_ebh5600.bin</code>	N/A
CN55XX	CN5500 Evaluation Board (54xx + RAID)	<code>u-boot-octeon_ebh5600.bin</code>	N/A
CN56XX	CN5600 Evaluation Board - host only	<code>u-boot-octeon_ebh5600.bin</code>	N/A
CN57XX	CN5700 Evaluation Board (56xx + RAID)	<code>u-boot-octeon_ebh5600.bin</code>	N/A
CN58XX	CN5800 Evaluation Board	<code>u-boot-octeon_ebt5800.bin</code>	<code>u-boot-octeon_ebt5800_pciboot.bin</code>
Notes			
Note: N/A stands for "Not Applicable".			

33.2 Save the old Bootloader Environment

Print the old bootloader environment, and save the output to a file by cut-and-pasting the data shown on the screen into a file.

```

target# printenv
bootdelay=0
baudrate=115200
download_baudrate=115200
bootloader_flash_update=protect off 0xbff430000 0xbff47ffff;erase 0xbff430000
0xbff)
linux_cf=fatload ide 0 $(loadaddr) vmlinux.64;bootoctlinux $(loadaddr)
burn_app=erase bf480000 +$(filesize);cp.b $(fileaddr) bf480000 $(filesize)
ls=fatls ide 0
bf=bootoct bf480000 forceboot numcores=$(numcores)
nuke_env=protect off BFBFE000 BFBFFFFF; erase BFBFE000 BFBFFFFF
autoload=n
ethact=octeth0
loadaddr=0x20000000
coremask_override=0xffff
numcores=16
stdin=serial
stdout=serial
stderr=serial
bootfile=u-boot-octeon_ebt3000.bin
gatewayip=192.168.51.254
netmask=255.255.255.0
ipaddr=192.168.51.174
serverip=192.168.51.1

Environment size: 715/8188 bytes

```

33.3 Updating the Bootloader on the Board

33.3.1 Download U-Boot to a PCI Target

When using a PCI target, a new bootloader may be downloaded over PCI. Simply type `oct-pci-boot` to download the new bootloader from the SDK, or a custom bootloader may be specified on the command line.

33.3.2 Download U-Boot to a Standalone Target

When using a standalone board, a new bootloader may be downloaded using `tftpboot`. In this example the board is an ebt3000.

33.3.2.1 Copy the Bootloader to the /tftpboot Directory

```

host$ sudo cp u-boot-octeon_ebt3000.bin /tftpboot
host$ ls -l /tftpboot
total 296
-rwxr-xr-x 1 root root 296736 Aug 1 14:54 u-boot-octeon_ebt3000.bin
The exact command line will depend on which bootloader is currently
running on the board.

```

33.3.2.2 Boot the New Bootloader

On a bootloader built before SDK 1.7:

```
target# tftpboot 0x100000 u-boot-octeon_ebt3000.bin
```

For bootloaders built for SDK 1.7 and higher:

```
target# tftpboot 0 u-boot-octeon_ebt3000.bin

Using octeth0 device
TFTP from server 192.168.51.254; our IP address is 192.168.51.174
Filename 'u-boot-octeon_ebt3000.bin'.
Load address: 0x20000000
Loading: ###
done
Bytes transferred = 296736 (48720 hex), 10349 Kbytes/sec
```

33.3.3 Update the Bootloader

An alias called `bootloader_flash_update` has been defined by default. The alias can be seen using the bootloader command `printenv`:

```
bootloader_flash_update=protect off 0xbff430000 0xbff47ffff;erase 0xbff430000
0xbff)
```

In the target console, type:

```
target# run bootloader_flash_update
Un-Protected 5 sectors

..... done
Erased 5 sectors
Copy to Flash... ..... done
```

(This process takes about 7 seconds.)

Then push the reset button. The header is now updated:

```
U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun
13)

EBT3000 board revision major:4, minor:0, serial #: 2006-00257-4.0
OCTEON CN3860-NSP pass 2.X, Core clock: 500 MHz, DDR clock: 266 MHz (532 Mhz
da)
PAL rev: 2.01, MCU rev: 2.11, CPU voltage: 1.20
DRAM: 2048 MB
Flash: 8 MB
*** Warning - bad CRC, using default environment

IPD backpressure workaround verified, took 29 loops
Clearing DRAM..... done
BIST check passed.
Net: octeth0, octeth1, octeth2, octeth3
Bus 0 (CF Card): not available
```

33.3.4 Erase the Prior Environment Settings

If the old bootloader on the board was bootloader 1.6 or lower, an additional step is needed to erase the prior environment settings.

A script, `nuke_env`, has been defined (the script is visible using `printenv`):

```
target# printenv
nuke_env=protect off $(env_addr) +$(env_size); erase $(env_addr) +$(env_size)
```

Run `nuke_env` to load the new default environment:

```
target# run nuke_env
Un-Protected 1 sectors

. done
Erased 1 sectors
```

33.3.5 Verify the New Default Environment has Been Loaded

The new default environment should now have been loaded:

```
target# printenv
bootdelay=0
baudrate=115200
download_baudrate=115200
bootloader_flash_update=protect off $(uboot_flash_addr)
+$uboot_flash_size;erv
burn_app=erase $(flash_unused_addr) +$(filesize);cp.b $(fileaddr)
$(flash_unuse)
bf=bootoct $(flash_unused_addr) forceboot numcores=$(numcores)
nuke_env=protect off $(env_addr) +$(env_size); erase $(env_addr)
+$env_size
linux_cf=fatload ide 0 $(loadaddr) vmlinu.64;bootoctlinux $(loadaddr)
ls=fatls ide 0
autoload=n
loadaddr=0x20000000
coremask_override=0xffff
numcores=16
stdin=serial
stdout=serial
stderr=serial
env_addr=0xbfbfe000
env_size=0x2000
flash_base_addr=0xbff400000
flash_size=0x800000
uboot_flash_addr=0xbff430000
uboot_flash_size=0x50000
flash_unused_addr=0xbff480000
flash_unused_size=0x77e000
ethact=octeth0

Environment size: 889/8188 bytes
```

The following table shows an example of how the original environment may be different from the new default environment.

Table 54: Example Bootloader Environment File Changes due to Upgrade

Original Bootloader Environment	Bootloader Environment Restored from Default after Upgrade
bootdelay=0	bootdelay=0
baudrate=115200	baudrate=115200
download_baudrate=115200	download_baudrate=115200
bootloader_flash_update=protect off 0xbff430000 0xbff47ffff;erase 0xbff430000 0xbff)	bootloader_flash_update=protect off \$(uboot_flash_addr) +\$(uboot_flash_size);erv
linux_cf=fatload ide 0 \$(loadaddr) vmlinux.64;bootoctlinux \$(loadaddr)	linux_cf=fatload ide 0 \$(loadaddr) vmlinux.64;bootoctlinux \$(loadaddr)
burn_app=erase bf480000 +\$filesize;cp.b \$(fileaddr) bf480000 \$filesize	burn_app=erase \$(flash_unused_addr) +\$filesize;cp.b \$(fileaddr) \$(flash_unuse)
ls=fatls ide 0	ls=fatls ide 0
bf=bootoct bf480000 forceboot numcores=\$(numcores)	bf=bootoct \$(flash_unused_addr) forceboot numcores=\$(numcores)
nuke_env=protect off BFBFE000 BFBFFFFF; erase BFBFE000 BFBFFFFF	nuke_env=protect off \$(env_addr) +\$env_size; erase \$(env_addr) +\$env_size
autoload=n	autoload=n
ethact=octeth0	ethact=octeth0
loadaddr=0x20000000	loadaddr=0x20000000
coremask_override=0xfffff	coremask_override=0xfffff
numcores=16	numcores=16
stdin=serial	stdin=serial
stdout=serial	stdout=serial
stderr=serial	stderr=serial
bootfile=u-boot-octeon_ebt3000.bin	
	env_addr=0xbfbfe000
	env_size=0x2000
	flash_base_addr=0xbff400000
	flash_size=0x800000
	uboot_flash_addr=0xbff430000
	uboot_flash_size=0x50000
	flash_unused_addr=0xbff480000
	flash_unused_size=0x77e000

Note 1: The gatewayip, netmask, ipaddr, and serverip environment variables will no longer be set after the upgrade. These IP addresses are set by either the `dhclient` command or `setenv` commands.

33.3.6 Reset IP Information

To restore the gatewayip, netmask, ipaddr, and serverip configuration variables, either use `dhcp` to reset the information, or set the variables individually. If static IP addresses are used, after setting them, `saveenv` to save them.

```
target# dhcp
Interface 1 has 4 ports (RGMII)
BOOTP broadcast 1
octeth0: Up 1000 Mbps Full duplex (port 16)
DHCP client bound to address 192.168.51.161
```

34 Appendix J: TFTP Boot Assistance (`tftpboot`)

The following documentation is provided for persons unfamiliar with TFTP, who need help getting `tftpboot` to work. The `tftpboot` utility is used to download an application to a standalone target board. Target boards which support CompactFlash may use a flash card instead of `tftpboot` to download the application, but `tftpboot` is faster.

In addition to being faster, `tftpboot` can be more convenient than loading via a flash card. Before inserting the flash card into the target, the target should be powered off.

34.1 TFTP Server Firewall

The `tftpboot` command will not work if the server's firewall is on. Be sure the firewall is off before beginning.

If the server's firewall is on, `tftpboot` will not work, but will not supply a convenient error message. This is particularly annoying to debug.

34.2 Verify that the TFTP Server RPM is Installed on the TFTP Server

Ensure that the `tftp-server` RPM is installed on the TFTP server and is running:

On the TFTP server, type:

```
host$ sudo rpm -qa | grep -i tftp
```

```
Expect to see the tftp-server package in the reply:
tftp-server-0.33-3
```

If the RPM package containing the `tftp` server is not installed, install it before continuing to the next step.

34.3 Verify the TFTP Server is Currently Enabled

Run the commands in this section after TFTP is installed to ensure the TFTP server is enabled properly.

34.3.1 Enable and Start the TFTP Server

TFTP configuration is stored in the TFTP startup file. In the example below, the startup file is /etc/xinetd.d/tftp. The exact startup file on your system can be determined by looking at the TFTP manual page.

Look at the configuration information in the TFTP startup file:

```
host$ cat /etc/xinetd.d/tftp
# default: off
# description: The tftp server serves files using the trivial file
transfer \
#      protocol. The tftp protocol is often used to boot diskless \
#      workstations, download configuration files to network-aware
printers, \
#      and to start the installation process for some operating systems.
service tftp
{
    socket_type          = dgram
    protocol             = udp
    wait                 = yes
    user                 = root
    server               = /usr/sbin/in.tftpd
    server_args          = -s /tftpboot
    disable              = yes # <<< HERE IS THE KEY DIFFERENCE
    per_source            = 11
    cps                  = 100 2
    flags                = IPv4
}
```

34.3.2 Turn on the TFTP Server

If the TFTP server is set to disable = yes, then turn on the TFTP server:

```
host$ /sbin/chkconfig tftp on
<no reply if all is well>
```

34.3.3 Verify the TFTP Server is Now Enabled

Look at the contents of the tftp startup file and look for “disable = no”.

```
host$ cat /etc/xinetd.d/tftp
# default: off
# description: The tftp server serves files using the trivial file
transfer \
#      protocol. The tftp protocol is often used to boot diskless \
#      workstations, download configuration files to network-aware
printers, \
#      and to start the installation process for some operating systems.
service tftp
{
    disable = no # <<< HERE IS THE KEY DIFFERENCE
    socket_type          = dgram
    protocol             = udp
    wait                 = yes
```

```

        user          = root
        server       = /usr/sbin/in.tftpd
        server_args  = -s /tftpboot
        per_source   = 11
        cps          = 100 2
        flags         = IPv4
    }
}

```

34.4 About the TFTP Download Directory on the TFTP Server

Note the contents of the tftp file shown above includes a line specifying the download directory:

```
server_args      = -s /tftpboot
```

If a different download directory is specified on this line, for instance /home, then downloads must occur relative to /home, not relative to /tftpboot. For instance, if /home is specified, then it is okay to create a sub-directory testname/dl inside of /home:

```
host$ mkdir /home/testname/dl
```

This is recommended if multiple users are sharing the same development host, because each user will then have a separate download directory.

Then, when typing the tftpboot command line in the target console, specify the download directory relative to /home:

For example, for bootloaders built with SDK 1.7 and higher:

```
target# tftpboot 0 testname/dl/hello
```

34.4.1 Tftpboot Directory Permissions

If the /tftpboot directory and the files inside it are not readable to *world*, then the tftp server will not be able to read the files. To resolve the directory permissions, either use a different directory, such as /home, or change the permissions on the /tftpboot directory:

```
# In the following command, the "1" will set the "sticky bit",
# so users may only delete their own files
# host$ chmod 1777 /tftpboot
```

See Section 26.4.1 – “File Basics” for an introduction to file permissions.

34.5 Verify serverip is set Correctly on the OCTEON Target Board

Ensure that the server IP variable, serverip is correctly set in the bootloader environment variables on the OCTEON target board.

34.5.1 If a DHCP Server is Available

If a DHCP Server is available, then in the target console, type:

```
target# dhcp
```

34.5.2 If a DHCP Server is Not Available

To set the bootloader's serverip variable to the IP of the TFTP server, first run the /sbin/ifconfig utility on the TFTP server to find out the TFTP server's IP address:

```
host$ /sbin/ifconfig
```

Then run the following commands from the target console for the OCTEON board (bootloader commands on the target board):

```
target# setenv serverip <host IP> # Use your host IP address
target# saveenv
```

For example:

```
target# setenv serverip 192.168.51.1 # use your TFTP Server's IP address
target# saveenv
```

34.5.3 Verify the Server IP Address and Physical Ethernet Connection

Check that the connection is good: In the target console, type:

```
target# ping <host IP> # Use your host IP address
```

For example:

```
target# ping 192.168.51.1
Using octeth0 device
host 192.168.51.1 is alive
```

34.6 Test tftpboot: Boot hello on the OCTEON Target Board

For a quick test, use the example application hello. If hello has not yet been built, see Section 11 – “Hands-on: Build and Run a SE-S Application (hello)”.

Copy the example application hello to the /tftpboot folder. This folder should have been created when the tftp-server RPM was installed. (Note that on some systems the default tftpboot directory is /var/lib/tftpboot. The exact directory can be determined by looking at the server_args value in the TFTP startup file.)

```
host$ cd $OCTEON_ROOT/examples/hello
host$ sudo cp hello /tftpboot
```

Note that the folder is only writable by the owner, root:

```
host$ cd /tftpboot
host$ ls -ld .
drwxr-xr-x 2 root root 4096 Feb 17 2004 .
```

Run the following command in the target console:

```
target# tftpboot 0 hello
```

If tftpboot is installed and configured correctly, expect to see:

```
Using octeth0 device
TFTP from server 192.168.16.41; our IP address is 192.168.16.61
Filename 'hello'.
Load address: 0x20000000
Loading: #### <<< tftpboot will print more "#" as data is downloaded
```

```
done
Bytes transferred = 315148 (4cf0c hex), 12823 Kbytes/sec
```

Otherwise, timeouts will occur.

If the `serverip` was not yet set, the following error will occur:

```
target# tftpboot 0 hello
Interface 1 has 4 ports (RGMII)
*** ERROR: 'serverip' not set
WARNING: Data loaded outside of the reserved load area, memory corruption
may occur.
WARNING: Please refer to the bootloader memory map documentation for more
information.
```

34.7 Further Information

For more information:

http://www.linuxhomenetworking.com/wiki/index.php/Quick_HOWTO:_Ch16:_Telnet,_TFTP,_and_xinetd#TFTP

35 Appendix K: Downloading Using the Serial Connection

Downloading over the serial connection is slow. This method is sometimes used to upgrade the bootloader on a standalone board. The instructions given here may be used for a bootloader, but `hello` is used in this example.

35.1 Kermit

The program `kermit` must have been installed on the development host. To check whether `kermit` has been installed type:

```
host$ which kermit
```

Expect to see:

```
/usr/bin/kermit
```

This document does not include instructions to install Kermit. Check a Linux System Administration manual for help if Kermit is not already installed on the development host.

Configure Minicom to use Kermit (these instructions assume the Minicom configuration was saved as `ttyS0`):

```
host$ sudo minicom -s ttyS0
```

Select “Filenames and paths”, and set the path for “Kermit program” to `/usr/bin/kermit`, then save the new configuration.

35.2 Copy `hello` to `/tmp`

Copy the application `$OCTEON_ROOT/examples/hello/hello` to `/tmp`. This step assumes it was already built.

35.3 Set up the .kermrc File

Create a `~/.kermrc` file. This file will contain the following information, assuming the development host's serial port is `/dev/ttyS0`:

```
set modem type none
set line /dev/ttyS0
set speed 115200
set streaming off
set prefixing all
set flow xon/xoff
set carrier-watch off
robust
```

35.4 Start Minicom

1. Start Minicom (`minicom -w`) .
2. Then, reset the board as needed to get a bootloader prompt.
3. Give the bootloader command:

```
target# loadb
```

4. Then call Kermit (Ctrl-A K).

You should see something like:

```
C-Kermit 8.0.209, 17 Mar 2003, for Red Hat Linux 8.0
Copyright (C) 1985, 2003,
Trustees of Columbia University in the City of New York.
Type ? or HELP for help.
(/home/testname/sdk/examples/hello/) C-Kermit>
```

5. Give the Kermit Commands in the Minicom Session (the download will take about 20 seconds):

```
(/home/testname/sdk/examples/hello/) C-Kermit> send /tmp/hello
(/home/testname/sdk/examples/hello/) quit
```

6. After quitting Kermit, type in the command to run the downloaded ELF file:

```
target# bootoc 0 coremask=0x1
Bootloader: Booting Octeon Executive application at 0x20000000, core
mask: 0x1,0
Bootloader: Done loading app on coremask: 0x1
PP0:~CONSOLE->
PP0:~CONSOLE->
PP0:~CONSOLE-> Hello world!
PP0:~CONSOLE-> Hello example run successfully.
```

36 Appendix L: Simple Executive Configuration

Ninety percent of users can use the Simple Executive, after making the minor configurations described in this section. This is the easiest way to speed your application to market.

The following items may be configured:

1. FPA Pools including: Packet Data Buffers, Work Queue Entry Buffers, PKO Command Queues, Timer Buffers, DFA Buffers, ZIP Buffers, and custom buffers.
2. Scratchpad (used for IOBDMA operations).
3. FAU Resources

For details on how to configure FPA Pools, see the FPA Chapter. For details on how to configure FAU Buffers, see the FAU Chapter.

37 Appendix M: Changing the ABI Used for Linux

Note that the *kernel* is always built for 64-bits. This section only applies to Linux example applications which may be compiled with either the default N64 ABI, or the N32 ABI.

Notice that in `sdk-examples.mk`, the exact make line depends on the variable `ABI`. This comes from the environment variable `TOOLCHAIN_ABI`.

To change this, go to the `$OCTEON_ROOT/linux/kernel_2.6/linux` directory. Type `make menuconfig` and select the desired ABI.

```
Global Options --->
[*] device-files
[*] busybox
[*]   Include the Busybox testsuite
[*] init-scripts
  NFS Root filesystem --->
[*] module init tools
--- libpcap
[*] libraries-n32
[ ] libraries-uclibc
--- libraries-64
<text omitted>
```

Select Global Options. The next menu will look similar to:

```
Toolchain ABI and C library (N64 ABI with GNU C Library (glibc))
[ ] Override the Linux Kernel configuration
[*] Include all kernel modules built in the kernel tree
[*] Enable IPV6 support
```

Select Toolchain ABI and C Library. The next menu will look similar to:

```
( ) N32 ABI with GNU C Library (glibc)
      (X) N64 ABI with GNU C Library (glibc)
      ( ) N32 ABI with uClibc
```

Select the ABI to be used for the examples, then exit, saving the new configuration, and re-make the `kernel` target in `$OCTEON_ROOT/linux`.

Note you may have to remove the file

`$OCTEON_ROOT/linux/embedded_rootfs/.root_complete` to force a filesystem rebuild on the next make.

38 Appendix N: Contents of the Embedded Root Filesystem

The 2.6 Linux `vmlinuz.64` ELF files contain a gzipped `cpio`-format archive, which is used to populate the in-memory root filesystem when the kernel boots up. This `cpio` file contains the busybox utilities, the example code, the init program, and other key files.

There are two files in \$OCTEON_ROOT/linux/embedded_rootfs: rootfs.cpio and rootfs.cpio.gz. The rootfs.cpio.gz file is the gzipped version embedded into the vmlinux.64 file.

To see the contents of the root filesystem before it is booted on the target, either look at the /tmp/root_rootfs directory, or examine the rootfs.cpio file.

The contents of the rootfs.cpio file can be seen using the cpio command.

```
host$ cd $OCTEON_ROOT/linux/embedded_rootfs

host$ ls -l rootfs.cpio
-rw-r--r-- 1 root root 42831360 Jan 22 14:14 rootfs.cpio

host$ cat rootfs.cpio | cpio -itv >/tmp/junk.txt

host$ head /tmp/junk.txt
drwxrwxrwx 20 root      root          0 Jan 22 14:10 .
drwxr-xr-x  2 root      root          0 Jan 22 14:14 sbin
lrwxrwxrwx  1 root      root         12 Jan 22 14:01 sbin/e2fsck ->
/bin/busybox
lrwxrwxrwx  1 root      root         12 Jan 22 14:01 sbin/klogd ->
/bin/busybox
-rwxr-xr-x  1 root      root       62000 Jan 22 14:14 sbin/ifconfig
lrwxrwxrwx  1 root      root         12 Jan 22 14:01 sbin/hdparm ->
/bin/busybox
lrwxrwxrwx  1 root      root         12 Jan 22 14:01 sbin/fdisk ->
/bin/busybox
-rwxr-xr-x  1 root      root       49408 Jan 22 14:14 sbin/arp
-rwxr-xr-x  1 root      root        932 Jan 22 14:01 sbin/rc
-rwxr-xr-x  1 root      root       49968 Jan 22 14:14 sbin/route
```

To look for a specific application, such as a custom application, use the following command. Note: substitute your search string for examples/fpa. (In the case, the FPA example has been successfully added as an example of a custom application. See Section 23.1 – “Adding Applications to the Embedded Root Filesystem”.)

```
host$ cat rootfs.cpio | cpio -itv | grep examples/fpa
-rwxr-xr-x  1 root      root       696296 Jan 30 10:35 examples/fpa
84904 blocks
```

39 Appendix O: Getting Ready to Use a Flash Card

A flash card can be used to:

1. Boot an ELF file (a SE-S application or the Linux kernel).
2. Provide the Debian root filesystem

Before using the flash card, some minor system administration must be done on the development host, and the flash card must be correctly formatted. This section describes these steps.

39.1 System Administration Steps

First attach the USB-connected CompactFlash reader/writer to the development host, and insert the flash card.

39.1.1 Determine the Flash Card's Device Name

When the USB CompactFlash reader/writer is plugged into a USB port on the development host, the Linux kernel will send information about it to the kernel log. After plugging in the reader/write, use dmesg to view the log. Look for an entry similar to the one below, near the end of the log. The device name is after the “Attached scsi removable disk” message. In this example, it is *sdb*:

```
usb 1-5: new high speed USB device using ehci_hcd and address 3
scsi3 : SCSI emulation for USB Mass Storage devices
usb-storage: device found at 3
usb-storage: waiting for device to settle before scanning
  Vendor: Generic Model: STORAGE DEVICE Rev: 0128
  Type: Direct-Access ANSI SCSI revision: 00
SCSI device sdb: 2001888 512-byte hdwr sectors (1025 MB)
sdb: assuming Write Enabled
sdb: assuming drive cache: write through
SCSI device sdb: 2001888 512-byte hdwr sectors (1025 MB)
sdb: assuming Write Enabled
sdb: assuming drive cache: write through
  sdb: sdb1
Attached scsi removable disk sdb at scsi3, channel 0, id 0, lun 0
Attached scsi generic sg1 at scsi3, channel 0, id 0, lun 0, type 0
usb-storage: device scan complete
```

Warning: It is possible to lose all data on the host root filesystem if the wrong device is specified! Before taking these steps, use the information above to verify the flash card device name (such *sdb*), so that the flash card is partitioned, not another filesystem on the development host!

39.1.2 Create the Mount Directories

The first partition is mounted on /mnt/cf1, the second is mounted on /mnt/cf2. The /mnt/cf1 directory configuration is required by \$OCTEON_ROOT/linux/Makefile when downloading a kernel to flash, so we will use that convention to simplify the system administration directions:

```
host$ sudo mkdir -p /mnt/cf1
host$ sudo mkdir -p /mnt/cf2
```

39.1.3 Prevent Automount of the Device

To prevent the device from being automounted, add the following entry to /etc/fstab:

/dev/sdb1	/mnt/cf1	auto	noauto,noatime,user	0 0
/dev/sdb2	/mnt/cf2	ext3	noauto,noatime,user	0 0

After the /etc/fstab entry is made, the command `mount /mnt/cf1` will look in /etc/fstab, determine the correct device (/dev/sdb1), and mount it, simplifying the command line and helping prevent errors.

39.1.4 Protect Yourself By Setting Up an Environment Variable

Because specifying the wrong device on the command line used to partition the device can destroy the data on that device, it is prudent to put the device name in an environment variable, then use that environment variable in the command line.

Specify the correct flash card device name in the bash command line below:

```
# be careful to substitute the correct device in the next step!
host$ export DISK=/dev/sdb
```

39.1.5 Partition the Flash card

If the flash card is already formatted, and the Debian root filesystem will not be used, this step can be skipped: only one FAT partition is required.

Flash cards are normally formatted with a single DOS FAT partition. Debian requires a flash card with two partitions: FAT and EXT3.

For simplicity, these instructions use the Debian Makefile to partition the flash even if the Debian root filesystem will not be used. Using this Makefile automates the system administration.

The steps here will erase all data on the flash card and repartition it to have these two partitions.

39.1.5.1 Debian Root Filesystem

If you are using the Debian root filesystem, the card is not formatted at this time. See Section 41 – “Appendix Q: Using the Debian Root Filesystem”.

39.1.5.2 ELF File

When using a flash card to simply copy an ELF file, such as an SE-S application or the Linux kernel, the first partition on the flash card is used. In order to simplify formatting the flash card, the Debian Makefile is used. This automates the system administration steps.

Warning: This step will delete all data on the flash card.

```

host$ cd $OCTEON_ROOT/linux/Debian
# add the directory /sbin to your path to get fdisk, mkfs, etc.
host$ PATH=$PATH:/sbin
# the DISK environment variable was set up during the
# System Administration step
host$ sudo make DISK=$DISK env-check safety-banner partition mkfs

```

After the flash card is partitioned, this step does not need to be repeated.

40 Appendix P: Booting an ELF File From a Flash Card

A CompactFlash device is usually available for standalone target boards, and is sometimes available for PCI target boards. Either an SE-S application or vmlinu \times .64 ELF file may be copied to a flash card, loaded into memory, and booted.

To use the Debian root filesystem from a flash card, see Section 41 – “Appendix Q: Using the Debian Root Filesystem”, otherwise follow the directions in this section.

40.1 System Administration Steps

Before using the flash card for the first time, see Section 39 – “Appendix O: Getting Ready to Use a Flash Card”.

40.2 Copying the ELF File to the Flash Card

There are two different directions: one for SE-S applications, and another for vmlinu \times .64.

40.2.1 Copy a SE-S Application to the Flash Card

In the following example, hello is copied to the flash card:

```

host$ cd $OCTEON_ROOT/examples/hello
host$ sudo mount /mnt/cf1 # mount the flash card
host$ cp hello /mnt/cf1
host$ sudo umount /mnt/cf1      # unmount the flash card

```

40.2.2 Copy the Kernel with Embedded Rootfs Onto the Flash Card

Note that the Makefile *requires* the flash card partition to be mounted on /mnt/cf1 before calling make flash:

```

host$ cd $OCTEON_ROOT/linux
host$ sudo mount /mnt/cf1      # mount the flash card
host$ sudo make kernel flash
host$ sudo umount /mnt/cf1      # unmount the flash card

```

40.3 Moving the Flash Card to the Target

Power off the development target, and then insert the flash card into the target board.

Warning: When inserting the flash card into the OCTEON development target, first power OFF the development target. The boot bus does not support hot plugging of devices. When hot plugging the flash card, there is a risk (low, but not zero) of damaging the board. To power off an OCTEON PCI target, power off the PCI host.

Note: If you are ignoring the safety warning above (at your own risk), so that the board has not been powered off, be sure to reset the board after inserting the flash card. If the development target is not reset, the bootloader will not detect the flash card. The bootloader command `ide reset` can also be used to cause the bootloader to recognize the flash card.

40.4 Loading the ELF File From the Flash Card into Memory

Remove the flash card from the development host.

To load an ELF file from the flash card (using bootloader version 1.7 and higher) use the following command, substituting the name of the ELF file for `vmlinux.64`.

```
target# fatload ide 0 0 vmlinux.64
```

40.5 Booting the ELF File From the Flash Card

Booting an SE-S Application:

```
target# bootoct 0 coremask=<coremask>
```

Booting Linux:

```
target# bootoctlinux 0 coremask=<coremask>
```

For more information on running Linux from a flash card, see the SDK document “*Linux on the OCTEON*”. For information on running the Debian root filesystem, See Section 41 – “Appendix Q: Using the Debian Root Filesystem”.

41 Appendix Q: Using the Debian Root Filesystem

This section provides a brief introduction to using the Debian root filesystem. For more information on the Debian root filesystem, see the SDK document “*Running Debian GNU/Linux on OCTEON*”.

Note: The `oncpu` utility may not be in the Debian root filesystem. If it is missing, copy it from the development host to the Debian root filesystem.

41.1 System Administration Steps

Before using the flash card for the first time, see Section 39 – “Appendix O: Getting Ready to Use a Flash Card”.

Warning: This step will delete all data on the flash card.

41.2 About the Debian Root Filesystem

When using the Debian root filesystem, the Linux kernel is compiled without the embedded root filesystem. The Debian root filesystem is on a flash card. The Makefile command which installs the Debian root filesystem also installs the kernel onto the flash card. The user may either use the kernel on the flash card, or a separate kernel.

41.3 Install Kernel Plus Debian Onto the Flash card

The Debian root filesystem is located in the \$OCTEON_ROOT/linux/debian directory. The Debian files are pre-packaged in the file debian_octeon.tgz.

To create the two partitions and their contents, use the linux/debian/Makefile:

```
host$ cd $OCTEON_ROOT/linux/debian
# add the directory /sbin to your path to access fdisk, mkfs, etc.
host$ PATH=$PATH:/sbin
# the DISK environment variable was set up during the
# System Administration step
host$ sudo make DISK=$DISK compact-flash
```

The first partition will have the kernel, and the second partition contain the root filesystem:

```
host$ sudo mount /mnt/cf1
host$ sudo mount /mnt/cf2
host$ /bin/ls -CF /mnt/cf1
vmlinuz.64*
host$ /bin/ls -CF /mnt/cf2
bin/  etc/  lib/  lost+found/  opt/  sbin/  tmp/  var/
boot/ home/ lib32@  media/  proc/  srv/  uclibc@
dev/  initrd/ lib64@  mnt/  root/  sys/  usr/
```

In detail, the Makefile target compact-flash will partition the flash card, then copy a stripped version of vmlinuz.64, extract the debian_octeon.tgz, file onto /dev/sdb, and add Cavium Networks additions (such as the Cavium Networks tool chain).

41.4 Moving the Flash Card to the Target

Power off the development target, and then insert the flash card card into the target board.

Warning: When inserting a flash card card into an OCTEON development target, first power OFF the development target. The boot bus does not support hot plugging of devices. When hot plugging the flash card, there is a risk (low, but not zero) of damaging the board. To power off an OCTEON PCI target, power off the PCI host.

Note: If you are ignoring the safety warning above (at your own risk), so that the board has not been powered off, be sure to reset the board after inserting the flash card. If the development target is not reset, the bootloader will not detect the flash card. The bootloader command ide reset can also be used to cause the bootloader to recognize the flash card.

41.5 Load the Kernel from the Flash Card into Memory

Load the kernel from the flash card into memory:

```
target# fatload ide 0 0 vmlinuz.64
```

41.6 Boot the Kernel

When booting the kernel, two changes are essential:

1. Specify root=/dev/cfa2 to use the Debian root filesystem on the flash card. If this step is omitted, Debian will fail with the message:

```
VFS: Unable to mount root fs via NFS, trying floppy.
VFS: Cannot open root device "<NULL>" or unknown-block(2,0)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on
unknown-block(2,0)
```

2. When booting Debian Linux, *do not* specify 0 as the load address. Debian will interpret the 0 to be the *run level* (0=shutdown), and Linux will shutdown immediately, giving a message similar to the following:

```
INIT: Entering runlevel: 0
Asking all remaining processes to terminate...done.
Killing all remaining processes...done.
Stopping portmap daemon.....
<text omitted>
System halted.
```

The correct boot command line is:

```
# using the value of loadaddr which was returned by the printenv command
target# bootoctlinux 0x2000000 coremask=1 root=/dev/cfa2
```

41.7 Upgrading the Kernel on the Flash Card

Use the \$OCTEON_ROOT/linux/Makefile target kernel-deb to create a kernel without the embedded root filesystem. Then use the target flash to copy the kernel to the flash card without disturbing the Debian root filesystem on the second partition:

```
host$ cd $OCTEON_ROOT/linux
# The Makefile requires the flash card partition be mounted
# on /mnt/cf1 before calling make flash
host$ sudo mount /mnt/cf1      # mount the flash card
host$ sudo make kernel-deb flash
host$ sudo umount /mnt/cf1     # unmount the flash card
```

42 Appendix R: About oct-pci-console

On OCTEON PCI targets, the PCI host command `oct-pci-console` may be used to view the development target console output. This command is only useful when using flash boot, not PCI boot (see note below), and an initial Minicom connection is required to configure the bootloader to send console messages over the PCI bus.

Note: *When using oct-pci-console, early bootloader messages will not be visible. These messages are sent to the serial console before the PCI console is enabled by the bootloader.*

In the following command the number 1 specifies one PCI console.

```
target# setenv pci_console_count 1
PCI console init succeeded, 1 consoles, 1024 bytes each
Using PCI console, serial port disabled.
```

(After changing the bootloader to send messages over PCI, the target console in the Minicom screen will no longer reply to the Enter key.)

Either exit Minicom (Ctrl-A X) or go to another terminal session, and connect to the target console over the PCI bus. Notice that the console is specified as 0 in the following command:

```
host$ oct-pci-console 0
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x8000000]
Connecting to PCI console 0
Using raw terminal mode, escape character is ^A, use ^A D to exit, ^A A
to send ^A
```

Save the value of `oct-pci-console` so it will be set when the board comes up the next time:

```
target# saveenv
```

(This step is done after the PCI console connection is verified to be okay.)

Note: The `oct-pci-console` command cannot be used effectively with PCI boot because there is no way to save the number of PCI consoles: After every reset, the number of consoles would have to be reconfigured. The only way to reconfigure them is via the serial connection (Minicom) to the target console, so that the serial console must always be connected.

The following error will occur when attempting to save environment variables when using PCI boot:

```
target# saveenv
Environment updates not supported
```

To exit `oct-pci-console` use Ctrl-A D (hold down control key (Ctrl) while pressing the letter “a”), then let go of the Ctrl-A and then press the letter “d”.

See the SDK document “*Developing with OCTEON as a PCI Target*” for more information.

43 Appendix S: About `oct-pci-reset` and `oct-pci-csr`

The `oct-pci-*` commands are used from a PCI host to an OCTEON PCI development target.

43.1 Reset: `oct-pci-reset`

This command will only reset the board. If the board is configured to boot from flash, the board will boot after the reset, otherwise it will be held in reset until the `oct-pci-boot` command is typed on the development host.

After `oct-pci-reset`, if the board is not configured to boot from onboard flash, then there will not be any response at the Minicom console, however the board’s reset state may be viewed using `oct-pci-csr` as shown in Section 43.2 –“Access Control and Status Registers (CSRs): `oct-pci-csr`”.

43.2 Access Control and Status Registers (CSRs): `oct-pci-csr`

The `oct-pci-csr` command can be used to examine OCTEON Command and Status Registers.

In the example below, `oct-pci-csr` is used to examine the board's reset state. The reset state is stored in the `CIU_PP_RST` Central Interrupt Unit Command and Status Register.

```
host$ oct-pci-csr CIU_PP_RST
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x8000000]
CIU_PP_RST(0x0001070000000700) = 0x000000000000ffff <> Board is in reset
  RESERVED_16_63      =          0 (0x0)
    RST                =      32767 (0x7fff)
    RST0               =          1 (0x1)
```

The low N bits of this register correspond to the N cores available on the OCTEON model, with core 0 in bit position 0. As shown above, the value of the register, when a 16-core OCTEON processor is reset, is 0xFFFF. Bits $<63:16>$ should read as zero. The value of the register (0x000000000000ffff) is displayed. The values of three fields in the register (RESERVED_16_63, RST, and RST0) are displayed separately.

Per the Hardware Reference Manual:

- When the external pin PCIBOOT is asserted, all cores are held in reset after any OCTEON processor hard or soft reset (i.e. on a 16-core OCTEON processor, `CIU_PP_RST` resets to 0xFFFF).
- When the external pin PCIBOOT is de-asserted, core 0 is not held in reset after any OCTEON processor hard or soft reset, but other cores are (i.e. `CIU_PP_RST` resets to 0xFFE).

Cores are taken out of reset by writing a 0 to the corresponding bit in `CIU_PP_RST`. This can be done either by a core or by a remote PCI host.

44 Appendix T: Multiple Embedded Root Filesystem Builds

If multiple users are sharing the same system, then a problem can occur during the embedded root filesystem build.

One of the steps in the embedded root filesystem build is to create a temporary directory:

```
/tmp/root-rootfs
```

The directions in this tutorial assume there is only one user on a system. If that is the case, then this problem will not occur. However, if multiple users are all doing this build on the same system, then they will over-write each other's work in this temporary directory.

The solution is to modify the `sudoers` file to add the NOPASSWD option:

```
testname    ALL = NOPASSWD: ALL
```

When this option is specified, `sudo` will not prompt for a password. This will allow the users to type `make kernel` without the preceding `sudo`. The temporary directory name will now be unique for each user: `/tmp/<username>-rootfs`.

For example, the user `testname` will use the directory:

```
/tmp/testname-rootfs
```

Warning: Setting the NOPASSWD option presents a security issue: If someone gets access to the system as a normal user, they now have root permission. Similarly, if the user walks away from the computer leaving the login session open, anyone who walks up has root permission.

Note: if you have been building Linux as root, and then switch to this NOPASSWD option, even if you first perform sudo make clean, build problems may occur from prior files owned by root which are not cleaned up.

For example, the following fatal error occurred (visible in \$OCTEON_ROOT/linux/make.out):

```
make -C .. /linux SUBDIRS=`pwd` modules;
make[5]: Entering directory `/home/testname/sdk18/linux/kernel_2.6/linux'
  CC [M]  /home/testname/sdk18/linux/kernel_2.6/intercept-example/intercept.o
  LD [M]  /home/testname/sdk18/linux/kernel_2.6/intercept-example/intercept-
example.o
/bin/sh: line 1:
/home/testname/sdk18/linux/kernel_2.6/intercept-example/.intercept-
example.o.cmd: Permission denied
```

An investigation shows a hidden file from the prior build which was not removed by the sudo make clean command:

```
host$ cd kernel_2.6/intercept-example
host$ ls -al
total 236
drwxr-xr-x  3 testname root      4096 Feb 22 14:18 .
drwxr-xr-x  4 testname root      4096 Jan 28 10:48 ..
-rw-r--r--  1 root     root      5901 Jan 28 10:48 intercept.c
-rw-r--r--  1 root     root      375  Feb 18 16:18 .intercept-example.ko.cmd
-rw-r--r--  1 root     root     12655 Feb 18 16:18 .intercept-example.mod.o.cmd
-rw-r--r--  1 testname software  77975 Feb 22 14:18 intercept-example.o
-rw-r--r--  1 root     root      287  Feb 18 16:18 .intercept-example.o.cmd
-rw-r--r--  1 testname software  77624 Feb 22 14:18 intercept.o
-rw-r--r--  1 testname software 19233 Feb 22 14:18 .intercept.o.cmd
-rw-r--r--  1 root     root      2562 Jan 28 10:48 Makefile
-rw-r--r--  1 root     root       0 Jan 28 10:48 Module.symvers
-rw-r--r--  1 root     root     3997 Jan 28 10:48 README.txt
drwxr-xr-x  2 testname root      4096 Feb 22 14:18 .tmp_versions
# Fix the problem by removing the .intercept* files
host$ rm .intercept*
```

Another make error:

```
cp: cannot create regular file
`/home/testname/sdk18/linux/embedded_rootfs/.. /kernel_2.6/linux/usr/initramfs_data.cpio':
Permission denied
```

Remove this file also. In kernel 2.6, these were all the changes required to allow the build to complete.

45 Appendix U: How to Find the Process's Core Number

When running Linux, it is possible to determine which core is running which process by the following technique.

In Section 22 – “Hands-on: Run `linux-filter` as a SE-UM Application on Multiple Cores”, `linux-filter` was started on cores 1 and 2 by using the command `oncpu 0x6 linux-filter`.

Use the `ps` command to get their Process IDs (PIDs):

```
host$ ps
  PID  Uid      VmSize Stat Command
    1 root        836 S   init
<text omitted>
  741 root       640 S   syslogd
  743 root       336 S   telnetd -l /bin/ash
  745 root       940 S   /bin/sh
759 root       324 S   /examples/linux-filter
760 root       184 S   /examples/linux-filter
<text omitted>
```

For each process, there is a `stat` file which contains the process information: the fourth-to-last item in the information is CPU number.

```
target# cat /proc/759/stat
 759 (linux-filter) S 745 759 745 1088 768 4194304 485 0 0 0 0 0 0 15 0 1 0
120
70 1290240 81 18446744073709551615 4831838208 4832438864 1099503726112
109950372
4976 4832217600 0 0 0 0 18446744071563596012 0 0 18 2 0 0 0

target# cat /proc/760/stat
1 1290240 46 18446744073709551615 4831838208 4832438864 1099503726112
1099503724
976 4832217600 0 0 0 0 18446744071563596012 0 0 18 1 0 0 0
```


Software Debugging Tutorial

TABLE OF CONTENTS

TABLE OF CONTENTS	1
LIST OF TABLES.....	4
LIST OF FIGURES	5
1 Introduction.....	6
1.1 Where to Get More Information	7
2 Getting Started Debugging	7
2.1 Types of Software Which May be Debugged.....	7
2.2 Different Debuggers	7
2.3 Runtime Environments	8
2.4 Cross-Debugging versus Native Debugging	9
2.4.1 Native Debugging Using the Debian Root Filesystem.....	9
2.4.2 Native Debugging Using the Embedded Root Filesystem	10
2.4.3 Native Debugging Using NFS	11
2.5 Cross-Debugging Connection Types.....	11
2.5.1 SE-S Applications and the Linux Kernel.....	11
2.5.2 Linux User-Mode Applications	12
2.5.3 Summary: Connection Choices	14
2.6 Hardware Configuration for Debugging.....	15
2.7 The First Breakpoint in the Application	15
2.8 The First Breakpoint in the Kernel	16
2.9 Multithread Debugging.....	16
2.10 Multicore Debugging.....	16
2.11 PCI Debugging GDB Commands.....	16
2.12 SDK Documentation.....	16
2.12.1 SE-S Application Debugging	16
2.12.2 Linux Kernel Debugging	17
2.12.3 Linux User-Mode Debugging.....	17
2.12.4 OCTEON Simulator Debugging.....	18
3 Building Applications and the Linux Kernel for Debugging	18
3.1 Building Applications for Debugging	18
3.1.1 Add the Debugging Flag (-g).....	18
3.1.2 Adjust the Optimization Level (-O0).....	19
3.2 Building the Linux Kernel for Debugging.....	20
4 Debugging Applications in the Embedded Root Filesystem.....	20
4.1 Verify Correct Installation.....	22

DEBUGGING
TUTORIAL

4.2	About Building the Embedded Root Filesystem	22
5	Hands-On: Debug a SE-S Application: <code>hello</code>	22
6	About Debugging SE-S Applications or the Linux Kernel	29
6.1	Quick Summary of <code>mipsisa64-octeon-elf-gdb</code>	29
6.2	Hardware Configuration for SE-S Applications and the Linux Kernel.....	30
6.3	Multicore Debugging Commands.....	30
6.4	Multicore Debugging and Barrier Sync.....	36
6.5	PCI Debugging Commands	36
6.5.1	Changes to Hands-On Steps When Using PCI Debugging	37
6.5.2	Multiple PCI Development Targets.....	40
6.5.3	Attaching to a Program Which is Already Running.....	40
6.6	Other Special Commands: <code>spawn-sim</code>	41
6.7	Summary: Directions for Different Connection Types	42
6.8	Software Breakpoints and Multicore Debugging	45
6.8.1	Race Condition: Cores Can Bypass the Breakpoint Without Stopping.....	45
6.8.2	Race Condition: Multiple Cores Stopped on the Same Breakpoint	46
6.9	Hardware Breakpoints	46
6.10	Hardware Watchpoints	47
6.11	Performance Counters	47
6.12	Finding the Cause of an Exception.....	48
7	Hands-On: Debug the Linux Kernel.....	48
7.1	Building the Linux Kernel for Debugging.....	49
7.1.1	Kernel Configuration	49
7.1.2	Rebuild Linux, Enable Frame Pointers.....	51
7.1.3	About the <code>make clean</code> Command.....	52
7.2	Debug the Linux Kernel	52
7.3	Example: Multicore Debugging and the Linux Kernel	57
8	About Debugging the Linux Kernel	58
8.1	Cavium Networks Proprietary GDB Protocol	58
8.2	The Standard Open Source Kernel Debugger	59
8.3	SMP Synchronization and <code>step-all</code>	59
8.4	The Kernel File Name: <code>vmlinux</code> vs <code>vmlinux.64</code>	59
9	Hands-On: Debug a SE-UM Application: <code>named-block</code>	59
10	About Linux User-Mode Application Debugging.....	66
10.1	Quick Summary of <code>mips64-octeon-linux-gnu-gdb</code>	66
10.2	Hardware Configuration for Linux User-Mode Debugging.....	67
10.3	Summary: Directions for Different Connection Types	68
10.4	The Management Port Ethernet Interface.....	69
11	EJTAG (Run-Control) Tools	71
12	About Debugging on the OCTEON Simulator.....	72
12.1	Debugging SE-S Applications on the Simulator	72
12.1.1	About <code>printf()</code> and the Simulator.....	76
12.1.2	Separating Console Output from Simulator Output	76
12.1.3	Using <code>simprintf()</code>	77
12.2	Simulator Magic Functions.....	78
12.3	Debugging Linux on the OCTEON Simulator	78

12.3.1	Building vmlinux to Run on the Simulator	78
12.3.2	Starting Linux on the Simulator	78
12.3.3	Running Linux User-Mode Applications on the Simulator.....	84
13	Appendix A: Common GDB Commands	84
14	Appendix B: Connecting Using a Terminal Server.....	86
14.1	Terminal Servers and “Garbage” Characters.....	87
15	Appendix C: How to Simplify the Command Lines	88
15.1	Script Files.....	88
15.2	Using an Alias to Simplify Start-Up.....	88
15.3	The .gdbinit file	89
15.4	Environment Variables	89
16	Appendix D: Graphical Debugger	89
17	Appendix E: Core Files	90
17.1	Core File Names	90
17.2	Example Core Dump	91
17.3	Example of Using ftpput to Transfer a Core File	91
17.4	Analyze Core File with GDB	92
17.5	The Executable Name is Required on GDB Command Line	93
18	Appendix F: The oct-debug Script.....	94
19	Appendix G: Debian and the Cavium Networks Ethernet Driver	95

LIST OF TABLES

Table 1: Different Debuggers Available.....	8
Table 2: Software Type and Cross or Native Debugging Availability.....	9
Table 3: Cross-Debugging Connection Types for SE-S and Linux Kernel.....	11
Table 4: Cross-Debugging Connection Types for Linux User-Mode Applications.....	13
Table 5: Connection Choices, Summarized.....	15
Table 6: SE-S Debugging - SDK Documentation	17
Table 7: Linux Kernel Debugging - SDK Documentation	17
Table 8: Linux User-Mode Debugging - SDK Documentation.....	18
Table 9: OCTEON Simulator Debugging - SDK Documentation	18
Table 10: Debug hello (SE-S) over Serial Connection, Part 1	25
Table 11: Debug hello (SE-S) over Serial Connection, Part 2	26
Table 12: Debug hello (SE-S) over Serial Connection, Part 3	27
Table 13: Debug hello (SE-S) over Serial Connection, Part 4	28
Table 14: Multicore Debugging Commands, Part 1	31
Table 15: Multicore Debugging Commands, Part 2	32
Table 16: PCI Debugging Commands	37
Table 17: Running GDB with PCI Development Targets, Part 1.....	39
Table 18: Running GDB with PCI Development Targets, Part 2.....	40
Table 19: Debug SE-S or Linux Kernel over PCI Bus	43
Table 20: Debug SE-S or Linux Kernel over Serial Connection.....	44
Table 21: Debug SE-S or Linux Kernel Using a Terminal Server	44
Table 22: Debug SE-S or Linux Kernel on the OCTEON Simulator.....	45
Table 23: Debug the Linux Kernel – Part 1	53
Table 24: Debug the Linux Kernel – Part 2	54
Table 25: Debug the Linux Kernel – Part 3	55
Table 26: Debug the Linux Kernel – Part 4	56
Table 27: Debug the Linux Kernel – Part 5	57
Table 28: Debug named-block (SE-UM) over TCP Sockets, Part 1.....	61
Table 29: Debug named-block (SE-UM) over TCP Sockets, Part 2.....	62
Table 30: Debug named-block (SE-UM) over TCP Sockets, Part 3.....	63
Table 31: Debug named-block (SE-UM) over TCP Sockets, Part 4	64
Table 32: Debug named-block (SE-UM) over TCP Sockets, Part 5	65
Table 33: Native Debugging of Linux User-Mode Applications	68
Table 34: Debugging Linux User-Mode Applications over Serial Connection	68
Table 35: Debug Linux User-Mode Applications over Ethernet.....	69
Table 36: Run Linux on the OCTEON Simulator, Part 1.....	81
Table 37: Run Linux on the OCTEON Simulator, Part 2.....	82
Table 38: Run Linux on the OCTEON Simulator, Part 3.....	83
Table 39: Run Linux on the OCTEON Simulator, Part 4.....	84
Table 40: A Few Common GDB Commands.....	85

LIST OF FIGURES

Figure 1: SE-S Applications or the Linux Kernel: Cross-Debugging	12
Figure 2: Linux User-Mode Applications: Cross or Native Debugging.....	14
Figure 3: Debug Hello on One Core.....	23
Figure 4: Hardware Configurations for SE-S and Linux Kernel Debugging	30
Figure 5: Debug <code>hello</code> on Two Cores	33
Figure 6: The <code>set focus</code> Command	33
Figure 7: The <code>set active-cores</code> Command	34
Figure 8: Active Cores All Stop when One Hits a Breakpoint.....	34
Figure 9: Effect When <code>step-all</code> is on.....	35
Figure 10: Effect When <code>step-all</code> is off	35
Figure 11: Race Condition: Cores Can Bypass Breakpoint.....	46
Figure 12: Debug the Linux Kernel.....	49
Figure 13: Debug <code>named-block</code>	60
Figure 14: Hardware Configurations for Linux User-Mode Debugging.....	67
Figure 15: Debugging SE-S Applications on the OCTEON Simulator	73
Figure 16: Debugging Linux on the OCTEON Simulator.....	80
Figure 17: Terminal Server - SE-S or Linux Kernel Debugging.....	87

1 Introduction

This chapter provides an introduction to debugging on the OCTEON processor, and introduces debugging SE-S applications, the Linux kernel, and Linux user-mode applications, including SE-UM applications.

This tutorial assumes the development target has already booted to a bootloader prompt, and can correctly run both SE-S and SE-UM applications.

The hands-on sections in this tutorial are:

- Section 5 – “Hands-On: Debug a SE-S Application: `hello`”
- Section 7 – “Hands-On: Debug the Linux Kernel”
- Section 9 – “Hands-On: Debug a SE-UM Application: `named-block`”
- Section 12.1 – “Debugging SE-S Applications on the Simulator”
- Section 12.3 – “Debugging Linux on the OCTEON Simulator”

It is recommended that the reader follow the hands-on directions for debugging `hello` to make sure the hardware setup is correct before moving on to more complex debugging configurations.

This chapter assumes the reader will use either an evaluation or a reference development target to follow the steps provided in the chapter. To maximize understanding of this chapter, the reader will need:

- An i386 or x86_64 development host, running Linux
- An OCTEON reference or evaluation development target
- The Cavium Networks OCTEON SDK
- Root privilege on the development host

Before reading this chapter, please read the *Packet Flow*, *Software Overview*, and the *SDK Tutorial* chapters. This chapter depends on the reader having correctly performed all the steps in the *SDK Tutorial* chapter.

Because this chapter is a tutorial, hands-on sections are labeled “Hands-On”. Discussion sections are labeled “About”. The chapter should be read in order, and the steps executed in order except for the reference section.

Note: In the text below,

- `host$` represents the development host command prompt. Execute the command as a normal user on the development host. Commands which require `root` privilege will be preceded by `sudo`, which is used to obtain the `root` privilege. When typing in the command as shown in the tutorial, omit the “`host$`” text.
- `target#` means to type in the terminal session connected to the development target (the `minicom` window). This prompt may represent either the U-Boot command prompt, or

the Linux command prompt. Target Linux commands should be executed as `root`. When typing in the command as shown in the tutorial, omit the “target#” text.

- `gdb>` means the GDB prompt. When typing in the command as shown in the tutorial, omit the “`gdb>`” text.
- The text which should be typed is shown in boldface.

1.1 Where to Get More Information

In addition to the material contained in this chapter, each API chapter in the *OCTEON Programmer's Guide*, such as the *FPA* chapter (Volume 2), contains essential information to avoid bugs. For example, the most common bug we see is freeing the same FPA buffer multiple times by accident. The *FPA* chapter includes the following sections:

1. Basic Code Review Checklist
2. Debugging (including a Common Mistakes section)
3. Performance Tuning Checklist
4. Advanced Code Review Checklist

Other resources include the extensive documentation which is supplied with the SDK, and the *Hardware Reference Manual (HRM)* for the specific OCTEON model used in your application.

2 Getting Started Debugging

The tool used in debugging depends on the type of software to be debugged. To avoid being lost in the complexity of choices, the reader should start by answering the following questions, in order:

1. The type of software to be debugged (determines choice of debugger).
2. Whether native debugging is available and desired (depends on type of software)
3. If cross-debugging, the type of runtime environment used (determines cross-debugging connection types for chosen debugger).

2.1 Types of Software Which May be Debugged

Several types of software may be debugged:

1. SE-S applications
2. Linux kernel
3. Linux user-mode applications, including SE-UM applications
4. Bootloader

DEBUGGING
TUTORIAL

2.2 Different Debuggers

There are two different debuggers:

1. The `mipsisa64-octeon-elf-gdb` debugger is used to debug SE-S applications and the Linux kernel. Cavium Networks-specific multicore GDB commands have been added to enable multicore debugging. These enhancements are mentioned in Section 2.10 – “Multicore Debugging”, and discussed in detail in Section 6.3 – “Multicore Debugging Commands”. Cavium Networks-specific enhancements have also been added to simplify debugging when using a PCI development target. These enhancements are discussed in detail in Section 6.5 – “PCI Debugging Commands”.

2. The `mips64-octeon-linux-gnu-gdb` debugger is used to debug Linux user-mode applications, including SE-UM applications. This debugger can only debug one process at a time (see Section 2.9 – “Multithread Debugging”).

Both of these debuggers will identify themselves as a Cavium Networks version in response to the `gdb -v` command.

```
host$ mipsisa64-octeon-elf-gdb -v
GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64
Copyright (C) 2006 Free Software Foundation, Inc.
<text omitted>
This GDB was configured as "--host=i686-pc-linux-gnu --target=mipsisa64-octeon-elf".

host$ mips64-octeon-linux-gnu-gdb -v
GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64
Copyright (C) 2006 Free Software Foundation, Inc.
<text omitted>
This GDB was configured as "--host=i686-pc-linux-gnu --target=mips64-octeon-linux-gnu".
```

In addition to these debuggers, EJTAG (run-control) tools are supplied by Cavium debugger tool partners. A brief introduction to debugging using EJTAG is provided in Section 11 – “EJTAG (Run-Control) Tools”.

Table 1: Different Debuggers Available

Software Type	Debugger Choices	
SE-S applications	<code>mipsisa64-octeon-elf-gdb</code>	EJTAG
Linux kernel	<code>mipsisa64-octeon-elf-gdb</code>	EJTAG
Linux user-mode applications, including SE-UM applications	<code>mips64-octeon-linux-gnu-gdb</code>	EJTAG
Bootloader	EJTAG	

2.3 Runtime Environments

There are 3 different runtime environments:

1. PCI development target
2. Stand-alone development target
3. Simulated development target (OCTEON simulator)

Debugging on the OCTEON simulator is covered Section 12 – “About Debugging on the OCTEON Simulator” and the SDK document “*OCTEON Simulator*”.

2.4 Cross-Debugging versus Native Debugging

The term *cross-debugging* refers to running the debugger (`gdb`) on the development host and connecting to the development target. The exact connection methods available depend on the type of software being debugged and the runtime environment (PCI, stand-alone, or simulator). If a Linux user-mode application is being debugged, then `gdbserver` is run on the development target, otherwise `gdbserver` is not needed.

The term *native debugging* refers to running the debugger on the development target. Native debugging is only supported for Linux user-mode applications. (Note: SDK 1.9 introduces new capabilities not available when this chapter was written.) Native debugging can be used with `embedded_rootfs`, or Debian Linux on CompactFlash. In the case of native debugging, `gdb` and `gdbserver` can be run on the development target as needed.

Table 2: Software Type and Cross or Native Debugging Availability

Software Type	Cross Debugging	Native Debugging
SE-S applications	Yes	No
Linux kernel	Yes	No
Linux user-mode applications, including SE-UM applications	Yes	Yes

2.4.1 Native Debugging Using the Debian Root Filesystem

When running the Debian filesystem on an OCTEON core, the Cavium Networks tool chain is located in `/usr/local/Cavium_Networks/OCTEON-SDK/tools/usr/bin` on the development target. The compiler is named `gcc` (without the `mips64-octeon-linux-gnu-*` prefix). The `PATH` environment variable must be modified to use the Cavium Networks toolchain instead of the standard toolchain that is part of the Debian filesystem:

```
target# PATH=/usr/local/Cavium_Networks/OCTEON-SDK/tools/usr/bin:$PATH
```

Use the command `gcc -v` to see if the `PATH` variable was set up correctly. If it is correct, the `gcc` version will include the string “Cavium Networks Version”.

Native tools run on the Debian root filesystem create an efficient debugging environment: the source code can be modified, re-compiled, and re-tested using native tools.

When debugging Linux user-mode applications which reconfigure the hardware units, see Section 19 – “Appendix G: Debian and the Cavium Networks Ethernet Driver” for directions on how to disable the automatic loading of the Ethernet driver.

2.4.2 Native Debugging Using the Embedded Root Filesystem

The GDB utility is installed in the embedded root filesystem in the `/usr/bin` directory, and can be used from there, as shown in the following example:

```

target# find . -name gdb -print
./usr/bin/gdb
target# cd /examples
target# ls
busybox-testsuite  fpa          named-block      testsuite
cause_core         linux-filter   openssl-testsuite zip
crypto             low-latency-mem passthrough
target# gdb -v
GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "mips64-octeon-linux-gnu".
target# gdb named-block
GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "mips64-octeon-linux-gnu"...Using host
libthread_db 1
library "/lib64/libthread_db.so.1".

gdb> b appmain
Breakpoint 1 at 0x120003acc: file named-block.c, line 39.
gdb> run
Starting program: /examples/named-block
CVMX_SHARED: 0x1201a0000-0x1201b0000
Active coremask = 0x1

Breakpoint 1, appmain () at named-block.c:39
39      named-block.c: No such file or directory.
      in named-block.c

```

Note that, for source-level debugging, the source code must be also copied to the embedded root filesystem. Also, the code cannot be recompiled using native tools (`gcc` is not present in the embedded root filesystem). Because of this limitation, native debugging on the embedded root filesystem is not as efficient as using the Debian root filesystem.

2.4.3 Native Debugging Using NFS

Native debugging can also be done on a root filesystem which is NFS-mounted from the development host onto the development target. Because NFS relies on Ethernet, this method cannot be used with applications which reconfigure hardware units which are needed by Ethernet. The Management Port Ethernet Interface, which is available on some models, is a good choice for native debugging using NFS. The driver for this interface does not use the hardware units. Debugging over NFS is not discussed in detail in this chapter.

2.5 Cross-Debugging Connection Types

The cross-debugging connection types available depend on the type of software being debugged and the runtime environment.

2.5.1 SE-S Applications and the Linux Kernel

The following table shows the connection types available for SE-S applications and the Linux kernel:

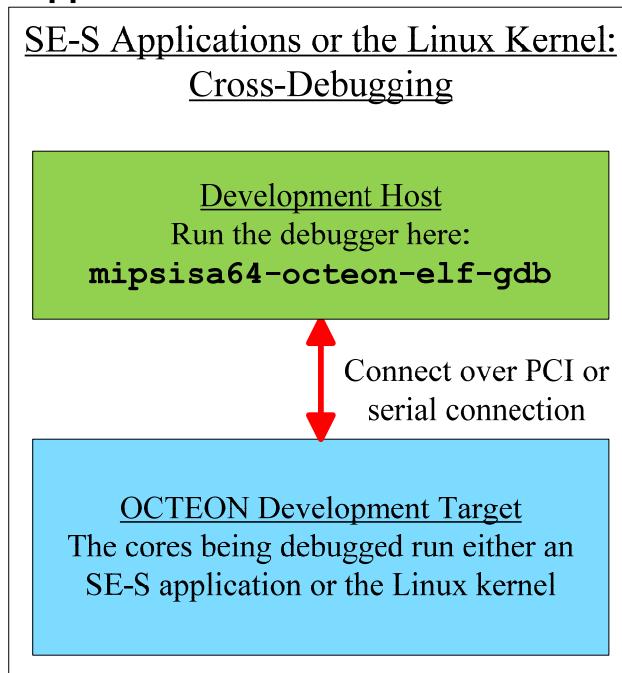
Table 3: Cross-Debugging Connection Types for SE-S and Linux Kernel

SE-S Applications and Linux Kernel (mipsisa64-octeon-elf-gdb)					
Software Type		SE-S		Linux Kernel	
Board Type		PCI Board	Standalone Board	PCI Board	Standalone Board
Connection Type	Serial Connection	✓	✓	✓	✓
	PCI Connection	✓	---	✓	---

Note 1: Serial Connection means an RS-232 cable to UART1.

Note: Debugging over PCI is supported for SE-S applications and the Linux kernel. See Section 6.5 – “PCI Debugging Commands” for more information.

GDB runs on the development host and connects to the development target, as shown in the following figure.

Figure 1: SE-S Applications or the Linux Kernel: Cross-Debugging

2.5.2 Linux User-Mode Applications

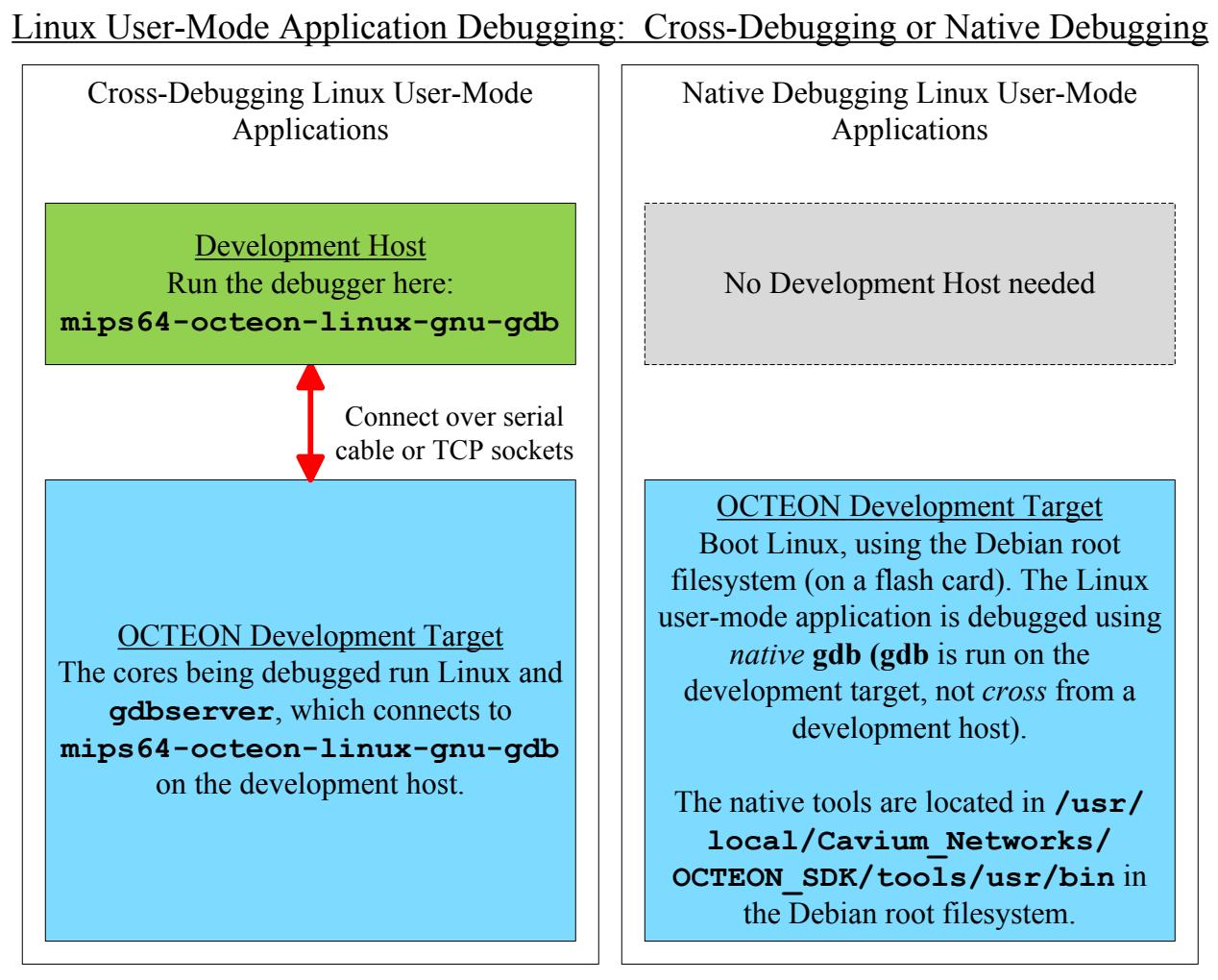
The following connection types are available for Linux user-mode applications, including SE-UM applications:

Table 4: Cross-Debugging Connection Types for Linux User-Mode Applications

Linux User-Mode Applications (including SE-UM applications) (mips64-octeon-linux-gnu-gdb)			
Software Type	Linux User-Mode Including SE-UM		
Board Type	PCI Board	Standalone Board	
Connection Type	Serial Connection	✓	✓
	Native Debugging	✓ (Preferred if CF is available)	✓
	TCP Sockets	✓ (Limited usefulness - See Note 2)	✓ (Limited usefulness - See Note 2)

Note 1: Serial Connection means an RS-232 cable to UART1.
Note 2: TCP Sockets cannot be used with SE-UM applications that re-initialize OCTEON hardware. The Ethernet driver has already initialized the hardware. Re-initializing the hardware will cause the Ethernet driver to stop working. An exception to this rule occurs when the management port Ethernet interface is used. This interface is available on some OCTEON models.

Figure 2: Linux User-Mode Applications: Cross or Native Debugging



2.5.3 Summary: Connection Choices

The following table summarizes the connection choices.

Table 5: Connection Choices, Summarized

The best connection choice is highlighted for each category. (✓ if available; --- if not available)							
Software Type		SE-S		SE-UM		Linux Kernel	Notes
Board Type		PCI Board	Stand-alone Board	PCI Board	Stand-alone Board	PCI Board	Standalone Board
Connection Type	Serial Connection	✓	✓	✓	✓	✓	✓ <i>Connect the null-modem serial cable to the debugging port, UART1, on target board.</i>
	PCI Connection	✓	---	---	---	✓	---
	Native Debugging	---	---	✓ (Preferred if CF is available)	✓	---	---
	TCP Sockets	---	---	✓ (limited usefulness)	✓ (limited usefulness)	---	---

2.6 Hardware Configuration for Debugging

Hardware Configuration is discussed in the following sections:

- SE-S Applications: See Section 6.2 – “Hardware Configuration for SE-S Applications and the Linux Kernel”.
- Linux Kernel: See Section 6.2 – “Hardware Configuration for SE-S Applications and the Linux Kernel”.
- Linux User-Mode Applications: See Section 10.2 – “Hardware Configuration for Linux User-Mode Debugging”.

2.7 The First Breakpoint in the Application

For SE-S applications, the first breakpoint is usually set at `main()`.

For Linux user-mode applications (including SE-UM applications), the first breakpoint is usually set at `appmain()`.

2.8 The First Breakpoint in the Kernel

The routine `r4k_wait()` is a convenient first breakpoint in the kernel.

2.9 Multithread Debugging

The `mips64-octeon-linux-gnu-gdb` debugger is used to debug Linux user-mode applications, including SE-UM applications. GDB's *multithread* debugging capability is standard for GDB.

Note that while `gdb` can be used to debug multiple threads, it can only debug one *process* at a time. When SE-UM applications are started on multiple cores under Linux, each core runs a different SE-UM process.

2.10 Multicore Debugging

The `mipsisa64-octeon-elf-gdb` command can be used to debug an SE-S application or the Linux kernel. Cavium Networks has added *multicore* debugging commands to this version of `gdb`. These commands are used when the same application (or the Linux kernel) has been loaded on multiple cores. See Section 6.3 – “Multicore Debugging Commands” for details about the commands.

Debugging SE-S applications is focused on the *core*, instead of the *process*. When using multicore debugging, the same image file (such as `hello`) must be loaded on multiple cores, and must be in the same load set.

From the perspective of the debugger, the Linux kernel is an SE-S application. Multicore debugging of the Linux kernel using `mipsisa64-octeon-elf-gdb` is the same as debugging an SE-S application, and multicore commands are supported. See Section 8 – “About Debugging the Linux Kernel” for more information.

2.11 PCI Debugging GDB Commands

PCI debugging is supported when debugging SE-S applications and the Linux kernel when using an OCTEON PCI target and the `mipsisa64-octeon-elf-gdb` debugger. See Section 6.5 – “PCI Debugging Commands” for more information.

2.12 SDK Documentation

2.12.1 SE-S Application Debugging

The following SDK documentation provides more information on SE-S application debugging:

Table 6: SE-S Debugging - SDK Documentation

Debugging SE-S Applications	
Connection Type	SDK Document
PCI	<i>Developing with OCTEON as a PCI Target</i>
Serial Line	<i>Simple Executive Debugger</i>
Terminal Server	<i>Simple Executive Debugger</i>

Note: A *terminal server* is a remote host which has a direct serial connection to the development target's debug port. This direct connection is identified by a server port number on the terminal server. Typically a terminal server is connected to multiple development targets.

2.12.2 Linux Kernel Debugging

The following SDK documentation provides more information on Linux kernel debugging:

Table 7: Linux Kernel Debugging - SDK Documentation

Debugging Linux Kernel	
Connection Type	SDK Document
PCI	<i>Linux on the OCTEON</i>
Serial Line	<i>Linux on the OCTEON</i>
Terminal Server	<i>Simple Executive Debugger</i>

Note: A *terminal server* is a remote host which has a direct serial connection to the development target's debug port. This direct connection is identified by a server port number on the terminal server. Typically a terminal server is connected to multiple development targets.

2.12.3 Linux User-Mode Debugging

The following SDK documentation provides more information on debugging Linux user-mode applications, including SE-UM applications:

Table 8: Linux User-Mode Debugging - SDK Documentation

Debugging Linux User-Mode Applications	
Cross-Connection Type	SDK Document
Serial Connection	<i>Linux Userspace Debugging</i> : includes how to use native gdbserver to connect to gdb on a host system over Serial connection connected at the Debug Port (UART1).
TCP Sockets	<i>Linux Userspace Debugging</i> : includes how to use native gdbserver to connect to gdb on a host system over TCP sockets.
Native Debugging	<ol style="list-style-type: none"> 1. <i>Linux on the OCTEON, Running Linux on the EBT3000 Hardware</i>: includes directions for copying vmlinuz to a flash card, and using the Debian root filesystem. 2. <i>Linux Userspace Debugging</i>: includes native debugging.

2.12.4 OCTEON Simulator Debugging

The following SDK documentation provides more information on debugging on the OCTEON simulator:

Table 9: OCTEON Simulator Debugging - SDK Documentation

Debugging on Simulated Hardware	
SE-S Simulator	<i>Simple Executive Debugger</i>
Linux Kernel on simulator	<i>Linux on the OCTEON - Running Linux on the Simulator</i>
The Bootloader and the Simulator	<i>OCTEON Bootloader - Simulator Specific Usage</i>

3 Building Applications and the Linux Kernel for Debugging

3.1 Building Applications for Debugging

The application must be compiled to include debugging information. It is also recommended that the optimization level be changed to simplify debugging.

3.1.1 Add the Debugging Flag (-g)

Before debugging an application, it must be compiled with the `-g` flag. The `-g` flag causes the compiler to add symbol information needed by the debugger.

Note that this flag is *on* by default in the examples.

3.1.2 Adjust the Optimization Level (-O0)

If seeing exact line numbers is needed when debugging, then change the optimization level to 0 (-O0). By default, the optimization level is set to 2 in \$OCTEON_ROOT/executive/cvmx.mk:

```
host$ cd $OCTEON_ROOT/executive
# use grep to find out where -O2 is set
# (use \ to escape the - character)
host$ grep -n "\-O2" *.mk
119:$ (OBJS_$(d)) :  CFLAGS_LOCAL := -I$(d) -O2 -g -W -Wall -Wno-unused-parameter
-Wundef
143:CFLAGS_SPECIAL := -I$(d) -I$(d)/cvmx-malloc -O2 -g -DUSE_CVM_THREADS=1
-D_REENTRANT
```

To change the optimization level when building the examples, change the value in \$OCTEON_ROOT/executive/cvmx.mk to the new optimization level:

```
$ (OBJS_$(d)) :  CFLAGS_LOCAL := -I$(d) -O0 -g -W -Wall -Wno-unused-parameter
-Wundef
<text omitted>
CFLAGS_SPECIAL := -I$(d) -I$(d)/cvmx-malloc -O0 -g -DUSE_CVM_THREADS=1
-D_REENTRANT
```

In the case of the named-block example, it may also be necessary to edit the local Makefile. The grep command can be used to locate Makefile which set the -O2 flag:

```
host$ cd $OCTEON_ROOT/examples/named-block
# use grep to find out where -O2 is set
# (use \ to escape the - character)
host$ grep "\-O2" Makefile
CFLAGS_LOCAL = -g -O2 -W -Wall -Wno-unused-parameter
```

Note: After editing the local Makefile, you may need to use the vi command w! to write out the file. The Makefile permissions are:

```
-rw-r--r-- 1 root root 1535 Mar 19 13:52 Makefile
```

After w!, the Makefile owner and group are:

```
-rw-r--r-- 1 testname software 1535 Mar 19 13:56 Makefile
```

Then build the application with the correct optimization level showing in the make output:

```
host$ cd $OCTEON_ROOT/examples/named-block
host$ make clean OCTEON_TARGET=linux_64
host$ make OCTEON_TARGET=linux_64
mips64-octeon-linux-gnu-gcc -I/home/testname/sdk/target/include -Iconfig
-DUSE_RUNTIME_MODEL_CHECKS=1 -DCVMX_ENABLE_PARAMETER_CHECKING=0
-DCVMX_ENABLE_CSR_ADDRESS_CHECKING=0 -DCVMX_ENABLE_POW_CHECKS=0
-DOCTEON_MODEL=OCTEON_CN38XX -DOCTEON_TARGET=linux_64 -mabi=64 -march=octeon
-msoft-float -Dmain=appmain -I/home/testname/sdk/executive -O0 -g -W -Wall
-Wno-unused-parameter -Wundef -MD -c -o obj-linux_64/cvmx-bootmem.o
/home/testname/sdk/executive/cvmx-bootmem.c
```

(Note the “magic” which happens in the command line just shown (-Dmain=appmain). This text changes the string “main” into “appmain” in the code when building a Linux SE-UM target.)

The difference can be seen in the size of the ELF file:

When compiled with -O2:

```
host$ ls -l named-block-linux_64
-rwxr-xr-x 1 testname software 3400228 Mar 18 16:02 named-block-linux_64
```

When compiled with -O0:

```
host$ ls -l named-block-linux_64
-rwxr-xr-x 1 testname software 3417201 Mar 18 15:53 named-block-linux_64
```

3.2 Building the Linux Kernel for Debugging

See Section 7.1 – “Building the Linux Kernel for Debugging” for directions.

4 Debugging Applications in the Embedded Root Filesystem

If the SE-UM application will be installed in the embedded root filesystem, then configure the embedded root filesystem Makefile to *not* strip the file. Otherwise, the Makefile will call the strip command, and strip will remove the debugging information from the ELF file.

For example programs, the strip command is located in the Makefiles in the `embedded_rootfs` directory.

In SDK 1.8, the strip command is called from

`$OCTEON_ROOT/linux/embedded_rootfs/pkg_makefiles/final-cleanup.mk`:

```
ifdef CFG_STRIP_BINARIES
    for f in `find ${ROOT}/bin ${ROOT}/sbin ${ROOT}/usr/bin
${ROOT}/usr/sbin
${ROOT}/examples -not -type l -and -not -type d`; \
        do if sh -c "file $$f | grep -q ELF"; then ${STRIP} $$f; fi; done
endif
```

For SDK 1.6, the `strip` command is called from

`$OCTEON_ROOT/linux/embedded_rootfs/pkg_makefiles/sdk-examples.mk:`

```
ifdef SDK_EXAMPLES_NAMED_BLOCK
    ${STRIP} -o ${ROOT}/examples/named-block ${OCTEON_ROOT}/examples/named-
block/named-block-$ABI
endif
```

For example (SDK 1.8), in the `make.out` file (created by the `sudo make kernel >make.out 2>&1` command), `strip` will be called for all the examples:

```
for f in `find /tmp/root-rootfs/bin /tmp/root-rootfs/sbin \
/tmp/root-rootfs/usr/bin /tmp/root-rootfs/usr/sbin /tmp/root-rootfs/examples - \
not -type l -and -not \
-type d`; \
do if sh -c "file $f | grep -q ELF"; then mips64-octeon-linux-gnu-strip $f; fi;
done
```

For SDK 1.8, to prevent the `strip` step from happening, reconfigure the embedded root filesystem build:

```
host$ cd $OCTEON_ROOT/linux/embedded_rootfs
host$ sudo make menuconfig
```

The first embedded root filesystem `menuconfig` screen looks similar to this:

```
Global Options --->
  [*] device-files
  [*] busybox
  [*]   Include the Busybox testsuite
<text omitted>
  [*] final-cleanup
  [*]   Strip debugging information from kernel modules
  [ ]   Strip debugging information from binaries <<< DESIRED SETTING
  [*]   Strip debugging information from libraries
  ()   Directory to copy extra files from
```

To navigate this screen, use the arrow keys on the keyboard. Highlight the option “Strip debugging information from binaries” by selecting the option. If the option is set (*), then press `Enter` to clear the option.

Select `Exit` to exit `menuconfig`. When prompted “Do you wish to save your new configuration?” select `Yes`.

The output of the `make menuconfig` command is the file `.config`:

```
host$ ls -ld .config
-rw-r--r--  1 root      root          2245 Mar 18 13:14 .config
```

Then remove the `.root_complete` file. Removing this file will cause the embedded root filesystem to rebuild when the `make kernel` command is executed in the `$OCTEON_ROOT/linux` directory.

```
host$ sudo rm .root_complete
host$ cd ..
# the following build command can take about 15 minutes to complete
host$ sudo make kernel >make.out 2>&1 &
```

In the `make.out` file, expect to see:

```
cp /home/testname/sdk/examples/named-block/named-block-linux_64 /tmp/root-rootfs/examples/named-block
```

4.1 Verify Correct Installation

To verify the file was correctly installed in the embedded root filesystem:

```
# show the size of the compiled file
host$ cd $OCTEON_ROOT/examples/named-block
host$ ls -l named-block-linux_64
-rwxr-xr-x 1 testname software 3417201 Mar 18 15:53 named-block-linux_64

# verify the file was copied properly (not stripped)
cd /tmp/root-rootfs/examples
# check the size of the installed file
host$ ls -l named-block
-rwxr-xr-x 1 root root 3417201 Mar 18 16:02 named-block
```

In the example above, the file sizes match (3417201), verifying correct installation.

4.2 About Building the Embedded Root Filesystem

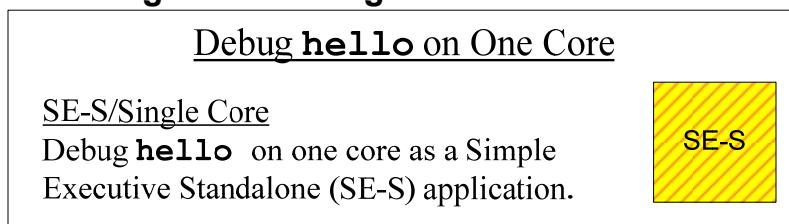
If any file in the embedded root filesystem is modified:

1. If the file needs to be built, type `make clean` in `$OCTEON_ROOT/linux/embedded_rootfs`, or use the following command in the `$OCTEON_ROOT/linux` directory:
`host$ make -s -C embedded_rootfs clean`
2. If the file is already built, and simply needs to be added to the root filesystem, remove the `$OCTEON_ROOT/linux/embedded_rootfs/.root_complete` file

Then type `make kernel` (or `make sim`) in `$OCTEON_ROOT/linux` directory.

5 Hands-On: Debug a SE-S Application: hello

The SE-S example `hello` is debugged in this hands-on step. A serial connection is used for cross-debugging because it works on both PCI and stand-alone development targets. This step should not be done until the prior steps in this chapter have been successfully completed.

Figure 3: Debug Hello on One Core

The step-by-step directions are provided in the tables below. A brief explanation precedes the tables. Detailed information is provided in Section 6 – “About Debugging SE-S Applications or the Linux Kernel”.

In this example, the development host serial port `/dev/ttyS1` is connected to the debug port on the development target (UART1). If a different serial port is used, modify the directions to use the correct value.

In this example, debugging over a serial connection will require two different sessions on the host: one to connect to the target console port, and one to connect to the target debugging port.

The development target configuration is illustrated in Figure 4 – “Hardware Configurations for SE-S and Linux Kernel Debugging”. The only addition to the configuration used in the *SDK Tutorial* chapter is the null-modem serial cable connection to the target’s debug port.

In the following table, the GDB prompt is shown as `gdb>`. In reality, the prompt looks more like `(Core#0-gdb)`. The *focus core* is shown in the prompt `(Core#0)`. (The focus core is the core which `gdb` is interacting with. Focus core is explained in more detail in Section 6.3 – “Multicore Debugging Commands”).

The directions in the following table are the hands-on instructions for debugging `hello`. After this section, more detailed information will be provided.

After compiling the program with the debug flag and the optimization level set appropriately, follow the instructions below. In this example, serial debugging is used to simplify the instructions.

When booting the application, specify `debug=<debug_port>` on the boot command line, as in:

```
target# bootct 0 coremask=<coremask> debug=1
```

The debug=1 command-line option sets the development target's debug port to UART1, and will cause the program enter the debug exception handler

(`__ocean_trigger_debug_exception()`) when it begins to run:

```
Sending bootloader command: bootoct 0x20000000 coremask=3 debug
Program received signal SIGTRAP, Trace/breakpoint trap.
0x10000388 in __ocean_trigger_debug_exception()
```

Note: The default UART used is UART1. Because UART1 is the default, the command line above can be entered as:

```
target# bootoct 0 coremask=<coremask> debug
```

Note: If UART0 is specified (debug=0), debugging and console output will use the same UART. It is important to close the minicom session to UART0 before beginning debugging in this case. Note that this configuration is not recommended because gdb can lock up.

At this point, the user can set the first breakpoint at `main()`.

Note: After the program stops in `__ocean_trigger_debug_exception()`, the `continue` command is used to resume execution; the `run` command is not used because the program is already running.

Table 10: Debug hello (SE-S) over Serial Connection, Part 1

Steps	Note
1. Connect the Hardware	
Connect serial cable to target console. Connect the Ethernet cable.	Follow directions in the <i>Quick Start Guide</i> . Note: The Ethernet cable is not needed to run <code>hello</code> on a PCI target board, but will be needed later in the <i>SDK Tutorial</i> . Be careful to isolate the test network from the office network so that experiments will not disturb the office network.
Connect the target debug port to the host using a serial cable.	GDB will communicate with the target using this connection.
2. Reset the board	
Power on or reset the target board.	The word <code>Boot</code> should appear on the red LEDs on the board. If not, the board is not configured to boot from flash, or something is wrong with the board, or the board does not have alpha LEDs.
3. Connect to the Target Console	
<code>host\$ minicom -w ttys0</code>	Substitute the serial port actually used on the host to connect to the OCTEON target board if it is not <code>ttys0</code> . Minicom will provide a connection to the target console. You should see the bootloader prompt.
4. Verify Bootloader Prompt is Visible	
<code>target# version</code>	The bootloader should reply with text similar to: <code>U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun 13)</code>
5. Verify Bootloader Version is at Least SDK 1.7	
If the bootloader's SDK version is not at least 1.7, then before continuing, upgrade the bootloader to a newer version.	Directions for upgrading the bootloader are included in the <i>SDK Tutorial</i> .
6. Build the Application	
<code>host\$ cd \$OCTEON_ROOT/examples/hello host\$ make clean host\$ make OCTEON_CFLAGS_GLOBAL_ADD=-O0</code>	The <code>make</code> command will create the executable file <code>hello</code> . The <code>OCTEON_CFLAGS_GLOBAL_ADD</code> will override default <code>-O2</code> optimization level set in the Makefile. The debugging flag (<code>-g</code>) is already set in the Makefile.
7. Copy the ELF file to the <code>tftpboot</code> Directory	
<code>host\$ sudo cp hello /tftpboot</code>	See <i>SDK Tutorial</i> directions for <code>tftpboot</code> .
<i>Continued in the next table...</i>	

Table 11: Debug hello (SE-S) over Serial Connection, Part 2

Steps	Note
8. Select Target IP Address, if Needed	
If a DHCP server is available, then selecting the target IP address is handled by the server. Otherwise, select a target IP address.	In this example, the target IP address is 192.168.51.159 .
9. Set the Development Target's IP Address	
9a. No DHCP Server	
First, set the IP address of the target (replace items in italic with your IP addresses). <i># Use your IP addresses instead of the example values!</i> target# setenv gatewayip 192.168.51.254 target# setenv netmask 255.255.255.0 target# setenv ipaddr 192.168.51.159 target# setenv serverip 192.168.51.1 <i># save the values so they will still be set after a reset</i> target# saveenv	Note: serverip is the IP address of the TFTP server.
9b. DHCP Server Available	
If a DHCP server is available substitute the following step: target# dhcp	Note: serverip is the IP address of the TFTP server. In this example the DHCP server is the same as the TFTP server.
10. Test the Ethernet Connection to the Host	
# <i>use your host IP address instead of the example value!</i> target# ping 192.168.51.254	Expect to see: Using octeth0 device host 192.168.51.254 is alive Note that the development target will <i>not</i> reply to a ping from the development host. Note: to see the development host's IP address, use the /sbin/ifconfig command on the development host.
11. Download the Application to the Development Target	
# <i>Use tftpboot to download the application.</i> target# tftpboot 0 hello	See the <i>SDK Tutorial</i> directions for tftpboot. If this step does not work, check the /etc/xinetd.d/tftp file on the host to verify that server_args = -s /tftpboot .
<i>Continued in the next table...</i>	

Table 12: Debug hello (SE-S) over Serial Connection, Part 3

Steps	Note
12. Boot the Application	
target# bootoct 0 coremask=0x1 debug	<p>This command will run hello on core 0.</p> <p>Expect to see:</p> <pre>setting debug flag! Bootloader: Booting Octeon Executive application at 0x20000000, core mask: 0x1, stack size: 0x100000, heap size: 0x300000 Bootloader: Done loading app on coremask: 0x1</pre>
13. Start GDB on the Host; Debug the Application	
host\$ cd \$OCTEON_ROOT/examples/hello	<p>Go to the directory where the hello source is located on the host.</p>
# start gdb host\$ mipsisa64-octeon-elf-gdb hello	<p>Use -q (quiet) for fewer start-up messages.</p> <p>Expect to see:</p> <pre>GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64 Copyright (C) 2006 Free Software Foundation, Inc. GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions. There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "--host=i686-pc-linux-gnu --target=mipsisa64-octeon-elf".</pre>
# connect to the debug port (substitute your tty port # for /dev/ttys1) gdb> target octeon /dev/ttys1	<p>Expect to see:</p> <pre>Remote target octeon connected to /dev/ttys1</pre>
# set a breakpoint in main() gdb> b main	<p>Expect to see:</p> <pre>Breakpoint 1 at 0x100004fc: file hello.c, line 44.</pre>
<i>Continued in next table...</i>	

Table 13: Debug hello (SE-S) over Serial Connection, Part 4

Steps	Note
13. Debug the Application, continued	
# continue to the breakpoint gdb> c	Expect to see: Continuing. Breakpoint 1, main () at hello.c:44 44 printf("\n");
# show 10 source lines at the breakpoint gdb> list	Expect to see: 39 40 /* printf() provides maximum flexibility, but is slow due to 41 ** the format string being processed in simulated code. Normal 42 ** buffering is done by the C library. 43 */ 44 printf("\n"); 45 printf("\n"); 46 printf("Hello world!\n"); 47 48 #if 0
# continue the program to the end gdb> c	Expect to see: Continuing. Program exited normally. (Core#0-gdb)
The output of hello will appear in the target console: PP0:~CONSOLE-> PP0:~CONSOLE-> PP0:~CONSOLE-> Hello world! PP0:~CONSOLE-> Hello example run successfully.	
14. Run the Application Again	
To re-run the program, start over by resetting the board and downloading the application again.	

6 About Debugging SE-S Applications or the Linux Kernel

In the prior section, `hello` was used as an example of debugging a SE-S application. This section provides more details about this debugging environment.

6.1 Quick Summary of `mipsisa64-octeon-elf-gdb`

The `mipsisa64-octeon-elf-gdb` debugger is used to debug:

- SE-S applications
- Linux kernel

Type of debugging available:

- Cross-Debugging

Debugger cross-connection types available:

- PCI bus (if using a PCI development target)
- Serial connection

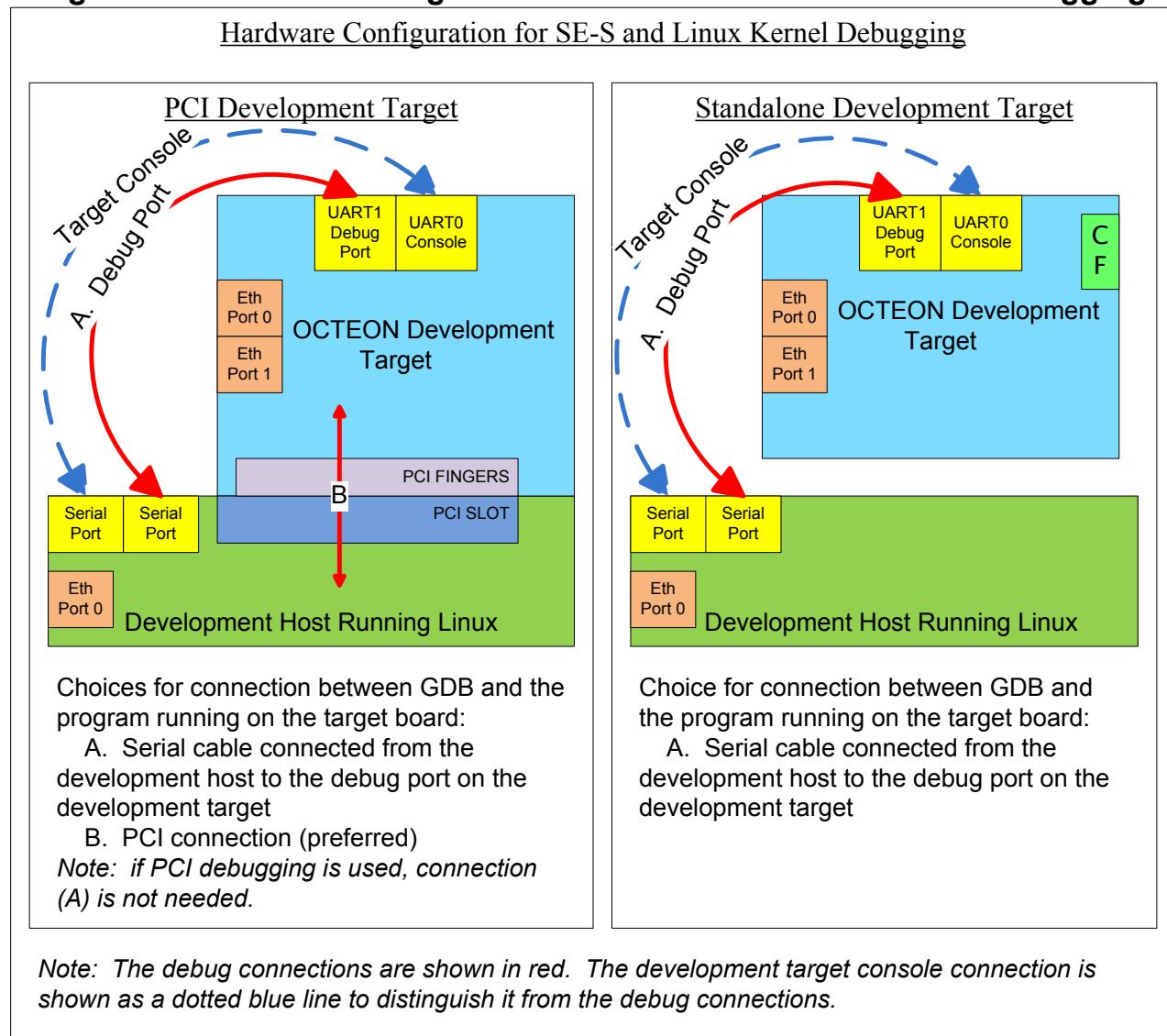
Special features:

- Multicore debugging commands
- Hardware breakpoints (on some models)
- Hardware watchpoints (on some models)
- Performance counters
- Attach to a running program
- Backtrace from an exception handler
- Special PCI debugging commands

6.2 Hardware Configuration for SE-S Applications and the Linux Kernel

The hardware configuration choices are shown in the following figures.

Figure 4: Hardware Configurations for SE-S and Linux Kernel Debugging



The serial port on the host should be configured identically for both the console and the debug port: 115200, 8, N, 1.

6.3 Multicore Debugging Commands

The following debugging commands are used to control multicore debugging. All cores to be debugged must be running the same image file (such as `hello`). More information may be found in the SDK document “Simple Executive Debugger”.

These commands control which core the debugger is interacting with, which cores stop when a breakpoint is hit, and which cores continue after the breakpoint.

Note: In the tables below, the prompt is shown as `gdb>`. The actual prompt looks similar to `(Core#0-gdb)`. The full prompt is not shown to allow the command text to fit into the column width of the tables.

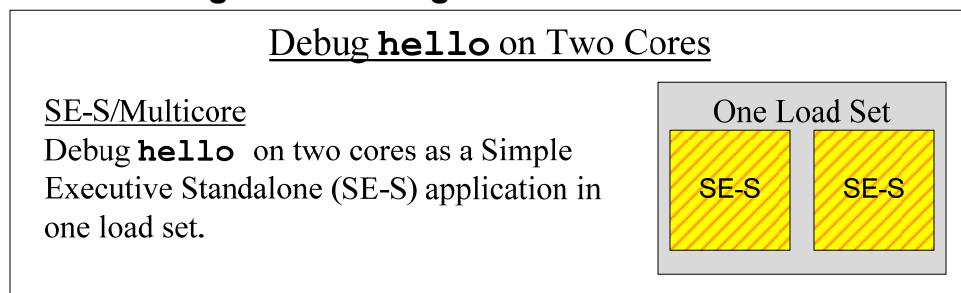
Table 14: Multicore Debugging Commands, Part 1

<i>* Note these commands do not work until after the target octeon command!</i>			
mipsisa64-octeon-gnu-gdb	Command	Description	Example
	set focus	Specify which core is directly interacting with the debugger. This is a number from 0-N where N is the maximum number of cores in the system minus one. The focus core is set automatically when a core hits a breakpoint. Data operations (memory or register reads and writes) are performed relative to the focus cores. Only a core currently stopped in a debug exception may become the focus core. (To make a non-active core the focus core requires a breakpoint, or changing the active-cores to include the core of interest.) The GDB prompt will show which core is the focus core. Note that the focus core may change if another core hits a breakpoint. The first core to hit a breakpoint becomes the focus core.	<pre>gdb> set focus 1 #0 0x10000388 in cvmx_write_csr (csr_addr=373292656, val=1844674407156281 9048) at /home/testname/sdk/t arget/include/cvmx.h :817 817 { (Core#1-gdb)</pre>
	show focus	Show which core is the current focus core.	<pre>gdb> show focus Expect to see: The currently debugged core is 1</pre>

Table 15: Multicore Debugging Commands, Part 2

<i>* Note these commands do not work until after the target octeon command!</i>			
	Command	Description	Example
mipsisa64-octeon-gnu-gdb	set active cores	Specify which cores are under active control of the debugger. All cores in this list will stop if <i>one</i> core in the list hits a breakpoint, allowing the debugger to control the set of active cores together. All cores which are not in this list (non-active cores) will continue (but they will suffer a performance hit). A non-active core will only stop when the core itself hits a breakpoint. The set of active cores are specified in a comma-separated list of cores. Setting the list to the NULL string is interpreted as setting all cores to active.	# type the following text on # one line: gdb> set active cores 0,1 Expect to see: (no reply from gdb)
	show active cores	Show which cores stopped on the breakpoint.	# type the following text on # one line: gdb> show active cores Expect to see (example): The cores stopped on execution of a breakpoint by another core is "0,1".
	set step-all	Specify which cores are affected by action commands (continue, step, or next). By default, step-all is off, so only the focus core performs the operation. Note that step-all also affects cores which are not in the active-cores list if they are currently stopped.	gdb> set step-all on Expect to see: (no reply from gdb)
	show step-all	Show the current value of step-all (on or off.	gdb> show step-all Expect to see (example): Step commands affect all cores is on.

The `hello` example can be run on multiple cores to test out the multicore commands. Simply boot it with the `coremask=0x3` to boot `hello` on 2 cores.

Figure 5: Debug hello on Two Cores

The following figures illustrate these commands.

Figure 6: The **set focus Command**

The **set focus** Command

Use the **set focus** command to specify which core is directly interacting with the debugger. Note in the second GDB prompt, **core1** is specified in the GDB command prompt, not **core0**. Cores number 0-N where N is the maximum number of cores in the system minus 1.

The **set focus** command is not the only way the focus core can be changed. When a core hits a breakpoint, the focus core is automatically set to that core.

```
(Core#0-gdb) set focus 1
(Core#1-gdb)
```

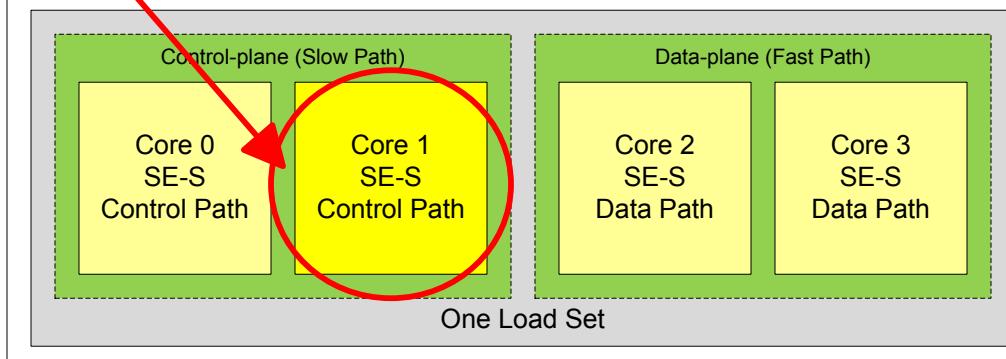
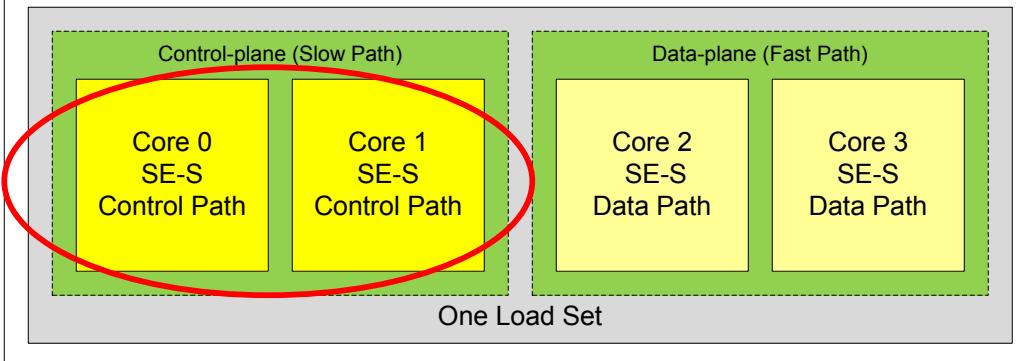


Figure 7: The `set active-cores` Command

The `set active-cores` Command
 Use `set active-cores` to specify which cores will stop when one of the cores in the set hits a breakpoint. In this example, the data plane cores continue running while software running on the control plane cores all stop when the breakpoint is hit by one core. Cores are specified in a comma-separated list.

(Core#0-gdb) `set active-cores 0,1`



When one of the active cores hits a breakpoint, all the active cores stop.

Figure 8: Active Cores All Stop when One Hits a Breakpoint

One of the Active Cores Hits a Software Breakpoint
 When one of the active cores hits a breakpoint, it stops and the other active cores also stop.

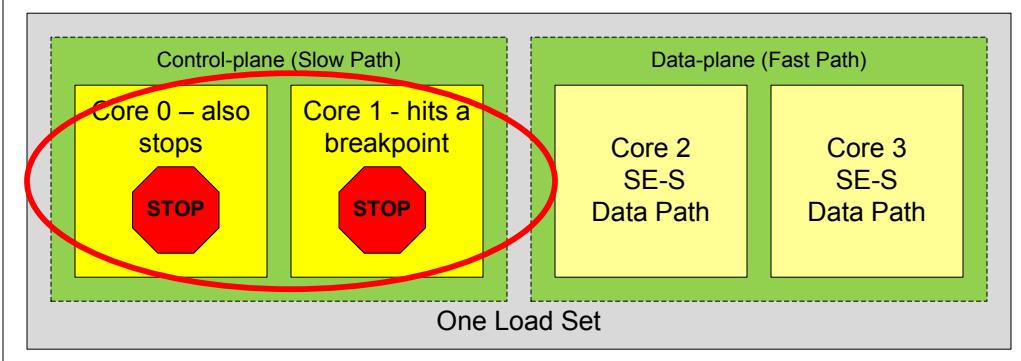
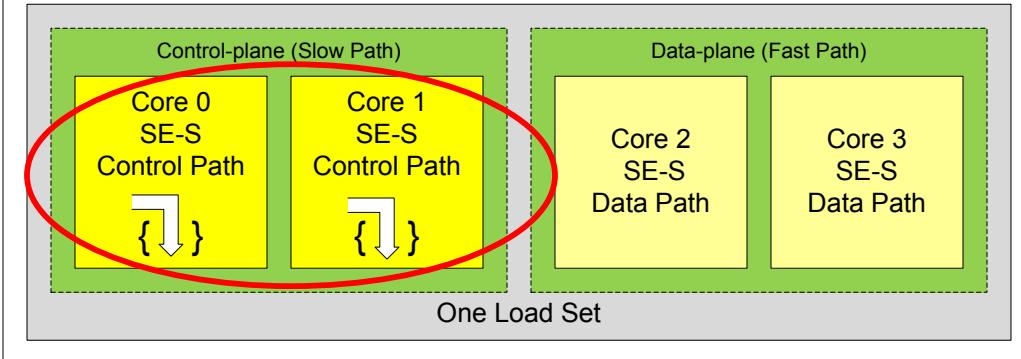


Figure 9: Effect When step-all is on**set step-all on**

If **step-all** is **on**, the **step**, **next**, and **continue** commands apply to all cores which are currently stopped (usually the active cores, but will also affect cores which are not active, but have stopped on their own breakpoints).

single step cores 0 and 1

```
(Core#1-gdb) s
```



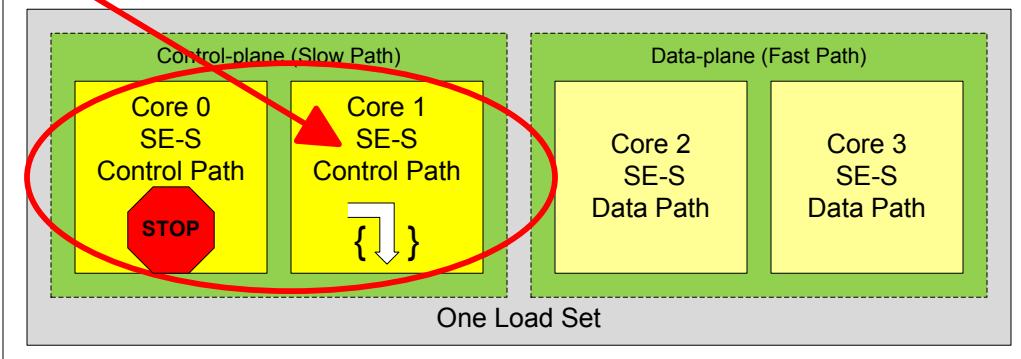
When step-all is OFF, only the focus core responds to the step, next, and continue commands.

Figure 10: Effect When step-all is off**set step-all off**

If **step-all** is **off**, the **step**, **next**, and **continue** commands apply only to the focus core. The focus core is automatically set to the core which hit the breakpoint.

single step core 1 only

```
(Core#1-gdb) s
```



After the application begins to run, it will stop automatically in the debug exception handler. To debug a set of cores as an active set, before continuing execution, set the desired breakpoints, then:

```
gdb> set active-cores <core-list>
gdb> set step-all on
```

When the first core hits the breakpoint, all cores in the active set will stop. As execution is resumed (`step`, `next`, `continue`), all active cores will obey the commands.

Note: All cores to be debugged must be running the same image file (such as `hello`). They should also all be in the same load set (loaded using the same `bootoct` command). (Cores running the same image file but loaded in different load sets may not work. This configuration has not been tested.)

6.4 Multicore Debugging and Barrier Sync

When using barrier synchronization (`cvmx_coremask_barrier_sync`) to synchronize the cores, place the first breakpoint after the barrier sync, set `step all` on, and then `continue` until after the barrier sync.

If the breakpoint is before the barrier sync, using `next` to single step will result in all cores hanging. An alternative is to set a breakpoint before the barrier sync, continue to that point, then set another breakpoint after the barrier sync and continue to that point.

Technical Note: The barrier sync is implemented using the load-link and store-conditional MIPS instruction pair. Load-link returns the value of a memory location. The store-conditional will only store a new value to that location if no updates have occurred since the load-link, or if no exception was taken. When using `next`, the debug exception will occur between the load-link and the store-conditional, causing the store-conditional to fail. The barrier sync condition will never be met, so the cores will hang.

6.5 PCI Debugging Commands

Debugging over PCI is very convenient. The debugger supports several PCI-only commands, including commands to reset the development target from inside the debugger and re-run the program.

When debugging over PCI, the command `target octeonpci bootoct <load_address> coremask=<coremask>` will reset the development target before loading the program. For development targets which don't boot from flash (or are not configured to boot from flash) the bootloader needs to be downloaded from the development host after reset.

Three debugging commands are only supported when debugging over PCI:

Table 16: PCI Debugging Commands

mipsisa64-octeon-gnu-gdb	Command	Description	Example
	set pci-bootcmd	Set to either oct-pci-boot (to download the bootloader after reset) or oct-pci-reset (if the bootloader is booted from flash). The default value is oct-pci-reset.	# type the following text on one line gdb> set pcibootcmd oct-pci-boot
	show pci-bootcmd	Show the current value of pci-bootcmd.	gdb> show pci-bootcmd
	show core-state	Displays the state of the core by dumping all the general-purpose registers, COP0 registers, and 32 TLB entries.	gdb> show core-state

Note: If pci-bootcmd is not set, then the following error may occur:

```
(Core#0-gdb) target octeonpci bootoct 0x20000000 coremask=1
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x80000000]
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x80000000]
```

Sending bootloader command: bootoct 0 coremask=1 debug

```
ERROR: Bootloader not ready for command.
Sending bootcmd failed
```

6.5.1 Changes to Hands-On Steps When Using PCI Debugging

The instructions are changed when debugging over PCI.

1. There is no need to start the program from the bootloader prompt: the target command will load the program
2. Start the debugger using the sudo command
3. Specify the pci-bootcmd in the debugger
4. Specify target octeonpci in the debugger
5. To run the application again, simply type r

To use PCI debugging, the debugger must be started with the sudo command. Without it, the following error will occur:

```
host$ mipsisa64-octeon-elf-gdb -q hello
(Core#0-gdb) target octeonpci bootoct 0 coremask=1
/proc/bus/pci/03/0d.0: Permission denied
Unable to open PCI connection to Octeon
```

When using PCI debugging, running the program again is easy. The debugger will reset the development target, download the application, boot it, and the test is ready to begin again:

```
gdb> run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/testname/sdk/OCTEON-SDK/examples/hello/hello
Found Ochteon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x8000000]
Found Ochteon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x8000000]

Sending bootloader command: bootoct 0x20000000 coremask=1 debug
```

These changes are shown in the table below. Note that this table does not include the correct commands for debugging the Linux kernel. See Section 7.2 – “Debug the Linux Kernel” for command information.

Table 17: Running GDB with PCI Development Targets, Part 1

<i>New and changed commands are highlighted in pale blue.</i>	
Steps for Debugging	Note
Start GDB on the Host; Debug the Application	
host\$ cd \$OCTEON_ROOT/examples/hello	Go to the directory where the hello source is located on the host.
# type the following text on one line host\$ sudo mipsisa64-octeon-elf-gdb hello	Note that sudo is needed to start the debugger when using PCI debugging. Without sudo, the target octeonpci command will fail.
gdb> set pci-bootcmd oct-pci-reset	Use either oct-pci-boot or oct-pci-reset, depending on whether the board will boot from flash or over PCI. The oct-pci-reset command is used to boot from flash.
# type the following text on one line gdb> target octeonpci bootoct 0x20000000 coremask=0x3	Use the load address received from the bootloader namedprint command. Specify the desired coremask. In this case, the load address is 0x20000000. (Do not specify a load address of 0x0). Note that the debug option is not added to the command line: the debug option is automatically passed to the bootoct command. Expect to see: (Core#0-gdb) target octeonpci bootoct 0x20000000 coremask=0x3 Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000], BAR1=0xd0000000[0x8000000] Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000], BAR1=0xd0000000[0x8000000] Using bootloader image: /home/testname/sdk/target/bin/u-boot-octeon_ebt3000_pciboot.bin Initialized 2048 MBytes of DRAM Sending bootloader command: bootoct 0x20000000 coremask=0x3 debug 0x100005c0 in __octeon_trigger_debug_exception ()
<i>Continued in next table...</i>	

Table 18: Running GDB with PCI Development Targets, Part 2

New and changed commands are highlighted in pale blue.

Steps for Debugging	Note
<pre># simply type r for "run" gdb> r # enter y when prompted to start the program over</pre>	<p>Expect to see:</p> <p>The program being debugged has been started already.</p> <p>Start it from the beginning? (y or n) y</p> <p>Starting program: /home/testname/sdk/examples/hello/hello Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000], BAR1=0xd0000000[0x8000000] Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000], BAR1=0xd0000000[0x8000000] Using bootloader image: /home/testname/sdk/target/bin/u-boot-octeon_ebt3000_pciboot.bin Initialized 2048 MBytes of DRAM</p> <p>Sending bootloader command: bootoct 0x20000000 coremask=0x3 debug</p> <p>Program received signal SIGTRAP, Trace/breakpoint trap. 0x100005c0 in __octeon_trigger_debug_exception ()</p>

6.5.2 Multiple PCI Development Targets

When using PCI debugging, the utilities access the first OCTEON board found on the PCI bus.

To access a different OCTEON device, set the OCTEON_PCI_DEVICE environment variable to a different PCI device index. For example, if the variable is set to “1”, the second OCTEON on the PCI bus is accessed.

```
# set the value, exporting it to any sub-shells spawned by the
# current shell
host$ export OCTEON_PCI_DEVICE=1
# confirm the value is now correct
host$ env | grep OCTEON_PCI_DEVICE
OCTEON_PCI_DEVICE=1
```

6.5.3 Attaching to a Program Which is Already Running

When using mipsisa64-octeon-elf-gdb and debugging over PCI, it is possible to attach to a program which is already running.

After the program is running, start the debugger. Once the target octeonpci command is entered, the cores are stopped and the debugger is connected to the program. Note that the target octeonpci command must be used *without* other arguments (no “bootoct...”).

Using either the `detach` command or exiting GDB resumes execution of the program.

First, start the Linux kernel running from the bootloader prompt. Don't add the debug flag, on the boot command, just let Linux run. Then connect to it with the debugger:

```
host$ sudo mipsisa64-octeon-elf-gdb vmlinuz
GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=mipsisa64-octeon-
elf"...
(Core#0-gdb) set pci-bootcmd oct-pci-boot
(Core#0-gdb) target octeonpci
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x8000000]
0xffffffff80121bc4 in arch_ptrace (child=0xffffffff81b30000,
request=<value optimized out>, addr=<value optimized out>,
data=-2141257904) at arch/mips/kernel/ptrace.c:471
471                  wake_up_process(child);
(Core#0-gdb) list
466
467
468
TIF_SYSCALL_TRACE;
469
470
471
472
473
474
475
(Core#0-gdb) detach
(Core#0-gdb)
```

Remember to use sudo when starting gdb, or the following error will occur:

```
(Core#0-gdb) target octeonpci
/proc/bus/pci/03/0d.0: Permission denied
Unable to open PCI connection to Octeon
```

6.6 Other Special Commands: `spawn-sim`

The mipsisa64-octeon-elf-gdb debugger also supports a special command `spawn-sim`.

The `spawn-sim` command is used to simplify debugging on the OCTEON simulator by starting the simulator from the debugger. If `spawn-sim` is on *and* the target is specified as `target octeon tcp:` (as in `target octeon tcp::2021`), then the debugger starts the OCTEON

simulator, using X to create a new terminal session. The simulator will run in the new terminal session. (Remember to set `xhost +` on the development host to allow the new terminal session to open.)

Note: When the application stops running (for instance, at the end), the simulator's terminal window will close. The data shown on the screen in that terminal session will be lost. If this is not desired behavior, start the simulator separately from gdb.

The `tcp` target is used in two cases:

1. When using `gdb` to connect to the OCTEON simulator:

```
gdb> target octeon tcp::<tcp_port_number>
```

2. When using `gdb` to connect to a development target through a terminal server:

```
gdb> target octeon  
tcp:<IP_address_of_terminal_server>:<server_port>
```

The `spawn-sim` command should be `on` when:

- The user wants `gdb` to start the OCTEON simulator

The `spawn-sim` command should be `off` when:

- The simulator has already been started
- X is not available
- A terminal server is being used to connect to actual hardware

If the simulator has already been started using the same TCP port number or X is not available, a warning will occur, which can be ignored. The only time when it is essential to make sure `spawn-sim` is `off` is when using a terminal server.

In some SDK versions, `spawn-sim` is `on` by default (in SDK 1.9 it is `off`). Use the command `show spawn-sim` to determine the value:

```
gdb> show spawn-sim  
Whether the simulator would be spawned upon the target command is on.
```

To set `spawn-sim` to `off`:

```
gdb> set spawn-sim off  
gdb> show spawn-sim  
Whether the simulator would be spawned upon the target command is off.
```

6.7 Summary: Directions for Different Connection Types

The following tables summarize the commands to use for the different connection methods for `mipsisa64-octeon-elf-gdb`.

Note that these tables do not include the correct commands for debugging the Linux kernel. See Section 7.2 – “Debug the Linux Kernel” for command information.

Table 19: Debug SE-S or Linux Kernel over PCI Bus

PCI Bus
<p><i>This method is highly preferred if PCI target board is available. Only a bootloader console and PCI connection are needed. Program can be run over and over from the debugger: the debugger will reboot board, initialize, and run program again.</i></p> <p>A. On host:</p> <pre>host\$ sudo mipsisa64-octeon-elf-gdb <filename></pre> <p>Specify:</p> <pre>gdb> set pci-bootcmd <oct-pci-reset oct-pci_boot> # type the following text on one command line. (Note the debug option is not # needed on the bootoct command line.) gdb> target octeonpci bootoct <tmp_download_address> coremask=<coremask> gdb> b main gdb> c</pre> <p>Note that <code>tmp_download_address</code> cannot be 0. At the bootloader console, use <code>namedprint</code> to see the start address of the temporary download block. In the following example, <code>namedprint</code> returns an address of 0x20000000.</p> <pre>target# namedprint List of currently allocated named bootmem blocks: Name: __tmp_load, address: 0x0000000020000000, size: 0x0000000006000000, index: 0 Name: __tmp_reserved_linux, address: 0x0000000000100000, size: 0x0000000008000000, index: 1</pre> <p>When debugging over PCI, it is easy to run the program multiple times because the debugger reboots the board automatically in response to the <code>run</code> command.</p>

Table 20: Debug SE-S or Linux Kernel over Serial Connection

Serial Connection
<p>Used for stand-alone target boards. Requires a serial connection from host to Debugging Port (UART1) on the target board. Specify the host side of the debugging connection as <serial_port_on_host> (such as /dev/ttyS1) in the command line below.</p> <p>A. Boot the target board, then download the file to the board.</p> <p>B. On Target: Boot the file using the command:</p> <pre>target# bootoct 0 coremask=<coremask> debug</pre> <p>C. On host:</p> <pre>host\$ mipsisa64-octeon-elf-gdb <filename></pre> <p>Specify:</p> <pre>gdb> target octeon <serial_port_on_host> gdb> b main gdb> c</pre>

Table 21: Debug SE-S or Linux Kernel Using a Terminal Server

Serial Connection Through a Terminal Server
<p>Used for stand-alone target boards where access is through a terminal server. Requires a serial connection from terminal server to Debugging Port (UART1) on the target board.</p> <p>A. Boot the target board, then download the file to the board.</p> <p>B. On Target: Boot the file using the command:</p> <pre>target# bootoct 0 coremask=<coremask> debug</pre> <p>C. On host:</p> <pre>host\$ mipsisa64-octeon-elf-gdb <filename></pre> <p>Specify:</p> <pre>gdb> set spawn-sim off # type the following text on one command line gdb> target octeon tcp:<IP_address_of_terminal_server>:<server_port> gdb> b main gdb> c</pre> <p>When using some terminal servers, there is no way to fully disable telnet negotiations. In this case, the initial characters sent by GDB can be misinterpreted and taken for a negotiation response. If, after the target command, GDB prints "garbage" characters, and then nothing more, see the SDK document "Simple Executive Debugger" in the section "Debugging via a terminal server" for assistance.</p>

If a terminal server is used, see Section 6.6 – “Other Special Commands: spawn-sim”, and Section 14 – “Appendix B: Connecting Using a Terminal Server”.

Table 22: Debug SE-S or Linux Kernel on the OCTEON Simulator

Connecting When Using the OCTEON Simulator
<p>A. Start two terminal interfaces on the development host. One will be used to start the OCTEON simulator. Output from the program will be seen in this terminal session. The other terminal session is used to start the debugger.</p> <p>B. In development host terminal session 1:</p> <pre># type the following text on one command line hostT1\$ oct-sim <filename> -quiet -noperf -numcores=<numcores> -uart1=2021 -debug</pre> <p>C. In development host terminal session 2:</p> <pre>hostT2\$ mipsisa64-octeon-elf-gdb <filename> Specify: gdb> set spawn-sim off # type the following text on one command line gdb> target octeon tcp::2021 -noperf -quiet -numcores=<numcores> gdb> load gdb> b main gdb> c</pre>

For more information on debugging on the OCTEON simulator, see Section 12 – “About Debugging on the OCTEON Simulator”.

6.8 Software Breakpoints and Multicore Debugging

When using mipsisa64-octeon-elf-gdb, when a software breakpoint is set, the breakpoint applies to all the cores using the same load set.

6.8.1 Race Condition: Cores Can Bypass the Breakpoint Without Stopping

This condition is rare. Software breakpoints are implemented by replacing the instruction at the breakpoint with the `sdbbp` instruction. All cores in the load set share the same code in memory. Once execution reaches this point, GDB will replace the `sdbbp` instruction with the original program instruction. The breakpoint will not be reinserted until the core that hit the `sdbbp` steps past the breakpoint. When the active-cores execute a single step, the non-focus core may single step before the focus core. If this happens, the `sdbbp` instruction will not be there, allowing the non-focus core to move past the expected breakpoint. This problem is rare. If avoiding this possibility is essential, the focus core must be stepped at least once with `step-all off` before any other cores are allowed to execute.

Figure 11: Race Condition: Cores Can Bypass Breakpoint

A	B	C
Code after breakpoint is set, but before breakpoint is hit.	Code when breakpoint is hit, before step instruction. (This is when the race condition could occur. If the non-focus core executes now, the breakpoint is missing.)	Code when focus core steps past the breakpoint: <code>sdbbp</code> instruction restored.
instruction 5	instruction 5	instruction 5
<code>sdbbp</code>	instruction 6	<code>sdbbp</code>
instruction 7	instruction 7	instruction 7
instruction 8	instruction 8	instruction 8

When a group of cores single steps together, the non-focus core might execute before the focus core, using the instructions as they appear in column B. By setting step-all off, the non-focus core cannot execute when the sdbbp instruction is missing.

6.8.2 Race Condition: Multiple Cores Stopped on the Same Breakpoint

This condition is rare. When debugging multiple cores, it is possible to have two or more cores hit a breakpoint on the same instruction. If `step-all` doesn't move one of the cores forward, verify that it isn't stopped on the same breakpoint as the focus core. If a non-focus core has stopped on a breakpoint, change the focus to the stopped core, and step past the breakpoint.

6.9 Hardware Breakpoints

Currently (SDK 1.8) hardware breakpoints are only supported in the `mipsisa64-octeon-elf-gdb` debugger. Unlike software breakpoints, which change the instructions which will be executed, hardware breakpoints do not modify the instructions. Because of this, hardware breakpoints are especially useful when debugging non-writable (or difficult to write) memory such as flash or EEPROM.

Hardware breakpoints may be set by using the GDB `hbreak` command. There are a maximum of four hardware breakpoints per application. The `hbreak` command is applied to the *focus core*, unlike software breakpoints which apply to all cores in the same load set.

Setting a hardware breakpoint is similar to the command to set a software breakpoint:

```
(Core#0-gdb) list
39
40      /* printf() provides maximum flexibility, but is slow due to
41      ** the format string being processed in simulated code.
Normal
42      ** buffering is done by the C library.
43      */
44      printf("\n");
45      printf("\n");
46      printf("Hello world!\n");
47
```

```

48      #if 0
(Core#0-gdb) hbreak 46
Hardware assisted breakpoint 2 at 0x10000514: file hello.c, line 46.
(Core#0-gdb) c
Continuing.

Breakpoint 2, main () at hello.c:46
46          printf("Hello world!\n");
(Core#0-gdb)

```

6.10 Hardware Watchpoints

Currently (SDK 1.8) hardware watchpoints are only supported in the mipsisa64-octeon-elf-gdb debugger. Both read and write watchpoints are supported. Hardware watchpoints are inserted into the program using the GDB `watch`, `rwatch`, and `awatch` commands. Watchpoints are applied to the focus core, unlike software breakpoints which apply to all cores in the same load set. There are a maximum of four hardware watchpoints per application. Hardware watchpoints do not significantly slow down program execution.

Watchpoints are useful for notification if the memory location is read (`rwatch`), or changed (`watch`). The `awatch` command is used if either the `rwatch` or `watch` conditions are met. Read watchpoints may not be implemented for all products or all releases. Check product-specific information such as the *HRM* for details.

6.11 Performance Counters

The OCTEON processor supports two performance counter registers. Commands to enable and read performance counters are applied to the focus core.

Each performance counter can be enabled by specifying an event. The GDB command `set perf-event [0|1]` will display the list of events supported for the OCTEON processor, for instance `set perf-event0`.

To initialize an event, type `set perf-event [0|1] <event>` at the GDB prompt.

To check the value of the performance counter register, type `show perf-event [0|1]` at the GDB prompt.

Use `set perf-event [0|1] none` to turn off the performance counter, which will also reset its value to zero.

For example:

```

(Core#0-gdb) set perf-event0 sissue
(Core#0-gdb) show perf-event0
Performance counter0 for "sissue" event is 0
(Core#0-gdb) list
41          ** the format string being processed in simulated code.
Normal
42          ** buffering is done by the C library.
43          */

```

```

44         printf("\n");
45         printf("\n");
46         printf("Hello world!\n");
47
48     #if 0
49         /* simprintf() passes the format string and up to 7 arguments
to the
50         ** simulator and is much faster than standard printf(). It is
limited
(Core#0-gdb) s
puts (s=0x1000ac50 "Hello world!")
        at
/usr/local/Cavium_Networks/toolchain/src/newlib/libc/stdio/puts.c:103
103
/usr/local/Cavium_Networks/toolchain/src/newlib/libc/stdio/puts.c: No
such file or directory.
        in
/usr/local/Cavium_Networks/toolchain/src/newlib/libc/stdio/puts.c
(Core#0-gdb) show perf-event0
Performance counter0 for "sisissue" event is 7
(Core#0-gdb) set perf-event0 none
(Core#0-gdb) show perf-event0
Performance counter0 for "none" event is 0
(Core#0-gdb) show perf-event1
Performance counter1 event is not set.

```

6.12 Finding the Cause of an Exception

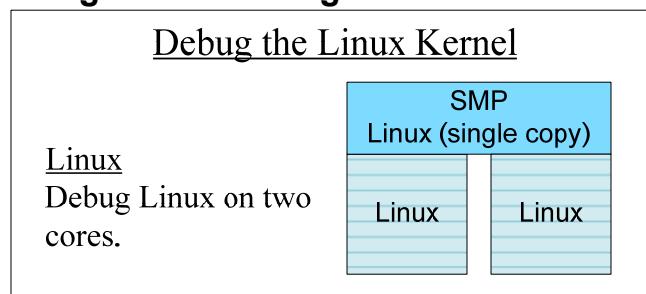
Because SE-S applications cannot provide a core dump, but it is possible to find the cause of an exception using the debugger.

Set a breakpoint in the default exception handler

(`_cvmx_interrupt_default_exception_handler()`), or in a custom exception handler. Then run the program in the debugger. When the exception occurs, the debugger `backtrace` command can be used to locate the cause of the exception. A detailed example is provided in the SDK document “*Simple Executive Debugger*”.

7 Hands-On: Debug the Linux Kernel

The Linux kernel may be debugged using the `mipsisa64-octeon-elf-gdb` debugger. This debugger provides features not available with standard `kgdb`. In particular, multicore debugging and debugging over PCI are available. See Section 6 – “About Debugging SE-S Applications or the Linux Kernel” for more information.

Figure 12: Debug the Linux Kernel

7.1 Building the Linux Kernel for Debugging

The following steps are needed to build kernel with debugging support, without the watchdog, and with frame pointers enabled. These directions are for kernel 2.6.21, SDK 1.8.

Precise information can be found in the SDK document “*Linux on the OCTEON*”. Note that the “Optimize for size” menuconfig option mentioned in the SDK documentation appears to be unnecessary.

7.1.1 Kernel Configuration

Set the kernel configuration needed for kernel debugging by running `make menuconfig` in the `$OCTEON_ROOT/linux/kernel_2.6/linux` directory.

When running `make menuconfig`, the configuration options are accessed via the “Machine selection”, and “Kernel hacking” sub-menus.

The first kernel menuconfig screen looks similar to this:

```

Machine selection --->
  Endianess selection (Big endian) --->
    CPU selection --->
      Kernel type --->
        Code maturity level options --->
          General setup --->
            Loadable module support --->
              Block layer --->
                Bus options (PCI, PCMCIA, EISA, ISA, TC) --->
                  Executable file formats --->
                    Networking --->
                      Device Drivers --->
                        File systems --->
                          Profiling support --->
Kernel hacking --->
  Security options --->
  Cryptographic options --->
  Library routines --->
  ---
  Load an Alternate Configuration File
  Save an Alternate Configuration File

```

To navigate this screen, use the arrow keys on the keyboard. The bottom of the screen provides some options that can be selected with the TAB key. In the first screen, these options are “Select, Exit, and Help”. To select a highlighted option, press **Enter** when the option “Select” (at the bottom of the screen) is highlighted.

7.1.1.1 Turn off Watchdog

The watchdog will reset the system if the kernel does not respond to it about every 5 seconds. If the debugger has stopped the kernel, then it cannot respond to the watchdog exception. Before beginning debugging, remove the watchdog.

To configure Linux for debugging, select “Machine selection”. The Machine selection screen will look similar to this:

```
Select the machine
System type (Support for the Cavium Networks Octeon reference
    [*] Enable Octeon specific options
    [*] Enable RI/XI extended page table bits
    [ ] Build the kernel to be used as a 2nd kernel on the same chip
    [*] Enable support for Compact flash hooked to the Octeon Boot Bus
    [*] Enable hardware fixups of unaligned loads and stores
    [*] Enable fast access to the thread pointer
    [*] Support dynamically replacing emulated thread pointer accesses
<text omitted>
<M> POW based internal only ethernet driver
<*> Management port ethernet driver (CN5XXX)
< > Octeon watchdog driver <<< DESIRED SETTING
< > Octeon trace buffer (TRA) driver
[ ] Enable enhancements to the IPsec stack to allow protocol offload.
[ ] Enable Cavium Octeon ip-offload module
```

Highlight “Octeon watchdog driver” by selecting the item. If the option is set (*), then press **Enter** to clear the option. Then select **Exit** to go up one menu.

7.1.1.2 Enable Remote Debugging

Select the next sub-menu, “Kernel hacking”. The Kernel hacking screen will look similar to this:

```
[ ] Show timing information on printks
    [*] Enable __must_check logic
    [ ] Magic SysRq key
    [ ] Enable unused/obsolete exported symbols
    [ ] Debug Filesystem
    [ ] Run 'make headers_check' when building vmlinux
    [*] Kernel debugging
    [ ] Debug shared IRQ handlers
<text omitted>
    [ ] Enable stack utilization instrumentation
    [ ] Remote GDB kernel debugging
    [*] Remote GDB debugging using the Cavium Networks Multicore GDB <<<
DESIRED SETTING
    [ ] Enable run-time debugging
```

Highlight “Remote GDB debugging using the Cavium Networks Multicore GDB” by selecting the item. If the option is clear (), then press **Enter** to set the option. Then select **Exit** to go up one menu.

This option will define `CONFIG_CAVIUM_GDB=y` in the kernel’s `.config` file.

7.1.1.3 Save and Exit

Select **Exit** to exit `menuconfig`. When prompted “Do you wish to save your new kernel configuration?” select **Yes**.

The output of the `make menuconfig` command is the file `.config`:

```
host$ ls -ld .config
-rw-r--r-- 1 root root 20640 Jan 23 11:58 .config
```

(Note: Saving a copy of the `.config` file is useful. The `$OCTEON_ROOT/linux/Makefile` clean target will remove the `$OCTEON_ROOT/linux/kernel_2.6/linux/.config` file.)

7.1.2 Rebuild Linux, Enable Frame Pointers

Then build the kernel with frame pointers enabled, packaging it along with the embedded root filesystem into `vmlinux`:

```
host$ cd ../..
host$ sudo ls # enter password now so it is stored for the next command
# this build will take about 10 minutes
host$ sudo make -s CONFIG_FRAME_POINTER=y kernel >make.out 2>&1 &
```

Note: If the `.config` file has changed, the build will rebuild the kernel appropriately. There is no need to do a `make clean` prior to the new build.

Note: If there is any need to do a `clean` on the kernel, be sure to change to `$OCTEON_ROOT/linux/kernel_2.6` before executing `make clean`. If `make clean` is executed in `$OCTEON_ROOT/linux`, then the kernel’s `.config` file (`$OCTEON_ROOT/linux/kernel_2.6/linux/.config`) will be removed, and the `make menuconfig` step will have to be repeated.

Note: If the `sudo` command is omitted when using the `make -s` command line, then the following cryptic error message will occur:

```
*** Error during writing of the kernel configuration.
```

```
make[4]: *** [silentoldconfig] Error 1
make[3]: *** [silentoldconfig] Error 2
make[2]: *** [include/config/auto.conf] Error 2
make[1]: *** [linux] Error 2
make: *** [kernel] Error 2
```

7.1.3 About the `make clean` Command

If the `make clean` command is executed in the `$OCTEON_ROOT/linux` directory, then the kernel's `$OCTEON_ROOT/linux/kernel_2.6/linux/.config` file will be deleted. On the next build, if the `$OCTEON_ROOT/linux/kernel_2.6/linux/.config` file is missing, then the pristine (original `kernel.config` supplied with the release) is copied to `.config`. This is done to allow the user to restore the original working kernel configuration.

This can be inconvenient if the user would like to save the `.config` file. The `.config` file can be saved to another name, such as `.config.save`. After the clean step, this file can be copied to `.config`.

To clean the kernel directory without removing the `.config` file, `cd` to `$OCTEON_ROOT/linux/kernel_2.6` before executing `make clean`, or use the following command in the `$OCTEON_ROOT/linux` directory:

```
host$ make -s -C kernel_2.6 clean
```

7.2 Debug the Linux Kernel

The debugger used for the Linux kernel (`mipsisa64-octeon-elf-gdb`) is the same as for SE-S applications, such as `hello`. This debugger supports multicore debugging and debugging using PCI as a connection.

Significant differences from SE-S application debugging directions:

1. The kernel is booted with the command:

```
bootoctlinux 0 coremask=<coremask> debug
```
2. The kernel will come up until just after “SMP Linux” is displayed in the LEDs on the development target
3. The kernel stops at `prom_init()`.
4. A reasonable first breakpoint is `r4k_wait()`

See Section 6 – “About Debugging SE-S Applications or the Linux Kernel” for more information.

Table 23: Debug the Linux Kernel – Part 1

Steps	Note
1. Connect the Hardware	
Connect serial cable to target console. Connect the Ethernet cable.	Follow directions in the <i>Quick Start Guide</i> . Note: The Ethernet cable is not needed to run <code>hello</code> on a PCI target board, but will be needed later in the <i>SDK Tutorial</i> . Be careful to isolate the test network from the office network so that experiments will not disturb the office network.
Connect the target debug port to the host using a serial cable.	GDB will communicate with the target using this connection.
2. Reset the board	
Power on or reset the target board.	The word <code>Boot</code> should appear on the red LEDs on the board. If not, the board is not configured to boot from flash, or something is wrong with the board, or the board does not have alpha LEDs.
3. Connect to the Target Console	
host\$ <code>minicom -w ttys0</code>	Substitute the serial port actually used on the host to connect to the OCTEON target board if it is not <code>ttys0</code> . Minicom will provide a connection to the target console. You should see the bootloader prompt.
4. Verify Bootloader Prompt is Visible	
target# <code>version</code>	The bootloader should reply with text similar to: U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun 13)
5. Verify Bootloader Version is at Least SDK 1.7	
If the bootloader's SDK version is not at least 1.7, then before continuing, upgrade the bootloader to a newer version.	Directions for upgrading the bootloader are included in the <i>SDK Tutorial</i> .
6. Configure the Kernel for Debugging	If this step was already done, no need to repeat it!
# type the following text on one line host\$ <code>cd \$OCTEON_ROOT/linux/kernel_2.6/linux</code> # see note for menuconfig options host\$ <code>make menuconfig</code>	Select the menuconfig options: <ul style="list-style-type: none"> * Machine selection: Octeon watchdog driver - off * General setup: optimize for size - on * Kernel hacking: Remote GDB debugging using the Cavium Networks Multicore - on
<i>Continued in the next table...</i>	

Table 24: Debug the Linux Kernel – Part 2

Steps	Note
7. Build the Kernel with Embedded Root Filesystem	If this step was already done, no need to repeat it!
<pre>host\$ cd ../../ # the directory should now be \$OCTEON_ROOT/linux # enter password now so it is stored for the next command host\$ sudo ls # type the following text on one line host\$ sudo make -s CONFIG_FRAME_POINTER=y kernel >make.out 2>&1 &</pre> <p># Note: This step can take up to 20 minutes. When it has completed successfully, the kernel_2.6/linux/vmlinux file will have been created.</p>	Note: The sudo ls command is merely used to set the root password. The root password will be stored for about 5 minutes for use in the next command (sudo make...).
8. Copy the ELF file to the tftpboot Directory	
<pre># verify the build is complete before continuing host\$ wait # go to the created vmlinux file host\$ cd kernel_2.6/linux host\$ sudo cp vmlinux /tftpboot</pre>	See <i>SDK Tutorial</i> directions for tftpboot.
9. Set the Development Target's IP Address	
If a DHCP server is available, then selecting the target IP address is handled by the server. Otherwise, select a target IP address.	In this example, the target IP address is 192.168.51.159.
9a. No DHCP Server	
First, set the IP address of the target (replace items in italic with your IP addresses). <i># Use your IP addresses instead of the example values!</i> <pre>target# setenv gatewayip 192.168.51.254 target# setenv netmask 255.255.255.0 target# setenv ipaddr 192.168.51.159 target# setenv serverip 192.168.51.1 # save the values so they will still be set after a reset target# saveenv</pre>	Note: serverip is the IP address of the TFTP server.
9b. DHCP Server Available	
If a DHCP server is available substitute the following step: <pre>target# dhcp</pre>	Note: serverip is the IP address of the TFTP server. In this example the DHCP server is the same as the TFTP server.
<i>Continued in the next table...</i>	

Table 25: Debug the Linux Kernel – Part 3

Steps	Note
10. Test the Ethernet Connection to the Host	<pre># use your host IP address instead of the example value! target# ping 192.168.51.254</pre> <p>Expect to see: Using octeth0 device host 192.168.51.254 is alive Note that the development target will <i>not</i> reply to a ping from the development host. Note: to see the development host's IP address, use the /sbin/ifconfig command on the development host.</p>
11. Download the Application to the Development Target	<pre># Use tftpboot to download the application. target# tftpboot 0 vmlinuz</pre> <p>See the <i>SDK Tutorial</i> directions for tftpboot. If this step does not work, check the /etc/xinetd.d/tftp file on the host to verify that server_args = -s /tftpboot.</p>
12. Boot the Application	<pre># type the following text on one line target# bootoctlinux 0 coremask=0x3 debug</pre> <p>This command will run Linux on cores 0 and 1. Expect to see: argv[2]: coremask=0x3 argv[3]: debug ELF file is 64 bit Attempting to allocate memory for ELF segment: addr: 0xfffffffff80100000 (adjusted to: 0x00000000000100000), size 0x1a99d58 <text omitted> Bootloader: Done loading app on coremask: 0x3 <Linux will stop here> Expect to see "SMPLinux" on the board's LEDs at this point.</p>

Continued in the next table...

Table 26: Debug the Linux Kernel – Part 4

Steps	Note
13. Start GDB on the Host; Debug the Application	
# type the following text on one line host\$ cd \$OCTEON_ROOT/linux/kernel_2.6/linux	Go to the directory where the vmlinu source is located on the host.
# start gdb host\$ mipsisa64-octeon-elf-gdb vmlinu	Use -q (quiet) for fewer start-up messages. Expect to see: GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64 Copyright (C) 2006 Free Software Foundation, Inc. GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions. There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "--host=i686-pc-linux-gnu --target=mipsisa64-octeon-elf"...
# connect to the debug port gdb> target octeon /dev/ttys1	Expect to see: Remote target octeon connected to /dev/ttys1
# set a breakpoint in r4k_wait() gdb> b r4k_wait	Expect to see: Breakpoint 1 at 0xffffffff801214b8: file arch/mips/kernel/cpu-probe.c, line 57.
# continue to the breakpoint gdb> c	Expect to see: Continuing. Breakpoint 1, r4k_wait () at arch/mips/kernel/cpu-probe.c:57 57 wmb();
<i>Continued in the next table...</i>	

Table 27: Debug the Linux Kernel – Part 5

Steps	Note
13. Debug the Application, continued	
# show 10 source lines at the breakpoint gdb> list	Expect to see: 52 * a non-enabled interrupt is requested. 53 */ 54 static void r4k_wait(void) 55 { 56 #ifdef CONFIG_CPU_CAVIUM_OCTEON 57 wmb(); 58#endif 59 __asm__(" .set mips3 \n" 60 " 61 wait \n" 62 ".set mips0 \n");
# r4k_wait() is called by the idle loop, so clear the # breakpoint in order to continue to the prompt gdb> clear	Expect to see: Deleted breakpoint 1
# continue to run the program gdb> c	Expect to see: Continuing.
The Linux prompt will now come up on the target console.	
14. Run Linux Again	
To reset the development target, reboot Linux, or: * for stand-alone boards: power on or reset the board. * for PCI boards: host\$ oct-pci-reset	To run Linux again, the board must be rebooted and vmlinux downloaded again.

7.3 Example: Multicore Debugging and the Linux Kernel

The following example shows using the multicore debugging commands on the Linux kernel.

Boot the kernel as shown in the above tables, then follow try out these **gdb** commands:

```
host$ cd $OCTEON_ROOT/linux/kernel_2.6/linux
host$ mipsisa64-octeon-elf-gdb vmlinux
GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
```

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "--host=i686-pc-linux-gnu --target=mipsisa64-octeon-elf"...

```
(Core#0-gdb) target octeon /dev/ttys1
Remote target octeon connected to /dev/ttys1
(Core#0-gdb) b r4k_wait
Breakpoint 1 at 0xffffffff801214b8: file arch/mips/kernel/cpu-probe.c,
line 57.
(Core#0-gdb) set step-all on
(Core#0-gdb) c
Continuing.
Core 1 taking focus. << Note the automatic focus change

Breakpoint 1, r4k_wait () at arch/mips/kernel/cpu-probe.c:57
57          wmb();
```

(Core#1-gdb) list

```
52      * a non-enabled interrupt is requested.
53      */
54      static void r4k_wait(void)
55      {
56      #ifdef CONFIG_CPU_CAVIUM_OCTEON
57          wmb();
58      #endif
59          __asm__ ("          .set      mips3      \n"
60                  "          wait      \n"
61                  "          .set      mips0      \n");
(Core#1-gdb) c
Continuing.

Breakpoint 1, r4k_wait () at arch/mips/kernel/cpu-probe.c:57
57          wmb();
```

(Core#1-gdb) clear

Deleted breakpoint 1

(Core#1-gdb) c

<The kernel now comes up to the interactive prompt in the target console.>

8 About Debugging the Linux Kernel

When debugging the Linux kernel, there are three choices:

1. Cavium Networks proprietary multicore debugger.
2. The standard open source kernel debugger
3. EJTAG

8.1 Cavium Networks Proprietary GDB Protocol

The kernel may be debugged using the Cavium Networks proprietary GDB protocol provided by mipsisa64-octeon-elf-gdb. This debugger supports multicore debugging of all kernel code, including interrupts and exceptions. This debugger does not handle TLB misses and other corner cases as well as the standard open source debugger.

When the kernel is booted with the debug option, it will enter the debug exception handler in `prom_init()` (`setup.c`):

```
#ifdef CONFIG_CAVIUM_GDB
    /* When debugging the linux kernel, force the cores to enter the debug
       exception handler to break in. */
    if (octeon_get_boot_debug_flag()) {
        cvmx_write_csr(CVMX_CIU_DINT, 1 << cvmx_get_core_num());
        cvmx_read_csr(CVMX_CIU_DINT);
    }
#endif
```

From here, the first logical breakpoint is `r4k_wait()`.

8.2 The Standard Open Source Kernel Debugger

The standard open source debugger can debug the kernel assuming the `gdb` stub is built into the kernel. The `gdb` stub is called `kgdb` in the kernel. This debugger works for debugging most kernel code but not exception and interrupt handlers. After `kgdb` is build into the kernel, use the `mips64-octeon-linux-gnu-gdb` to debug the kernel. (Note that other versions of `gdb` will also work.)

Directions are in the SDK Document “*Linux on the OCTEON*”, in the section “*Debugging the Kernel with KGDB*”.

8.3 SMP Synchronization and step-all

The kernel uses `smp_call_function()` to synchronize operations across all cores. This function will hang forever if a core is currently stopped in the debug exception. Make sure `step-all` is on whenever SMP operations are expected.

8.4 The Kernel File Name: `vmlinux` VS `vmlinux.64`

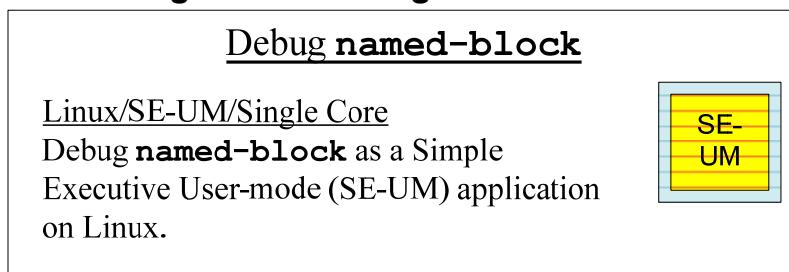
In the SDK documentation, the name `vmlinux.64` is used instead of `vmlinux`. Both files have the same content, so when the directions specify `vmlinux.64`, it is okay to use `vmlinux` instead.

The presence of two files is for historical reasons. At one point in the past there was a difference between `vmlinux` and `vmlinux.64`. The kernel build process made a 32-bit binary and then changed it into a 64-bit binary. This is no longer the case, but the `vmlinux.64` name was kept for backwards compatibility.

9 Hands-On: Debug a SE-UM Application: `named-block`

The SE-UM example `named-block` is debugged in this hands-on step. A serial connection is used for cross-debugging because it works on both PCI and stand-alone development targets. This step should not be done until the prior steps in this chapter have been successfully completed.

Figure 13: Debug named-block



Detailed information will be provided in Section 10 – “About Linux User-Mode Application Debugging”.

In this example, the development host serial port `/dev/ttys1` is connected to the debug port on the development target (UART1). If a different serial port is used, modify the directions to use the correct value.

Debugging over serial connection will require two different sessions on the host: one to connect to the target console port, and one to connect to the target debugging port.

The development target configuration is illustrated in Figure 4 – “Hardware Configurations for SE-S and Linux Kernel Debugging”. The only addition to the prior configuration is the serial cable to the target’s debug port.

The directions in the following table are the hands-on instructions for debugging named-block. After this section, more detailed information will be provided.

After compiling the program with the debug flag, and optimization level set appropriately, follow the instructions below. Select the debug method in the table below, and follow the instructions. For more information, see the SDK document for the type of debugging chosen.

The first breakpoint is usually set at `appmain()`. Note that there is no symbol named `main()`.

In the default configuration, example applications are stripped of debugging information when they are installed in the embedded root filesystem. For information on how to disable the `strip` step, see Section 4 – “Debugging Applications in the Embedded Root Filesystem” for details.

Note: When invoking `mips64-octeon-linux-gnu-gdb` to debug example code, be sure to add the `-linux_64` suffix. Without the executable name, `gdb` assumes the ABI is O32 ABI (32-bit registers). `GDB` will then issue an error message (bad register size):

```
.../.../src/gdb/mips-tdep.c:606: internal-error: bad register size
A problem internal to GDB has been detected,
further debugging may prove unreliable.
```

Table 28: Debug named-block (SE-UM) over TCP Sockets, Part 1

Steps	Note
1. Connect the Hardware	
Connect serial cable to target console. Connect the Ethernet cable.	Follow directions in the <i>Quick Start Guide</i> . Note: The Ethernet cable is not needed to run Linux on a PCI target board, but will be needed later in the <i>SDK Tutorial</i> . Be careful to isolate the test network from the office network so that experiments will not disturb the office network.
2. Reset the board	
For <i>stand-alone</i> boards: power on or reset the board. For <i>PCI</i> boards: host\$ oct-pci-reset	The word Boot should appear on the red LEDs on the board. If not, the board is not configured to boot from flash, or something is wrong with the board, or the board does not have alpha LEDs.
3. Connect to the Target Console	
host\$ minicom -w ttys0	Substitute the serial port actually used on the host to connect to the OCTEON target board if it is not <i>ttys0</i> . Minicom will provide a connection to the target console. You should see the bootloader prompt.
4. Verify Bootloader Prompt is Visible	
target# version	The bootloader should reply with text similar to: U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time: Jun 13)
5. Verify Bootloader Version is at Least SDK 1.7	
If the bootloader's SDK version is not at least 1.7, then before continuing, upgrade the bootloader to a newer version.	Directions for upgrading the bootloader are included in the <i>SDK Tutorial</i> .
6. Build the Application for Debugging	If this step was already done, no need to repeat it!
<pre data-bbox="176 1233 866 1453"> host\$ cd \$OCTEON_ROOT/examples/named-block # edit the Makefiles as needed to # change the optimization level to -O0 host\$ make clean OCTEON_TARGET=linux_64 host\$ make OCTEON_TARGET=linux_64 </pre>	
7. Modify Embedded Root Filesystem to Not Call Strip	If this step was already done, no need to repeat it!
<pre data-bbox="176 1493 866 1691"> # type the following text on one line host\$ cd \$OCTEON_ROOT/linux/embedded_rootfs # change the configuration to not strip the files host\$ sudo make menuconfig # deselect Strip debugging information from binaries </pre>	Directions are for SDK 1.8.
<i>Continued in the next table...</i>	

Table 29: Debug named-block (SE-UM) over TCP Sockets, Part 2

Steps	Note
8. Build Linux	If this step was already done, no need to repeat it!
<pre>host\$ cd \$OCTEON_ROOT/linux</pre> <p># enter password now so it is stored for the next command</p> <pre>host\$ sudo ls</pre>	Note: The <code>sudo ls</code> command is merely used to set the <code>root</code> password. The <code>root</code> password will be stored for about 5 minutes for use in the next command (<code>sudo make...</code>).
<pre>host\$ sudo make kernel >make.out 2>&1 &</pre>	Build the kernel and embedded root filesystem in the background. Use <code>tail -f</code> to see the output. The <code>make</code> command will create the executable file <code>vmlinux</code> .
9. Copy the ELF file to the <code>tftpboot</code> Directory	
<pre># verify the build is complete before continuing</pre> <pre>host\$ wait</pre> <p># go to the created vmlinux file</p> <pre>host\$ cd kernel_2.6/linux</pre> <pre>host\$ sudo cp vmlinux /tftpboot</pre>	The <code>vmlinux</code> file is located in the <code>kernel_2.6/linux</code> directory. See <i>SDK Tutorial</i> directions for <code>tftpboot</code> .
10. Set the Development Target's IP Address	
If a DHCP server is available, then selecting the target IP address is handled by the server. Otherwise, select a target IP address.	In this example, the target IP address is <code>192.168.51.159</code> .
10a. No DHCP Server	
First, set the IP address of the target (replace items in italic with your IP addresses). <i># Use your IP addresses instead of the example values!</i> <pre>target# setenv gatewayip 192.168.51.254</pre> <pre>target# setenv netmask 255.255.255.0</pre> <pre>target# setenv ipaddr 192.168.51.159</pre> <pre>target# setenv serverip 192.168.51.1</pre> <i># save the values so they will still be set after a reset</i> <pre>target# saveenv</pre>	Note: <code>serverip</code> is the IP address of the TFTP server.
10b. DHCP Server Available	
If a DHCP server is available substitute the following step: <pre>target# dhcp</pre>	Note: <code>serverip</code> is the IP address of the TFTP server. In this example the DHCP server is the same as the TFTP server.
<i>Continued in the next table...</i>	

Table 30: Debug named-block (SE-UM) over TCP Sockets, Part 3

Steps	Note
11. Test the Ethernet Connection to the Host	
# use your host IP address instead of the example value! target# ping 192.168.51.254	Expect to see: Using octeth0 device host 192.168.51.254 is alive Note that the development target will <i>not</i> reply to a ping from the development host. Note: to see the development host's IP address, use the /sbin/ifconfig command on the development host.
12. Download Linux to the Development Target	
Use tftpboot to download the application. target# tftpboot 0 vmlinuz	See the <i>SDK Tutorial</i> directions for tftpboot. If this step does not work, check the /etc/xinetd.d/tftp file on the host to verify that server_args = -s /tftpboot.
13. Boot Linux	
target# bootoctlinux 0 coremask=0x1	This command will run vmlinuz on core 0. Note: Linux will now run, and output a prompt on the target console. Type ls to verify Linux is up and running.
14. Start Ethernet on the Development Target	
target# modprobe cavium-ethernet	
15. Set the Development Target's IP Address	
If a DHCP server is available, then selecting the target IP address is handled by the server. Otherwise, select a target IP address.	
15a. No DHCP Server	
First, set the IP address of the target (replace items in italic with your IP addresses). # Use your IP addresses instead of the example values! # type the following command on one line target# ifconfig eth0 192.168.51.159 netmask 255.255.255.0	In this example, the target IP address is 192.168.51.159.
15b. DHCP Server Available	
If a DHCP server is available substitute the following step: target# udhcpc -i eth0	Note: serverip is the IP address of the TFTP server. In this example the DHCP server is the same as the TFTP server.
<i>Continued in the next table...</i>	

Table 31: Debug named-block (SE-UM) over TCP Sockets, Part 4

Steps	Note
16. Start gdbserver on the Target	
target# cd examples	Go to the directory where named-block is located.
target# gdbserver :2349 named-block	Expect to see: <pre>/examples # gdbserver :2349 named-block GNU gdbserver 6.5 Cavium Networks Version: 1_6_0, build 34 Process named-block created; pid = 737 Listening on port 2349</pre>
17. Start GDB on the Host; Debug the Application	
# type the following command on one line host\$ cd \$OCTEON_ROOT/examples/named-block	Go to the directory where the named-block source is located on the host.
# type the following command on one line host\$ mips64-octeon-linux-gnu-gdb named-block-linux_64	Use -q for quiet
# type the following command on one line # (substitute the your target IP address for the example # value) gdb> target remote 192.168.51.159:2349	Expect to see: <pre>Remote debugging using 192.168.51.159:2349 0x0000000120003910 in _ftext ()</pre>
# set a breakpoint in appmain () gdb> b appmain	Expect to see: <pre>Breakpoint 2 at 0x120003acc: file named-block.c, line 39.</pre>
# continue to the breakpoint gdb> c	Expect to see: <pre>Continuing. Breakpoint 1, appmain () at named-block.c:39 39 cvmx_user_app_init();</pre>
<i>Continued in the next table...</i>	

Table 32: Debug named-block (SE-UM) over TCP Sockets, Part 5

Steps	Note
17. Debug the Application, continued	
# show 10 source lines at the breakpoint gdb> list	Expect to see: <pre> 34 { 35 36 cvmx_bootmem_named_block_desc_t 37 *block_desc; 38 39 int status; 40 void *block_ptr; 41 42 cvmx_user_app_init(); 43 if 44 (!cmvx_coremask_first_core(cvmx_sysi 45 nfo_get()->core_mask)) 46 return 0; 47 48 printf("INFO: Size of 49 pointer is %llu bytes\n", 50 (ULL)sizeof(void*)); 51 (gdb) </pre>
# let the program run to completion gdb> c	Expect to see: Continuing. Program exited normally.
	When the steps above are followed correctly, the target console should show: <pre> /examples # gdbserver :2349 named- block GNU gdbserver 6.5 Cavium Networks Version: 1_8_0, build 64 Process named-block created; pid = 775 Listening on port 2349 Remote debugging from host 192.168.51.254 CVMX_SHARED: 0x1201a0000-0x1201b0000 Active coremask = 0x1 INFO: Size of pointer is 8 bytes PASS: All tests passed Child exited with retcode = 0 Child exited with status 0 GDBserver exiting </pre>

10 About Linux User-Mode Application Debugging

In the prior section, named-block was used as an example of debugging a Linux user-mode application (in this case, a SE-UM application). This section provides more details about this debugging environment.

10.1 Quick Summary of *mips64-octeon-linux-gnu-gdb*

The *mips64-octeon-linux-gnu-gdb* debugger is used to debug:

- Linux user-mode applications, including SE-UM applications

Type of debugging available:

- Cross-Debugging
- Native debugging

Debugger cross-connection types available:

- Serial connection
- TCP sockets

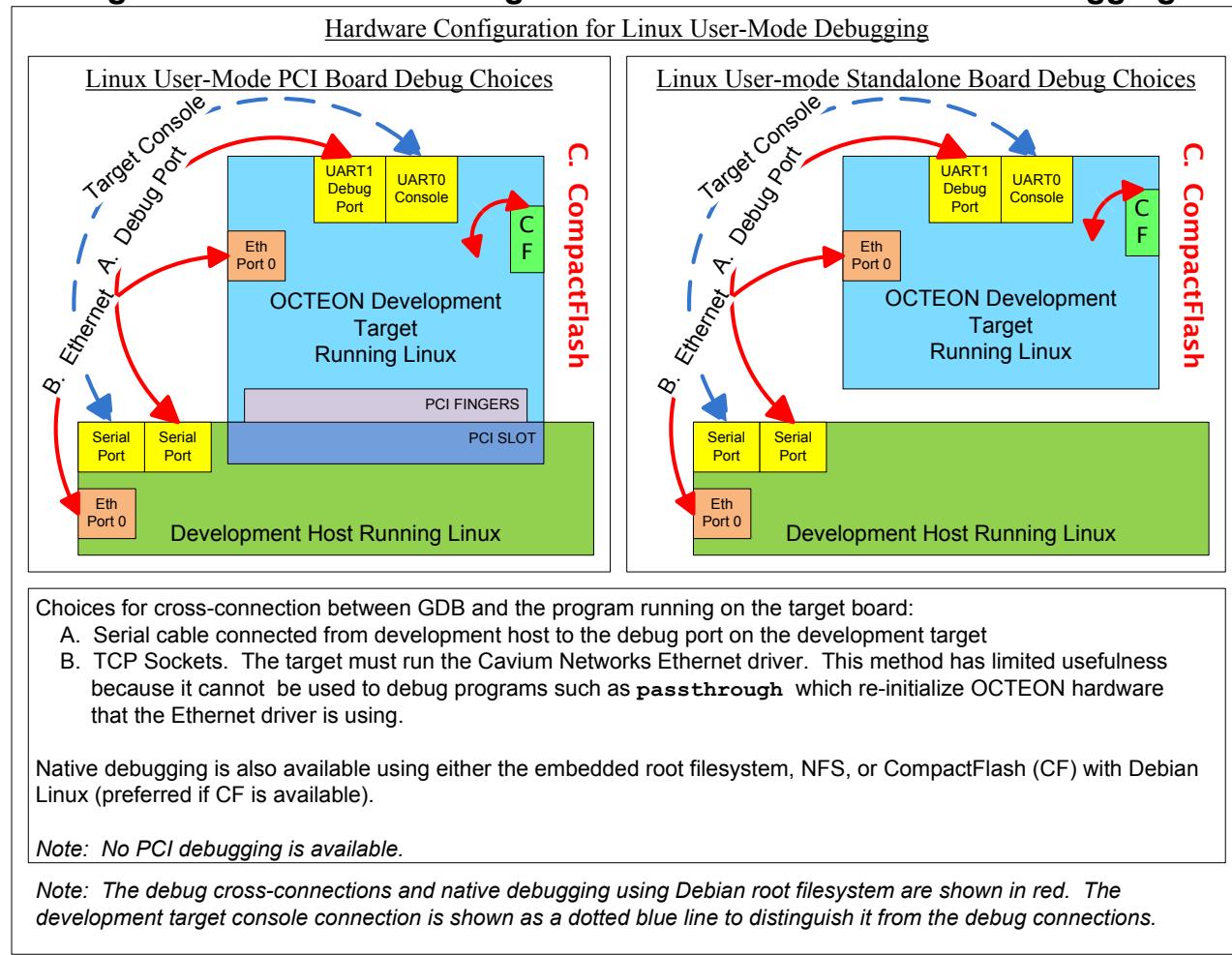
Notes

- This debugger cannot debug a multicore SE-UM application because *gdb* can only debug one process at a time, and SE-UM applications are processes, not threads.
- TCP Sockets cannot be used with SE-UM applications that re-initialize OCTEON hardware. The Ethernet driver has already initialized the hardware. Re-initializing the hardware will cause the Ethernet driver to stop working. It is possible to use TCP Sockets if the Ethernet connection is to the management port Ethernet interface. This interface is available on some OCTEON models. See Section 10.4 – “The Management Port Ethernet Interface”.

10.2 Hardware Configuration for Linux User-Mode Debugging

The hardware configuration is shown in the following figures.

Figure 14: Hardware Configurations for Linux User-Mode Debugging



The serial port on the host should be configured identically for both the console and the debug port: 115200, 8, N, 1.

10.3 Summary: Directions for Different Connection Types

The following tables summarize the commands to use for the different connection methods.

Table 33: Native Debugging of Linux User-Mode Applications

Native Debugging (preferred)
<p><i>Requires Compact Flash drive running Debian Linux.</i></p> <p>A. Install Debian Linux on a flash card, then copy the application from the development host to the flash card. Boot Linux using the Debian root filesystem on the flash card.</p> <p>B. On the target:</p> <p>Add the /usr/local/Cavium_Networks/OCTEON_SDK/tools/usr/bin directory to the <i>start</i> of the PATH environment variable:</p> <pre># type the following text on one command line target#</pre> <p>PATH=/usr/local/Cavium_Networks/OCTEON_SDK/tools/usr/bin:\$PATH</p> <p>Use gdb -v to verify the correct version of GDB is running. If it is correct, the gdb version will include the string "Cavium Networks Version".</p> <pre>target# gdb -v target# gdb <filename></pre> <p>Specify:</p> <pre>gdb> b appmain gdb> c</pre> <p><i>With this method, the application may be run over and over, and can be re-compiled on the target using native tools.</i></p>

Table 34: Debugging Linux User-Mode Applications over Serial Connection

Serial Port
<p><i>Requires a serial connection from host to Debugging Port on the target board.</i></p> <p>A. On the host, configure Linux to enable the second serial port, and rebuild the Linux kernel. The directions for this step are in the SDK document "Linux on the OCTEON".</p> <p>B. On the target:</p> <pre>target# gdbserver <serial_port> <filename></pre> <p>C. On the host:</p> <pre># be sure to add the -linux_64 suffix to the filename for example applications host\$ mipsisa64-octeon-linux-gnu-gdb <filename></pre> <p>Specify:</p> <pre>gdb> target remote <serial_port_on_host> gdb> b appmain gdb> c</pre>

*Note: The SDK documentation may contain the instruction to set the **remotebaud** rate to 9600 when using the serial connection to **gdbserver**. This information applied to*

an older version of the debugger and is no longer true. The baud rate is now set by default to 115200 for both serial and USB converter devices, and does not need to be changed when connecting to gdbserver.

Note: To exit gdbserver, use the disconnect GDB command.

Table 35: Debug Linux User-Mode Applications over Ethernet

Ethernet/TCP Sockets
<p><i>Note: This method cannot be used to debug SE-UM applications (such as passthrough) which initialize OCTEON hardware: the Ethernet driver will stop working.</i></p> <p>A. On the target:</p> <p>Boot Linux, then start the Ethernet driver:</p> <pre>target# modprobe cavium-ethernet</pre> <p>set Ethernet address of target</p> <pre># if dhcp is available: target# udhcpc -i eth0 # otherwise: target# ifconfig eth0 address <TARGET_IP_ADDRESS> netmask 255.255.255.0</pre> <p><i># then start gdbserver, specifying the tcp port number 2349:</i></p> <pre>target# gdbserver :2349 <filename></pre> <p>B. On the host:</p> <p><i># be sure to add the -linux_64 suffix to the filename for example applications</i></p> <pre>host\$ mipsisa64-octeon-linux-gnu-gdb <filename></pre> <p>Specify:</p> <pre>gdb> target remote <IP_ADDRESS_OF_TARGET>:2349 gdb> b appmain gdb> run</pre>

Warning: *TCP Sockets cannot be used with SE-UM applications that re-initialize OCTEON hardware. The Ethernet driver has already initialized the hardware. Re-initializing the hardware will cause the Ethernet driver to stop working.*

Note: The best choice for debugging over Ethernet, if available, is to use the management port Ethernet interface. The driver for this interface does not configure the hardware units. This interface is available on some OCTEON models.

10.4 The Management Port Ethernet Interface

Some OCTEON models support the management port Ethernet interface. This interface is useful for debugging Linux user-mode applications over Ethernet because the driver does not configure the hardware units. (Their interface names are "mgmt0" and "mgmt1".)

If the OCTEON model supports this interface, it can be configured into the kernel by running `make menuconfig` in the `$OCTEON_ROOT/linux/kernel_2.6/linux` directory.

When running `make menuconfig`, the configuration option is accessed via the “Machine selection” sub-menu.

The first kernel menuconfig screen looks similar to this:

```

Machine selection --->
Endianess selection (Big endian) --->
CPU selection --->
Kernel type --->
Code maturity level options --->
General setup --->
Loadable module support --->
Block layer --->
Bus options (PCI, PCMCIA, EISA, ISA, TC) --->
Executable file formats --->
Networking --->
Device Drivers --->
File systems --->
Profiling support --->
Kernel hacking --->
Security options --->
Cryptographic options --->
Library routines --->
---
Load an Alternate Configuration File
Save an Alternate Configuration File

```

To navigate this screen, use the arrow keys on the keyboard. The bottom of the screen provides some options that can be selected with the TAB key. In the first screen, these options are “Select, Exit, and Help”. To select a highlighted option, press Enter when the option “Select” (at the bottom of the screen) is highlighted.

Select “Machine selection”. The Machine selection screen will look similar to this:

```

Select the machine
System type (Support for the Cavium Networks Octeon reference
    [*] Enable Octeon specific options
    [*] Enable RI/XI extended page table bits
    [ ] Build the kernel to be used as a 2nd kernel on the same chip
    [*] Enable support for Compact flash hooked to the Octeon Boot Bus
    [*] Enable hardware fixups of unaligned loads and stores
    [*] Enable fast access to the thread pointer
    [*] Support dynamically replacing emulated thread pointer accesses
<text omitted>
<M> POW based internal only ethernet driver
<*> Management port ethernet driver (CN5XXX) <<< DESIRED SETTING>
< > Octeon watchdog driver
< > Octeon trace buffer (TRA) driver
[ ] Enable enhancements to the IPsec stack to allow protocol offload.
[ ] Enable Cavium Octeon ip-offload module

```

Then follow the usual steps to save the configuration and rebuild the kernel. (See Section 7.1 – “Building the Linux Kernel for Debugging”.)

11 EJTAG (Run-Control) Tools

Enhanced JTAG (EJTAG) is a MIPS standard based on the IEEE Joint Test Action Group (JTAG) standard. EJTAG allows direct control of the cores via a special serial test/access (JTAG) port on the development target.

Run-control tools and probes (EJTAG tools) use EJTAG to debug/probe the hardware. EJTAG tools are particularly useful during board bring-up and bootloader debugging. (The development target must have Linux running well enough to provide a prompt before GDB can be used for debugging.) EJTAG tools can also be used to debug the Linux kernel and SE-S applications.

Run-control tool vendors may also provide an Integrated Development Environment (IDE) software product which runs on the development host. The IDE can be a simple command-line interface, or a GUI.

EJTAG allows a user to:

- Set instruction and data breakpoints which are monitored by hardware within the chip
- Dump memory contents
- Read and write registers (peek and poke)
- Step over single instructions

Run control tools typically provide special features to help speed debugging on new hardware platforms, including:

- Built-in memory testing
- ROM-less booting
- Start-up files which can pre-configure and test OCTEON communications before any software is running on the processor

The ability to perform these operations on a new board is extremely powerful when:

- The boot flash content has not been verified to be correct
- The DDR controller initialization has not been proven to provide a reliable memory interface

This capability is extremely powerful in shortening the time needed to identify whether the board bring-up issue is related to hardware or software issues, and to find the root cause of the problem.

EJTAG provides both a non-intrusive (target continues to run when connecting) and intrusive (target is reset on connect) means of interactively debugging software.

EJTAG-based debugging tools for OCTEON are available from several vendors. A current list of EJTAG vendors is located at <http://www.caviumnetworks.com/>. Please contact the EJTAG vendors for more information.

12 About Debugging on the OCTEON Simulator

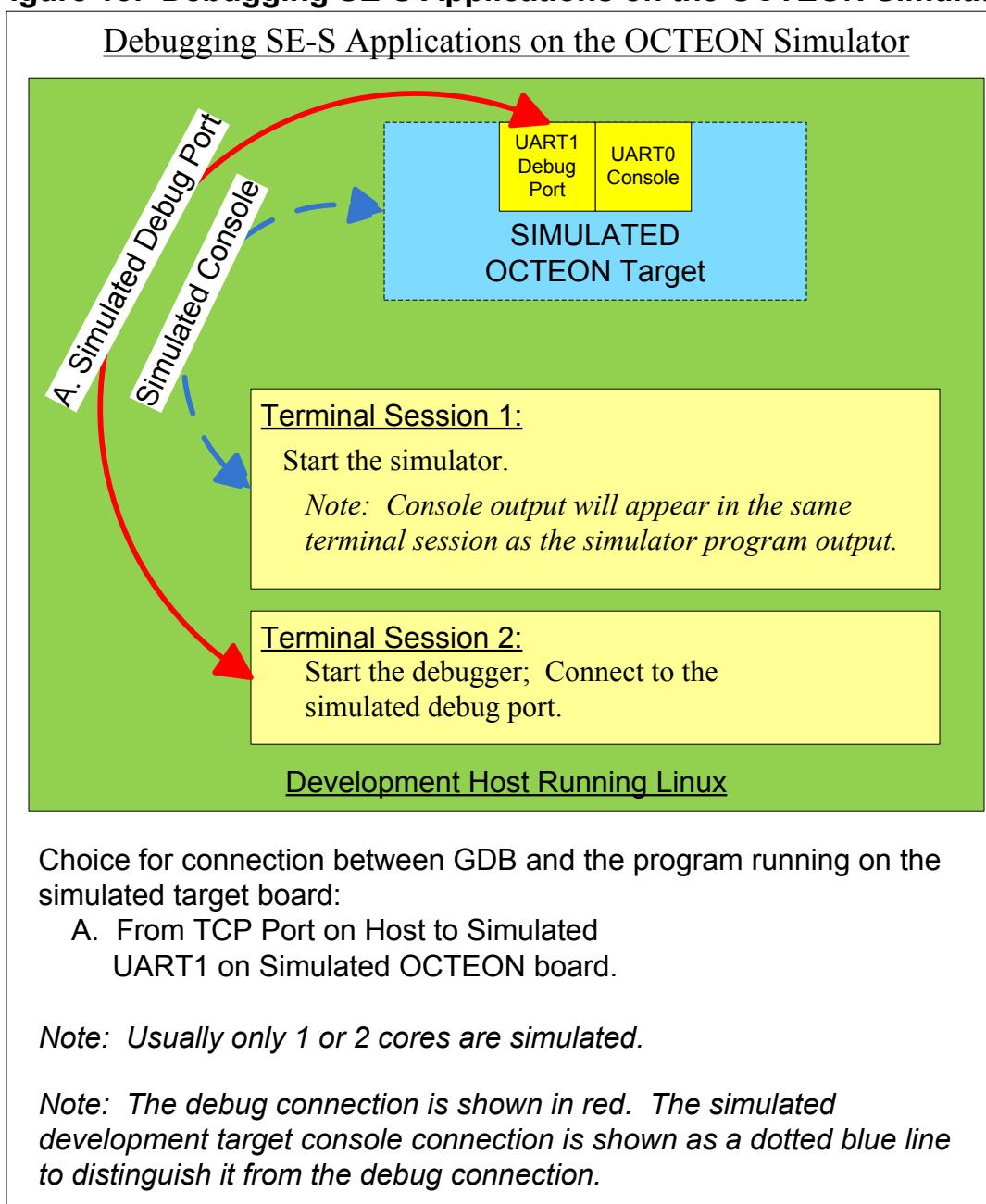
Either SE-S applications or the Linux kernel may be debugged on an i386 or x86_64 development host, running the OCTEON simulator.

When using the OCTEON simulator, the debugger connection is made via TCP sockets using telnet.

Note: The debugger may launch the OCTEON simulator by default when the `target octeon tcp...` command is entered on the command line. See Section 6.6 – “Other Special Commands: `spawn-sim`” for more information.

12.1 Debugging SE-S Applications on the Simulator

The SE-S application `hello` will be used to illustrate debugging SE-S applications on the OCTEON simulator.

Figure 15: Debugging SE-S Applications on the OCTEON Simulator

DEBUGGING
TUTORIAL

Two terminal sessions are needed for debugging: one to start the simulator, and one to run the debugger.

A TCP socket is used as the simulated UART. To determine whether the desired TCP socket (socket 2021, in this case) is already in use, use the Linux command netstat:

```
host$ netstat -an | grep :2021
```

If the socket is in use, then pick an unused number. (Similarly, netstat -a --inet will display only the raw, UDP, and TCP protocol sockets.)

Terminal session #1: start the simulator:

```
hostT1$ cd $OCTEON_ROOT/examples/hello
hostT1$ oct-sim hello -quiet -noperf -numcores=1 -uart1=2021 -debug
```

Note: Target console output is routed to this terminal session. See Section 12.1.2 – “Separating Console Output from Simulator Output” for help separating the OCTEON simulator output from the console output.

Terminal session #2, start the debugger:

```
hostT2$ cd $OCTEON_ROOT/examples/hello
hostT2$ mipsisa64-octeon-elf-gdb hello
# the spawn-sim command must be off before the target command
gdb> set spawn-sim off
gdb> target octeon tcp::2021
Remote target octeon connected to tcp::2021
gdb> b main
Breakpoint 1 at 0x100004fc: file hello.c, line 44.
gdb> c
Continuing.

Breakpoint 1, main () at hello.c:44
44      printf("\n");
gdb> next
45      printf("\n");
gdb> list
40      /* printf() provides maximum flexibility, but is slow due to
41      ** the format string being processed in simulated code.
Normal
42      ** buffering is done by the C library.
43      */
44      printf("\n");
45      printf("\n");
46      printf("Hello world!\n");
47
48      #if 0
49      /* simprintf() passes the format string and up to 7 arguments
to the
gdb> c
Continuing.

Program exited normally.
gdb>
```

The output of `hello` will be seen on the first terminal session, where the simulator was started:

```
Simulating past cycle (approximate instruction) 62.5M ()
Simulating past cycle (approximate instruction) 63.0M ()
Simulating past cycle (approximate instruction) 63.5M ()
Simulating past cycle (approximate instruction) 64.0M ()
Simulating past cycle (approximate instruction) 64.5M ()
Simulating past cycle (approximate instruction) 65.0M ()
Simulating past cycle (approximate instruction) 65.5M ()
Simulating past cycle (approximate instruction) 66.0M ()
PP0:~CONSOLE->
PP0:~CONSOLE-> Hello world!
PP0:~CONSOLE-> Hello example run successfully.
PP0: stopping due to BREAK instruction

SIMULATION COMPLETE at cycle (approximate instruction) 66416751 (0 global
stop phases)
```

Note: If `spawn-sim` is on, and X is not available, the `target tcp...` command will return the following error message (which can be ignored):

`gdb> target tcp::2021`
`xterm Xt error: Can't open display:`

Note: If `spawn-sim` is on, and the simulator has already been started, then the TCP port is already in use. A warning will be issued, and gdb will continue without problem. In SDK 1.9 and above, `spawn-sim` is off by default.

The Simple Executive libraries are built with debugging information, but the source for the libraries is not available. If the command `step` is used instead of `next` in the example above, then the following information will be printed. This happens because `gdb` has stepped into the library function `printf()` (`printf()` is simply a `putchar()` at this point). Without the toolchain source, `gdb` cannot show the source file:

```
gdb> step
putchar (c=10)
  at
/usr/local/Cavium_Networks/toolchain/src/newlib/libc/stdio/putchar.c:95
95
/usr/local/Cavium_Networks/toolchain/src/newlib/libc/stdio/putchar.c: No
such file or directory.
  in
/usr/local/Cavium_Networks/toolchain/src/newlib/libc/stdio/putchar.c
```

12.1.1 About `printf()` and the Simulator

For SE-S applications, the `printf()` function calls different console output functions depending on the runtime environment. The different console types are: UART, PCI console, or simulated console. The simulated console type is automatically set when the application is run on the simulator, causing the output to go to the simulator program's standard output (`stdout`). Because of this, the application output appears in the same terminal session as the simulator output, not a separate connection to a simulated UART.

Note: This is not true when running the Linux kernel on the simulator. Linux does not use a different print function depending on the runtime environment; Linux always sends the output to the UART. When Linux is run on the simulator, the target console (UART) is simulated. The user must connect to the simulated target console to see the Linux prompt.

12.1.2 Separating Console Output from Simulator Output

When running SE-S applications on the simulator, the console output comes out on the same screen as the simulator output. The application output is always labeled “CONSOLE”. By using the Linux command `grep`, the console output can be separated from the simulator output

In the following example, the debug option is not used:

```
# type the following command on one line
host$ oct-sim hello -quiet -noperf -numcores=1 | grep CONSOLE
PP0:~CONSOLE->
PP0:~CONSOLE->
PP0:~CONSOLE-> Hello world!
PP0:~CONSOLE-> Hello example run successfully.
```

Or try this command to create a log file, look at it as it is created, then filter it later:

```
# type the following text on one line
host$ oct-sim hello -quiet -noperf -numcores=1 -uart1=2021 -debug >my_logfile
2>&1 &

host$ tail -f my_logfile
# after exiting the tail command, pick out the CONSOLE lines
host$ grep CONSOLE my_logfile
```

The contents of `my_logfile` will look similar to:

```
No -memsize argument provided, using default of 128 Megabytes
mem size is 128 Megabytes
Using simulator: cn38xx-simulator
Loading hello to boot bus address 0x1000000
Starting simulator....
<text omitted>
Simulating past cycle (approximate instruction) 39.0M ()
```

```
Simulating past cycle (approximate instruction) 39.5M ()
WARNING: use of the magic write function is only supported in the
simulation environment
PP0:~CONSOLE->
PP0:~CONSOLE->
PP0:~CONSOLE-> Hello world!
```

*PP0:~CONSOLE-> Hello example run successfully.
 PP0: stopping due to BREAK instruction*

SIMULATION COMPLETE at cycle (approximate instruction) 39875606 (0 global stop phases)

12.1.3 Using simprintf()

The `simprintf()` function is a special function which can only be used when running SE-S applications on the simulator (there is no library support for this function when building Linux user-mode applications).

When the normal `printf()` function is called by an application which is running on the simulator, even though the output does not go to the simulated UART, the normal string processing and argument conversion is simulated. (See Section 12.1.1 – “About `printf()` and the Simulator”.) This simulation consumes many simulator cycles, slowing down simulation. By using `simprintf()`, the `printf()` simulation is bypassed. The `simprintf()` function simply calls the development host’s `printf()` function run on the host to output to the simulator’s standard output (`stdout`). The result is that the simulator runs much faster because `printf()` is not simulated. Details about the `simprintf()` function can be found in the SDK document “*OCTEON Simulator*”.

Note that `simprintf()` is limited: it can only take seven arguments. Each argument must be of type integer, and use the `%llx` format.

For example, the `hello.c` example includes a call to `simprintf()` which can be easily enabled:

```
#if 1
    /* simprintf() passes the format string and up to 7 arguments to the
     * simulator and is much faster than standard printf. It is limited
     * to 7 arguments of integer types, and all must use long long (%llx)
     * formats in order to be processed properly by the host.
     * No buffering is done - output of each simprintf call is immediate.
     * See README in the 'runtime' directory for more info.
    */
    simprintf("Hello again - a big number is 0x%llx\n", 0x1234567887654321ULL);
#endif
```

The `simprintf()` function is discussed in more detail in the SDK document “*OCTEON Simulator*”, in the section “*Simulator “Magic”*”.

12.1.3.1 Do Not Use `simprintf()` When Running on Hardware

For SE-S applications, the `simprintf()` function should not be used when running the application on hardware because the application will hang.

When running SE-S applications, the `simprintf()` function address is hard-coded to an address which is illegal when running on actual hardware. When `simprintf()` is run on actual hardware, the program will hang.

This is easy to see with the debugger:

1. Build `hello.c` with the `simprintf()` enabled.
2. Set a breakpoint at `main()`, continue to the breakpoint, then
3. Use `n` to step over the `printf()` function calls until the `simprintf()` function would be executed.
4. Expect: When the `simprintf()` function is stepped over, the program will hang (there is no `simprintf()` code at that address when running on actual hardware). The `gdb` prompt will return in response to `Ctrl-C`, but the program cannot be continued. The development target must be reset.

12.2 Simulator Magic Functions

The `simprintf()` function is one of several simulator “magic” functions. These functions, when run on the simulator, do not consume simulation cycles. Instead, their tasks are performed by executing directly on the development host, using operations native to the development host.

Other magic functions are: `open()`, `close()`, `read()`, and `write()`. Using these calls on `stdout`, `stderr`, or `stdin` is not supported.

The simulator magic functions are discussed in more detail in the SDK document “*OCTEON Simulator*”, in the section “*Simulator “Magic”*”.

12.3 Debugging Linux on the OCTEON Simulator

This section provides an overview and then hands-on directions for running Linux on the OCTEON simulator.

12.3.1 Building vmlinuz to Run on the Simulator

First configure Linux for debugging as described in Section 7.1 – “Building the Linux Kernel for Debugging”.

Then build `vmlinuz` to run on the simulator.

```
host$ cd $OCTEON_ROOT/linux
host$ sudo ls
host$ sudo make sim >make.out 2>&1 &
```

Technical Note: Although the same kernel is used for both the `kernel` and `sim` targets, the root filesystem is different. In the `kernel` case, the root filesystem is a compressed initramfs filesystem packaged with the kernel into `vmlinuz`. In the `sim` case, the root filesystem is an uncompressed ext3 filesystem which is not packaged into `vmlinuz`. Instead, it is loaded at a fixed location in memory by the simulator.

12.3.2 Starting Linux on the Simulator

A script, `oct-linuX`, is used to start the simulator with the options needed by the Linux kernel. The contents of the `oct-linuX` script are:

```
#!/bin/bash

memory=384
uart=2020
packet_port=2000

oct-sim linux/vmlinux.64 -envfile=u-boot-env -memsize=${memory} -uart0=${uart}
-serve=${packet_port} -ld0x40000000:../embedded_rootfs/rootfs.ext3 $*
```

(Note: The files vmlinux.64 and vmlinux are identical.)

Running vmlinux on the OCTEON simulator will require three terminal sessions: one to start the simulator, one to run the debugger, and one to connect to the simulated target console.

The tables below contain the directions. Here are a couple of notes which accompany the instructions in the tables:

1. To exit the simulator, type **Ctrl-C** in the terminal session where it was started, or exit **gdb**.
2. Once Linux boots to the interactive shell prompt, it can be useful to change **telnet** to character mode instead of line mode. In the **telnet** session, press **Ctrl-]** and enter mode **char** at the prompt. Then hit **Enter** a few times. Shell tab completion and other interactive aspects should now work.

Figure 16: Debugging Linux on the OCTEON Simulator

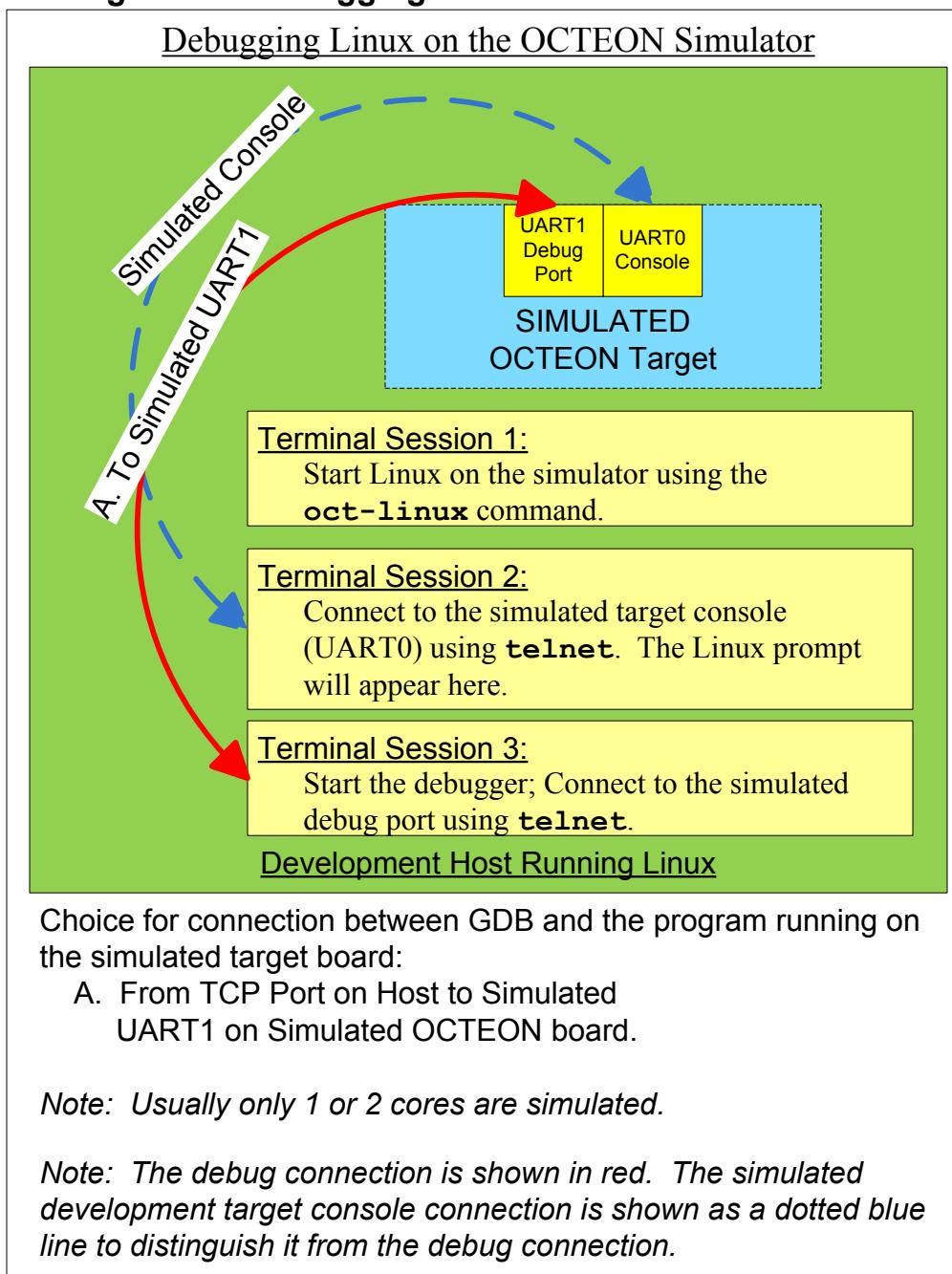


Table 36: Run Linux on the OCTEON Simulator, Part 1

Steps	Note
1. Configure the Kernel for Debugging	If this step was already done, no need to repeat it!
<pre># type the following text on one line host\$ cd \$OCTEON_ROOT/linux/kernel_2.6/linux # see note for menuconfig options host\$ make menuconfig</pre>	Select the menuconfig options: <ul style="list-style-type: none"> * Machine selection: Octeon watchdog driver - off * General setup: optimize for size - on * Kernel hacking: Remote GDB debugging using the Cavium Networks Multicore - on
2. Build the Kernel with Embedded Root Filesystem	If this step was already done, no need to repeat it!
<pre>host\$ cd ../.. # the directory should now be \$OCTEON_ROOT/linux # enter password now so it is stored for the next command host\$ sudo ls # type the following text on one line host\$ sudo make -s CONFIG_FRAME_POINTER=y sim >make.out 2>&1 & # Note: This step can take up to 20 minutes. When it has # completed successfully, the kernel_2.6/linux/vmlinux # file will have been created.</pre>	Note: The sudo ls command is merely used to set the <i>root</i> password. The <i>root</i> password will be stored for about 5 minutes for use in the next command (sudo make...).
# verify the build is complete before continuing host\$ wait	
3. Create a Total of Three Terminal Sessions on the Development Host	
Create three terminal sessions on the host, for these three purposes: 1. run the simulator 2. connect to the target console 3. run the debugger	
<i>Continued in the next table...</i>	

Table 37: Run Linux on the OCTEON Simulator, Part 2

Steps	Note
4. Boot the Kernel on the OCTEON Simulator	
<pre>First Terminal Session (hostT1\$): host\$ cd \$OCTEON_ROOT/linux/kernel_2.6 # type the following text on one line hostT1\$ oct-linux -noperf -quiet -numcores=1 -uart1=2021 -debug</pre> <p>Note: The following error will occur if the make sim step was omitted:</p> <pre>Error: Unable to open binary file: ./embedded_rootfs/rootfs.ext3</pre>	<p>This command will run Linux on the OCTEON simulator.</p> <p>Expect to see:</p> <pre>Loading u-boot environment from file: u-boot-env mem size is 384 Megabytes Using simulator: cn38xx-simulator Loading linux/vmlinu.64 to boot bus address 0x1000000 Starting simulator.... <text omitted> Live Packet Listening at pak.caveonetworks.com, port 2000 (0x7d0) Uart Listening at pak.caveonetworks.com, port 2020 (0x7e4) Uart Listening at pak.caveonetworks.com, port 2021 (0x7e5) waiting for a connection to uart 0 1 waiting for a connection to uart <text omitted></pre>
5. Connect to the target console	
<pre>Second Terminal Session (hostT2\$): hostT2\$ telnet localhost 2020</pre> <p>Note: The following error will occur if the telnet to 2020 is started before the simulator "waiting for a connection to uart 0 1" message appears:</p> <pre>Trying 127.0.0.1... telnet: connect to address 127.0.0.1: Connection refused</pre>	<p>This command will connect to the simulated target console. Note: Wait until "waiting for a connection to uart 0 1" appears in the simulator output, but don't wait too long. The simulator will timeout after about 45 seconds if no console connection is made.</p> <p>Expect to see:</p> <pre>argv[2]: coremask=0x1 argv[3]: debug ELF file is 64 bit Attempting to allocate memory for ELF segment: addr: 0xffffffff80100000 (adjusted to: 0x000000000000100000), size 0x1a99d58 <text omitted> Bootloader: Done loading app on coremask: 0x1 <Linux will stop here></pre>

Continued in the next table...

Table 38: Run Linux on the OCTEON Simulator, Part 3

Steps	Note
6. Start GDB on the Host; Debug the Kernel	
Third Terminal Session (hostT3\$) : <i># type the following text on one line</i> hostT3\$ cd \$OCTEON_ROOT/linux/kernel_2.6/linux	Go to the directory where the <code>vmlinux</code> source is located on the host.
<i># start gdb</i> host\$ mipsisa64-octeon-elf-gdb vmlinux	Use <code>-q</code> (quiet) for fewer start-up messages. Expect to see: GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64 Copyright (C) 2006 Free Software Foundation, Inc. GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type "show copying" to see the conditions. There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "--host=i686-pc-linux-gnu --target=mipsisa64-octeon-elf"...
<i># don't start the simulator from the debugger; it is already started</i> gdb> set spawn-sim off	<i>This step is only needed if spawn-sim is set to on by default.</i>
<i># connect to the debug port</i> gdb> target octeon tcp:::2021	Expect to see: Remote target octeon connected to tcp:::2021
<i># set a breakpoint in r4k_wait()</i> gdb> b r4k_wait	Expect to see: Breakpoint 1 at 0xffffffff801214b8: file arch/mips/kernel/cpu-probe.c, line 57.
<i># continue to the breakpoint</i> gdb> c	Expect to see: Continuing. Breakpoint 1, r4k_wait () at arch/mips/kernel/cpu-probe.c:57 57 wmb();
<i>Continued in the next table...</i>	

Table 39: Run Linux on the OCTEON Simulator, Part 4

Steps	Note
6. Debug the Kernel, continued	
# show 10 source lines at the breakpoint gdb> list	Expect to see: 52 * a non-enabled interrupt is requested. 53 */ 54 static void r4k_wait(void) 55 { 56 #ifdef CONFIG_CPU_CAVIUM_OCTEON 57 wmb(); 58 #endif 59 __asm__("
	.set mips3 \n" 60 " wait \n" 61 " .set mips0 \n");
# r4k_wait() is called by the idle loop, so clear the # breakpoint in order to continue to the prompt gdb> clear	Expect to see: Deleted breakpoint 1
# continue to run the program gdb> c	Expect to see: Continuing.
The Linux prompt will now come up on the simulated target console.	

12.3.3 Running Linux User-Mode Applications on the Simulator

Eventually, an interactive shell will appear in the simulated target console. The amount of time before the prompt appears can be 5-20 minutes, depending on the speed of the host processor, the amount of memory installed, and the number of cores requested.

Once the interactive shell prompt appears, a SE-UM application, such as `named-block`, can be run on the simulator.

Linux user-mode applications cannot be debugged on the simulator. There is no facility for GDB to connect to the target application.

13 Appendix A: Common GDB Commands

The following table provides a few of the commonly-used GDB commands. All of these commands are standard commands provided with GDB.

When using `mipsisa64-octeon-elf-gdb`, additional commands are available, including multicore and PCI commands. These commands are documented in Section 6 – “About Debugging SE-S Applications or the Linux Kernel”.

Table 40: A Few Common GDB Commands

Command	Notes
<code>awatch expr</code>	Set a watchpoint for the memory location specified by <code>expr</code> . The watchpoint will trigger if the memory location is changed, or the contents of the memory location is changed (a combination of <code>watch + rwatch</code>).
<code>file [filename]</code>	File name to debug. GDB can also be started using the <code>gdb filename</code> command.
<code>b [file:]line</code>	(break) Set a software breakpoint in <code>file</code> at the specified <code>line</code> number.
<code>bt</code>	(backtrace) Print trace of all frames in stack.
<code>c [count]</code>	(continue) continue running. If <code>count</code> is specified, ignore this breakpoint the next <code>count</code> times.
<code>detach</code>	Release target from GDB control. Useful to get <code>gdbserver</code> to exit on the target.
<code>dir name</code>	Add directory <code>name</code> to front of source path.
<code>hbreak [file:]line</code>	Set a hardware breakpoint in <code>file</code> at the specified <code>line</code> number.
<code>help</code>	List classes of commands. Can also specify <code>help class</code> or <code>help command</code> to get more help.
<code>info break</code>	Show defined breakpoints.
<code>info watch</code>	Show defined watchpoints.
<code>list</code>	Show next 10 lines of source.
<code>n</code>	(next) Execute next source line, stepping over function calls.
<code>ni</code>	(next instruction) Execute next machine instruction, rather than next source line.
<code>quit</code>	Exit GDB.
<code>p [/f] [expr]</code>	(print) Show value of expression. Optionally, formatting is specified with <code>/f</code> , for instance <code>/x</code> for hexadecimal.
<code>run [arglist]</code>	Run the program from the beginning. If connection is PCI, board will be rebooted, program downloaded and run.
<code>rwatch expr</code>	Set a watchpoint for the memory location specified by <code>expr</code> . The watchpoint will trigger if the memory location is read.
<code>s</code>	(step) Execute next source line, stepping into function calls.
<code>si</code>	(step instruction) Execute next machine instruction, rather than next source line.
<code>source script</code>	Read and execute GDB commands from file <code>script</code> .
<code>target octeon</code>	Connect to SE-S application on the target over serial connections. This command is also used with the simulator to connect to the simulated debug port.
<code>target octeonpci</code>	Connect to SE-S application on the target over PCI.
<code>target remote</code>	Connect to <code>gdbserver</code> running on the target.
<code>watch expr</code>	Set a watchpoint for the memory location specified by <code>expr</code> . The watchpoint will trigger if the contents of the memory location are changed.

When booting the program, specify `debug` on the load command line. The `debug` option causes the bootloader to load and run the program, but the program will be stopped in the debug exception

handler. Note that the GDB command to *continue* is usually used in the examples below instead of *run*, because the program is already running (it is waiting in the debug exception handler). Using *run* after the program has begun (instead of *continue*) will result in this message:

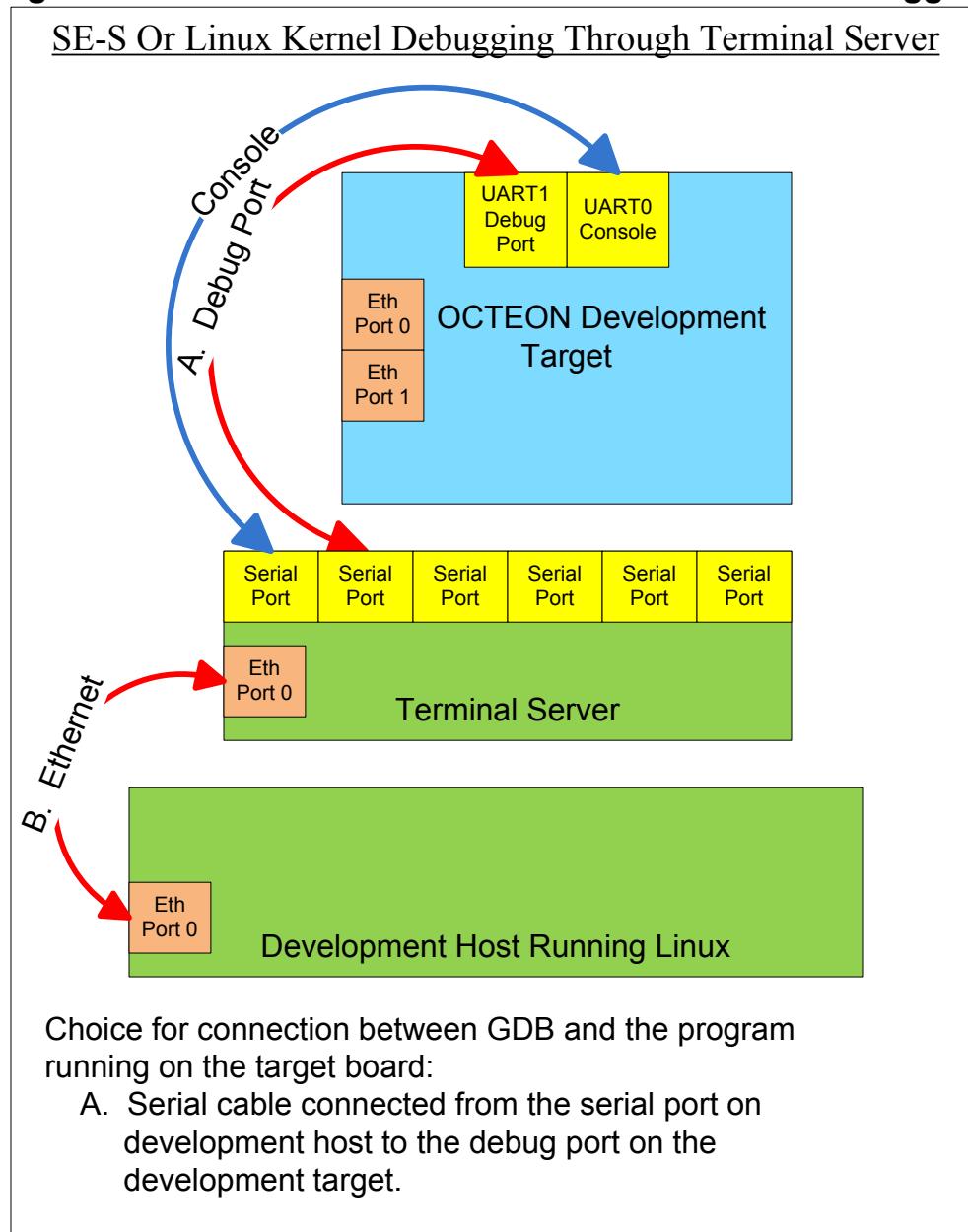
```
gdb> run
The program being debugged has been started already.
Start it from the beginning? (y or n) n
```

An exception to this rule occurs when native gdb is used to debug a Linux user-mode application. In this instance, the *r* (run) gdb command is used instead of *c* (continue).

14 Appendix B: Connecting Using a Terminal Server

A terminal server is a remote host which has a direct serial connection to the development target's debug port. This direct connection is identified by a server port number on the terminal server. Typically a terminal server is connected to multiple development targets.

The following figures illustrate using a terminal server to connect to the OCTEON processor. This connection can be used with the `mipsisa64-octeon-elf-gdb` debugger.

Figure 17: Terminal Server - SE-S or Linux Kernel Debugging

14.1 Terminal Servers and “Garbage” Characters

There may be a problem where “garbage” characters appear in the `gdb` command window after the `target` command, and then nothing further is printed. This may be caused by the terminal server misinterpreting the initial characters sent by `gdb`.

On some terminal servers, there is no way to fully disable `telnet` negotiations. In this case, the initial characters sent by `GDB` can be misinterpreted by the terminal server and interpreted as a negotiation response.

Inserting a delay before GDB sends the first packet can help. The GDB variable `transmit-delay` is used to specify how many seconds GDB should wait before sending the first packet. Set this delay to 2 seconds before typing in the `target` command.

```
gdb> set transmit-delay 2
gdb> target octeon tcp:<ip_address_of_terminal_server>:<server_port>
```

15 Appendix C: How to Simplify the Command Lines

15.1 Script Files

Instead of typing each command on the command line, a script can be used for the start-up commands. After the script is run, GDB will still respond to typed commands on the command line.

The following text is in the file `gdb_script`:

```
file hello
set pci-bootcmd oct-pci-reset
target octeonpci bootoct 0x20000000 coremask=3
set step-all on
b main
```

One option is to invoke the script on the GDB command line (in this example, the script is in the user's home directory):

```
host$ mipsisa64-octeon-elf-gdb -x ~/gdb_script
```

Another option is to source the script once GDB is started (in this example, the script is in the current directory):

```
host$ sudo mipsisa64-octeon-elf-gdb -q
(Core#0-gdb) source gdb_script
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x80000000]
Found Octeon on bus 3 in slot 13. BAR0=0xd8000000[0x1000],
BAR1=0xd0000000[0x80000000]

Sending bootloader command: bootoct 0x20000000 coremask=3 debug
0x10000388 in __octeon_trigger_debug_exception ()
Breakpoint 1 at 0x100002e0
(Core#0-gdb)
```

15.2 Using an Alias to Simplify Start-Up

The debugging command lines can be difficult to type (`mipsisa64-octeon-elf-gdb ...`). The command can be made into an alias, such as:

```
host$ alias pcidbg="mipsisa64-octeon-elf-gdb -x ~/gdb_script"
```

After the alias is created, simply type:

```
host$ pcidbg
```

15.3 The .gdbinit file

In addition to script files, a per-project .gdbinit file can be used. This file is located in the user's home directory, and is executed each time gdb is started.

The following text is in the file ~testname/.gdbinit:

```
file hello
set pci-bootcmd oct-pci-reset
target octeonpci bootoct 0x20000000 coremask=3
set step-all on
b main
c
list
```

To run it, simply cd to the \$OCTEON_ROOT/examples/hello directory, and start gdb:

```
host$ sudo mipsisa64-octeon-elf-gdb -q
```

15.4 Environment Variables

To create a simpler tools prefix name, add two environment variables to the ~/.bash_profile file:

```
# use lt to reference tools used to build Linux SE-UM applications
export lt=mips64-octeon-linux-gnu
# use st to reference tools used to build SE-S applications and the Linux Kernel
export st=mipsisa64-octeon-elf
```

Source in the file to set the environment variables into the shell's environment:

```
host$ source ~/.bash_profile
```

To use the new environment variable, type:

```
host$ cd $OCTEON_ROOT/examples/hello
host$ make hello
host$ $st-size hello
      text      data      bss      dec      hex filename
    49928      4104      768    54800      d610 hello
```

16 Appendix D: Graphical Debugger

The graphical tool, mipsisa64-octeon-elf-ddd, can be used to invoke graphical debugging for SE-S applications and the Linux Kernel. Directions are provided at <http://www.gnu.org/software/ddd/>.

The graphical debugger provides drop-down menus which simplify finding the right command to display the content of data variables and registers.

The script `oct-debug` can be used to start `ddd` as described in Section 18 – “Appendix F: The `oct-debug` Script”.

17 Appendix E: Core Files

Linux user-mode applications will generate core files as usual, but will not include any OCTEON-specific features such as special registers.

The following information is standard Linux:

1. Make sure the directory the program is executed in is writable.
2. The program must not have the `setuid` or `setgid` bits set.
3. Set the `ulimit` large enough to accommodate the core file, for example:

```
target# ulimit -c unlimited
```

To recover the core file, use either the CompactFlash, or `ftpput` from the embedded root filesystem to a FTP server connected on the same network. Note that recovering the file can be difficult because:

- `ftpput` requires the Ethernet driver which collides with programs which initialize the hardware units, so this may not be the best way to copy a core file back to the development host
- flash disks should not be inserted or removed while the development target is powered on

17.1 Core File Names

Standard Linux allows the core file to be named according to user-supplied specifications:

```
%% output one '%'  
%p pid  
%u uid  
%g gid  
%s signal number  
%t UNIX time of dump  
%h hostname  
%e executable filename
```

Put the core filename specification into the `/proc/sys/kernel/core_pattern` file.

For example:

```
target# echo core.%e.%t > /proc/sys/kernel/core_pattern
```

In addition to the core file naming conventions shown above, the PID can be included in the core file name. The `/proc/sys/kernel/core_uses_pid` file is used to suffix the core filename with the process PID. This will only happen if the file contains a non-zero value.

For example:

```
target# echo 1 > /proc/sys/kernel/core_uses_pid
```

17.2 Example Core Dump

The following directions will create a separate example application which will cause a core dump.

Note: If necessary, a faster way to see core file creation is to simply edit an existing example, so that the example is already part of the embedded root filesystem. In this case, a separate example is created, using the fpa_simplified code as the starting point because it has nested function calls.

To see an example core dump:

1. Copy the `fpa_simplified` example to a new directory:
`$OCTEON_ROOT/examples/cause_core`
2. Change the file name (`fpa.c` to `cause_core.c`)
3. Edit the Makefile to change the file name: `OBJS = $(OBJ_DIR)/cause_core.o`
4. Add the new example to the embedded root filesystem as shown in the *SDK Tutorial* chapter.

Then add the following function to `cause_core.c`:

```
static void cause_core(void)
{
    volatile char *bad_address = 0x0;

    printf("ATTEMPTING TO CAUSE CORE DUMP...\n");
    // write to bad_address, causing a core dump
    bad_address[0] = 'b';
}
```

To cause the coredump, this function can be executed in `populate_one_fpa_pool()`:

```
<text omitted>
    result = cvmx_fpa_setup_pool(pool_num, pool_name, memory,
buffer_size,
                                num_buffers);

    // Cause the core dump now
    cause_core();

    return result;
}
```

Then build `cause_core` for debugging. Ensure that `cause_core` is not stripped when it is copied to the embedded root filesystem. Boot the new vmlinu file, and run `cause_core` to create the core dump.

17.3 Example of Using `ftpput` to Transfer a Core File

In the following example, the Ethernet driver is configured, and then `ftpput` is used to transfer the core file to the FTP server.

```

target# modprobe cavium-ethernet
target# udhcpc -i eth0
udhcpc (v1.2.1) started
Jan 1 00:05:51 (none) local0.info udhcpc[753]: udhcpc (v1.2.1) started
Sending discover...
Jan 1 00:05:51 (none) local0.debug udhcpc[753]: Sending discover...
Sending select for 192.168.51.173...
Jan 1 00:05:51 (none) local0.debug udhcpc[753]: Sending select for
192.168.51.1
73...
Lease of 192.168.51.173 obtained, lease time 86400
Jan 1 00:05:51 (none) local0.info udhcpc[753]: Lease of 192.168.51.173
obtained
, lease time 86400
deleting routers
SIOCDELRT: No such process
adding dns 192.168.51.254
target# ifconfig

eth0      Link encap:Ethernet HWaddr 00:0F:B7:10:03:E2
          inet addr:192.168.51.173 Bcast:192.168.51.255 Mask:255.255.255.0
          inet6 addr: fe80::20f:b7ff:fe10:3e2/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
            RX packets:447 errors:0 dropped:0 overruns:0 frame:0
            TX packets:639 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:38864 (37.9 Kb) TX bytes:858458 (838.3 Kb)

lo       Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING MTU:16436 Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

# note that your password is transmitted in clear text
# type the following text on one line, entering your data where <> is specified
target# ftpput -v -u <username> -p <password> <development_host_IP_address>
core core

# back on the host, the core file is now located in your home directory (if
# username was your login name).

```

17.4 Analyze Core File with GDB

Once the core file has been copied to the development host, the core file can be analyzed with GDB as usual. Note that GDB sometimes is unable to give an accurate back trace for MIPS core files.

Note: Be careful to specify the ELF filename on the GDB command line. Without the ELF filename, GDB assumes the ELF file is compiled for the O32 ABI. The O32 ABI only supports 32-bit registers. Without the ELF file name, GDB will crash while reading the register information.

In this example, the ELF file is correctly specified on the command line:

```
host$ mips64-octeon-linux-gnu-gdb cause_core-linux_64 -c core
GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=mips64-octeon-
linux-gnu"...

warning: core file may not match specified executable file.
Core was generated by `./cause_core'.
Program terminated with signal 11, Segmentation fault. << The fault is shown
#0 0x0000000120003e60 in populate_one_fpa_pool (pool_num=5,
    buffer_size=32768, num_buffers=33,
    pool_name=0x12008ed20 "This is my THIRD pool") at cause_core.c:63
63          bad_address[0]='b';


```

The GDB command **bt** can now be used to view the stack back trace:

```
gdb> bt << The stack backtrace, below, shows the function call sequence
#0 0x0000000120003e60 in populate_one_fpa_pool (pool_num=5,
    buffer_size=32768, num_buffers=33,
    pool_name=0x12008ed20 "This is my THIRD pool") at cause_core.c:63
#1 0x0000000120004680 in appmain (argc=<value optimized out>,
    argv=<value optimized out>) at cause_core.c:234
#2 0x000000012001039c in main (argc=1, argv=0xfffff9dde18)
    at /home/testname/sdk/executive/cvmx-app-init-linux.c:418
```

17.5 The Executable Name is Required on GDB Command Line

Note that the executable name (`cause_core-linux64`, in the example) is required on the command line along with the core filename. Without the executable name, gdb assumes the ABI is O32 ABI (32-bit registers). GDB will then issue an error message (bad register size), and the **bt** command will not work, as shown below:

```
host$ mips64-octeon-linux-gnu-gdb -c core
GNU gdb 6.5 Cavium Networks Version: 1_8_0, build 64
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=mips64-octeon-
linux-gnu".
Core was generated by `./cause_core'.
Program terminated with signal 11, Segmentation fault.
.../src/gdb/mips-tdep.c:606: internal-error: bad register size
A problem internal to GDB has been detected,
further debugging may prove unreliable.
Quit this debugging session? (y or n) n
.../src/gdb/mips-tdep.c:606: internal-error: bad register size
A problem internal to GDB has been detected,
```

```
further debugging may prove unreliable.  
Create a core file of GDB? (y or n) n  
(gdb) bt  
.../.../src/gdb/mips-tdep.c:606: internal-error: bad register size  
A problem internal to GDB has been detected,  
further debugging may prove unreliable.  
Quit this debugging session? (y or n) y
```

When the executable filename is included on the command line, `gdb` reads the file and determines the correct ABI.

18 Appendix F: The `oct-debug` Script

The `oct-debug` script can be used with either SE-S applications or the Linux kernel.

The different command-line options are:

```
host$ oct-debug <executable> <serial_device>  
host$ oct-debug <executable> pci [PCI_BOOTCMD=pci_bootcmd] <pci-options>  
host$ oct-debug <executable> <simulator_options>
```

Depending on the command line options, this script will start either:

1. The `mipsisa64-octeon-elf` toolchain debugger (if either a serial device or `pci` is the second argument)
 - a. either the graphical debugger DDD as a front-end to GDB
 - b. or if DDD is not present on the system, the command-line debugger GDB is started
2. The OCTEON simulator (if neither a serial device or `pci` is specified as the second argument on the command line)

When the `oct-debug` script is used to start the debugger, if `mipsisa64-octeon-elf-ddd` is found on the system, then it will be invoked, along with the debugger, `mipsisa64-octeon-elf-gdb`. Otherwise, the command-line debugger, `mipsisa64-octeon-elf-gdb`, will be invoked without the graphical front-end.

When the debugger is started, by default DDD is started. DDD is a graphical front-end used with `gdb`, therefore DDD requires X.

If X is not available, the following error message will occur:

```
host$ oct-debug hello -quiet -noperf -numcores=1
Error: Can't open display:
```

To start GDB without DDD, either remove DDD from the system, or comment out the line which starts DDD in the file \$OCTEON_ROOT/host/bin/oct-debug, as shown below:

```
#if mipsisa64-octeon-elf-ddd -v &> /dev/null
#then
#    debugCmd="mipsisa64-octeon-elf-ddd --debugger mipsisa64-octeon-elf-gdb --
command ${cmdfile} ${program}"
#else
#    echo "*****"
#    echo "The \"ddd\" debugger isn't installed. Defaulting to the command
line"
#    echo "*****"
#    debugCmd="mipsisa64-octeon-elf-gdb -x ${cmdfile} ${program}"
#fi
```

19 Appendix G: Debian and the Cavium Networks Ethernet Driver

By default, when the Debian root filesystem comes up, the Cavium Networks Ethernet driver module is loaded.

This is only a problem when *both* of the following conditions occur:

1. Debian is used for native debugging
2. The application being debugged is a Linux user-mode application which reconfigures hardware units also used by the Cavium Networks Ethernet driver

If the Ethernet driver is loaded as a module:

After Debian is installed on the compact flash, mount the second partition and remove the cavium-ethernet.ko file from the /lib/modules directory.

```
# first mount partition 2 of the CompactFlash on the
# development host (in this case the mount directory is /mnt/cf2)
# Note that the kernel directory will vary. In this
# case, it is 2.6.21.7-Cavium-Octeon
# type the following text on one line
host$ cd /mnt/cf2/lib/modules/2.6.21.7-Cavium-Octeon/drivers/net/cavium-
ether
# Then remove the module:
host$ rm cavium-ether.ko
```

If the Ethernet drive is compiled into the kernel:

Remove the driver from the kernel by using the kernel menuconfig option. Select “Device Drivers”, “Networking Device Support”, “Ethernet (1000 Mbit)”, and then unselect “Cavium Networks Octeon Ethernet Support”. (These directions are for the 2.6.27 kernel.)

OCTEON Application Performance Tuning Whitepaper

TABLE OF CONTENTS

TABLE OF CONTENTS	1
LIST OF TABLES.....	4
LIST OF FIGURES	4
1 Introduction.....	5
1.1 References	5
2 Performance Tuning Overview.....	6
2.1 Test Setup	6
2.2 Evaluating whether Performance Tuning is Appropriate	7
2.3 Start with the Minimum Set of Cores, then Scale Up.....	7
2.4 Locate the Bottleneck in the Code.....	9
2.5 Performance Testing Tools.....	10
2.5.1 Performance Tools on Simple Executive	10
2.5.2 Performance Tools on Linux	11
2.5.3 A Note about the Simulator and Viewzilla.....	11
2.6 Instrumenting the Code and Using Performance Counters	12
2.6.1 Cycle Counters.....	12
2.6.2 CP0 Performance Counters.....	12
2.6.2.1 Sample Code: Enable and Read CP0 Counters	13
2.6.3 L2 Cache Performance Counters	16
2.6.4 DRAM Utilization Information	17
2.7 When Is Performance Optimization Complete?	18
3 Performance Tuning Checklist	18
4 Hardware Architecture Overview	20
5 Software Architecture for High Performance	22
5.1 L2 Cache configuration: Aliased Cached Indexing.....	22
5.1.1 Unaliased Caching Indexing Algorithm	22
5.1.2 Aliased Cache Indexing Algorithm	23
5.2 Configuring the Right Amount of Packet Data Buffers and WQE Buffers	23
5.3 Simple Executive versus Linux or other OS.....	24
5.4 Concurrent Programming Techniques	24
5.4.1 Critical Regions and Locks.....	25
5.4.2 Minimize use of Shared Data	25
5.4.3 Minimize Critical Regions.....	25

PERFORMANCE
TUNING

5.5	Pipelined versus Run-to-Completion Software Architecture	25
5.6	Event-driven Loop versus Interrupt Handling for Packet Processing	26
6	Tuning the Minimum Set of Cores	26
6.1	Compiler Choice	26
6.2	Compiler Optimization (-O3)	26
6.3	Re-Configuring the Right Amount of Packet Data Buffers and WQE Buffers	26
6.4	Memory Alignment	26
6.4.1	Align Data on Natural Address Boundaries	26
6.4.2	Simple Executive Facilities to Support Memory Alignment	28
6.4.3	Pad Structures to Align on Natural Boundaries	28
6.5	Data Structure Compaction (packing), Re-arranging Structures	28
6.5.1	Packing: Space versus Speed Tradeoff	28
6.5.2	Re-Arrange Structure Fields to Save Space	29
6.6	Large Data Structures: Working with Cache-line Size	30
6.7	Group Common Data Together	30
6.8	Loop Unrolling	31
6.8.1	Sample Code: Loop Unrolling	31
6.9	Replace <code>memset()</code> and <code>memcpy()</code> when Needed	32
6.9.1	Sample Code: Replacing <code>memcpy()</code> and <code>memset()</code>	32
6.10	Using Free Pool Allocator (FPA) Memory Pools to Manage Free Buffers	33
6.11	Cache Prefetch	33
6.11.1	Sample Pseudo code: Prefetch	34
6.12	Prepare-For-Store	35
6.13	Scratchpad: Core-local Storage	36
6.14	Asynchronous FPA Allocation	37
6.15	Don't Write Back (DWB) Commands	37
6.15.1	DWB Commands from the Core	37
6.15.2	DWB Commands from other Hardware Units	39
6.16	Hardware CRC Engine	40
6.16.1	Sample Code: Using the OCTEON Processor CRC Engine	40
6.16.2	Performance Comparison: Hardware versus Software CRC	42
6.17	Hardware Hash Engine	43
6.17.1	Sample Code: Using the OCTEON Processor HASH Engine	43
6.17.2	Performance Comparison: Hardware versus Software Hashing	45
6.18	Hardware Timers	45
6.19	Hardware Fetch and Add (FAU) Unit	46
6.20	Asynchronous Fetch and Add Operations	46
6.21	Work prefetch: Asynchronous <code>get_work</code>	46
6.22	Interleaving Prefetch with Computational Instructions	46
6.22.1	Sample Code: Interleaving Prefetch	46
6.23	Hardware TCP/UDP Checksum Calculation	47
6.24	Use Functions Wisely	47
6.25	Update Bit-Fields Wisely	47
6.25.1	Sample Code: Updating Bit-Fields Wisely	49
6.26	Read After Write	49
6.26.1	Sample Code: Read after Write	51

7	Tuning Multi-core Applications (Scaling).....	51
7.1	Re-Configuring the Right Amount of Packet Data Buffers and WQE Buffers.....	51
7.2	Tune Initial Tag Values to Separate Flows.....	51
7.3	Set Initial Tag Type to ORDERED if Possible	51
7.4	Switch Tag Type to ORDERED or NULL when Possible.....	51
7.5	Use Asynchronous Switch Tag Operations	51
7.6	Critical Regions	52
7.7	Replace Spinlocks with Packet-Linked Locks When Possible.....	52
7.7.1	Spinlocks.....	52
7.7.2	The Scheduling / Synchronization / Order Unit and ATOMIC Tag Type	53
7.7.3	Example of Spinlock versus SSO ATOMIC Locking	54
7.7.3.1	Sample Pseudo code: Using Spinlock	54
7.7.3.2	Sample Pseudo Code: Using SSO / ATOMIC Tag Type	54
7.8	Arena-based Memory Allocation	55
7.9	L2 Cache Configuration: Way Partitioning and Cache-Block Locking.....	56
8	Linux-specific Tuning	56
8.1	TLB Exceptions and Huge Page Size.....	56
8.2	Use CPU Affinity for Processes/Threads	56
8.3	Direct all Packet RX Interrupts to the Same Core.....	56

LIST OF TABLES

Table 1: CP0 Performance Counter Events	14
Table 2: L2 Performance Counter Events	16
Table 3: PERFORMANCE TUNING CHECKLIST	18
Table 4: Performance Comparison: Hardware versus Software CRC	42
Table 5: Performance Comparison: Hardware versus Software Hashing	45
Table 6: Spinlock versus SSO Packet-Linked Locking.....	54

LIST OF FIGURES

Figure 1: Physical Test Configuration.....	6
Figure 2: Logical Test Configuration	7
Figure 3: Start with One Core or Minimum Set of Cores.....	8
Figure 4: Scaling up to Full Number of Cores	9
Figure 5: Isolate the Function which is causing the Bottleneck	10
Figure 6: Hardware Architecture Overview	21
Figure 7: Bits 7-17 in Address Used to Index into L2 cache.....	22
Figure 8: Limited L2 Cache Index Choices When Data is Aligned on 2K Boundary	23
Figure 9: Align Data on Natural Boundaries.....	27
Figure 10: Pad Structures as Needed for Alignment	28
Figure 11: Packed Data Structures (Space versus Speed Tradeoff)	29
Figure 12: Re-arrange the Fields in the Data Structure to Save Space.....	29
Figure 13: Large Data Structure Alignment	30
Figure 14: Group Common Data Together.....	31
Figure 15: Prefetch Choices.....	35
Figure 16: Don't Write Back (DWB) Commands from the Core	38
Figure 17: DWB Commands sent by IOB on Behalf of other Hardware Units	40
Figure 18: Graph of Hardware Versus Software CRC – 750 MHz Processor	43
Figure 19: Cache Miss When Doing Bit-field Writes	48
Figure 20: Read after Write Performance Penalty	50
Figure 21: Arena Memory Allocation Reduces Contention	55

1 Introduction

This document describes common areas where changes can bring big performance improvements. Many of the performance improvement techniques presented in this document are industry-standard; others take advantage of Cavium Networks-specific hardware acceleration.

This document addresses both designing for high performance, and post-development performance tuning. Both single core and multiple-core (scaling) issues are addressed.

Performance tuning issues are separated into four sections:

1. Software Architecture for High Performance
2. Tuning the Minimum Set of Cores
3. Tuning Multi-core Applications (Scaling)
4. Linux-Specific Tuning

Within each section, performance tuning choices are presented from the easiest to most difficult to implement.

Information is also provided on performance evaluation tools.

Performance tuning is both an art and a science. This document does not attempt to cover all the possibilities, only some of the more common ones.

For OCTEON white papers on other topics, please contact your Cavium Networks representative.

1.1 References

The following references provide additional information:

- Definition of spinlock:
 - http://en.wikipedia.org/wiki/Spin_lock
- Concurrent programming:
 - Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. ISBN: 0-13-711821
 - Butenhof, David. *Programming With Posix Threads*. ISBN: 0201633922
- *OCTEON Hardware Reference Manual (HRM)*
- *OCTEON Programmer's Guide*
- The documentation set provided with the OCTEON processor Software Development Kit (SDK)
- Hardware Simulator: See the SDK documentation
- Viewzilla: See the SDK documentation
- Perozilla: See the SDK documentation
- Oprofile: See the SDK Documentation and
 - Oprofile Website - <http://oprofile.sourceforge.net/>
 - Oprofile Manual - <http://oprofile.sourceforge.net/doc/index.html>

- Profile-Feedback Optimization – See the SDK Documentation
- Oct-Profile – See the SDK Documentation

2 Performance Tuning Overview

Knowing where the different units are located relative to the buses is useful in avoiding and detecting bottlenecks. Some performance changes can improve performance in one area, while creating a problem elsewhere. It is important to understand the overall system hardware architecture before making changes. After making changes, verify that the performance change actually improved performance.

If possible, design and code for high performance. After the application is running and debugged, then use this technique to tune it for the highest possible performance:

1. Set up a performance-tuning test bed.
2. Reduce the number of cores to the smallest set: only one if at all possible.
3. Tune the system for the highest performance with the smallest set.
4. Scale up the system. The resultant performance should be the minimum set performance times the scaling factor: linear improvement. If not, then performance issues relative to scaling need to be addressed.

Stay alert to the possibility that, for your application, a performance tuning change may worsen performance because of application-specific issues.

Note: If possible, make changes one at a time, checking the performance before and after the change to verify that the change improved performance.

2.1 Test Setup

Typically, performance testing is done by using a third-party Traffic Generator connected to the OCTEON processor. The Traffic Generator can transmit/receive at different speeds, and report the actual measured speed which results when OCTEON is “in the middle” (as shown in the logical view of the test bed, below).

Figure 1: Physical Test Configuration

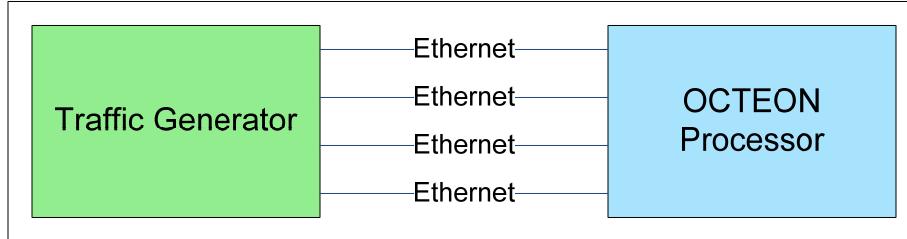
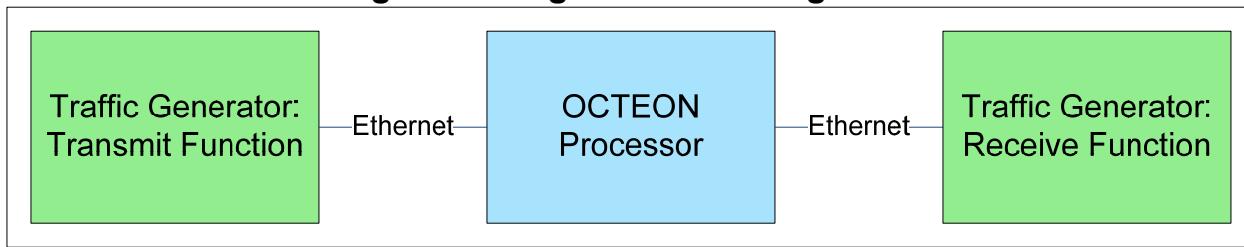


Figure 2: Logical Test Configuration

2.2 Evaluating whether Performance Tuning is Appropriate

The first step in testing is to hook up the test bed and measure actual performance. If the actual performance is close to the desired performance, the performance tuning will be sufficient to close the gap. For instance if you want 10 Gigabit and have 8 Gigabit of throughput, then try performance tuning. If the performance is much farther off target, then the design should be re-evaluated: performance tuning will not be enough to reach the goal.

2.3 Start with the Minimum Set of Cores, then Scale Up

When tuning performance, first reduce the test to the smallest possible set of cores. After the performance is maximized on the smaller number of cores, then add cores. At this point, check for scaling issues.

In scaling, more cores are added with the expectation that the single core performance numbers (N) will be multiplied by the number of added cores (X), resulting in a speed of N times X instead of N (a linear improvement). If adding more cores does not result in the expected improvement, the problem is one specific to a scaling issue.

Figure 3: Start with One Core or Minimum Set of Cores

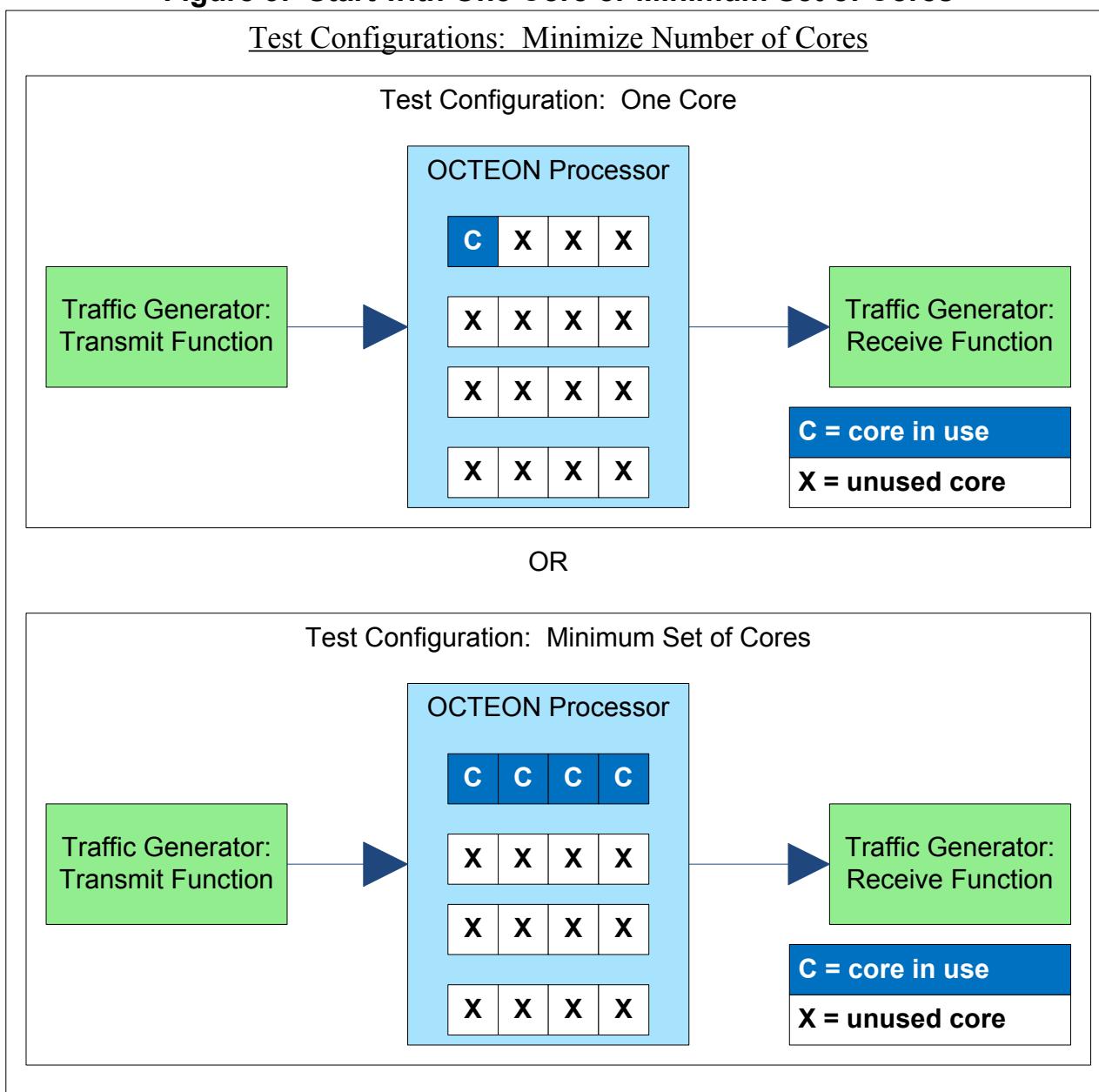
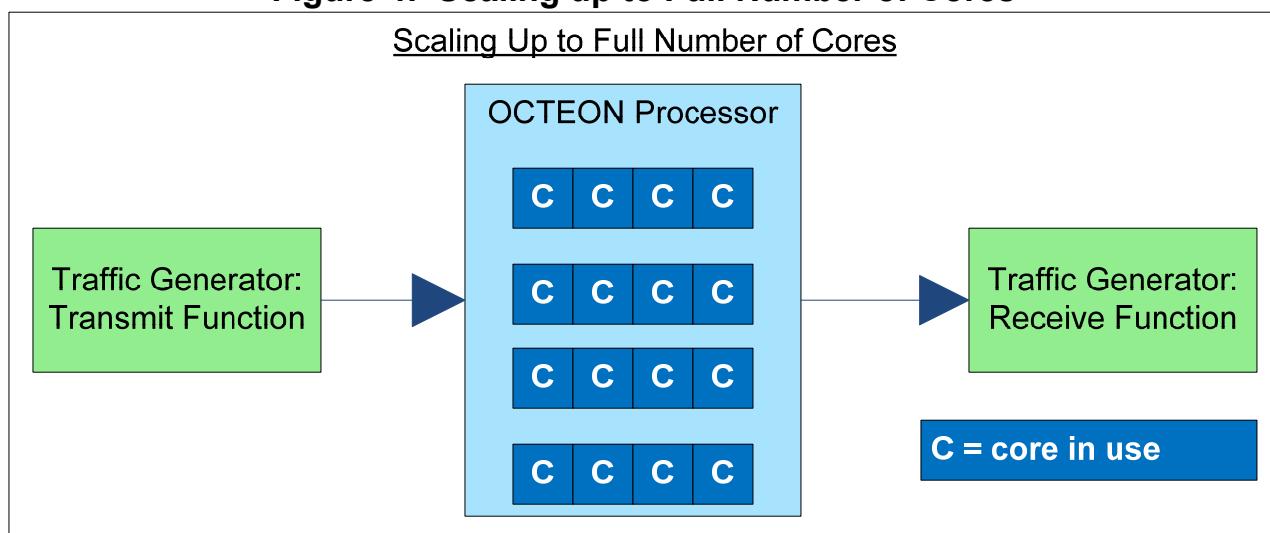


Figure 4: Scaling up to Full Number of Cores

2.4 Locate the Bottleneck in the Code

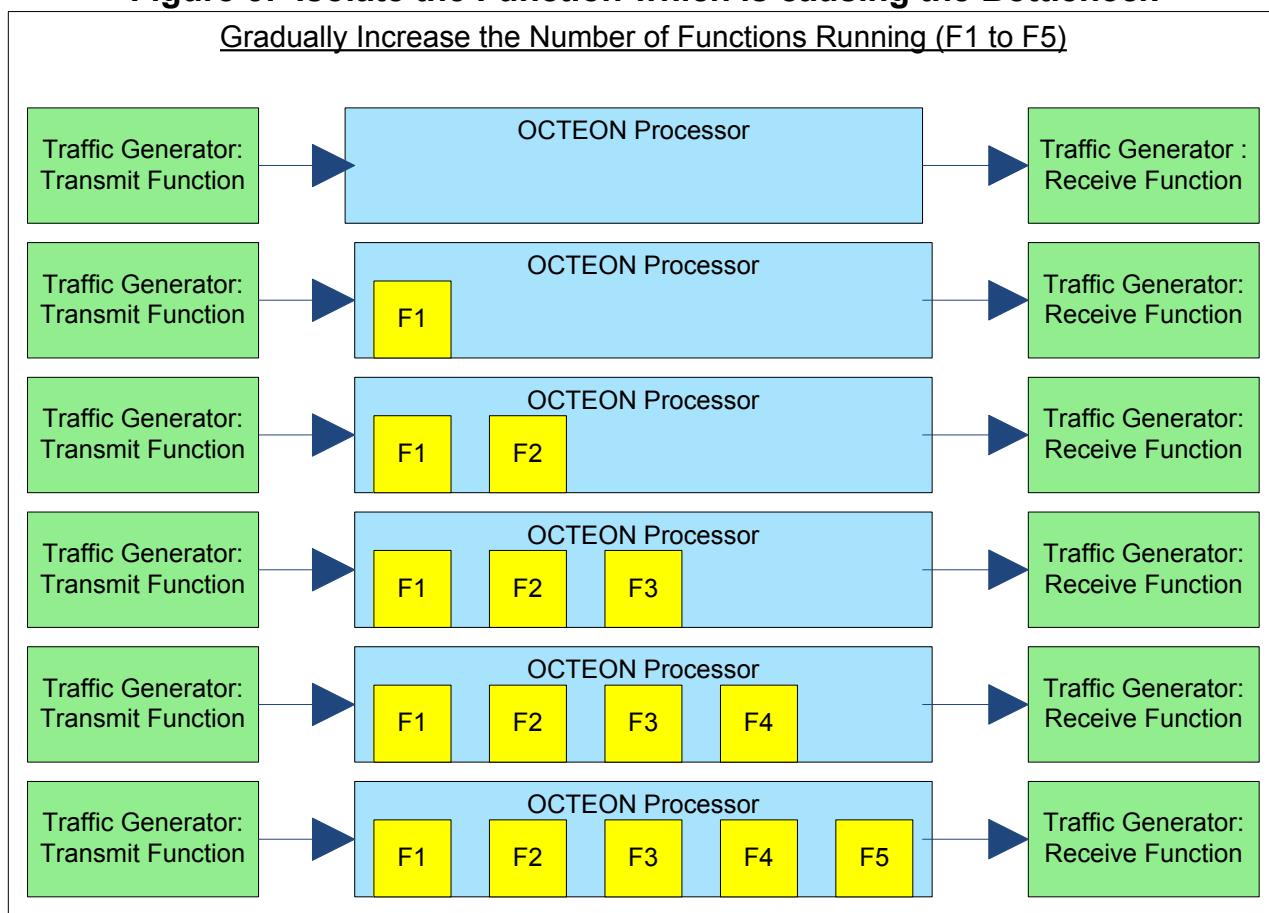
If performance tuning is needed, then the goal is to isolate the function containing the bottleneck.

For example: If the code consists of a loop where work comes into the software, goes through functions F1, F2, F3, F4, and F5, and then begins the loop with new data:

1. Measure performance with all five functions.
2. Measure performance with no functions.
3. Then add the functions back one at a time, measuring performance at each step.

Using this technique, it should be possible to pick out one or two functions with problems. The other functions can be ignored.

Figure 5: Isolate the Function which is causing the Bottleneck



2.5 Performance Testing Tools

The following performance testing tools are available.

2.5.1 Performance Tools on Simple Executive

The following tools are helpful in tuning performance on Simple Executive:

- Hardware Simulator : The OCTEON hardware simulator is provided with the SDK. See the SDK documentation for more information.
- PCI Profiling: For PCI profiling, use the `oct-profile` utility. This utility provides real-time profiling of a running application over PCI. More information is provided in the SDK documentation.
- Perfzilla: This utility is a graphical interface wrapper for the `oct-profile` results. It can also be run on the simulator output. More information is provided in the SDK documentation.
- Profile-Feedback Optimization: This is used to guide the compiler in making optimization choices. *Profile-feedback optimization* also known as *Feedback-Driven Optimization (FDO)* can significantly improve the compiler's decision on what optimizations would be beneficial in which parts of the program. More information is provided in the SDK documentation.

- Viewzilla: This program processes the simulator's output. It provides profiling information. More information is provided in the SDK documentation.

2.5.2 Performance Tools on Linux

The following tools are helpful in tuning performance on Linux:

- Hardware Simulator: Simulates the hardware. Useful in finding the number of TLB Exceptions.
- oprofile utility: This utility uses the internal core counters to provide a wide range of performance statistics. Counters may be used to statistically find ICACHE and DCACHE misses, branch misses, unaligned memory accesses, as well as the more common cycle counts.
- top utility: This utility provides updates in real-time to show system versus user time. The most important information is the percent of CPU (core) utilization. This tool can also show if processes or threads are bound to particular core. Note that there is big performance impact if threads/processes are not bound to specific cores.
- time utility: This utility shows amount of time spent in different parts: real time, user time, system time.
- prof utility: The commonly-used profile utility.
- gprof utility: The GNU version of the commonly-used profile utility.

2.5.3 A Note about the Simulator and Viewzilla

Viewzilla is the best tool to analyze the code and optimize the performance, but this tool only runs along with the simulator (on the output of the simulator).

When writing code for the Simple Executive, the programmer should try to make sure the code also runs on the simulator throughout the entire development process. It is common to abandon the simulator when the actual hardware becomes available. Then, when performance testing becomes important near the end of the project, the code may no longer work on the simulator. This means the key performance analysis tool, Viewzilla, will not work.

There are a couple of reasons why code will not run on the simulator. Some of them are simple, while the others may be more complicated.

First, make sure the software can be configured to a minimal configuration. When doing performance testing with the simulator, use a minimal setup (limited number of ports, etc), which can pass packets. This minimal setup can then be analyzed with the simulator and Viewzilla. The most common issue is a hardware setup which is difficult to replicate under the simulator (more ports, PCI etc). The simulator does not support PCI. To simulate PCI, modify the code to mimic PCI messages.

If PCI is used for initialization, provide some hard-coded initialization values which could be used under the simulator instead. This solution is very application specific.

2.6 Instrumenting the Code and Using Performance Counters

The code may be instrumented to use per-core (CP0) or L2 counters to help with performance testing. Performance counters, such as DRAM Controller registers, may also be used to detect bottlenecks.

Note: Remove the instrumentation before final test and ship, since the instrumentation hurts performance.

Note that tools such as Oprofile, Simulator, and Viewzilla use/interpret the performance counters. This may be more convenient to use than instrumenting the code.

2.6.1 Cycle Counters

Use the Simple Executive function `cvmx_get_cycle()` to record the cycle count at various points in the code. The programmer can define an array of these cycle values, and during can collect the cycle count at various stages of packet processing.

These values can be collected over a large set of packets (accumulated or averaged values), and the results can be dumped once a specific criteria is met (for example, dump the cycles after one million packets).

The function `cvmx_get_cycle()` reads the cycle count from hardware register 31. Note that the full 64-bit CvmCount value is provided when 64-bit operations are enabled, and only the lower 32 bits of CvmCount are provided (sign-extended) when 64-bit operations are enabled.

In the SDK, the passthrough example contains `cvmx_get_cycle()` function calls.

2.6.2 CP0 Performance Counters

There are two 64-bit CP0 performance counters per core that can simultaneously count events. To select the event to count, write to CP0 Register 25, select 0 or 2. To read, select 1 or 3 in the same register (see code sample, below). The list of selectable events, as shown in Table 1 – “CP0 Performance Counter Events” and Table 2 – “L2 Performance Counter Events”, below.

There are four 36-bit L2 performance counters that can simultaneously count events. Each counter’s even is selected via the corresponding CNTxSEL field. The list of selectable events is shown in Table 1 – “CP0 Performance Counter Events” and Table 2 – “L2 Performance Counter Events”.

The CP0 performance counters are most commonly used.

Performance counters can be used to locate areas where performance improvement is needed. For example to determine the Icache or Dcache miss for each core, examine the `PERF_CNT_CIMISS` and `PERF_CNT_DMLDS` events.

The most useful CP0 events are `PERF_CNT_CLK`, `PERF_CNT_CIMISS`, `PERF_CNT_UULOAD`, `PERF_CNT_UUSTORE`, and `PERF_CNT_DMLDS`.

2.6.2.1 Sample Code: Enable and Read CP0 Counters

The following macros can be used to configure and read the performance counters:

```
/* write to COP0, register 25, select 0 (perf 0) */
#define CVMX_MT_PERF0 (val)    asm volatile ("mtc0 %[rt],$25,0" : : [rt] "d" (val))

/* write to COP0, register 25, select 2 (perf 1) */
#define CVMX_MT_PERF1 (val)    asm volatile ("mtc0 %[rt],$25,2" : : [rt] "d" (val))

/* read from COP0, register 25, select 1 (perf 0) */
#define CVMX_MF_PERF0 (val)    asm volatile ("dmfc0 %[rt],$25,1" : [rt] "=d" (val):)

/* read from COP0, register 25, select 3 (perf 1) */
#define CVMX_MF_PERF1 (val)    asm volatile ("dmfc0 %[rt],$25,3" : [rt] "=d" (val):)

/* setup perf control word */
#define PERF_WORD(exl,k,s,u,ie,event,w,m) ( (((uint64_t)exl) << 0) | \
                                         (((uint64_t)k) << 1) | \
                                         (((uint64_t)s) << 2) | \
                                         (((uint64_t)u) << 3) | \
                                         (((uint64_t)ie) << 4) | \
                                         (((uint64_t)(event & 0x3f)) << 5) | \
                                         (((uint64_t)w) << 30) | \
                                         (((uint64_t)m) << 31) )
```

Setup the performance counters to be monitored (once only; should be setup by each core):

```
/* perf control */
uint64_t perf_config;

perf_config = 0;
perf_config = PERF_WORD(1,1,1,1,0,29,1,0); /* 29 is CIMISS */
CVMX_MT_PERF0 ((uint32_t)perf_config);

perf_config = 0;
perf_config = PERF_WORD(0,1,1,1,0,46,1,0); /* 46 is DMLDS */
CVMX_MT_PERF1 ((uint32_t)perf_config);
```

Now each core can read its performance counter. Note that these counters will provide the accumulated values from the start of the code execution. You have to maintain the deltas in the code.

```
uint64_t perf_val1 = 0;
uint64_t perf_val2 = 0;

CVMX_MF_PERF0 (perf_val1); /* using the macro to read the register */
CVMX_MF_PERF1 (perf_val2); /* using the macro to read the register */
```

Table 1: CP0 Performance Counter Events

Num	Item	Meaning (<i>The most useful items are highlighted.</i>)
0	Reserved	
1	PERF_CNT_CLK	Conditionally clocked cycles (as opposed to count/cvm_count which counts even with no clocks)
2	PERF_CNT_ISSUE	Instructions issued but not retired
3	PERF_CNT_RET	Instructions retired (retired means you fetched the instruction and executed it)
4	PERF_CNT_NISSUE	Cycles no issue
5	PERF_CNT_SISSLUE	Cycles single issue
6	PERF_CNT_DISSUE	Cycles dual issue
7	PERF_CNT_IFI	Cycle ifetch issued (but not necessarily commit to pp_mem)
8	PERF_CNT_BR	Branches retired (equals branches taken - retired means you fetched the instruction and executed it)
9	PERF_CNT_BRMIS	Branch mispredicts (fetches but not taken)
10	PERF_CNT_J	Jumps retired (jumps taken)
11	PERF_CNT_JMIS	Jumps mispredicted (fetched but not taken)
12	PERF_CNT_REPLAY	Mem Replays
13	PERF_CNT_IUNA	Cycles idle due to unaligned_replays
14	PERF_CNT_TRAP	trap_6a signal
15	Reserved	
16	PERF_CNT_UUPLOAD	Unexpected unaligned loads (REPUN=1)
17	PERF_CNT_UUSTORE	Unexpected unaligned store (REPUN=1)
18	PERF_CNT_UPLOAD	Unaligned loads (REPUN=1 or USEUN=1)
19	PERF_CNT_USTORE	Unaligned store (REPUN=1 or USEUN=1)
20	PERF_CNT_EC	Exec clocks (must set CvmCtl[DISCE] for accurate timing)
21	PERF_CNT_MC	Mul clocks (must set CvmCtl[DISCE] for accurate timing)
22	PERF_CNT_CC	Crypto clocks (must set CvmCtl[DISCE] for accurate timing)
23	PERF_CNT_CSRC	Issue_csr clocks (must set CvmCtl[DISCE] for accurate timing)
24	PERF_CNT_CFETCH	Icache committed fetches (demand+prefetch)
25	PERF_CNT_CPREF	Icache committed prefetches
26	PERF_CNT_ICA	Icache aliases
27	PERF_CNT_II	Icache invalidates
28	PERF_CNT_IP	Icache parity error
29	PERF_CNT_CIMISS	Cycles idle due to imiss (must set CvmCtl[DISCE] for accurate timing)
30	Reserved	
31	Reserved	
32	PERF_CNT_WBUF	Number of Write Buffer entries created

Num	Item	Meaning (<i>The most useful items are highlighted.</i>)
33	PERF_CNT_WDAT	Number of Write Buffer data cycles used (may need to set CvmCtl[DISCE] for accurate counts)
34	PERF_CNT_WBUFLD	Number of Write Buffer entries forced out by loads
35	PERF_CNT_WBUFFL	Number of cycles that there was no available Write Buffer entry (may need to set CvmCtl[DISCE] and CvmMemCtl[MCLK] for accurate counts)
36	PERF_CNT_WBUFTR	Number of stores that found no available Write Buffer entries
37	PERF_CNT_BADD	Number of address bus cycles used (may need to set CvmCtl[DISCE] for accurate counts)
38	PERF_CNT_BADDL2	Number of address bus cycles not reflected (i.e. destined for L2) (may need to set CvmCtl[DISCE] for accurate counts)
39	PERF_CNT_BFILL	Number of fill bus cycles used (may need to set CvmCtl[DISCE] for accurate counts)
40	PERF_CNT_DDIDS	Number of Dstream DIDs created (trying to load Dcache)
41	PERF_CNT_IDIDS	Number of Istream DIDs created (trying to load Icache)
42	PERF_CNT_DIDNA	Number of cycles that no DIDs were available (may need to set CvmCtl[DISCE] and CvmMemCtl[MCLK] for accurate counts)
43	PERF_CNT_LDS	Number of load issues
44	PERF_CNT_LMLDS	Number of local memory load issues
45	PERF_CNT_IOLDS	Number of I/O load issues
46	PERF_CNT_DMLDS	Number of loads that were not prefetches and missed in the cache
47	Reserved	
48	PERF_CNT_STS	Number of store issues
49	PERF_CNT_LMSTS	Number of local memory store issues
50	PERF_CNT_IOSTS	Number of I/O store issues
51	PERF_CNT_IOBDMA	Number of IOBDMAs
52	Reserved	
53	PERF_CNT_DTLB	Number of dstream TLB refill, invalid, or modified exceptions
54	PERF_CNT_DTLBAD	Number of dstream TLB address errors
55	PERF_CNT_ITLB	Number of istream TLB refill, invalid, or address error exceptions
56	PERF_CNT_SYNC	Number of SYNC stall cycles (may need to set CvmCtl[DISCE] for accurate counts)
57	PERF_CNT_SYNCIOB	Number of SYNCIOBDMA stall cycles (may need to set CvmCtl[DISCE] for accurate counts)
58	PERF_CNT_SYNCW	Number of SYNCWs or SYNCWSs
59-63	Reserved	

See the *HRM* for more details on how to use the CP0 performance counter events.

2.6.3 L2 Cache Performance Counters

To detect whether the L2 cache is thrashing, look at the L2 performance counters. For example, at the instruction and data hit and miss counters.

L2 Performance Events

Num	Event
1	L2 Instruction Miss
2	L2 Instruction Hit
3	L2 Data Miss
4	L2 Data Hit
5	L2 Miss I/D - Sum of events 1 and 3
6	L2 Hit I/D - Sum of events 2 and 4

In the table below:

XMC = ADD bus
 XMD = STORE bus
 RSC = COMMIT bus
 RSD = FILL bus

Table 2: L2 Performance Counter Events

Num	Item (<i>The most useful items are highlighted.</i>)
0	Cycles
1	L2 Instruction Miss
2	L2 Instruction Hit
3	L2 Data Miss
4	L2 Data Hit
5	L2 Miss (I/D) (sum events 1 and 3)
6	L2 Hit (I/D) (sum events 2 and 4)
7	L2 Victim Buffer Hit (Retry Probe)
8	LFB-NQ Index Conflict
9	L2 Tag Probe (issued - could be VB-Retry)
10	L2 Tag Update (completed - note: some CMD types do not update)
11	L2 Tag Probe Completed (beyond VB-RTY window)
12	L2 Tag Dirty Victim
13	L2 Data Store NOP
14	L2 Data Store READ
15	L2 Data Store WRITE
16	Memory Fill Data valid (1 strobe/32B)
17	Memory Write Request
18	Memory Read Request
19	Memory Write Data valid (1 strobe/32B)

Num	Item (<i>The most useful items are highlighted.</i>)
20	XMC NOP (XMC Bus Idle)
21	XMC LDT (Load-Through Request)
22	XMC LDI (L2 Load I-Stream Request)
23	XMC LDD (L2 Load D-stream Request)
24	XMC STF (L2 Store Full cacheline Request)
25	XMC STT (L2 Store Through Request)
26	XMC STP (L2 Store Partial Request)
27	XMC STC (L2 Store Conditional Request)
28	XMC DWB (L2 Don't WriteBack Request)
29	XMC PL2 (L2 Prefetch Request)
30	XMC PSL1 (L1 Prefetch Request)
31	XMC IOBLD
32	XMC IOBST
33	XMC IOBDMA
34	XMC IOBRSP
35	XMD Bus valid (all)
36	XMD Bus valid (DST=L2C) Memory Data
37	XMD Bus valid (DST=IOB) REFL Data
38	XMD Bus valid (DST=PP) IOBRSP Data
39	RSC NOP
40	RSC STDN
41	RSC FILL
42	RSC REFL
43	RSC STIN
44	RSC SCIN
45	RSC SCFL
46	RSC SCDN
47	RSD Data Valid
48	RSD Data Valid (FILL)
49	RSD Data Valid (STRSP)
50	RSD Data Valid (REFL)
51	LRF-REQ (LFB-NQ)
52	DT RD-ALLOC (LDD/PSL1 Commands)
53	DT WR-INVAL (ST* Commands)

See the *HRM* for more details on how to use the L2 performance counter events.

2.6.4 DRAM Utilization Information

The DRAM Controller (LMC) performance registers can be used to check for bottlenecks in accesses to DRAM.

For instance, to calculate DRAM utilization, use `LMC_OPS_CNT_HI`, `LMC_OPS_CNT_LO`, `LMC_DCLK_CNT_HI`, and `LMC_DCLK_CNT_LO` registers. (The “HI” and “LO” are the upper and lower bits of the 64-bit performance counter.):

$$\text{Bus utilization} = (\text{LMC_OPS_CNT_HI}, \text{LMC_OPS_CNT_LO}) / (\text{LMC_DCLK_CNT_HI}, \text{LMC_DCLK_CNT_LO})$$

If DRAM utilization is 70% or more, then the DRAM interface is very busy.

See the *HRM* for more details on how to use the LMC performance counter registers.

2.7 When Is Performance Optimization Complete?

Performance optimization is complete when you satisfy your system requirements.

Note that if you measure the Instructions Per Cycle (IPC). For cnMIPS cores, the maximum Instructions Per Cycle is two (dual issue pipeline). Given pipeline stalls, the average will be less:

- IPC = 1 (one) is good
- IPC > 1 (one) is very good
- IPC approximately 2 (two) is excellent

3 Performance Tuning Checklist

The following checklist can be used to select and prioritize the performance improvements you make.

Table 3: PERFORMANCE TUNING CHECKLIST

Linux, Simple Exec, Both, or Architecture	Hardware Assist	Item to Check (Ordered in the same order presented in this document. Within each section, the items are ordered from easiest to hardest to implement.)	Possible Improvement (Big, Medium, Small)
Software Architecture for High Performance			
Simple Exec	X	L2 Cache Configuration: Aliased Cache Indexing	Medium (See Note 1)
Both	X	Configuring the Right Amount of Packet Data Buffers and WQE Buffers	Big
Architecture		Choosing Simple Executive versus Linux or other OS	Big
Both		Concurrent Programming Techniques	Medium
Architecture		Pipelined versus Run-to-Completion Software Architecture	Big
Architecture		Event-driven loop versus Interrupt Handling for Packet Processing	Big
Tuning the Minimum Set of Cores			
Both		Compiler Choice	Medium
Both		Compiler Optimization (-O3)	Medium

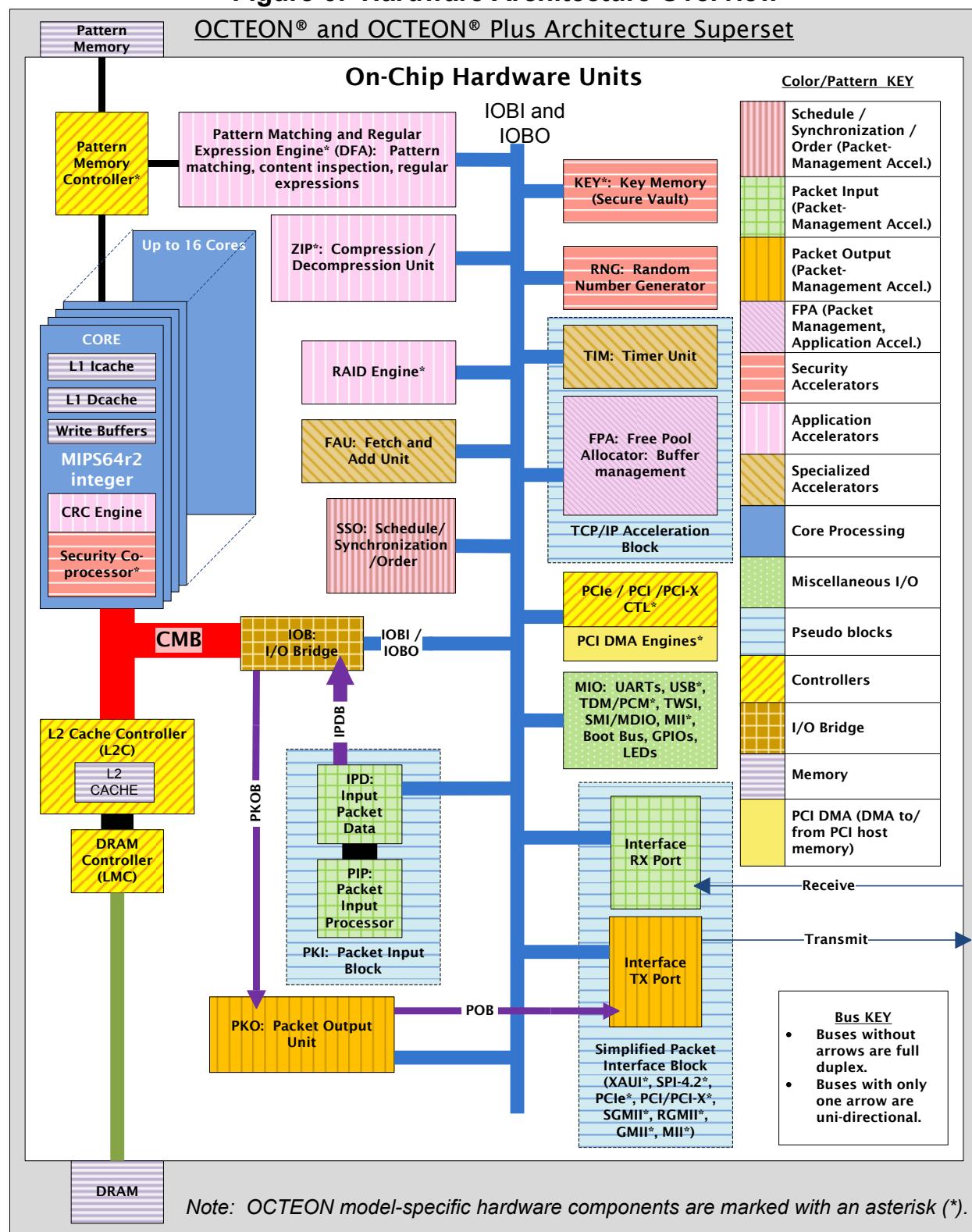
Linux, Simple Exec, Both, or Architecture	Hardware Assist	Item to Check (Ordered in the same order presented in this document. Within each section, the items are ordered from easiest to hardest to implement.)	Possible Improvement (Big, Medium, Small)
Both	X	Re-Configuring the Right Amount of Packet Data Buffers and WQE Buffers	Medium
Both	X	Memory Alignment	Medium
Both		Data Structure Compaction (packing), Re-arranging Structures	Big
Both		Large Data Structures: Working with Cache-line Size	Big
Both		Group Common Data Together	Medium
Both		Loop Unrolling	Medium/Small
Both		Replace memset() and memcpy() When Needed	Medium
Simple Exec	X	Use Free Pool Allocator (FPA) Memory Pools to Manage Free Buffers	Small
Both	X	Cache Prefetch	Big (See Note 1)
Both	X	Prepare for Store	Big
Simple Exec	X	Scratchpad: Core-local Storage	Small
Simple Exec	X	Asynchronous FPA Allocation	Medium
Both	X	Don't Write Back (DWB) Commands	Small (See Note 1)
Both	X	Hardware CRC Engine	Medium
Both	X	Hardware Hash Engine	Medium
Simple Exec	X	Hardware Timers	Medium
Simple Exec	X	Hardware Fetch and Add (FAU) Unit	Medium
Simple Exec	X	Asynchronous Fetch and Add Operations	Medium
Both	X	Work prefetch: Asynchronous get_work	Medium
Both	X	Interleaving Prefetch with Computational Instructions	Small
Both	X	Hardware TCP/UDP Checksum Calculation	Medium
Both		Use Functions Wisely	Medium
Both		Update Bit-fields Wisely	Medium
Both		Read after Write	Medium
Tuning Multi-core Applications (Scaling)			
Both	X	Re-Configuring the Right Amount of Packet Data Buffers and WQE Buffers	Medium
Simple Exec	X	Tune Initial Tag Values to Separate Flows	Small
Simple Exec	X	Set Initial Tag Type to ORDERED if Possible	Medium
Simple Exec	X	Switch Tag Type to ORDERED or NULL when Possible	Medium
Simple Exec	X	Use Asynchronous Tag Switch Operations	Small

Linux, Simple Exec, Both, or Architecture	Hardware Assist	Item to Check (Ordered in the same order presented in this document. Within each section, the items are ordered from easiest to hardest to implement.)	Possible Improvement (Big, Medium, Small)
Both		Minimize Critical Regions	Medium
Simple Exec	X	Replace Spinlocks with ATOMIC Tag Types when Possible	Medium
Both		Arena-based Memory Allocation	Medium
Both	X	L2 Cache Configuration: Way Partitioning and Cache-Block Locking	Medium (See Note 1)
Linux-specific Tuning			
Linux	X	TLB Exceptions and Huge Page Size	Big (See Note 1)
Linux		Use CPU Affinity for Processes/Threads	Big
Linux		Direct all Packet RX Interrupts to the Same Core	Small
<i>Note 1: This may have a large positive or a large negative effect, depending on the application/code.</i>			

4 Hardware Architecture Overview

Knowing where the different units are located relative to the buses is useful in avoiding and detecting bottlenecks. Some performance changes can improve performance in one area, while creating a problem elsewhere. It is important to understand the overall system architecture before making changes, and to be careful to verify that the performance change actually improved performance.

Figure 6: Hardware Architecture Overview



PERFORMANCE TUNING

5 Software Architecture for High Performance

When designing an application or tuning at the design level, the following items can affect performance. Note that it is best to design the application so it can be scaled up or down easily. This helps performance testing, and also allows the application to flexibly meet customer needs.

5.1 L2 Cache configuration: Aliased Cached Indexing

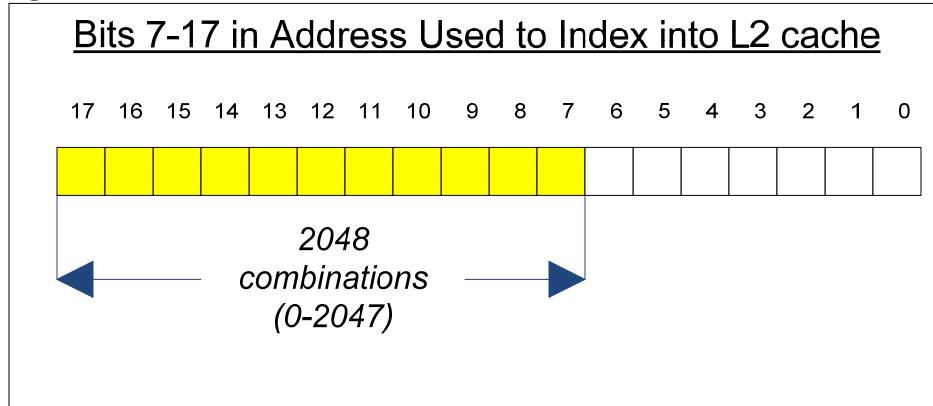
The high-end OCTEON L2 cache is 8-way set associative consisting of 2048 sets. There are two different set indexing algorithms. It is very simple to change the algorithm, so try both choices and see which performs best for your application.

5.1.1 Unaliased Caching Indexing Algorithm

Normally, a cache line is indexed with address bits 7 to 17 (shown as 17:7). This index selects one of the 2048 sets:

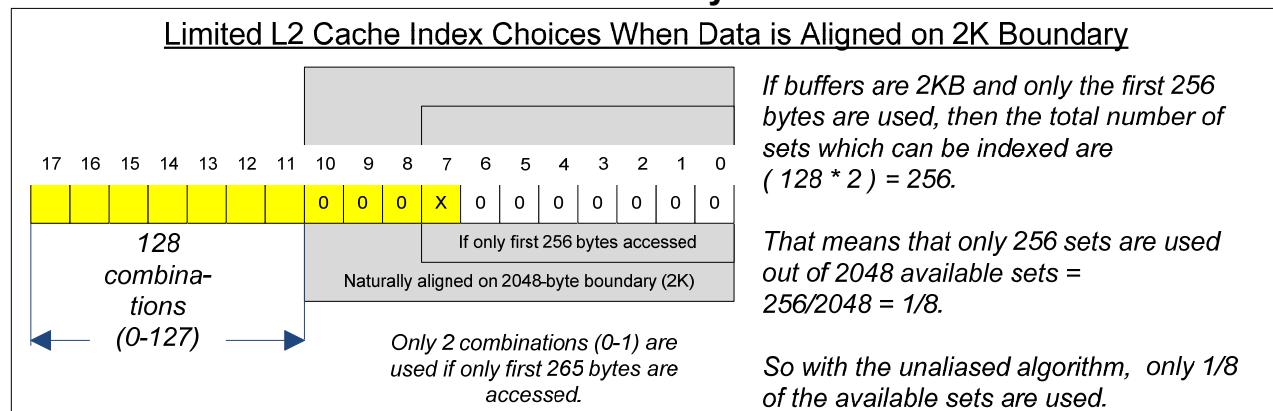
Index<10:0> = address<17:7>

Figure 7: Bits 7-17 in Address Used to Index into L2 cache



This unaliased algorithm may not be ideal for packet processing. For example, if the Packet Data Buffers are 2 KB and always naturally-aligned, AND only the first 256 bytes are used, the unaliased algorithm can only use (2*128) sets out of 2048 (only 1 out of every 8 sets is used). Because fewer sets are used, it is more likely that a needed cache block will be evicted: the cache appears to be smaller than it really is.

Figure 8: Limited L2 Cache Index Choices When Data is Aligned on 2K Boundary



5.1.2 Aliased Cache Indexing Algorithm

When the index aliasing option is enabled, the cache line index is computed as the exclusive or of address bits <17:7> and <27:17>.

$$\text{Index}_{10:0} = (\text{address}_{17:7} \text{ XOR } \text{address}_{28:18})$$

The aliased algorithm can spread cache blocks across more sets. If the program is accessing the first part of many different 2KB blocks, then the aliased algorithm will provide a larger number of sets.

There is no rule to decide when aliasing improves system performance. The best way to evaluate it is by prototyping the intended algorithm and evaluating performance using both modes.

Configuration of the caching algorithm must be done prior to any block being loaded into the L2 cache. Given this restriction, the OCTEON evaluation board bootloader has an environment variable to control this called `disable_12_index_aliasing`. Setting this variable via the `setenv` command will disable setting the `L2C_CFG[IDXALIAS]` bit during the board initialization sequence. The code that sets this bit can be found in the function `init_ebt3000_dram()` in the `bootloader/u-boot/lib_mips/board.c` file located in the OCTEON SDK directory tree.

5.2 Configuring the Right Amount of Packet Data Buffers and WQE Buffers

It is important to configure in sufficient numbers of Packet Data Buffers and Work Queue Entry Buffers so that the system does not run out of this resource. If the system does not have enough of these buffers, the PIP/IPD will not be able to receive new packets.

Note: The SSO has a limited number of Work Descriptors (internal memory it uses to hold both in-flight and pending work). It uses these Work Descriptors to create Cached Input Queues inside the SSO (pending work). When there are no more available Work Descriptors, the SSO will put

new work in Overflow Input Queues located in DRAM. As internal space becomes available, work from the Overflow Input Queues are brought from DRAM into the Cached Input Queues. The process of bringing in work from the Overflow Input Queues slows down the average latency between the `get_work` request and the satisfaction of the request when work is returned to the core. The net result is a slightly reduced overall system throughput. There are many factors that affect the magnitude of the performance reduction. One sure way to avoid this issue is to allocate the number of Work Queue Entries and Packet Data Buffers to be less than or equal to the number of SSO Work Descriptors.

5.3 Simple Executive versus Linux or other OS

When designing the system, the Simple Executive should be used for data-plane applications. The Simple Executive is optimized for the highest possible performance. When using Linux, the following items add processing overhead:

- Exception and context switch
- Copy of data between user and kernel space
- TLB management
- Scheduling timer overhead

It is possible to run Linux on some cores and Simple Executive on others. Often, Linux is used for the control-plane and Simple Executive is used for data-plane.

Also note that Linux SMP scales performance up to a limited number of cores. Some designs run multiple Linux SMP instances: the cores are divided into two groups, each running a different instance of Linux SMP.

Other operating systems (OS) are also available. Ask your Cavium Networks representative for more information.

5.4 Concurrent Programming Techniques

Since the OCTEON processor has multiple cores, one should be aware of concurrent programming techniques. There are many books available on this subject. For engineers without extensive experience in this area, see the concurrent programming references provided in Section 1.1 – “References”. This section will provide a brief introduction a key issue in concurrent programming: shared data.

Data can be shared by:

- All threads in a process (global, static, C++ objects, core-local shared memory, or Cavium Networks core-shared memory)
- All processes on a core (core-local shared memory, or Cavium Networks core-shared memory)
- Processes on different cores (Cavium Networks core-shared memory)

The problem with unprotected shared data structures is data corruption from multiple reader/writers. The classic example is a read-modify-write where the function increments a counter:

1. Process1 reads the value (read=0)
2. Process2 does a quick read-modify-write (read=0, 0+1=1, write 1)
3. Process1 modifies its stored value and writes it back (0+1=1, write 1).
4. As a result, the value goes from 0 to 1 to 1 instead of 0 to 1 to 2. The counter is now incorrect.

If a linked list is being modified, then chaos may result, for instance when “next” pointers are corrupted.

5.4.1 Critical Regions and Locks

It is important to know whether data is shared when writing code to access it. If it is shared, then use a locking mechanism to protect it.

When possible, use Cavium Networks-specific locking mechanisms. This will be addressed in Section 7.6 – “Critical Regions”.

5.4.2 Minimize use of Shared Data

Eliminate global, static, and unnecessary shared C++ objects. They not only cause reentrancy problems, they also make debugging difficult because it is difficult to see which process/thread modified the data. It is better to pass information by reference (provide a pointer to the data) as an argument to functions accessing the data.

5.4.3 Minimize Critical Regions

The section of code which modifies a locked shared data structure is a critical region. Functions must use a locking mechanism before entering the critical region.

Make sure each critical region is as short as possible. Remove unnecessary code from inside the locked section. This will speed up processing because processes wanting the lock will not have an unneeded delay.

The purpose of any lock is data and data structure integrity. The lock only needs to be held while traversing the data structure, while changing the data structure in such a way that would disrupt traversal (the “next” pointer, for instance), or changing a particular data item (in which case only the item should be locked, not the whole structure).

The lock should only be held for the minimum time needed to do the few manipulations that affect the entire structure. For example, in a linked list insertion, all of the processing to set up the new node should be handled first. When everything is ready, lock the list, determine the right place to insert the new node, insert it, and unlock the list.

5.5 Pipelined versus Run-to-Completion Software Architecture

The OCTEON processor has no per-core instruction-size limitation. It is not necessary to use pipeline software architecture; however the OCTEON processor supports both pipeline and run-to-

completion software architectures. In pipeline architecture, each processor handles one function and the packet moves through the pipeline, changing processors as needed to pass through the series of functions. In run-to-completion architecture, one processor handles all the functions, and the packet stays on the processor as it moves through the series of functions.

If possible, designing your application to use run-to-completion will result in higher-performance code because time is not spent on unnecessary switches to other processors.

5.6 Event-driven Loop versus Interrupt Handling for Packet Processing

If you choose to implement your application on Linux, you choose to use interrupt-driven packet processing. If you choose to implement your application on Simple Executive, then polling is used for packet processing. Polling is a higher performance processing architecture.

6 Tuning the Minimum Set of Cores

After the application is running and debugged, reduce it to run on the smallest possible set of cores. If possible, run on only one core. Optimize the performance on this sub-set before scaling up the number of cores. This will help you isolate where the performance problem is located, because scaling issues related to performance are different than those related to single-core performance.

6.1 Compiler Choice

Use the OCTEON SDK compiler, which is supplied with the Software Development Kit. It makes use of the Cavium Networks-specific instruction set, which will greatly improve performance.

6.2 Compiler Optimization (-O3)

Turn on compiler optimization using the -O3 option to the compiler.

6.3 Re-Configuring the Right Amount of Packet Data Buffers and WQE Buffers

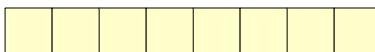
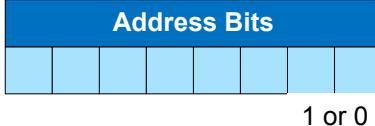
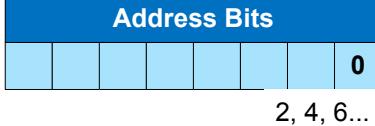
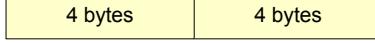
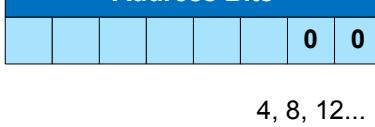
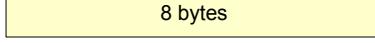
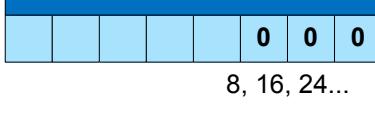
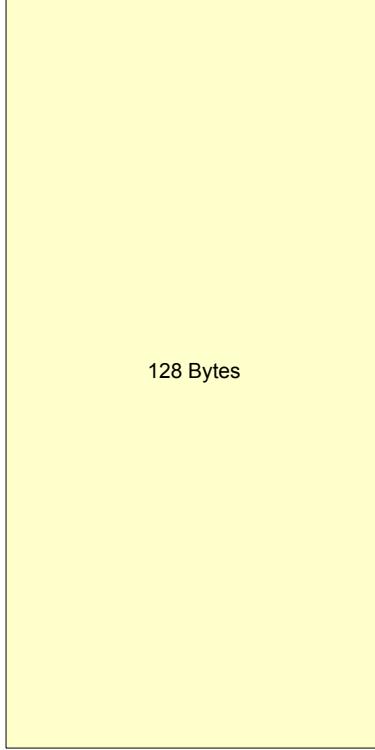
Review the system requirements and verify that sufficient numbers of Packet Data Buffers and Work Queue Entry Buffers have been configured into the Free Pool Allocator pools.

6.4 Memory Alignment

6.4.1 Align Data on Natural Address Boundaries

On RISC architectures, memory operations are most efficient when data is aligned on natural address boundaries. For instance, 4-byte types should reside on even 4-byte addresses.

Figure 9: Align Data on Natural Boundaries

Desired Boundary	Address Bits Which Must be 0
Align Character (8-bit data) on 1-Byte Boundary 	Address Bits  Alignment = 8 bits: 1-byte alignment (Address ends in 0 or 1, a multiple of 1.)
Align 16-bit data on 2- Byte Boundary 	Address Bits  Alignment = 16 bits: 2-byte alignment (Address is always even, a multiple of 2.).
Align 32-bit data on 4- Byte Boundary 	Address Bits  Alignment = 32 bits: 4-byte alignment (Address is a multiple of 4.) (32 bits / 8 bits per byte = 4- byte alignment)
Align 64-bit data on 8-Byte Boundary 	Address Bits  Alignment = 64 bits: 8-byte alignment (Address is a multiple of 8.) (64 bits / 8 bits per byte = 8-byte alignment)
Align 128-Byte data on 128-Byte Boundary  128 Bytes	Address Bits  Alignment = 128 bytes 128-bytes alignment (Address is a multiple of 32.) This is the same as the cache line size. ((128 bytes * 8 bits per byte) / 8 bits per byte = 128-byte alignment.)

6.4.2 Simple Executive Facilities to Support Memory Alignment

To support aligned memory allocation, Simple Executive has such facilities and can be used as follows:

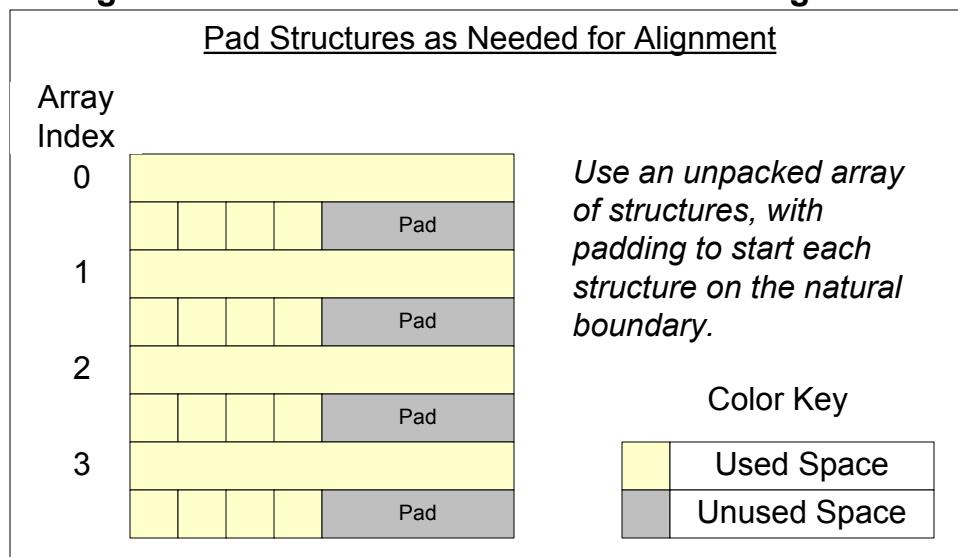
```
uint64_t *ptr;
uint64_t size = 256;
uint64_t alignment = 128;
// Allocate a 256-byte chunk of memory aligned on 128-byte boundary.
ptr = cvmx_bootmem_alloc (size, alignment);
```

This will allocate a 256-byte chunk of memory aligned to 128 bytes (cache line). Additionally, the physical address returned by this facility can be shared across all OCTEON cores.

6.4.3 Pad Structures to Align on Natural Boundaries

When creating arrays of structures, pad the structures so that each starts on a natural boundary:

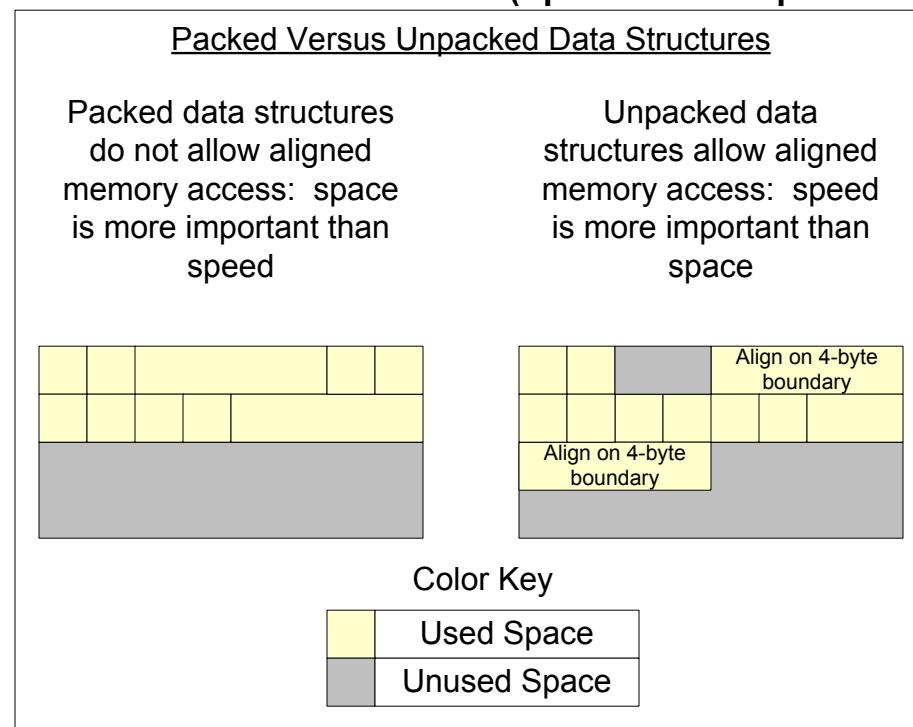
Figure 10: Pad Structures as Needed for Alignment



6.5 Data Structure Compaction (*packing*), Re-arranging Structures

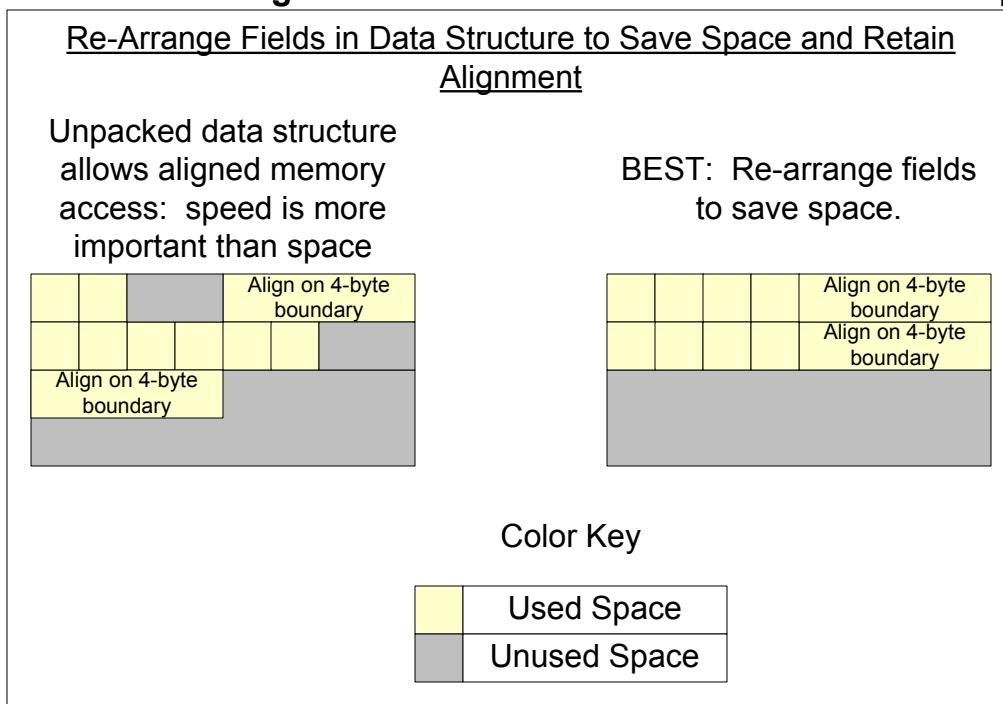
6.5.1 Packing: Space versus Speed Tradeoff

Because using natural memory alignment can speed performance, review whether packed data structures (which save space) are the best choice. There may be places where using unpacked data structures can improve performance and be worth the speed/space trade-off.

Figure 11: Packed Data Structures (Space versus Speed Tradeoff)

6.5.2 Re-Arrange Structure Fields to Save Space

Re-arranging elements in the structure may save wasted space by filling in the gaps with useful data instead of wasted gaps.

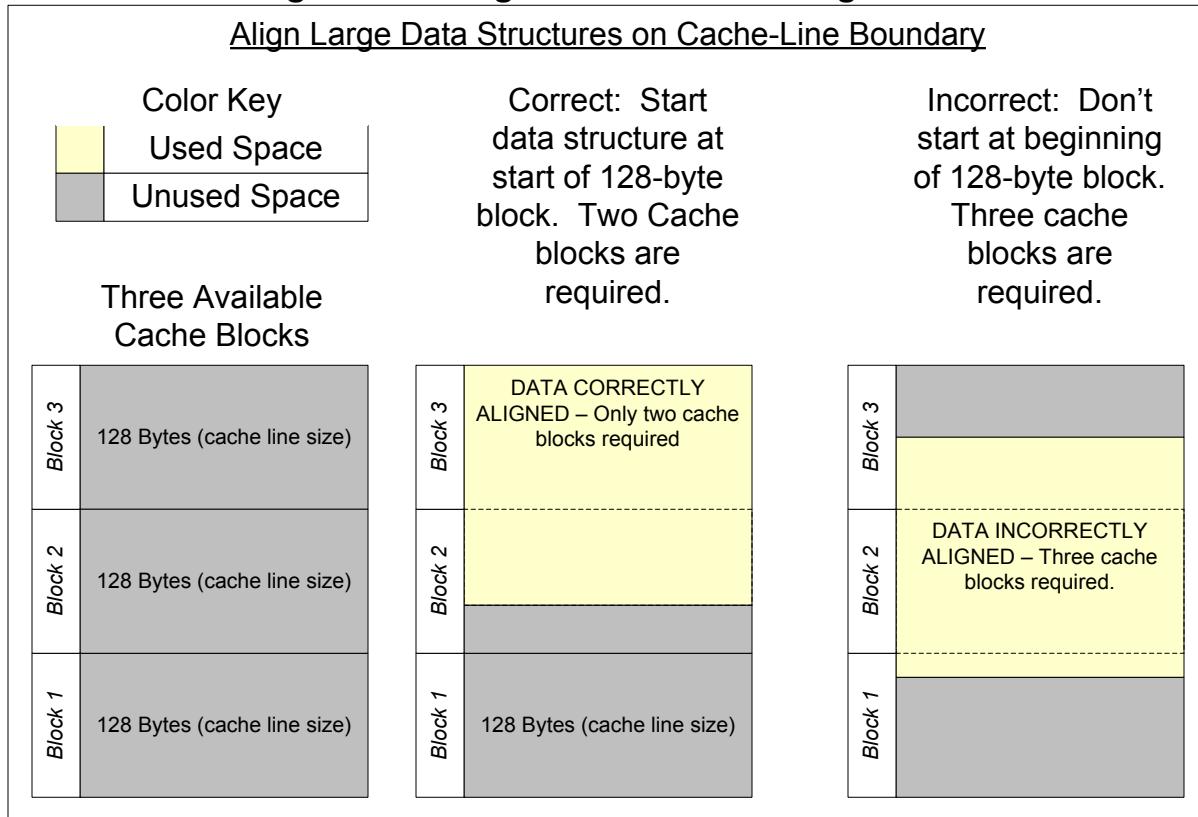
Figure 12: Re-arrange the Fields in the Data Structure to Save Space

6.6 Large Data Structures: Working with Cache-line Size

If the data being manipulated exceeds the cache line size (128 bytes), then make sure it is aligned to the start of the cache block. This will cause it to use the smallest possible number of cache blocks. The Simple Executive function which will allocate aligned memory is discussed in Section 6.4 – “Memory Alignment”.

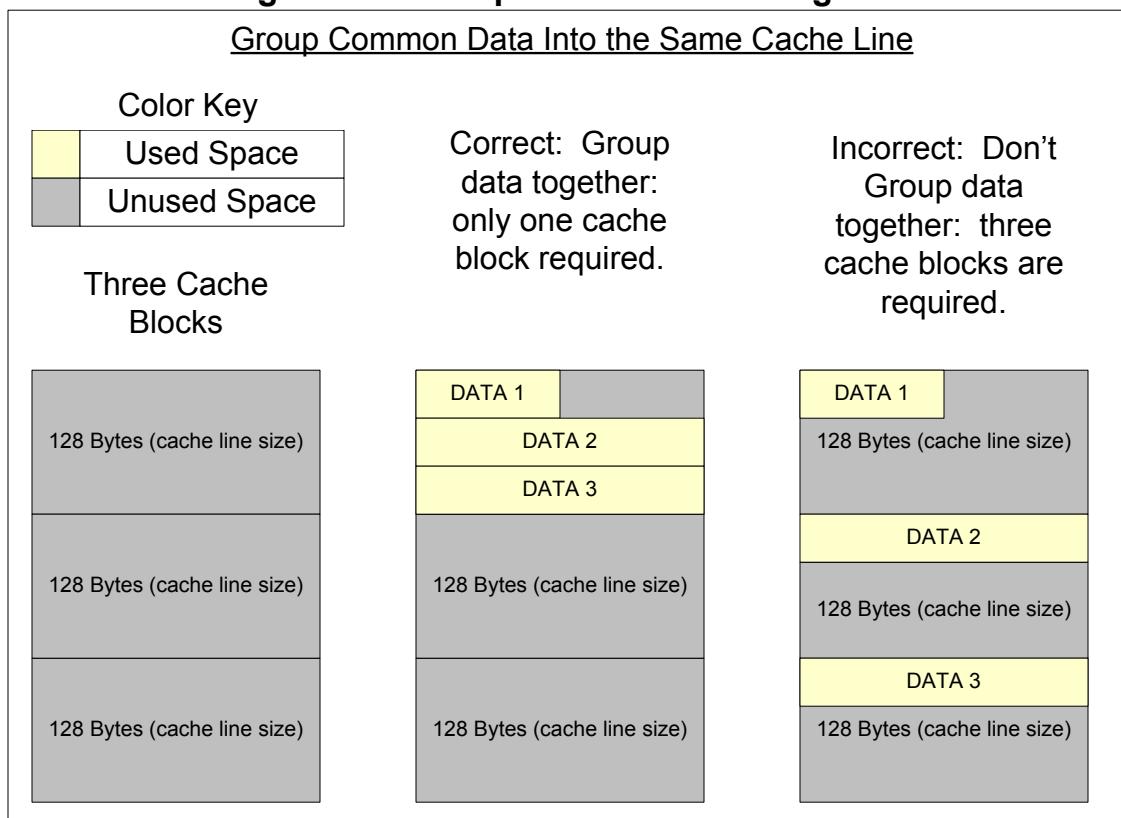
Also, if the data object is larger than one cache block, prefetch the entire data object prior to processing it. See Section 6.11 – “Cache Prefetch”.

Figure 13: Large Data Structure Alignment



6.7 Group Common Data Together

If the data is not in the cache when it is needed, the system will stall while waiting for it to be fetched into the cache. Group data objects which are needed together into the same area of memory, so a minimal number of cache blocks need to be fetched to get all the necessary data. Also, remember that the L1 cache size is limited: if the data is scattered over many cache blocks, the different fetches may cause eviction of still-needed data from the cache.

Figure 14: Group Common Data TogetherGroup Common Data Into the Same Cache Line

6.8 Loop Unrolling

This optimization is most useful in situations where a loop is used to perform multiple operations on arrays. Unrolling part of the loop to take advantage parallel operations can increase performance considerably.

There is also a compiler option, “`funroll-loops`”, that signals the GCC optimizer to automatically unroll loops in some cases.

6.8.1 Sample Code: Loop Unrolling

The loop:

```

char *src, *dst; // Here we use a character
int i;

// 32 operations, 8 bits at a time
for (i = 0; i < 32; i++) {
    dst[i] = src[i];
}

```

The above code fragment can be unrolled, as shown in the next code fragment:

```
uint64_t *src, *dst; // Note instead of char, the type is unit64_t

// 4 operations, 64 bits at a time
dst[0] = src[0];
dst[1] = src[1];
dst[2] = src[2];
dst[3] = src[3];
```

Not only does this remove the loop overhead, it also takes advantage of the CPU's natural 64-bit word size in that the load and store instructions utilize 64-bit operations per cycle. In the case of the char data types, the instructions will move only 8 bits per cycle.

A larger example of code which uses loop unrolling can be found in Section 6.16 – “Hardware CRC Engine”.

6.9 Replace `memset()` and `memcpy()` when Needed

When operating on large blocks of data which is aligned on a word boundary, `memset()` or `memcpy()` are not efficient. Both of these routines use byte-by-byte copy which does not take advantage of the 64-bit word size.

The following code is an example of code used to replace the `memset()` function. Similar code can be used to replace the `memcpy()` function. The exact code will depend on your data size. In this example, entire cache lines are written, and the data aligned on the 128-byte boundary (cache line size).

6.9.1 Sample Code: Replacing `memcpy()` and `memset()`

The following code avoids the use of `memcpy()` and `memset()`. It will zero out a cache line.

```
// p is assumed to be cache-line aligned and a valid pointer
// p must also be a multiple of cache line size (fill a whole cache line)
// anything else will cause memory corruption errors

static inline void buffer_init_fast(void *p, uint64_t num_cache_lines)
{
    uint64_t *ptr = (uint64_t *) p;

    while(num_cache_lines-- > 0) { // write one cache line at a time
        *ptr++ = 0x0L;
        *ptr++ = 0x0L;
```

```

        *ptr++ = 0x0L;
        *ptr++ = 0x0L;
        *ptr++ = 0x0L;
    };
}

```

The following code will simply write 64-bits at a time:

```

// p is assumed to be word-aligned and a valid pointer
// p is also assumed to be a multiple of word (fill a whole word)
// anything else will cause memory corruption errors
static inline void buffer_init_fast(void *p, uint64_t size)
{
    uint64_t num_words = (size >> 3);
    uint64_t *ptr = (uint64_t *) p;

    while(num_words-- > 0)
    {
        *ptr++ = 0x0L; // write 64-bits at a time
    };
}

```

6.10 Using Free Pool Allocator (FPA) Memory Pools to Manage Free Buffers

The Free Pool Allocator (FPA) provides a mechanism to pre-allocate and manage pools of free memory. These pools consist of chunks of memory, usually divided into equal-sized buffers. The memory is pre-allocated at system initialization. Because it is pre-allocated and is not fragmented (because all the buffers in a pool are usually the same size), it is more likely that a request for memory will succeed. The time to fill the request is also deterministic: there is no need for the system to search a linked list of free fragments of memory looking for one large enough to fill the request.

Core software may allocate and free these buffers. The buffers may also be asynchronously allocated which can speed processing and prevent the core from stalling while waiting for a buffer (see Section 6.14 – “Asynchronous FPA Allocation”).

6.11 Cache Prefetch

The action of loading data from DRAM into either L1 or L2 cache (fetch) takes a certain number of system cycles. To avoid stalling the core during the fetch delay, the Simple Executive API provides functions which prefetch data into the L1 cache, L2 cache, or both.

Note that more than one prefetch operation can occur at the same time.

There are two key reasons to use prefetch:

1. Keep the core busy: Software starts the prefetch, and then does other processing. This can improve performance because the core stays busy, and data is in L1 data cache when it is needed.
2. Optimize use of space in L2 cache: The L2 cache size is limited. The optimal use of the space is for data needed by multiple cores. A prefetch can bypass the L2 cache, and bring

the data directly into the L1 cache. Saving space in L2 can also prevent eviction of cache blocks which are still needed. Software needing the evicted cache block will stall because of the cache miss and the re-fetch. This also adds more work to the bus. Cache misses cause a large performance penalty.

There are two key things to consider before adding prefetches to the code:

1. Don't over-use prefetch: Over-using prefetch can either fail to improve or hurt system performance. This is because L1 cache size is limited. Prefetched data may evict other prefetched data needed sooner, causing extra work and stalling the core while it waits for the data. For instance, if data A, B, and C are prefetched, and needed in that order, C may evict B from the L1 cache. When the software needs B, it is not available.
2. Add prefetch to the correct location in the code: The best place to put a prefetch in the code depends on the structure of the code. After starting a prefetch, the code needs to have something to do until the prefetch completes. It is best to start the prefetch as early as possible to ensure the data is in the cache when needed.

The best solution is to prototype without prefetching, and then run tests to determine the code locations which will most benefit from prefetch.

Two situations where prefetch can be particularly useful are:

1. If the data object is larger than one cache block, prefetch the entire data object prior to processing it.
2. When operating on elements in a linked list, prefetch the next element before it is needed.

The Simple Executive API provides three different prefetch modes:

1. `CVMX_PREFETCH(address, offset)` - The block will be read into the L1 cache and the L2 cache.
2. `CVMX_PREFETCH_NOTL2(address, offset)` - The block will be read into the L1 cache, but will not be put into the L2 cache.
3. `CVMX_PREFETCH_L2(address, offset)` - The block will be read into the L2 cache without putting it into the L1 cache.

The hardware instruction is `PREF` (prefetch).

6.11.1 Sample Pseudo code: Prefetch

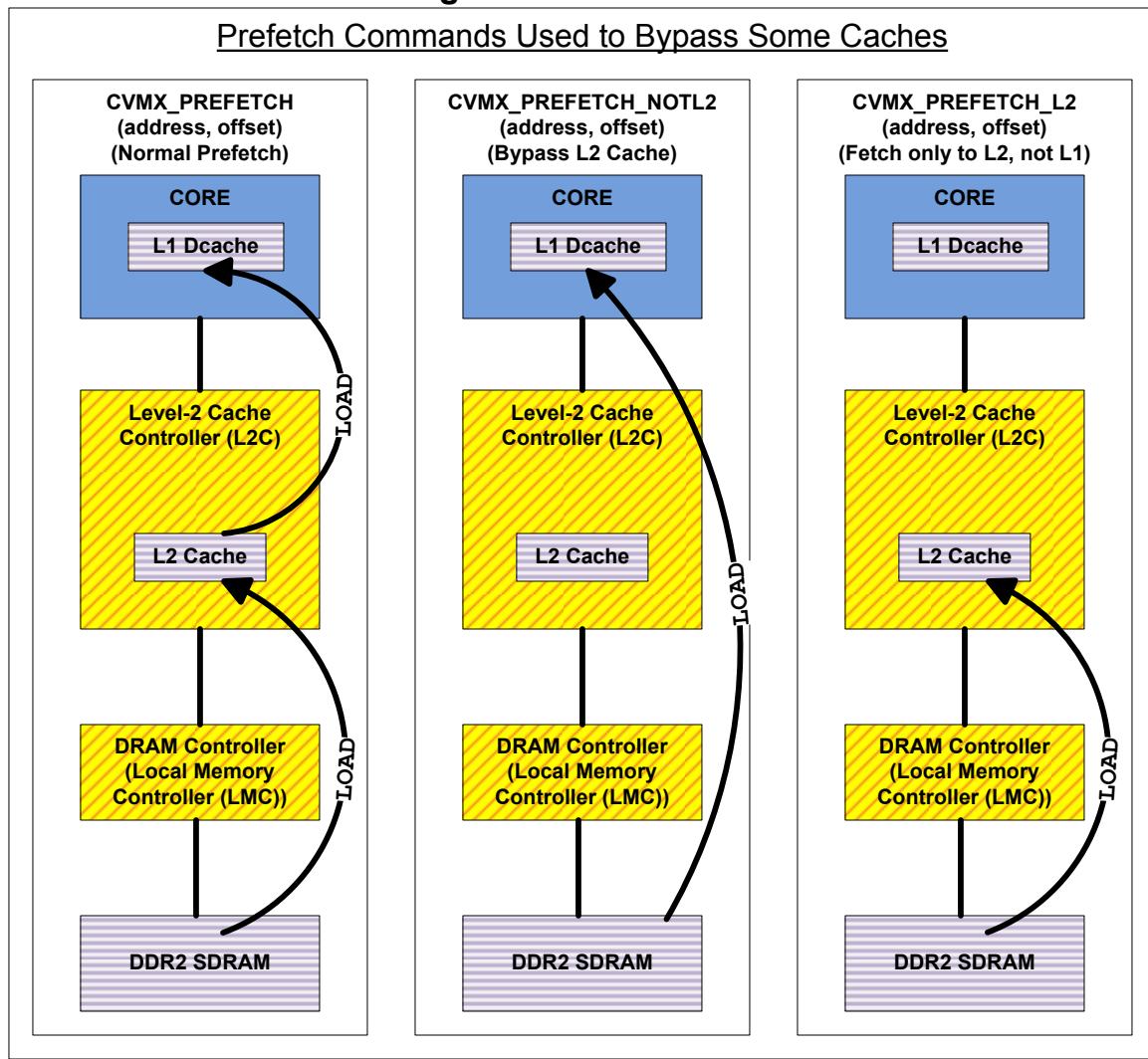
The following pseudo code illustrates prefetching the next element in a linked list before it is needed. In this example, the “next” record is prefetched into L1, bypassing L2.

```

linked_list_t *this = list_to_search;
while (this) {
    CVMX_PREFETCH_NOTL2(this->next, 0);
    if (process_current_record (this) == DONE) {
        break;
    }
    this = this->next;
}

```

A larger example using prefetch can be found in the code in Section 6.16 – “Hardware CRC Engine”.

Figure 15: Prefetch Choices

6.12 Prepare-For-Store

The prepare-for-store operation is used when all of the old data in the cache block can be thrown away.

In a typical write, if the cache block is not in L2 (a cache miss occurs), the prior data is loaded into L2 cache from DRAM. Then the new data is written to the cache block (often only a subset of the cache block is modified), and then the new cache block data is written out.

When using the prepare-for-store operation, the *prior* contents of the cache block will be discarded.

When the core stores stored to L2/DRAM (store operation):

1. Prepares a Write Buffer for the write.

2. If the memory is not in L2 cache (a cache miss occurs), the *prior* data is loaded into L2 cache from DRAM.

All of this takes system overhead.

Prepare-for-store operations can be used to avoid unnecessary DRAM reads for memory locations whose prior value does not matter.

The prepare-for-store function provides two performance-enhancing operations:

1. Creates the Write Buffer in advance of the actual write operation.
2. Prevents reading the old data into L2 cache if there is a cache miss.

Note: Prepare to store is only used if the old data in the cache block can be “thrown away”. The entire cache block must be overwritten by the write. The contents of the cache block in L2 cache must not be relied on in any way.

The Simple Executive prepare-for-store function is: CVMX_PREPARE_FOR_STORE (address, offset).

The hardware instruction is “PREF” (prefetch).

6.13 Scratchpad: Core-local Storage

Part of the core’s L1 data cache may be reserved to as core-local “scratchpad” memory. The scratchpad can be used to store local variables that are accessed frequently but do not need to be stored in DRAM.

Because the scratchpad is local memory, the data does not have to be written to L2/DRAM. This reduces the Write Buffer traffic on the coherent memory bus (CMB).

The OCTEON Simple Executive provides a configuration interface to reserve scratchpad storage via the *executive-config.h* header file. Within this file, there are declarations that define macro names and reserve space for these variables, for example:

```
#ifdef CAVIUM_COMPONENT_REQUIREMENT
    cvmxconfig
    {
        scratch LOCAL_8_BYTE_SCRATCH
        size = 8
        description = "A local 8 byte scratch area";
    }
#endif
```

This code snippet will reserve an eight-byte location in the scratchpad and define the macro **LOCAL_8_BYTE_SCRATCH**, specifying which location it was assigned. The code can then merely dereference this address to use the scratchpad storage.

The Simple Executive functions to access the scratchpad are:

- `cvmx_scratch_read8()`
- `cvmx_scratch_read16()`
- `cvmx_scratch_read32()`
- `cvmx_scratch_read64()`
- `cvmx_scratch_write8()`
- `cvmx_scratch_write16()`
- `cvmx_scratch_write32()`
- `cvmx_scratch_write64()`

Detailed information about this configuration facility is in the documentation set included with the OCTEON SDK.

6.14 Asynchronous FPA Allocation

The Free Pool Allocator (FPA) buffers may be fetched asynchronously. In the Simple Executive API the function is `cvmx_fpa_async_alloc()`. While waiting for the operation to complete, the core can do other processing.

Note that this operation requires configuration of a scratchpad location for the FPA to store the returned pointer.

6.15 Don't Write Back (DWB) Commands

Don't Write Back (DWB) commands can be issued from the core or from other hardware units.

6.15.1 DWB Commands from the Core

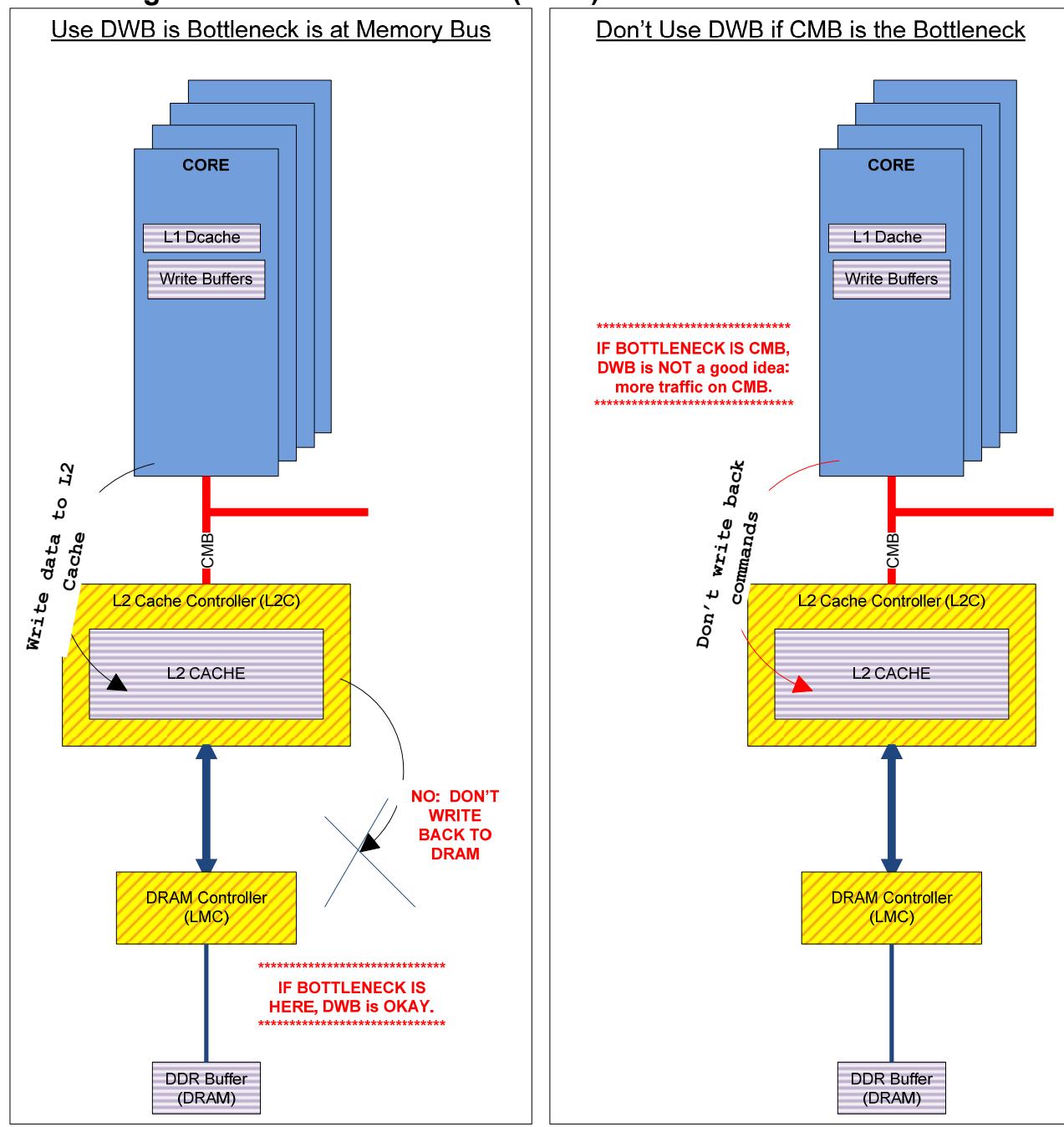
Writes from the core which are destined for L2/DRAM are buffered in the core's Write Buffer. From there, they are sent to the L2 Cache. The L2 cache controller sets the "dirty bit" and later writes the dirty cache block to DRAM (flush).

The Don't-Write Back (DWB) operation can be used to avoid unnecessary write backs from the L2 cache (to DRAM) for memory locations which were used, but the data may now be discarded.

When Don't Write Back (DWB) is used, the core issues commands to clear the cache block's dirty bit. The L2 cache controller clears the dirty bit so when the block is evicted, the L2 cache controller will not write back data to DRAM.

Depending on the exact moment of the write to DRAM, the command to clear the dirty bit might prevent a write from L2 to DRAM. There is no guarantee, since the exact moment of the flush varies.

Figure 16: Don't Write Back (DWB) Commands from the Core



Guidelines:

- If the bus connecting the DDR Controller to the DRAM is overloaded (it is much narrower than the CMB), then Don't Write Back (DWB) can improve system performance. If this bus is not overloaded, do not use DWB.
- If the CMB is overloaded, then DWB might hurt performance, because of the added DWB commands sent on the CMB.

The Simple Executive function `CVMX_DONT_WRITE_BACK(address, offset)` is used to specify DWB.

The hardware instruction is “`PREF`” (prefetch).

6.15.2 DWB Commands from other Hardware Units

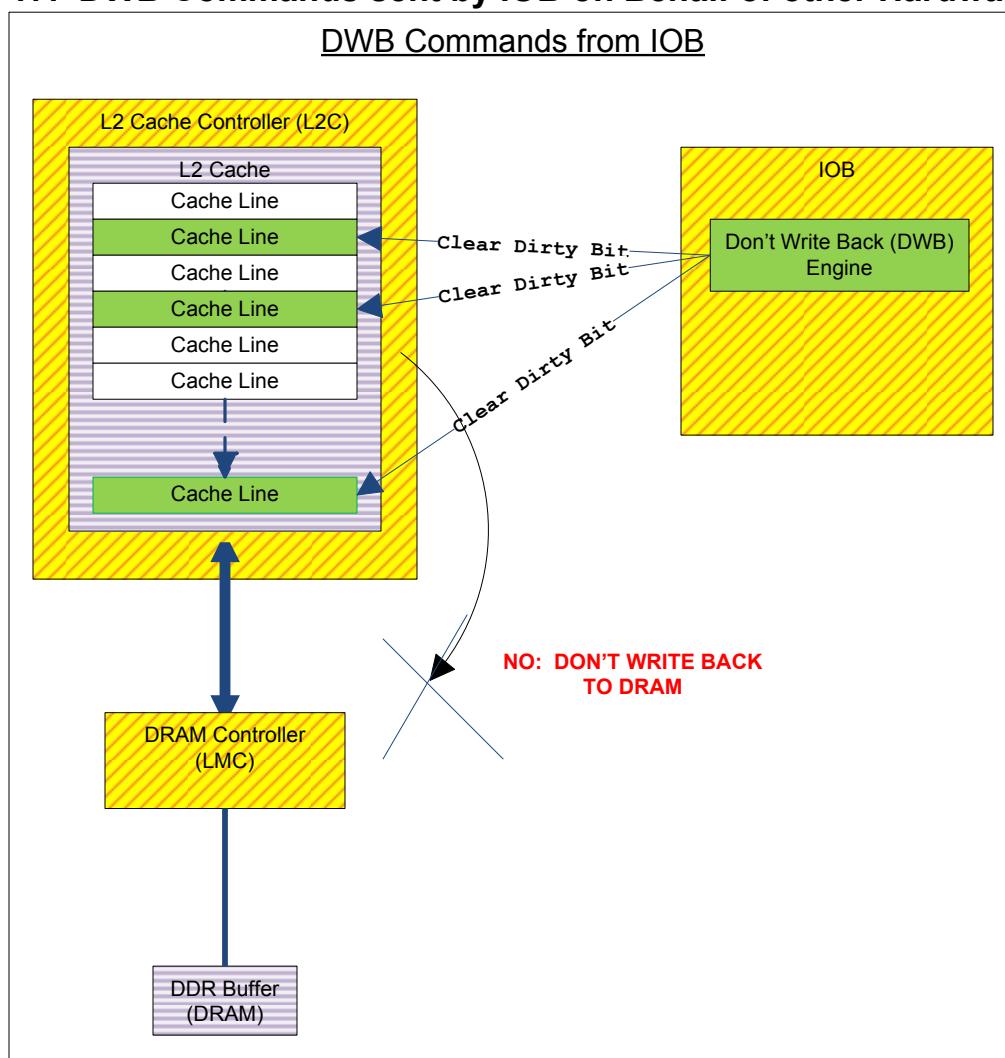
In addition to the core DWB commands, other units can cause DWB commands to be issued. For these units, the IOB Controller actually issues the commands, using its DWB Engine.

For instance, to enable DWB to prevent writing the contents of the Packet Data Buffer back to memory when the buffer is freed, set the register: `PKO_REG_CMD_BUF[ENA_DWB]`.

Other hardware units, such as DFA, PCI, TIM, and ZIP also support DWB.

As noted in the section above, DWB commands add a load to the CMB, while potentially offloading the DDR Controller. DWB commands will not prevent the data from being written back; the exact timing of the dirty-bit clear and the L2 flush to DRAM are not predictable. See the *Hardware Reference Manual* for details.

Figure 17: DWB Commands sent by IOB on Behalf of other Hardware Units



6.16 Hardware CRC Engine

OCTEON has a powerful general purpose CRC engine built into each core. There are many applications that can benefit from such hardware acceleration. For instance, software that creates a hash value by combining several discrete data items can use this engine to compute the hash much faster than software-alone implementations.

A very good description of the CRC algorithm can be found at:
http://en.wikipedia.org/wiki/Cyclic_redundancy_check.

A good description of the Adler-32 algorithm can be found at: <http://en.wikipedia.org/wiki/Adler-32>.

6.16.1 Sample Code: Using the OCTEON Processor CRC Engine

As an example, the Adler-32 algorithm specified in RFC 3309 (SCTP) can be implemented using the OCTEON CRC unit as follows:

```

#define POLY 0x1EDC6F41      // CRC Polynomial
#define POLY_WIDTH 32         // Bit width of the CRC Polynomial
// Note: the initial value "iv" is 0xffffffff
// The "REFLECT" operations are used to reverse the shift of the bits
// into the CRC engine. Both MSB first and LSB first are available
// The Adler-32 algorithm uses LSB first

uint32_t corecrc(uint32_t iv, uint64_t *data, int size) {
    uint64_t my_iv;
    uint64_t t1, t2; // Notice the data is 64-bit aligned

    CVMX_MT_CRC_POLYNOMIAL(((uint64_t) POLY) << (32 - POLY_WIDTH));
    CVMX_MT_CRC_IV(((uint64_t) iv) << (32 - POLY_WIDTH));

    size /= sizeof(uint64_t);

    while (size > 15) { // Notice the loop unrolling (16 64-bit operations)
        t1 = *data++; // Notice the interleaving
        t2 = *data++;
        CVMX_MT_CRC_DWORD_REFLECT(t1);
        CVMX_MT_CRC_DWORD_REFLECT(t2);
        t1 = *data++;
        t2 = *data++;
        CVMX_MT_CRC_DWORD_REFLECT(t1);
        CVMX_MT_CRC_DWORD_REFLECT(t2);
        CVMX_PREFETCH(data,128); // Notice the prefetch here
        size -= 16;
    }

    while (size > 1) {
        t1 = *data++;

```

```

        t2 = *data++;
        CVMX_MT_CRC_DWORD_REFLECT(t1);
        CVMX_MT_CRC_DWORD_REFLECT(t2);
        size -= 2;
    }

    if (size) {
        CVMX_MT_CRC_DWORD_REFLECT(*data++);
    }

    CVMX_MF_CRC_IV_REFLECT(my_iv);
    return((uint32_t) (my_iv >> (32 - POLY_WIDTH)));
}

```

Notice that there are several previously discussed optimizations utilized in this function. The main loop is unrolled for a single cache line (128 bytes processed per loop) and utilizes a prefetch of the next cache line. This substantially improves performance for large blocks. Additionally, memory access occurs on 64-bit boundaries and is interleaved to minimize pipeline stalling.

6.16.2 Performance Comparison: Hardware versus Software CRC

Each core has a dedicated security coprocessor which can be used to accelerate security applications and CRC or hash generation. Once the core issues an instruction to the coprocessor, the core can continue to do other work while the coprocessor completes the instruction, or the core can wait for the coprocessor to complete the task.

Hardware CRC or hashing provides a strong performance improvement over using software to perform these functions.

For example, using security hardware acceleration, CRC generation for a 1024-byte block consumes only 358 cycles versus 13,720 cycles for software CRC, as shown in the table below.

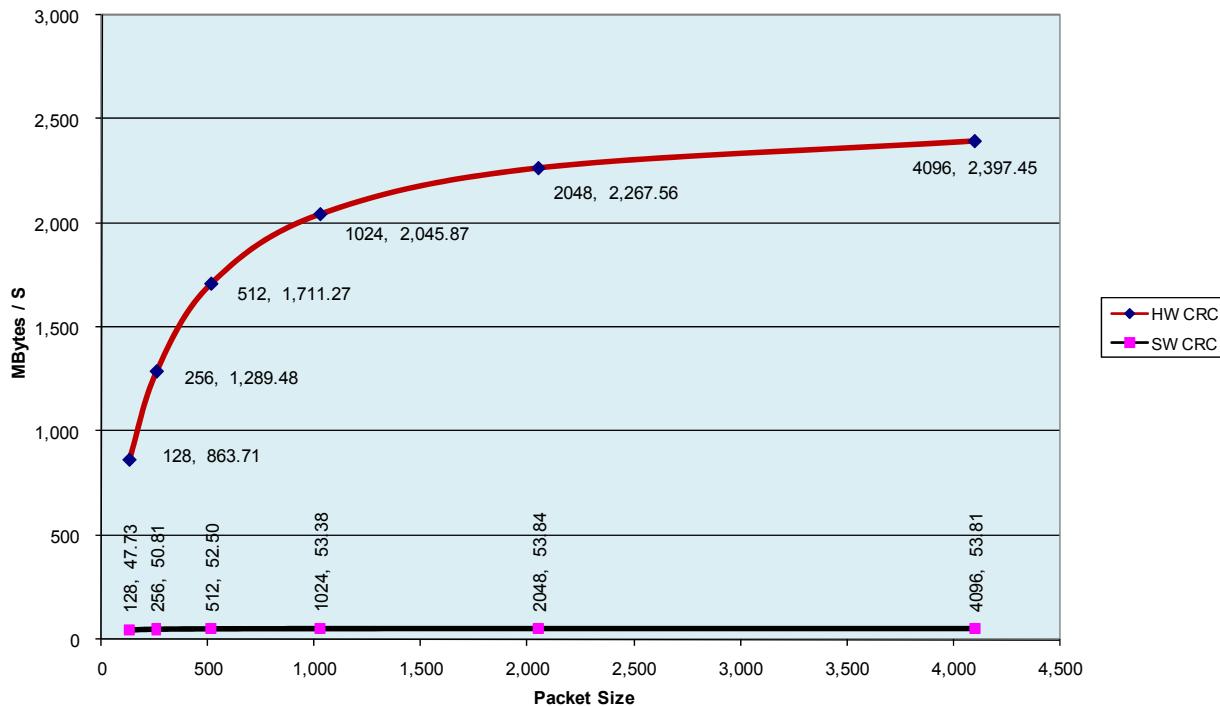
Table 4: Performance Comparison: Hardware versus Software CRC

Block Size	128	256	512	1024	2048	4096
Hardware CRC (cycles/block)	106	142	214	358	646	1,222
Software CRC (cycles/block)	1,918	3,604	6,976	13,720	27,208	54,448

In the following graph, the comparison between hardware and software CRC is converted from cycles to Mbytes/second, using a 750 MHz processor. The black line on the bottom in the graph is the software CRC performance: low throughput and independent of the packet size. The red line on the top is the hardware CRC performance: much higher throughput than software, even at small packet sizes. As the packet size increases, the hardware CRC performance increases, especially between 128 bytes and 1024 bytes.

Figure 18: Graph of Hardware Versus Software CRC – 750 MHz Processor

Performance Comparison: Hardware CRC vs Software CRC
Maximum hardware throughput much larger than software, even at small packet sizes.



6.17 Hardware Hash Engine

All OCTEON cores have a built-in hash unit that performs MD5 and SHA hashing algorithms. Software needing to implement such processing-intensive operations should be modified to take advantage of this hardware acceleration.

6.17.1 Sample Code: Using the OCTEON Processor HASH Engine

RFC 1321 specifies the MD5 algorithm and has an implementation that computes the MD5 hash of an arbitrary memory buffer called MD5String. This same functionality can be implemented on OCTEON cores using the built-in hash unit as shown below.

```
#include "cvmx.h"
/**
 * Calculate the MD5 hash of a block of data
 *
 * @param md5          Filled with the 16 byte MD5 hash
 * @param buffer       Input data
 * @param buffer_len   Inout data length
 */
static void hash_md5(uint8_t *md5, const uint8_t *buffer, int buffer_len)
{
    const uint64_t bits = swap64(buffer_len * 8); /* MD5 expects
                                                Little Endian */
}
```

```

const uint64_t *ptr = (const uint64_t *)buffer;
uint8_t chunk[64];

/* Set the IV to the MD5 magic start value */
CVMX_MT_HSH_IV(0x0123456789abcdefull, 0);
CVMX_MT_HSH_IV(0xfedcba9876543210ull, 1);

/* MD5 input is in the following form:
   1) User data
   2) Byte 0x80
   3) Optional zero padding
   4) Original Data length in bits as an 8 byte unsigned integer
      Zero padding is added to make the 1-4 an even multiple of 64
      bytes */

/* Iterate through 64 bytes at a time */
while (buffer_len >= 64)
{
    CVMX_MT_HSH_DAT(*ptr++, 0);
    CVMX_MT_HSH_DAT(*ptr++, 1);
    CVMX_MT_HSH_DAT(*ptr++, 2);
    CVMX_MT_HSH_DAT(*ptr++, 3);
    CVMX_MT_HSH_DAT(*ptr++, 4);
    CVMX_MT_HSH_DAT(*ptr++, 5);
    CVMX_MT_HSH_DAT(*ptr++, 6);
    CVMX_MT_HSH_STARTMD5(*ptr++);
    buffer_len-=64;
}

/* The rest of the data will need to be copied into a chunk */
if (buffer_len > 0)
    memcpy(chunk, ptr, buffer_len);

chunk[buffer_len] = 0x80;
memset(chunk + buffer_len + 1, 0, sizeof(chunk) - buffer_len - 1);

ptr = (const uint64_t *)chunk;
CVMX_MT_HSH_DAT(*ptr++, 0);
CVMX_MT_HSH_DAT(*ptr++, 1);
CVMX_MT_HSH_DAT(*ptr++, 2);
CVMX_MT_HSH_DAT(*ptr++, 3);
CVMX_MT_HSH_DAT(*ptr++, 4);
CVMX_MT_HSH_DAT(*ptr++, 5);
CVMX_MT_HSH_DAT(*ptr++, 6);

/* Check to see if there is room for the bit count */
if (buffer_len < 56)
    CVMX_MT_HSH_STARTMD5(bits);
else
{
    CVMX_MT_HSH_STARTMD5(*ptr);
    /* Another block was needed */
    CVMX_MT_HSH_DATZ(0);
    CVMX_MT_HSH_DATZ(1);
    CVMX_MT_HSH_DATZ(2);
    CVMX_MT_HSH_DATZ(3);
    CVMX_MT_HSH_DATZ(4);
}

```

```

    CVMX_MT_HSH_DATZ(5);
    CVMX_MT_HSH_DATZ(6);
    CVMX_MT_HSH_STARTMD5(bits);
}

/* Get the final MD5 */
CVMX_MF_HSH_IV(((uint64_t*)md5)[0], 0);
CVMX_MF_HSH_IV(((uint64_t*)md5)[1], 1);
}

```

6.17.2 Performance Comparison: Hardware versus Software Hashing

Using the OCTEON MD5 hashing hardware acceleration increases the performance of this functionality significantly for all block sizes and is shown in the table below. The numbers indicate the number of CPU cycles required to perform the hash over the block of data.

Table 5: Performance Comparison: Hardware versus Software Hashing

Block Size	128	256	512	1024	2048	4096
OCTEON MD5 Hardware (cycles per block)	724	817	1,446	2,706	5,226	10,266
RFC1321 Software (cycles per block)	5,242	6,513	10,661	18,971	35,595	68,843

The OCTEON hardware MD5 hashing unit yields six to seven times performance improvement over optimized software implementations.

6.18 Hardware Timers

The OCTEON processor provides 16 timer rings. The Simple Executive Timer API configures one timer ring per core, eliminating the need to lock the timer ring data structure. Software configures the timer ring's interval.

Software can create a timer event by allocating a Work Queue Entry (WQE) Buffer from the FPA Timer pool, and initializing it. It adds the WQE pointer to the timer ring's to-do list corresponding to how far in the future the event needs to be processed.

The hardware Timer unit will traverse the timer ring data structure, processing one to-do list per interval. For each Timer Entry in the to-do list, the Timer unit adds the Work Queue Entry pointer to the SSO's Input Queue using the `add_work` operation. Software will receive the Work Queue Entry in response to a `get_work` operation, and then process the Work Queue Entry.

After traversing the to-do list, hardware zeroes the list. These timers are one-shot timers. The Timer unit traverses up to 80 million timer entries per second.

An example of utilizing the hardware Timer unit is the TCP acknowledgement timer. When a packet is sent out, the TCP stack expects to receive an acknowledgement for the packet from the receiver. If the stack does not receive the acknowledgment within a certain amount of time, it retransmits the packet. Software can create a hardware timer entry with a specific time interval. If

the acknowledgment arrives with in this time, software cancels the timer; otherwise the Timer unit schedules the Work Queue Entry. On receiving the Work Queue Entry (in response to a `getwork` operation), software knows that it has to re-transmit the packet.

6.19 Hardware Fetch and Add (FAU) Unit

Collecting global statistics with multiple cores can be expensive and requires locking operations to protect the integrity of the statistics. The Fetch and Add (FAU) unit provides atomic update operations and can be used for these types of operations. Using the FAU to do these operations will offload software.

Example code is available in the SDK. One example which uses the FAU unit is `linux-filter`.

6.20 Asynchronous Fetch and Add Operations

The Fetch and Add (FAU) unit supports both synchronous and asynchronous mode of operation. In synchronous mode, an update request is sent to the FAU unit and the core waits for the response to come back. For certain operations (for example updating statistics) the value of the count is not required at the point it is being updated. For these types of operations, an asynchronous fetch-and add-operation can be used, which is very fast. In this mode, an update request is sent and the core does not wait for the response. The value of the statistics can be extracted from the FAU later.

Note that both the synchronous and asynchronous operations require configuration of a scratchpad location for the SSO to store the returned pointer.

6.21 Work prefetch: Asynchronous `get_work`

The `get_work` operation may also be done asynchronously. The core can continue doing processing while waiting for the asynchronous `get_work` to return the next work to do (usually this corresponds to a packet to process). The Simple Executive API call is `cvmx_async_get_work`.

Note that this operation requires configuration of a scratchpad location for the SSO to store the returned pointer.

6.22 Interleaving Prefetch with Computational Instructions

Interleaving is requesting the data slightly before it is actually used. This can reduce the amount of time the core spends waiting for the data to arrive (minimize pipeline stalling). Ideally, prefetch is interleaved with computational instructions.

6.22.1 Sample Code: Interleaving Prefetch

An example of interleaving is seen in the sample code in Section 6.16 – “Hardware CRC Engine”. The following code, taken from that example, uses interleaving:

```
t1 = *data++; // start the fetch of t1
t2 = *data++; // start the fetch of t2 (delay before using t1)

CVMX_MT_CRC_DWORD_REFLECT(t1); // use t1
CVMX_MT_CRC_DWORD_REFLECT(t2); // use t2
```

6.23 Hardware TCP/UDP Checksum Calculation

The Packet Output (PKO) Unit can add the TCP Checksum, which improves performance. The software does not need to calculate the checksum.

If WORD0 [IPoffp1] of the command to the PKO is non-zero, the PKO hardware will generate and insert the TCP/UDP checksum.

See the *Hardware Reference Manual* for details.

6.24 Use Functions Wisely

When doing small operations, especially in a time-critical loop, stay alert to the overhead added by a function call. If the operation is small, it may be that a function call is not the best choice: it may cost many instructions versus only a few.

For example, when copying six bytes of data (for instance, MAC addresses), is it more efficient to use six assignment statements instead of `memcpy()`.

6.25 Update Bit-Fields Wisely

When updating bit fields, be alert to whether the data being updated is still in the L1 Dcache. If it is then there will be no performance issue from the operation.

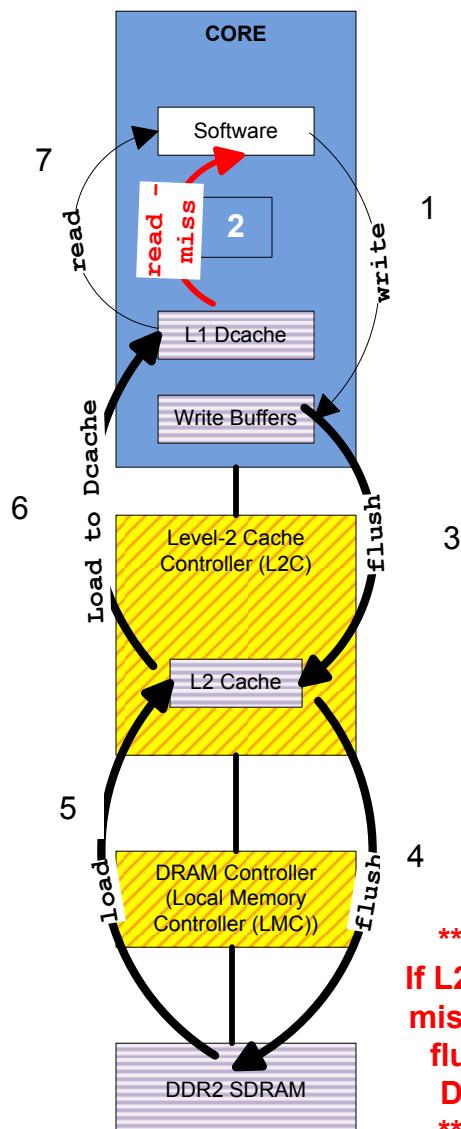
If the data is no longer in L1 Dcache then it is important to know that the OCTEON processor can only do loads and stores on bytes or words, not on bits. To write the bit-field, the processor will need to load the byte or word into L1 Dcache, modify it, and then write it out. The performance is slowed by the cache miss penalty.

By contrast, if the entire byte or word is stored, it does not need to be loaded into Dcache first: the write goes from the Write Buffer to L2 cache without a load into L1 Dcache.

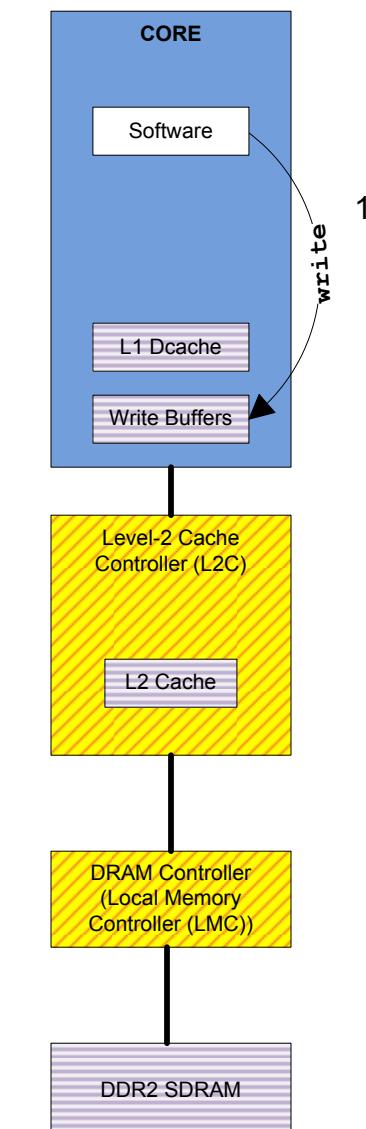
Note that if you know that the data is already present in the L1 cache, you can do the usual individual structure bit field update (first case mentioned above) without paying the price of a cache miss.

Figure 19: Cache Miss When Doing Bit-field Writes

Incorrect: Write Bits in Byte Only: Read/Modify/Write – Cache miss if data is not in Dcache



Correct: Write Whole Byte: Write without need to read data into Dcache: no cache miss.



6.25.1 Sample Code: Updating Bit-Fields Wisely

To order to avoid the cache miss penalty, update the individual bit fields locally and write them back as a complete byte or a word. For example, you may have a structure like:

```
typedef union
{
    uint16_t u16;
    struct {
        uint16_t bit_15: 1;
        uint16_t bit_14: 1;
        uint16_t bit_13: 1;
        uint16_t bit_12: 1;
        uint16_t unused: 12;
    } s;
} bit_struct;
```

The following code will cause a cache miss:

```
bit_struct bstr;

uint16_t bit_15_value = 1;
uint16_t bit_14_value = 0;
uint16_t bit_13_value = 0;
uint16_t bit_12_value = 0;
uint16_t unused_value = 0;

bstr.s.bit_15 = bit_15_value; // this line will cause a data cache miss
bstr.s.bit_14 = bit_14_value;
bstr.s.bit_13 = bit_13_value;
bstr.s.bit_12 = bit_12_value;
bstr.s.unused = unused_value;
```

The following code will avoid the cache miss:

```
bit_struct bstr;
uint16_t bit_15_value = 1;
uint16_t bit_14_value = 0;
uint16_t bit_13_value = 0;
uint16_t bit_12_value = 0;
uint16_t unused_value = 0;
uint16_t local_val = 0;

local_val = ((bit_15_value << 15) |
             (bit_14_value << 14) |
             (bit_13_value << 13) |
             (bit_12_value << 12) |
             (unused_value << 11)) & 0xffff;

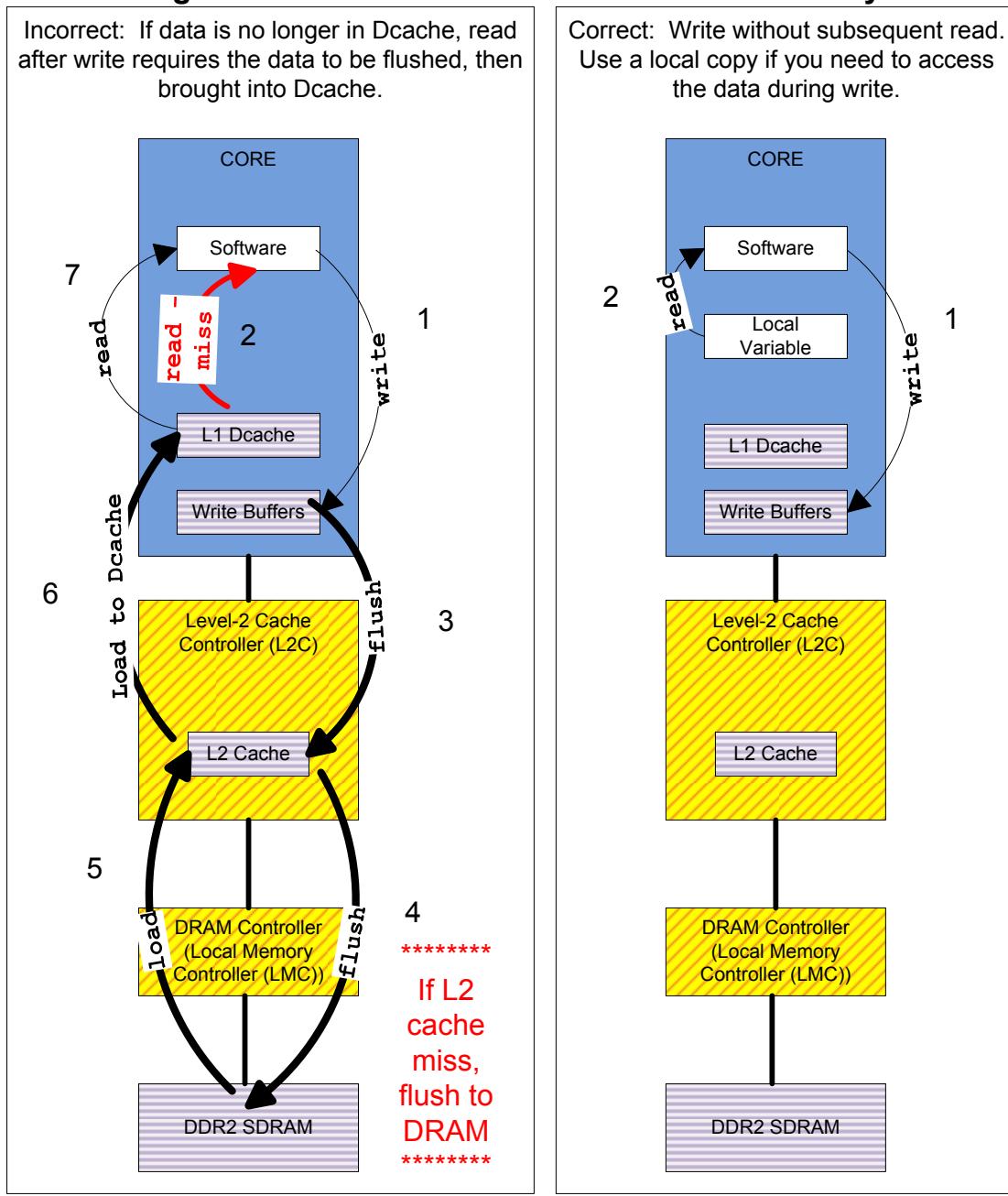
bstr.u16 = local_val; // this will write the complete 16 bits without
                     // a cache miss penalty
```

6.26 Read After Write

If a write is being done on data which is not present in L1 Dcache, the writes will go through the Write Buffers without touching Dcache. If we read the data after the write, the data may not be in

Dcache. If the data is not in Dcache, the system will need to flush the writes to L2/DRAM, and then bring the cache line into L2 Dcache. There is a big performance penalty for this. It is better to save a local copy of the data and read that copy as needed, than to read the copy just written.

Figure 20: Read after Write Performance Penalty



6.26.1 Sample Code: Read after Write

Here is an example (assuming that “swp” is not in L1 Dcache):

```

swp->len = 40;
swp->packet_ptr.s.i = 1;
swp->packet_ptr.s.addr += CVM_TCP_BUFFER_START_SKIP;
swp->packet_ptr.s.size = swp->len; // Here we READ the value just
// written: inefficient
swp->word2.s.not_IP = 0;

```

In order to avoid this problem, use a local variable for the value needed later in the code, as shown in the following code sample:

```

int tlen = 40; // local variable for later use: efficient

swp->len = tlen;
swp->packet_ptr.s.i = 1;
swp->packet_ptr.s.addr += CVM_TCP_BUFFER_START_SKIP;
swp->packet_ptr.s.size = tlen;
swp->word2.s.not_IP = 0;

```

7 Tuning Multi-core Applications (Scaling)

7.1 Re-Configuring the Right Amount of Packet Data Buffers and WQE Buffers

Review the system requirements and verify that sufficient numbers of Packet Data Buffers and Work Queue Entry Buffers have been configured into the Free Pool Allocator pools.

7.2 Tune Initial Tag Values to Separate Flows

The packet’s initial tag value is set on ingress by the PIP/IPD. The initial tag value is used to separate flows. More unique flows will improve scaling by creating more unique Tag Tuples. When ATOMIC locks are used, more unique Tag Tuples equates to more locks. There will be fewer processes contending for the same lock, improving throughput.

7.3 Set Initial Tag Type to ORDERED if Possible

The packet’s initial tag type is set on ingress by the PIP/IPD. The initial tag type may be set to ORDERED, ATOMIC, or NULL. Packets with the same tag value and an ATOMIC tag type are processed one at a time. This creates a bottleneck. In some cases the bottleneck cannot be avoided, but whenever possible, use the ORDERED tag type. Packets with the ORDERED tag type may be processed in parallel by multiple cores.

7.4 Switch Tag Type to ORDERED or NULL when Possible

Whenever possible, software should use the `switch_tag` operation to change the tag type from ATOMIC to ORDERED or NULL.

7.5 Use Asynchronous Switch Tag Operations

When using the `switch_tag` operation, the core may continue to do other processing while waiting for the tag switch to complete. The core is notified via the Switch Complete Bit (hardware

register 30) that the switch has completed. By using the RDHWR instruction, the core may check on the status of the Switch Complete Bit. When the bit is set, the switch is complete.

7.6 Critical Regions

Critical regions are areas of code where only one process may enter at a time. The processes/threads accessing the region might be one of many on the same core (multi-process/multi-thread), or on different cores (multi-processor). These critical regions are usually brief. An introduction to concurrent processing issues is provided in Section 5.4 – “Concurrent Programming Techniques”. In Section 7.7 – “Replace Spinlocks with Packet-Linked Locks When Possible”, locking will be discussed in more detail.

An example of a critical region is modification of the TCP/IP control block, which is usually located in shared memory. Multiple processes or multiple processors need to update it when a packet is sent or received on that particular connection.

In a single-processor environment with multiple processes (multi-processing), mutexes are commonly used to protect critical regions. In multi-processor environment, mutexes are implemented by a spinlocks. Spinlocks causes the CPU to wait until the lock is granted. The CPU cannot do anything else while waiting.

Architectural goal #1: minimize, if not eliminate “serialized processing regions” (critical regions which only one process can enter at a time). There are Cavium Networks-specific ways to minimize critical regions (such as improving Tuple Hash calculation (see Section 7.2 – “Tune Initial Tag Values to Separate Flows”).

Architectural goal #2: Where critical regions exist and are highly contended, use packet-linked locks instead of spinlocks (see Section 7.7 – “Replace Spinlocks with Packet-Linked Locks When Possible”).

7.7 Replace Spinlocks with Packet-Linked Locks When Possible

The OCTEON processor provides packet-linked locks which can be used instead of spinlocks. These packet-linked locks are implemented using the ATOMIC tag type. See the *Packet Flow* chapter of the *OCTEON Programmer's Guide* for more information.

7.7.1 Spinlocks

Spinlocks are used to protect critical regions, allowing only one process to access the region at a time.

Spinlocks can have the following problems, especially when the level of contention is high (a lot of processes or processors want the lock):

1. They do not grant fair access, may starve some processes.
2. They do not provide access in any particular order.
3. They do not allow the process to do anything while waiting for the lock.
4. They take an indeterminate amount of time before the lock is granted (since there is no particular order to grant – the same core may get the lock multiple times, while another core may not get it at all).
5. They become even more problematic (higher contention) with scaling: more processes want the lock, more overhead of trying to get it, more chance of starvation. Less probability of winning the lock in a larger group of contenders.

7.7.2 The Scheduling / Synchronization / Order Unit and ATOMIC Tag Type

The Scheduling / Synchronization / Order (SSO) unit (also known as the Packet / Order / Work (POW) unit) on the OCTEON chip provides an ideal locking mechanism, especially useful when scaling applications. This mechanism is referred to as a *packet-linked lock*. Note that this mechanism does not provide a general-purpose lock. This mechanism is designed to use during a packet processing ATOMIC phase, such as locking the TCP/IP control block.

Using this hardware assistance:

- The lock is granted fairly: All packets from the same flow will receive the lock in ingress order. This eliminates starvation, and provides fairness in granting the lock.
- Access is in ingress order: All packets from the same flow will receive the lock in ingress order.
- The process does not wait: The core may request the lock before it is needed, continue processing up to the point of actually needing the lock, or work on some other part of processing, then when it gets the lock, enter the critical region. This is impossible to do with a spinlock.
- The time before the lock is granted is deterministic: Since all packets from the same flow receive the lock in ingress order, the delay depends only on the length of the critical region and the number of prior packets in the flow waiting for the lock.
- Performance will not degrade when scaling. The time before the lock grant is still determined by the length of the critical region and the number of prior packets in the flow waiting for the lock.
- Additionally, due to the hardware assistance from the SSO, the core doesn't spend any time figuring out whom to grant the lock to next. All this overhead is offloaded to the SSO.

Note that a core may only request/hold one lock at a time.

This feature uses the ATOMIC tag type. When the process wants the lock, it switches the tag type to ATOMIC, then checks later to see if the lock has been granted. More details can be found in the SSO (POW) chapter in the *Hardware Reference Manual*.

The packet's initial tag type is assigned by the PIP/IPD on ingress. The core may later switch the tag type. For instance, the initial tag type may be ORDERED. ORDERED packets may be

processed in parallel, on multiple cores. When the core wants to lock a critical region, it switches the tag type to ATOMIC.

Note that packet-linked locks are used as part of packet processing, not as general-purpose locks. Also, since the core may have only one outstanding lock request at a time, this lock cannot be used in situations where two locks are required.

Table 6: Spinlock versus SSO Packet-Linked Locking

Features	SSO Packet-Linked Locks	Spinlock
Lock is granted fairly	yes	no
Processes may be starved waiting for the lock	no	yes
Access is in ingress order	yes	no
Process is idle while waiting	no	yes
Lock grant time is deterministic	yes	no
Performance will not degrade when scaling	yes	no
Core is offloaded, work shifted to SSO	yes	no

7.7.3 Example of Spinlock versus SSO ATOMIC Locking

The following two examples (in pseudo code) illustrate the CPU continuing to work while waiting for the lock:

7.7.3.1 Sample Pseudo code: Using Spinlock

In this case, the thread of execution will remain blocked in `Spinlock_lock()` until it is able to obtain the lock. If this were a highly contested spinlock, the wait period would be non-deterministic.

```

Do_non_serialized_region_processing();
Spinlock_lock();                                // Inefficient: Blocked while
                                                // waiting
Do_serialized_region_processing();
Spinlock_unlock();

```

7.7.3.2 Sample Pseudo Code: Using SSO / ATOMIC Tag Type

The following code uses a tag switch to the ATOMIC tag type to lock the region. While waiting for the lock to be granted, the core does other work.

```

Do_some_non_serialized_region_processing();
Initiate_atomic_tag_switch();
Complete_non_serialized_region_processing(); // Efficient: Work while
                                              // waiting
Wait_for_atomic_tag_switch_completion();
Do_serialized_region_processing();
Initiate_tag_switch_to_some_other_tag_or_NULL();

```

The SSO is responsible for figuring out which process gets the lock next, and notifying the process when the lock has been granted. Since the SSO maintains the order of the requests, the lock may be granted in a fair order. The order in which the lock is granted depends on the tag type before the switch, how many requesters are waiting, and how long the critical region is.

7.8 Arena-based Memory Allocation

Memory allocation of variably sized shared memory regions can be another bottleneck. Imagine the free memory is in a memory pool. Multiple cores wish to get the free memory. To allocate free memory, the core locks the pool, removes the needed chunk of memory, then unlocks the pool. Multiple cores contending for the same lock can introduce a delay.

In arena-based memory allocation, the free memory is divided into multiple free pools (arenas). These are put into a list of arenas. Any core can allocate from or free to any arena. The system will look for one which is unlocked, and use that one for the requesting core. The virtual addresses returned from this allocation are mapped into all cores, and thus can be shared.

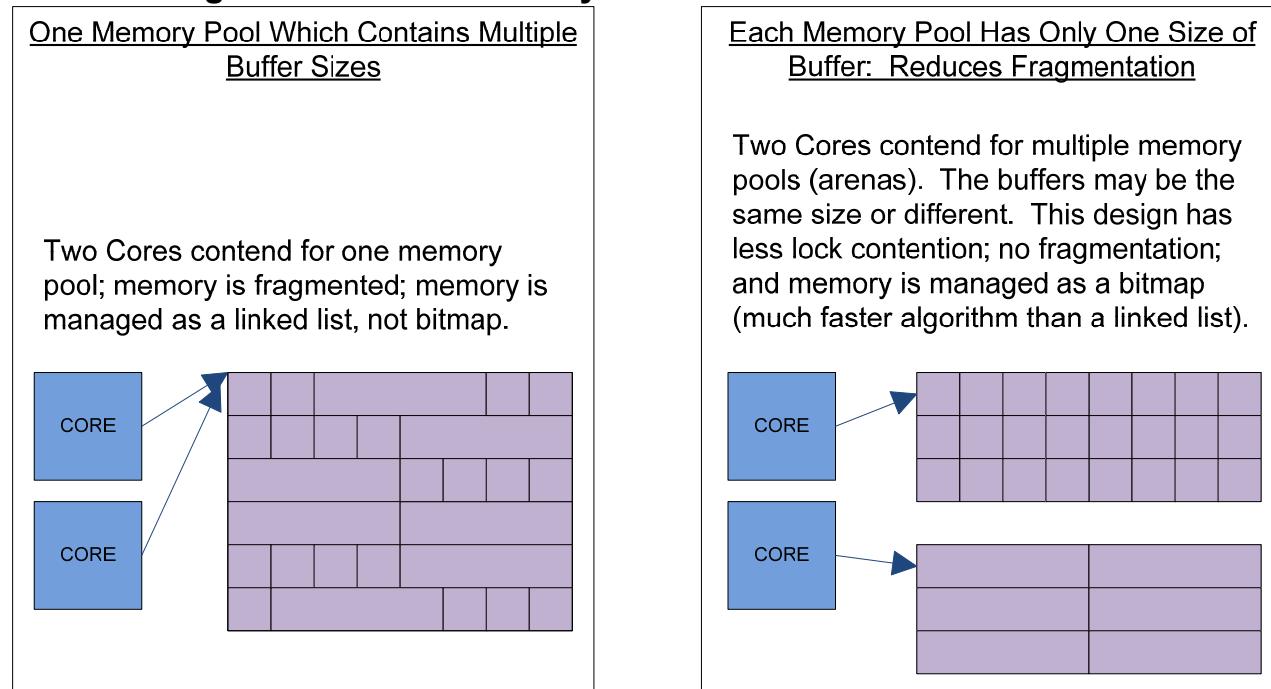
Support for arena-based memory allocation is built-into the Simple Executive API.

Note that non-shared memory (on the core-local heap), is always available by using the standard `malloc()` and `free()` function calls. These calls provide virtual addresses private to the core. This memory is not subject to contention.

The Simple Executive functions which support arena-based memory allocation are:

`cvmx_add_arena()`, `cvmx_malloc()`, `cvmx_calloc()`, `cvmx_realloc()`,
`cvmx_memalign()`, and `cvmx_free()`.

Figure 21: Arena Memory Allocation Reduces Contention



7.9 L2 Cache Configuration: Way Partitioning and Cache-Block Locking

Way partitioning can prevent specific cores or I/O devices from polluting the L2 Cache. See the *Hardware Reference Manual* for more information.

L2C can lock individual cache blocks into the L2 cache. A locked block is not replaced (until it is later flushed from the cache via the procedure described in the *Hardware Reference Manual*, or until the chip is reset), so fast access to the block is guaranteed when it is locked.

See the *Hardware Reference Manual* for more information.

8 Linux-specific Tuning

In evaluating Linux-specific performance bottlenecks, it is important to measure whether most of the time is spent in application time versus kernel time.

The following tuning suggestions may be worth your time: in one case, the improvement was from 200 connections per second to 25,000 changes per second.

8.1 TLB Exceptions and Huge Page Size

The default page size is 4KBytes. If performance testing shows a large number of TLB exceptions, configure the TLB to use a larger page size, such as 32KBytes.

In one specific application, a shift from 4KBytes to 32KBytes pages yielded a 30% performance improvement. The ideal page size will depend on your application.

8.2 Use CPU Affinity for Processes/Threads

Performance will be improved if CPU affinity is used to bind a process/thread to a core. For example, the RX process can run on one core (or a small collection of cores). For each of these cores, the RX process can be the only process running on the core, so it will not have to context switch and scheduling overhead is small.

8.3 Direct all Packet RX Interrupts to the Same Core

If packet RX interrupts are sent to all cores, not just those which can process the interrupt, then the other cores will have unnecessarily degraded performance.

Glossary

5-tuple

5-tuple is a common networking term which refers to classification of a packet by its IP protocol, IP source address, IP destination address, and (if present) source port and destination port.

address

In this document, the word *address* refers to a physical address.

ALU

Arithmetic Logic Unit. The part of the core which performs arithmetic operations such as add, multiply, shift, and compare. See the *HRM* core block diagram for details.

API

Application Programming Interface. Typically, this is code which provides a convenient interface to the hardware units.

ATOMIC tag type

The *ATOMIC tag type* is used for locking. Only one in-flight packet with the same tag tuple can have the ATOMIC lock. ATOMIC tag tuple processing is serialized: one-at-a-time in ingress-order. This tag type can be used to protect critical regions. This is how packet-linked locks are implemented. The SSO will grant the lock in ingress order, not the order the request for the ATOMIC tag type was made.

barrier sync, barrier synchronization

Barrier synchronization is used to synchronize the cores. Typically, one core performs program initialization while the other cores wait. When initialization is complete, the initializing core writes a value to a memory location. The other cores wait until the value is written.

base SDK

The *base SDK* is one of the RPM files supplied by Cavium Networks. The base SDK includes:

- The complete GNU-based tool chain including the compiler, linker, and generic libraries, optimized to take advantage of the cnMIPS cores contained within the OCTEON processor.
- The OCTEON software simulator, which includes performance measuring tools.
- Cavium Networks Simple Executive: software that enables quick application development. This software provides a C or C++ API to the underlying hardware.
- Several example applications.

bit field

Bit field refers to a subset of a byte or a word. Bit field accesses are common when processing packet headers.

bootloader

The *bootloader* is the program which initializes the board. This program is typically stored in flash, but can also be downloaded and booted from a PCI host.

buffer pool

A *buffer pool* is a collection (pool) of free buffers used for a common purpose. Buffer pools are managed by the FPA Unit.

bus

A *bus* is one specific connection with a dedicated purpose, direction(s), and bandwidth. More than one device can be on the same bus.

Cached Input Queue

The *Cached Input Queue* is the portion of the SSO QoS Input Queue which fits into the SSO's internal memory. Each Cached Input Queue is comprised of linked WD. Each WD in the list contains a pointer to a corresponding WQE.

CMB

See Coherent Memory Bus.

CMI

Coherent Memory Interconnect. The OCTEON II enhanced version of the CMB.

cnMIPS

The term *cnMIPS* refers to Cavium Networks version of MIPS, which contains additional Cavium Network-specific instructions.

code locality

Code locality refers to limiting the instructions run by a core to a small subset of code which will fit in Icache. This is done to improve performance in some applications. Once the instructions are loaded into Icache, the Icache misses drop as close to zero as possible. Note that this is not the optimal design in all situations.

Coherent Memory Bus

CMB. The *Coherent Memory Bus* is actually an interconnect, not a bus. This interconnect connects the cores, the L2 cache controller, and the I/O Bridge. The coherent bus is responsible for making sure the data in Dcache is invalidated if cache block is changed in the L2 cache.

control-plane application

Application functions may be divided into two categories: *control-plane* (slow path), and data-plane (fast path). The control-plane usually handles exceptions.

Coprocessor0

Coprocessor0 (Cop0) is a standard MIPS coprocessor which is used for system control, TLB, and exception handling.

Coprocessor1

Coprocessor1 (Cop1) is a standard MIPS coprocessor which is used for floating point processing. This coprocessor is not supported on cnMIPS.

Coprocessor 2

Coprocessor2 (Cop2) is used to provide the Security Engine and the CRC Engine.

Core State Descriptor

Inside the SSO, there is one *Core State Descriptor* data structure for each core. When the core performs a successful `get_work` operation, a Work Descriptor is removed from the Cached Input Queue and assigned to the core. A pointer to the assigned Work Descriptor is stored in the Core State Descriptor. The Work Descriptor contains a pointer to the WQE. The `get_work` operation returns the WQE pointer to the core.

CRC Engine

The *CRC Engine* is used to accelerate Cyclic Redundancy Check (CRC) generation. There is one CRC Engine per core.

critical region

A *critical region* is typically a short section of code where one-at-a-time access is critical. For example, code which modifies shared data structures. Critical regions, such as code which modifies shared data structures, may be protected by using packet-linked locking, which is implemented by the ATOMIC tag type. When a core needs to access a critical region, it changes the Work Descriptor's tag type from ORDERED to ATOMIC.

Crypto Unit

The *Crypto Unit* is also called the Security Coprocessor or Security Engine.

CSR

Control and Status Register. These registers are used to configure the hardware units, and query status.

cvmseg

Part of the per-core data cache (Dcache) may be set aside for IOBDMA operations and scratchpad memory. This area of virtual memory is referred to as *cvmseg*. The amount of Dcache used for *cvmseg* is set when either Simple Executive or Linux is configured. Note that since space for *cvmseg* comes from Dcache, keeping the size of *cvmseg* to a minimum will help system performance by leaving more Dcache blocks available for the application. The special *cvmseg* memory is configured at build time for both Simple Executive applications and Linux.

cvmx

A prefix commonly found in the code, which stands for CaViuM networks eXecutive (the Simple Executive).

cvmx_shared region

The *cvmx_shared* region is a special section in the ELF file which can be used for a small amount of shared memory. This memory is shared between the members of a load set.

data cache

DCACHE. There is one *data cache* per core, located in the core's L1 cache area. Writes from the core go to the Write Buffer (also in L1 cache). From there, they are written simultaneously to the DCACHE and the L2 cache. DCACHE is both readable and writable.

data-plane application

Application functions may be divided into two categories: control-plane (slow path), and *data-plane* (fast path). The data-plane handles normal packet processing.

Dcache

See data cache.

deschedule

The core may perform an operation to *deschedule* the Work Descriptor, so that the Work Descriptor is no longer assigned to the core. The SSO will reschedule the work descriptor to the same or a different core, using the normal scheduling criteria. A descheduled work descriptor which is runnable has a higher priority than a Work Descriptor which has never been scheduled.

Descheduled-Now-Ready List

Descheduled Work Descriptors which are runnable are put on the *Descheduled-Now-Ready List* (DS-Now-Ready List). When a core requests more work, the scheduler will check this list before checking the Input Queues.

development host

To avoid confusion with the term *PCI host*, the term *development host* is used to describe the i386 or x86_64 machine which is used as a cross-development platform.

development target

The term *development target* refers to the OCTEON evaluation board connected to the development host.

DFA Unit

See Pattern Matching and Regular Expression Engine.

DMA

Direct Memory Access. Hardware units can access memory independently from the CPU.

Don't Write Back

DWB. This OCTEON feature is used to avoid unnecessary L2 data writes to memory. For example, the packet data in L2 cache can be discarded after the packet is transmitted. In a conventional L2 cache design, all dirty data is written back to memory. The OCTEON processor provides the option not to write back selective data.

DS-Now-Ready List

See Descheduled-Now-Ready List

dual-issue

More than one ALU instruction may be processed at a time by a core.

DWB

See Don't Write Back

ELF executable file, ELF file

The *ELF executable file*, or (*ELF file*) is an executable ELF-format application, the output of the linker (for example, the a.out file).

FAU

Fetch and Add Unit. This unit is used to add a number to a memory location, and can be used to manage counters.

flow

Flow is a common networking term which refers to a uni-directional collection of packets which share the same 5-tuple.

FPA Unit

Free Pool Allocator Unit. This unit manages pools of free buffers, including Packet Data Buffers. It can be used as a general buffer manager, not only as a packet buffer manager.

group (Grp)

The packet's *group* value is assigned by the PIP/IPD on input. Work groups are used to balance the processing load among the different cores, providing architectural flexibility. Each group can have from 0 to all of the cores. Cores can be in more than one group. The number of cores in each group can easily be changed by software, allowing the application to dynamically adapt to varying workload requirements. When a core requests more work to do, the SSO will only schedule work from an appropriate group to the core.

HRM

Cavium Networks *Hardware Reference Manual*.

hybrid system

A *hybrid system* is a multicore OCTEON processor where more than one boot command has been used to load the cores, such as a system with both SE-S and Linux SE-UM applications.

I/O Bridge

IOB. I/O Bridge. The bridge connecting the CMB interconnect and the I/O interconnect.

I/O Interconnect

IOI. The *I/O Interconnect* is a collection of buses which connect hardware units which are not connected to the CMB. The I/O Bridge is connected to both the IOI and the CMB.

I/O Space, I/O Space Address

The *I/O space* contains the OCTEON configuration and status registers for the various hardware units and also contains the PCI configuration, I/O and memory space. If physical address bit 48 is 1, the access is to I/O space.

Icache

See instruction cache.

In-Flight Queue

Once the WD has been assigned to a core, it is considered to be in-flight, even if it is pending a tag switch or descheduled. Work with the same tag tuple is on the same *In-Flight Queue*. If there are no in-flight Work Descriptors for a particular tag tuple, there is no In-Flight Queue for that tag tuple. Note that more than one WD from the same flow can be in-flight simultaneously. The In-Flight Queues are internal to the SSO, and maintained by the SSO. They are essential to maintaining packet order, critical region locks, and packet serialization.

in-flight Work Descriptor

Once a Work Descriptor has been scheduled on a core, it is considered to be *in-flight* until it is discarded by a subsequent get_work operation or a switch to the NULL tag type. Descheduled Work Descriptors are also considered to be in-flight, since processing on the associated WQE has started, but has not completed.

ingress order

The *ingress order* is the order that packets from the same flow arrived at the PKI and were submitted to the SSO.

initial In-Flight Queue

The *initial In-Flight Queue* is the In-Flight Queue the Work Descriptor is on when the tag switch request begins.

in-memory image

After storing the ELF file in the Reserved Download Block, the bootloader reads the ELF file, parses it, allocates system memory for the in-memory image, and creates the *in-memory image(s)* in different system memory location(s). Note the application's in-memory image size is larger than the ELF file size because it includes memory allocated for the stack and heap. The program will run from the in-memory image. There is one in-memory image per instance in a load set; if a load set consists of 4 cores, there are 4 in-memory instances of the program.

Input Queue

The *Input Queues* are the QoS queues maintained by the SSO. The SSO has 8 input queues (0-7), one per QoS value. When a new WQE is added to the SSO, the WQE goes onto the input queue which matches its QoS value. When the WQEs are added to the SSO's input queue, the "Next Pointer" is used to link them into a list. The Input Queues can consist of two parts: the Cached Input Queues and the Overflow Input Queues.

instruction cache

Icache. There is one per core, located in the core's L1 cache. The instruction cache is read-only.

interconnect

An *interconnect* is a group of buses with a specific common purpose, such as to connect a specific collection of devices.

IOB

See I/O Bridge

IOBDMA

The term *IOBDMA* refers to an I/O Bridge DMA operation. IOBDMA operations are asynchronous (the program does not wait for the result). When the program is ready to use the buffer, it issues a SYNCIOBDMA operation to make sure all the IOBDMA operations for that core have completed, and then retrieves the returned buffer address from the scratchpad.

IOBI

I/O Input Bus. One of the buses in the I/O Interconnect.

IOBO

I/O Output Bus. One of the buses in the I/O Interconnect.

IOI

See I/O Interconnect.

IPD Unit

Input Packet Data Unit: This unit works with PIP to manage packet input.

IPDB

IPD Bus. The IPDB is used to DMA received packet data from the IPD to L2/DRAM.

KEY Unit

The KEY Unit provides and manages secure on-chip memory which can be used to store a hardware key, and can be reset using an external pin. This unit is not present in all OCTEON models.

kseg0

The *kseg0* segment is in the kernel address space (unmapped, uncached) in the 32-bit MIPS virtual memory address map. On cnMIPS, accesses to this segment access system memory, which is always cached on OCTEON. SE-S 32-bit applications run in kernel mode and access system memory through *kseg0* addresses.

kseg1

The *kseg1* segment is in the kernel address space (unmapped, cache attribute not defined for generic MIPS) in the 32-bit MIPS virtual memory address map. On cnMIPS, accesses to this segment access system memory, which is always cached on OCTEON.

kseg3

The *kseg3* segment is in the kernel address space (mapped) in the 32-bit MIPS virtual memory address map. On cnMIPS, user mode access is allowed only to *cvmseg*, which is inside *kseg3*.

L1 cache

Each core has a private *L1 cache*. This cache is divided into the data cache (Dcache) and instruction cache (Icache).

L1 data cache

See data cache.

L1 instruction cache

See instruction cache.

L2 cache

The *L2 cache* is shared by all the cores. Different OCTEON models offer different L2 cache sizes and features.

L2 Cache Controller

L2C. The *L2 cache controller* is responsible for managing the L2 cache and the interface to the DRAM controller.

L2C

See L2 Cache Controller.

LLMEM

Low Latency Memory. Some OCTEON models use low latency memory as the pattern memory used by the Pattern Matching and Regular Expression Engine.

LMC

Low Latency Memory Controller

Load Set

All cores booted by the same load command are in the same *load set*. For SE-S applications, all cores loaded via the same boototc command are in the same load set. For SE-UM application started from Linux, all cores started by the same oncpu command are in the same load set.

named block

Global boot memory can be allocated and accessed via names. These *named blocks* are useful for creating memory blocks which can be shared between different load sets.

NULL tag type

NULL tag types are neither ordered nor serialized: multiple cores can process multiple packets, and no packet order is maintained by the SSO. Examples of packets where ordering may not be important include ICMP packets (such as ping packets), UDP packets where ordering is not required, and non-IP packets. If a WD assigned a NULL tag type on ingress, then the SSO will assign it on a core, but will not put it in an In-Flight Queue. The NULL tag type means that the SSO will not keep the packets with the same tag tuple in ingress order.

ORDERED tag type

Multiple packets from the same flow with an *ORDERED tag type* may be processed in parallel by multiple cores. Thus, “*ORDERED*” does *not* mean “only one packet at a time”.

Output Queue

See PKO Output Queue.

Overflow Input Queue

The *Overflow Input Queue* is the portion of the SSO QoS Input Queue which does not fit into the SSO's internal memory. This queue is comprised of linked WQE Buffers.

PABITS

On MIPS, *xkphys* addresses are not mapped, and are never translated by the operating system or TLB. The *xkphys* addresses provide a “window” into the physical address space. The high bits are stripped off the virtual address, and the low *PABITS* (Physical Address BITS) are used as a physical address. On OCTEON, PABITS is 49: bits <48:0>, matching the number of SEGBITS (49).

Packet Data Buffer

The *Packet Data Buffer* is used to store packet data if the packet data will not fit in the WQE. Packet Data Buffers are managed by the FPA, automatically allocated by the PIP/IPD, and automatically freed by the PKO.

packet-linked lock

The ATOMIC tag type is used to provide *packet-linked locks*. These locks can be used instead of spinlocks. They provide ATOMIC access to packets in ingress order, and can be used asynchronously so that the core may continue processing while waiting for the lock to be granted. These locks off-load the cores, because they are managed by the SSO.

pattern memory

The *pattern memory* is off-chip memory which is used with the Pattern Matching and Regular Expression Engine. This memory can also be accessed via instructions from the core.

Pattern Matching and Regular Expression Engine

This unit is used to perform string matching. The unit has different names on different OCTEON models, for example Deterministic Finite Automata (DFA). This unit is not present in all OCTEON models.

physical address

At the hardware level, transactions requiring addresses use *physical* addresses. For instance, the “allocate” and “free” operations use the physical address of the buffer in DRAM, not a virtual address. When accessing hardware registers directly, be aware that addresses sent and returned are physical, not virtual addresses.

PIP Unit

Packet Input Unit. This unit is responsible for receiving, buffering, classifying, and tagging an input packet. It then gives it to the SSO for scheduling. It may also drop the packet due to configurable admission control.

PKI Pseudo Block

PacKet Input Pseudo Block. This pseudo block contains the PIP and IPD hardware units.

PKO Output Ports

The PKO supports up to 40 *PKO Output Ports*, depending on the OCTEON model. Different ports correspond to the different hardware interfaces.

PKO Output Queue

The PKO has up to 256 *PKO Output Queues*, depending on the OCTEON model. The Output Queues are mapped to the Output Ports. The Output Queues can have different priorities, which are configured at system initialization time. To insure that all packets from the same flow are transmitted in ingress order, send them all to the same Output Queue.

PKO Unit

Packet Output Unit. This unit manages packet output.

PKOB

PKO Bus. Used to DMA packet data from L2/DRAM to the PKO's internal memory.

POB

Packet Output Bus. Used to transfer packet data from the PKO's internal memory to the output port.

pointer

In this document, the word *pointer* refers to a C or C++ data type which holds a *virtual* address, NULL, or an invalid address.

POW Unit

Same as SSO Unit. The term POW is used in the *HRM*.

PP

Packet Processor. This term is used primarily in the *HRM*.

prefetch

The word *prefetch* is used to describe IOBDMA operations which can be used to allocate free buffers before they are needed. See also prefetch instructions.

prefetch instructions

OCTEON processors provide multiple *prefetch instructions* to move data into L1 and/or L2 caches prior to the application needing the data. These instructions are used to avoid cache misses. See also prefetch.

processor

In this document, the word *processor* refers to the entire chip, with all of the different functional hardware blocks including all of the cores on the chip. Individual cores are called *cores*, not processors.

QoS Input Queues

See Input Queue.

QoS Value

See Quality of Service Value.

Quality of Service value

The *Quality of Service value* is a number (0-7) which represents the priority of the packet. When packets are received, the PIP/IPD computes the QoS number for the packet, and saves the value in the Work Queue Entry. There is no requirement for 0 or 7 to be the highest priority, and there is no requirement for the priority to be linear.

RAID Engine

The RAID Engine provides RAID/XOR Acceleration. This unit is not available in all OCTEON models.

RED

Random Early Dropping. If internal buffers are approaching full, packets may be dropped.

reserve32

A special region of free memory which is low enough to have 32-bit physical addresses (the “shallow end” of the memory pool). This region is only used by 32-bit Linux processes (SE-UM 32-bit) which cannot access *kseg0*. Instead, they access system memory through memory mapped into *useg* (the *reserve32* area).

Reserved Download Block

The *Reserved Download Block* is an area which is reserved by the bootloader, and which is used to download the application. This area may be seen with the bootloader command `namedprint`.

Reserved Linux Block

Unlike Simple Executive applications, which can be loaded anywhere in memory, Linux is linked to run at specific physical addresses. The *Reserved Linux Block* is a block of memory which is reserved by the bootloader so that when the Simple Executive application’s in-memory image is created, the bootloader will not locate it in the area of memory Linux requires. If Linux is not loaded, this area of memory is reclaimed. This area may be seen with the bootloader command `namedprint`.

RNG Unit

Random Number Generator.

RX**Receive****scheduled**

A Work Descriptor that has been assigned to a core is considered to be *scheduled*.

scratchpad

Core-local memory. Part of the per-core data cache (Dcache) may be set aside for IOBDMA operations and *scratchpad* memory. The scratchpad area can be accessed from the core. When an IOBDMA operation is performed, the result of the operation is stored in the scratchpad. An example IOBDMA operation is `cvmx_fpa_alloc_async()`, which starts an IOBDMA operation. This operation will asynchronously get the address of a free buffer from the FPA, and store the buffer's address in scratchpad memory. See IOBDMA for more information.

SDK

Software Development Kit. The SDK is defined to be two packages: the base SDK, and OCTEON Linux. All other RPM packages are not included in the definition of the SDK. The SDK and other RPM files are supplied by Cavium Networks. They include the compiler, libraries, API source files, example code, and documentation needed to develop applications for the OCTEON processor family.

Security Coprocessor

The *Security Coprocessor* is a special coprocessor used to accelerate security algorithms. There is one Security Engine per core. This unit is not available in all OCTEON models. This unit is sometimes called the Crypto Unit.

Security Engine

Same as Security Coprocessor.

SEGBITS

In the 64-bit virtual address map, the high two bits of the virtual address (<63:62>) are used to select one of four segments. These address bits are always translated by the hardware, not the operating system. Of the remaining 62 bits in the virtual address, some of the high bits are ignored if the processor does not support that many virtual address bits within a segment (*SEGBITS*). On OCTEON, SEGBITS equals 49, so only bits <48:0> of the virtual address define the address space within the segment. The remaining bits (<61:49>) are ignored.

segment

In MIPS architecture, the address space is divided into segments: it is not an undifferentiated virtual address space. For 64-bit address space: *xuseg*, *xsseg*, *xkseg*, *xkphys*. For 32-bit address space: *useg*, *sseg*, *kseg0*, *kseg1*, *kseg3*. *Cvmseg* is in *kseg3* and *xkseg*.

SE-S

Simple Executive provides an API to the hardware units. Simple Executive may be run Standalone (*SE-S*), which means without the support of an operating system.

SE-UM

Simple Executive provides an API to the hardware units. Simple Executive may be run Standalone (*SE-S*), or as a user-mode (*SE-UM*) application on an operating system such as Linux.

Simple Executive

Simple Executive is the name for the code supplied with the SDK which provides an API to the hardware units. This code may be run as a *SE-S* or *SE-UM* application, or called from an operating system or driver.

spinlock

Spinlock is an industry-standard term. A spinlock is a synchronization device. A thread of execution will remain blocked in the spinlock lock routine until it is able to obtain the lock. If this were a highly contested spin lock, the wait period would be non-deterministic. When possible, it is preferable to use packet-linked locks instead of spinlocks.

sseg

The *sseg* segment is in the supervisor address space (mapped) in the 32-bit MIPS virtual memory address map. On cnMIPS, this segment is usually not used.

SSO Unit

Schedule/Synchronization and Order Unit. This unit manages packet scheduling and ordering.

Switch Complete Bit

The *Switch Complete Bit* is used for communication between the core and the SSO. When the core performs the *switch_tag* operation, it requests the SSO change the Work Descriptor's tag tuple to the new tag tuple. The tag switch does not necessarily complete immediately. The Cavium Networks-specific hardware register, 30, contains the status of the *switch_tag* request. The RDHWR instruction is used to read this register. Note this register is not the same as general purpose register 30. The value of this register is referred to as the Switch Complete Bit. When the core requests a tag switch, the core's Switch Complete Bit is set to zero. When the tag switch is complete, the SSO sets the Switch Complete Bit to one

switch tag

A core may perform a *switch_tag* operation to change the Work Descriptor's tag type, tag value, or both. Both may be changed with the same operation. A separate operation may be performed to change the Work Descriptor's Group value.

tag switch

See *switch tag*.

tag tuple

The combination of tag type and tag value is a *tag tuple*. Software can change the tag tuple to move packets from the same flow through different processing phases, including ATOMIC sections when packet-linked locking is needed.

tag type

The Work Descriptor's *tag type* is one of: ORDERED, ATOMIC, or NULL. The first tag type is set by the PIP/IPD when the packet is received.

tag value

The *tag value* is a 32-bit number, usually containing the port number and the tuple hash value. The first tag value is set by the PIP/IPD when the packet is received.

target In-Flight Queue

The *target In-Flight Queue* is the In-Flight Queue the Work Descriptor is on when the tag switch request completes. This queue corresponds to the requested tag tuple.

TCP/IP Acceleration Pseudo Block

The *TCP/IP Acceleration Pseudo Block* is a pseudo block containing the Timer Unit and the FPA Unit, which are both used to accelerate TCP/IP packet processing.

TIM Unit

Timer: This unit provides timers, which can be used for TCP timeouts as well as other more general purposes.

TLB

Translation Look-aside Buffer.

TLB hit

The requested virtual address is found in the TLB.

TLB miss

The requested virtual address to physical address mapping is not found in the TLB. A *TLB miss* exception will occur. The TLB miss handler is responsible for handling the exception, usually by looking up the virtual to physical address mapping, and loading the information into the TLB.

tuple hash value

The *tuple hash value* is commonly used in packet processing. A Cyclic Redundancy Check (CRC) algorithm is used to reduce the 13-byte 5-tuple to 16 bits. The resultant 16-bit value is referred to as the tuple hash value. The tuple hash value is used to identify the flow. The PIP is responsible for reading the packet header and computing the tuple hash value.

TX

Transmit.

useg

The *useg* segment is the user address space (mapped) in the 32-bit MIPS virtual memory address map. On cnMIPS, SE-S 64-bit applications run in kernel mode, but are mapped into *useg*.

virtual address

On MIPS processors, the addresses used by a program are always *virtual addresses*. Virtual addresses are not the same as physical addresses, even if their 64-bit values are the same. Virtual addresses are always interpreted differently by the hardware (segment selector, ignored bits, and SEGBITS). C and C++ programs must therefore always use virtual addresses (pointers), not physical addresses, when accessing memory.

way partitioning

The L2 cache ways can be partitioned among the cnMIPS cores and the I/O sub-system. This is referred to as *way partitioning*. This OCTEON feature enables intelligent management of the L2 cache to minimize cache pollution and the resulting loss of performance. See the *HRM* for details.

WD

See Work Descriptor.

Work Descriptor

WD. The SSO contains internal memory. Part of the internal memory has been used to create a limited number of *Work Descriptors*. Each Work Descriptor contains the key information needed by the SSO to schedule the work on a core, and to keep the packets in the correct order. The key fields in the Work Descriptor are: WQE pointer, tag value, tag type (TT), QoS and Group (Grp). (The Group field is not discussed in this example.)

Work Group

See Group.

Work Queue

A QoS Input Queue is also referred to in our manuals as a *work queue*.

Work Queue Entry

The *Work Queue Entry* (WQE) is a data structure which contains the tag type, tag value, QoS value, Group, and a pointer to the Packet Data Buffer. The IPD allocates the WQE Buffer from the FPA. The PIP/IPD fill in the WQE fields, then sends the WQE pointer to the SSO, using the `add_work` operation.

WQE

See Work Queue Entry.

Write Buffer

The *Write Buffer* is a special buffer located in each core. Writes from the core go to this buffer. The Write Buffer is used to minimize the number of CMB writes. When the Write Buffer is flushed, the buffer contents are written simultaneously to the Dcache and the L2 cache. The Write Buffer is periodically flushed. A Write Buffer flush can also be requested by the core (for instance, when processing on that section of data is complete). The MIPS64 ISA has a SYNC instruction for controlling memory order. Cavium Networks added several variations which provide finer memory order control for higher performance.

xkphys

The *xkphys* segment is the kernel address space in the 64-bit MIPS virtual memory address map. It is an unmapped address space: a window into the physical address space (system memory and I/O space). On cnMIPS, SE-UM 64-bit applications may be allowed to access *xkphys* addresses. SE-S 64-bit applications always have access to *xkphys* addresses because they run in kernel mode. Accesses to system memory are always cached. Accesses to I/O space are never cached.

xkseg

The *xkseg* segment is the kernel address space (mapped) in the 64-bit MIPS virtual memory address map. On cnMIPS, the *xkseg* segment contains the OCTEON-specific *cvmseg* segment. User-mode access is allowed to *cvmseg*.

xsseg

The *xsseg* is the supervisor address space (mapped) in the 64-bit MIPS virtual memory address map. This segment is usually not used in OCTEON cnMIPS.

xuseg

The *xuseg* segment is the user address space (mapped) in the 64-bit MIPS virtual memory address map. On cnMIPS, SE-S 64-bit applications run in kernel mode, but are mapped to *xuseg*.

ZIP Unit

The ZIP Unit is a compression/decompression engine. This unit is not available in all OCTEON models.



Corporate Headquarters

Cavium Networks, Inc.
805 East Middlefield Road
Mountain View, CA 94043
USA
Tel: 650-623-7000
Fax: 650-625-9761
www.caviumnetworks.com