Software Overview

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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

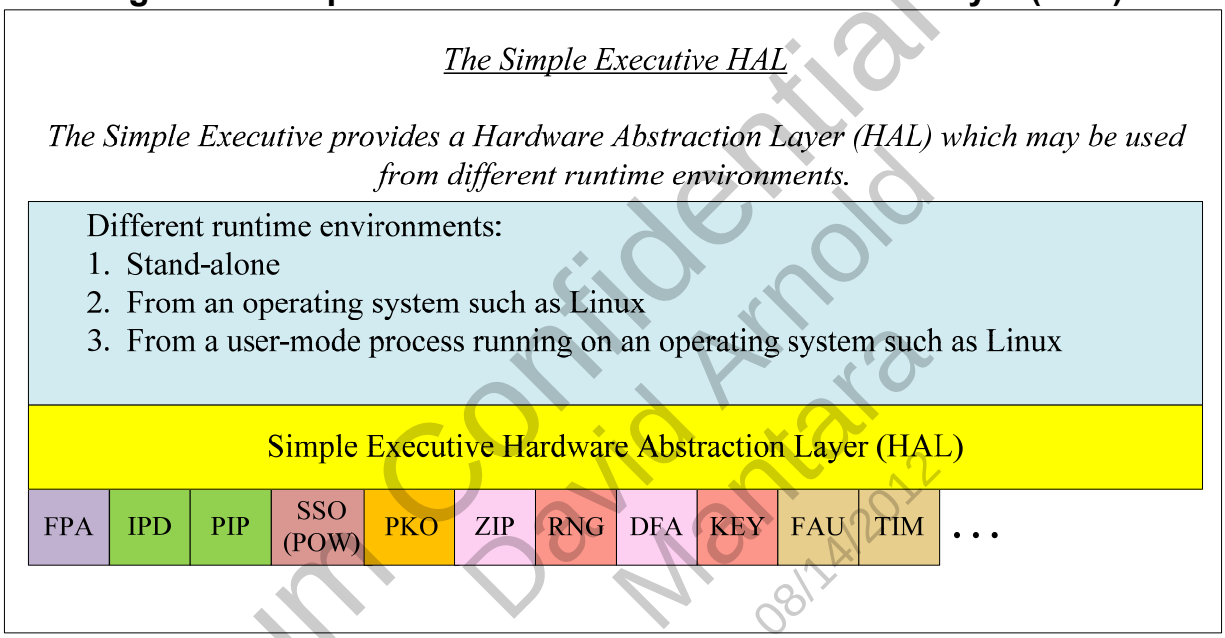
Instruction Categories
Unsigned byte add.
Bit-test branches
Cache manipulation instruction
Instructions to use the in-core 3DES coprocessor (must have the Security Engine)
Instructions to use the in-core AES coprocessor
Instructions to use the in-core CRC coprocessor
Instructions to use the Galois Field 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

Note that different OCTEON models have different features: some hardware units are not available on all models.	
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

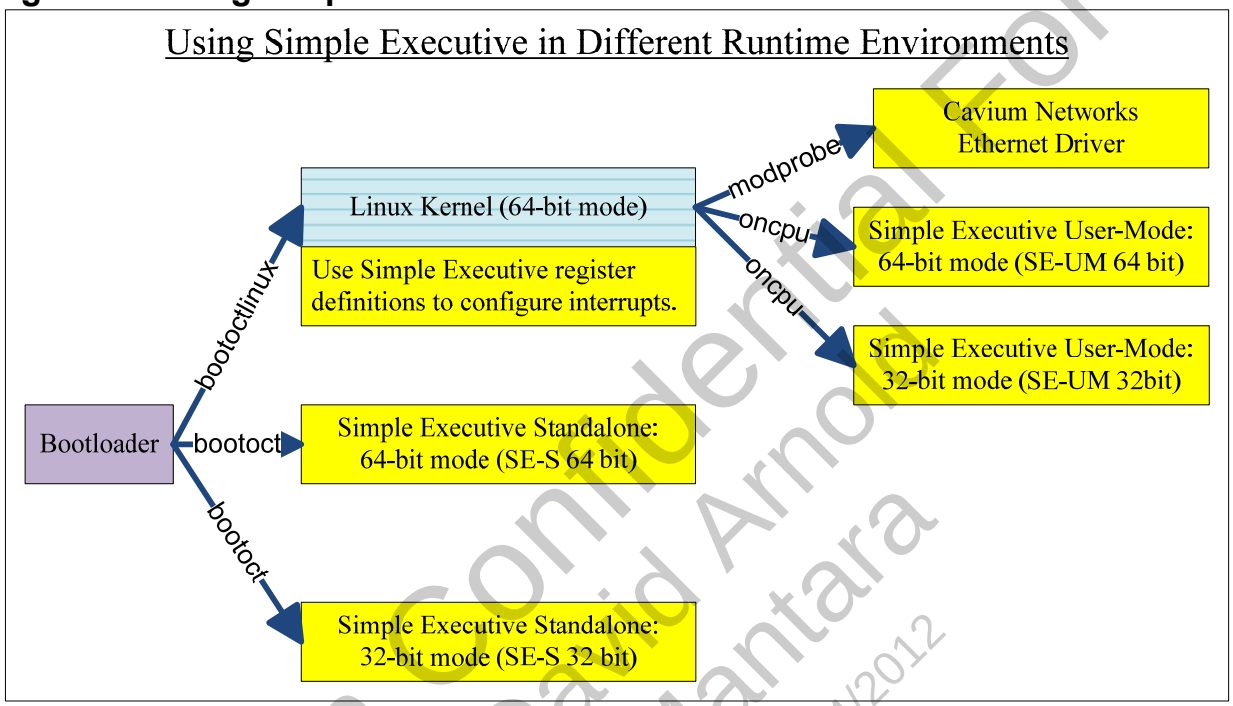
Note that different OCTEON models have different features: some functions are not supported on all models.	
Synchronization Support	
Packet-linked Locks	Packet-linked locks are implemented by the SSO, and provide ATOMIC access to critical regions.
Basic Spinlocks	Non-recursive spinlocks.
Recursive Spinlocks	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.

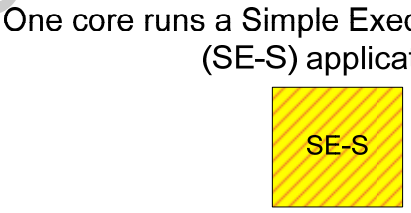
4.2 *Simple Executive*

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.

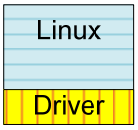
Figure 3: Simple Executive Standalone Application (SE-S)



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

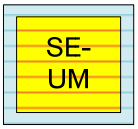
One core runs Linux with Cavium Networks Ethernet Driver



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-interrrupt-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/exectutive/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 "eth*N*". SPI4000 ports are devices "spi0" through "spi*N*". XAUI devices are "xaui0" through "xaui*N*". (*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 `passthough` over NFS will not work because passthough 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
#ifdef __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 `passthough`) 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

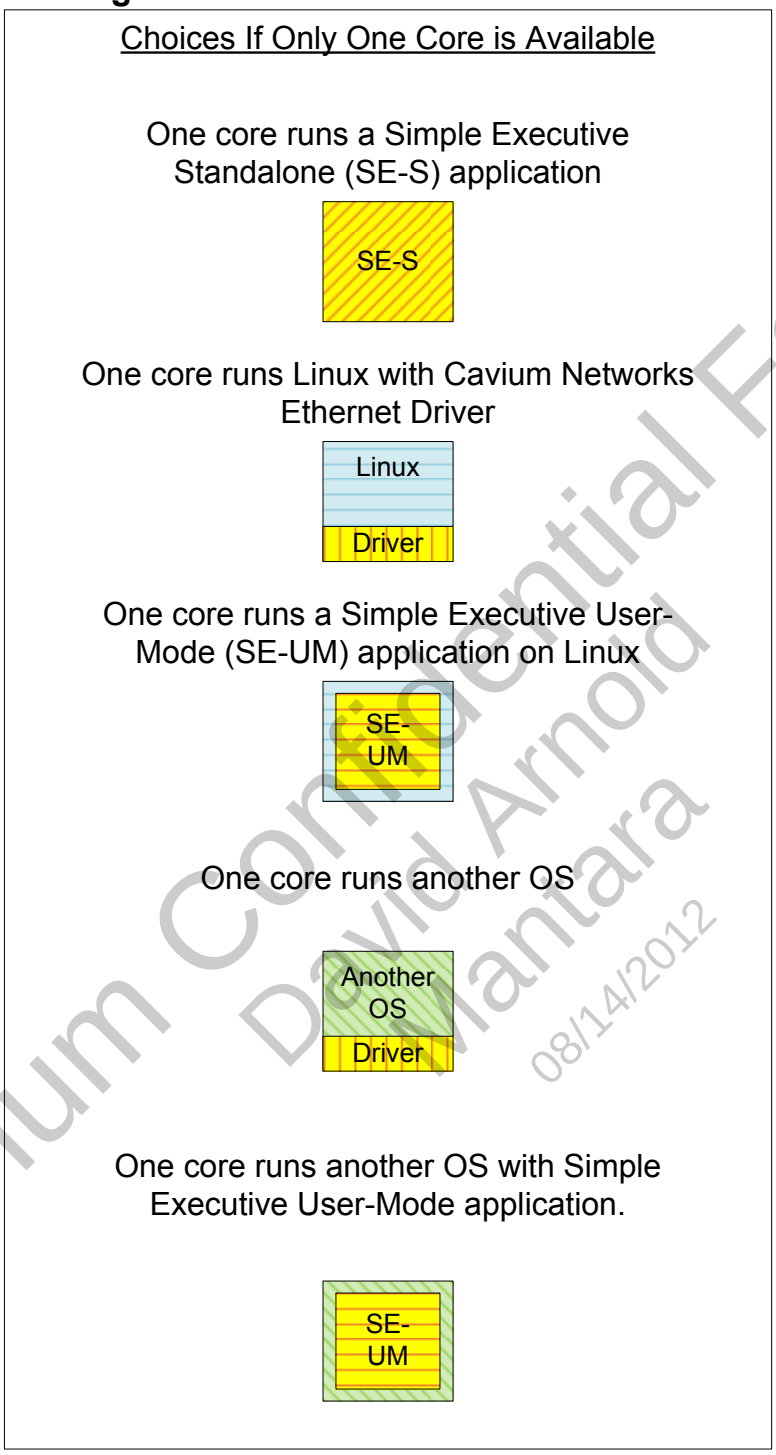
The following figures show chips running combinations of runtime environments, without showing the control-plane/data-plane configuration.

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



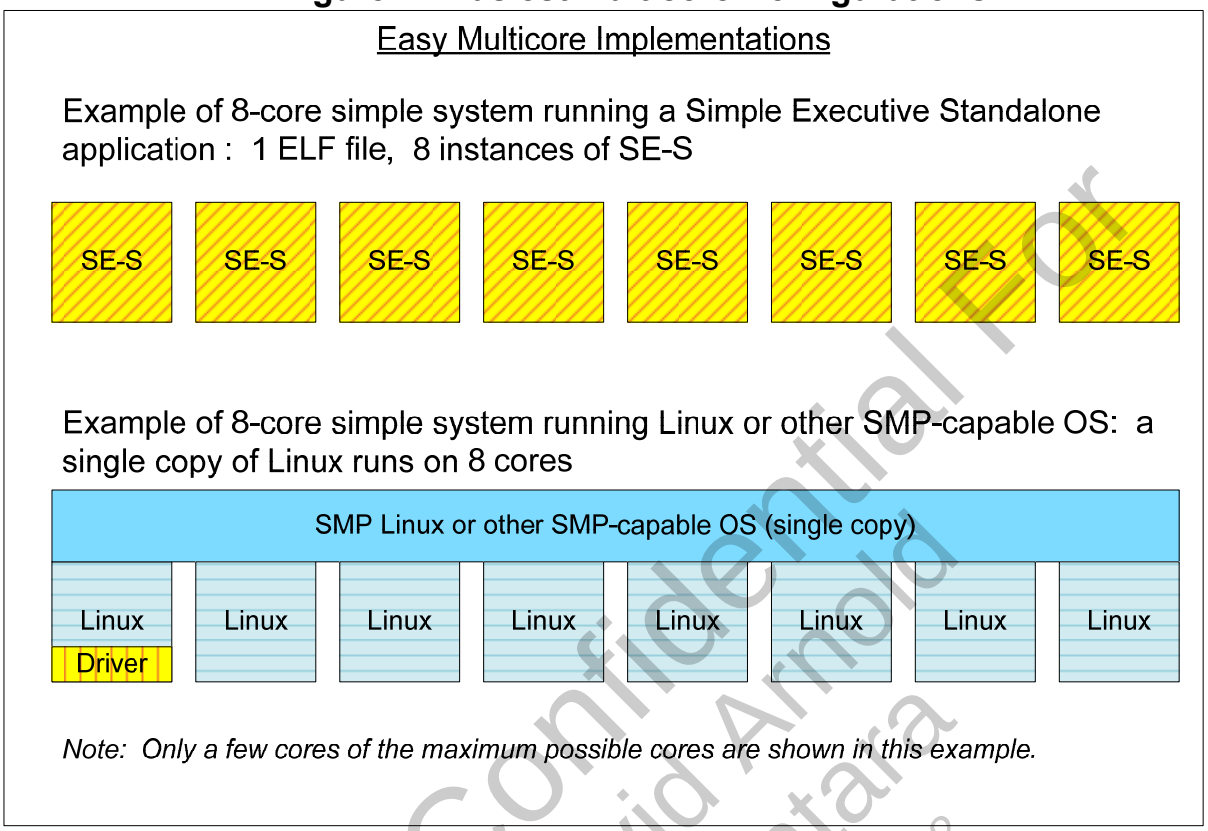
5.2 *Multicore Runtime Choices*

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.

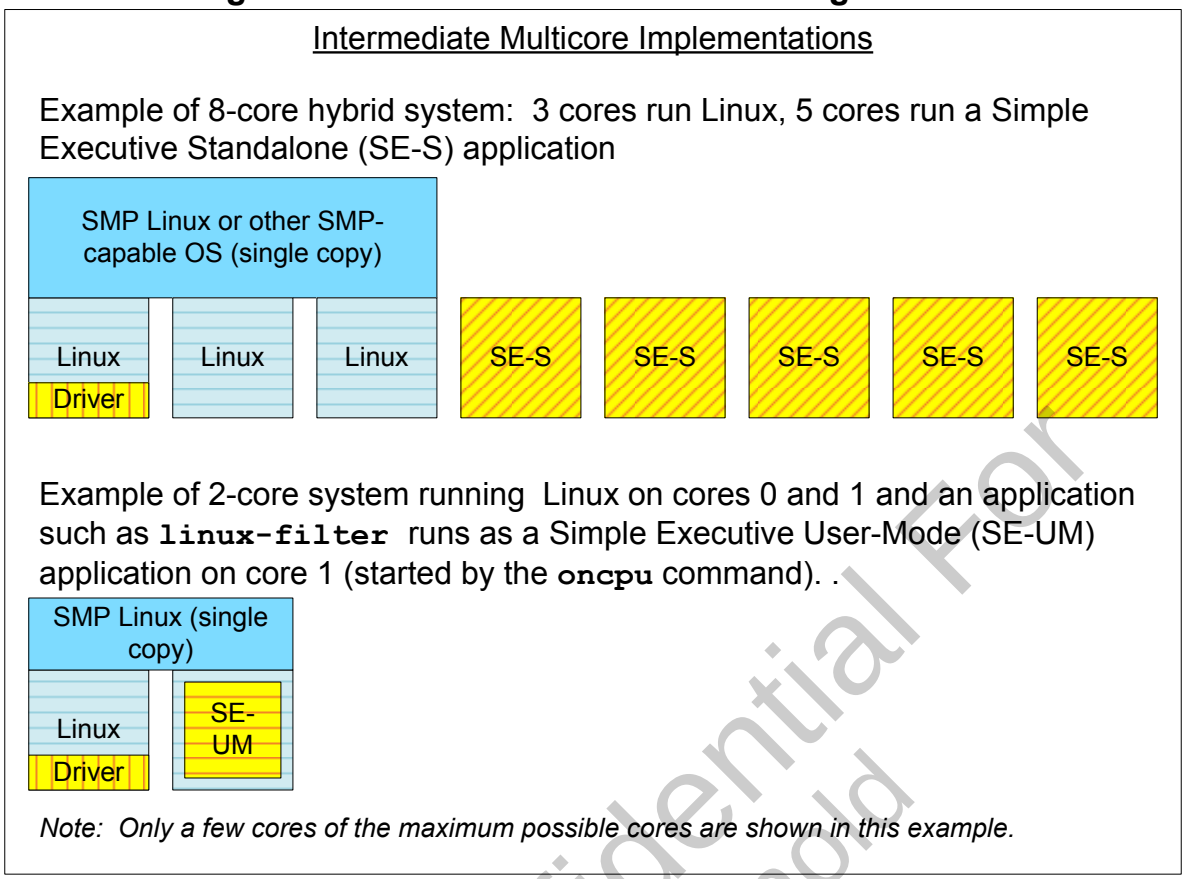
Figure 7: Easiest Multicore Configurations



5.2.2 Intermediate Configurations

The following configurations in the midrange of complexity to implement.

Figure 8: Intermediate Multicore Configurations

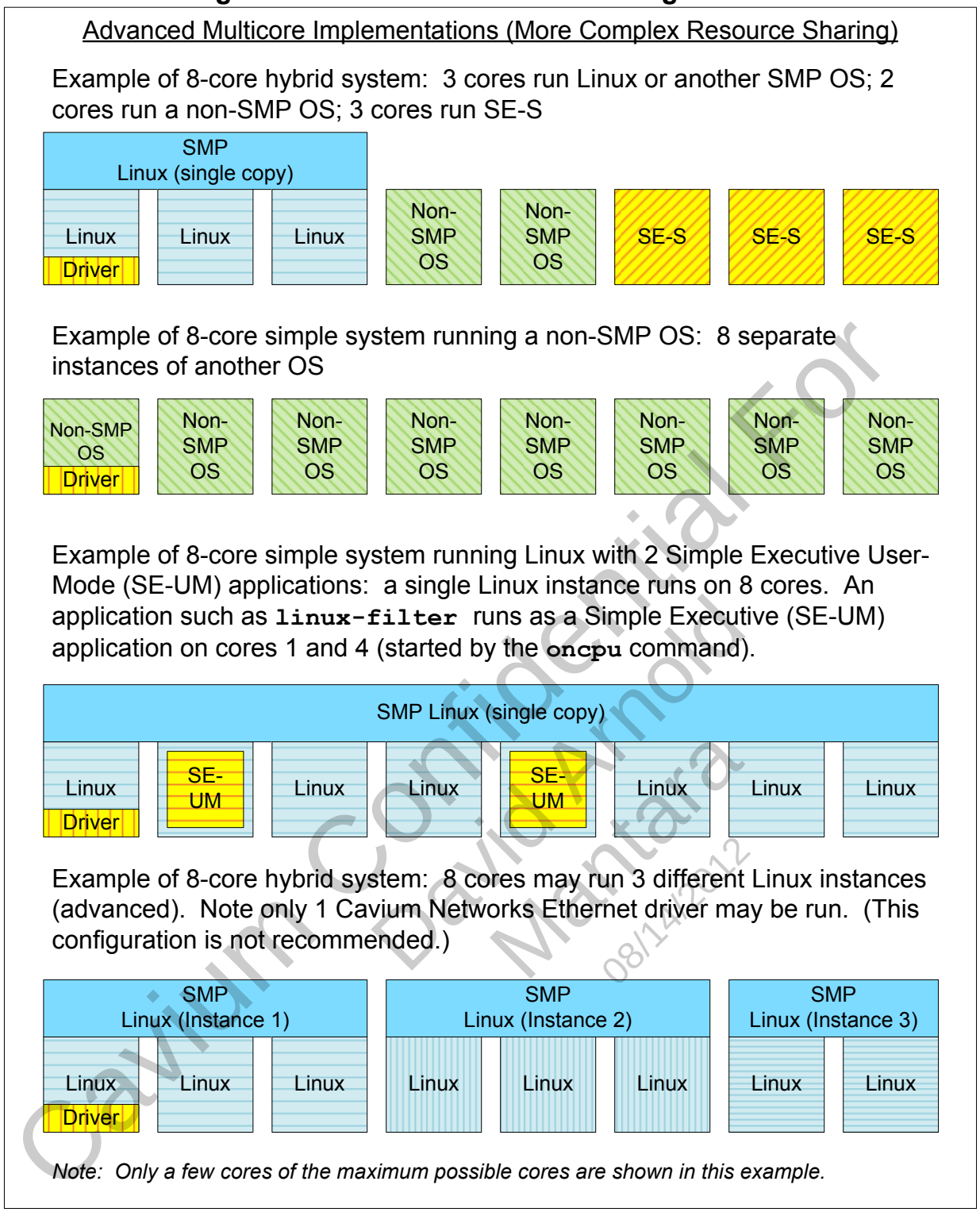


Note: Although Linux is usually run on core 0, this is not a requirement.

5.2.3 Advanced Configurations

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

Figure 9: Advanced Multicore Configurations



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	
`__cvmx_app_init()`	Application entry point. Defined in `cvmx-app-init.c`.
`main()`	Defined in applicaton code such as `linux-filter.c`. Called after `__cvmx_app_init()`.
`cvmx_user_app_init()`	Called by `main()`, defined in `cvmx-app-init.c`.

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 `cvmx-app-init-linux.c`.
`appmain()`	Defined in application code such as `linux-filter.c`: "`main`" is aliased to "`appmain`" by `common.mk`.
`cvmx_user_app_init()`	Called by `appmain()`, defined in `cvmx-app-init-linux.c`.

5.4 *Booting SE-S or SE-UM Applications*

To boot Simple Executive applications:

- for SE-S applications: `bootoct` 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.

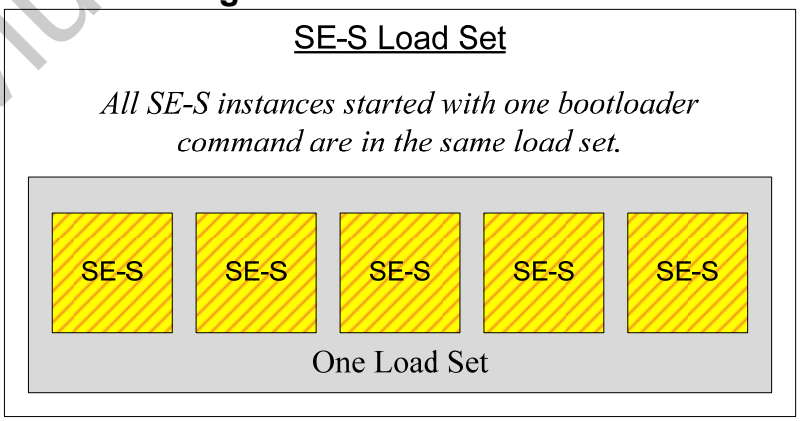
5.5 *Booting One ELF File on Multiple Cores: Load Sets*

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

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

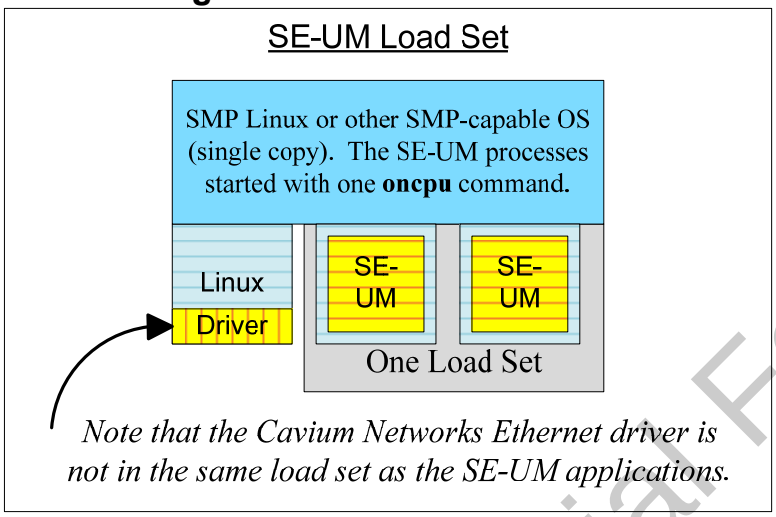
All cores booted by the same load command are in the same load set. The following figure shows cores running Simple Executive Standalone in a load set.

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

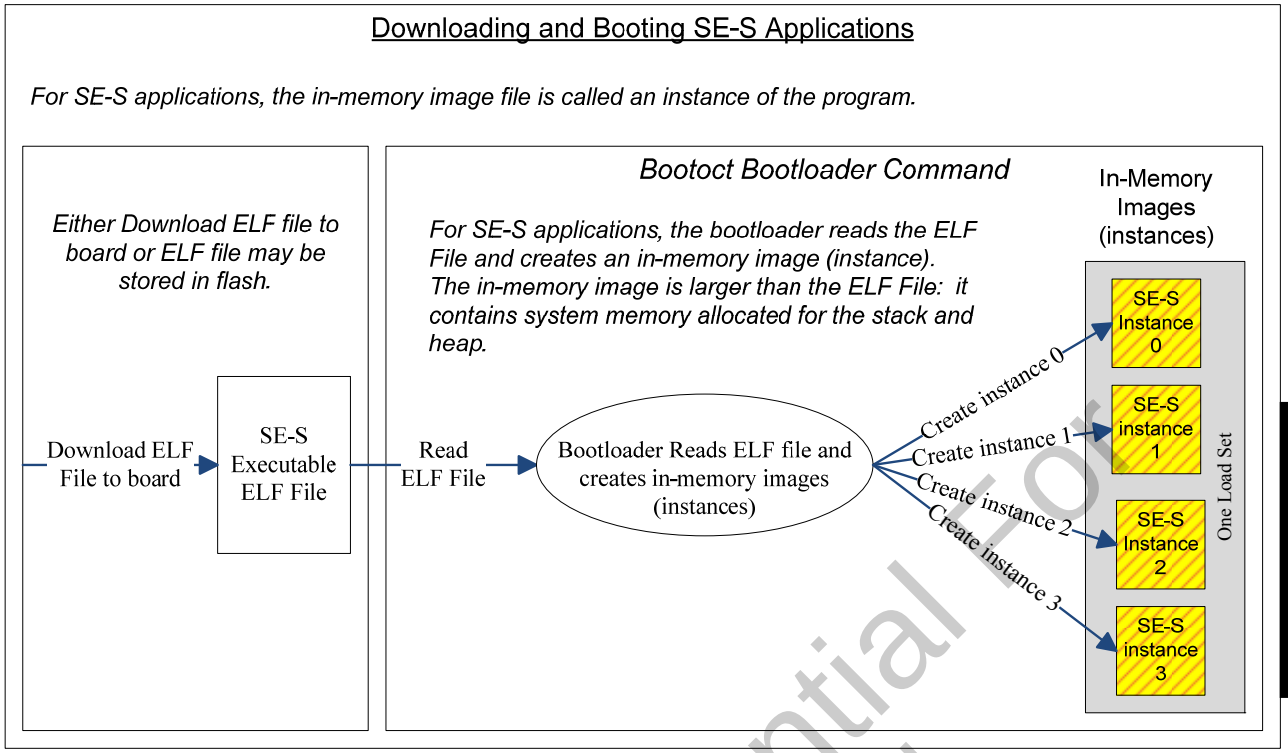


Load sets are discussed in more detail in Section 8.1.1.2 – "The *cvmx_shared* Section"

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

Starting Simple Executive Standalone applications with the `bootoct` 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 `Bootoct` Command



5.5.2 Starting Linux With the `bootoctlinux` Command

Linux may be started with the `bootoctlinux` 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 `bootoctlinux` 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 0x*XXXX*. 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.

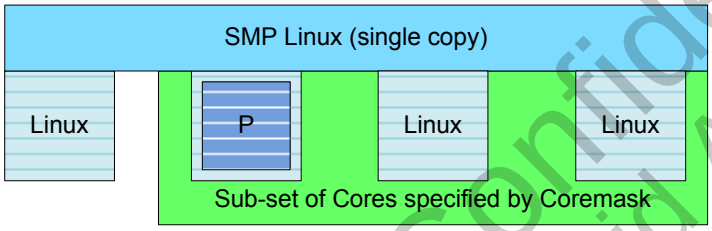
This special processing begins when the SE-UM application begins to run on a core. The function **`main()`** will **`fork()`** until a copy of the SE-UM process is running on every core which has the corresponding bit set in the **`coremask.`** The **`main()`** will call the function **`sched_setaffinity()`**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

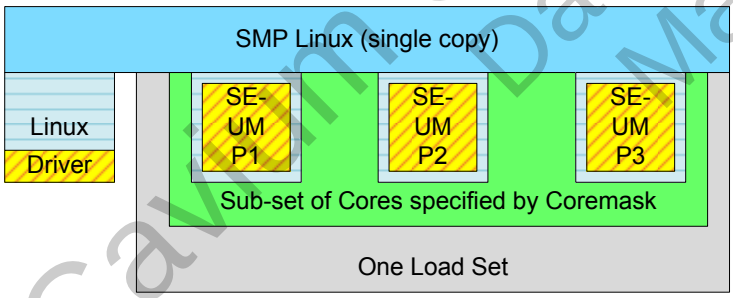
If the SE-UM application is started from the command line (for instance: **`target# ./linux-filter`**), then **`main()`** will start one instance of the SE-UM application on each SMP Linux core.

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



N cores, 1 process may run on any of N cores specified in the coremask.

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



N cores and N SE-UM processes.

The SE-UM **`main()`** *calls* **`fork()`** *to* **`fork`** *(N-1) processes, and calls* **`sched_setaffinity()`** *to bind each SE-UM process to a core.*

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

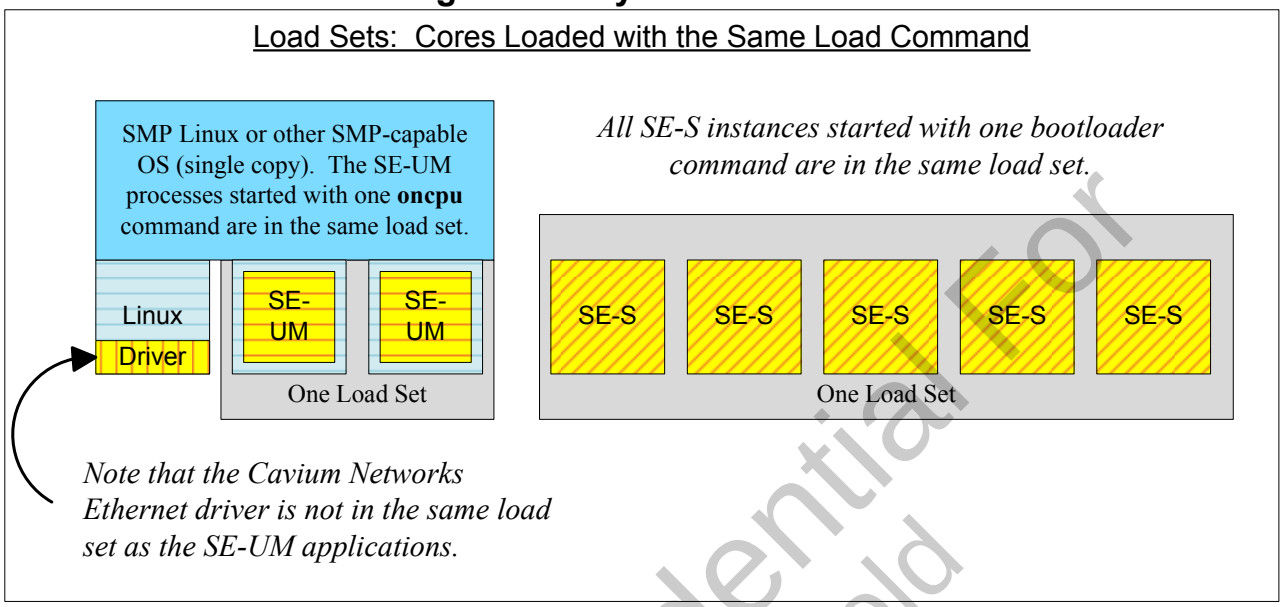
Details may be found in the SDK document "*Linux Userspace on the OCTEON*" in the section "*Controlling Core Affinity With oncpu*".

SW OVERVIEW

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\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.

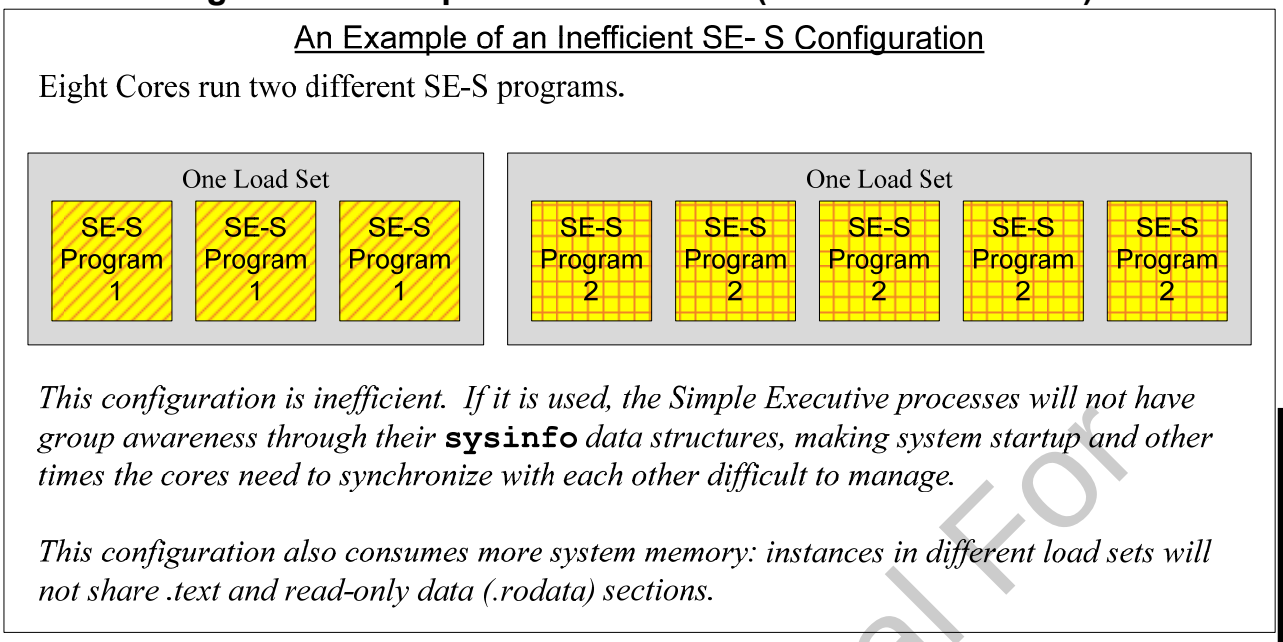
5.7.3 SMP Linux Synchronization

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".

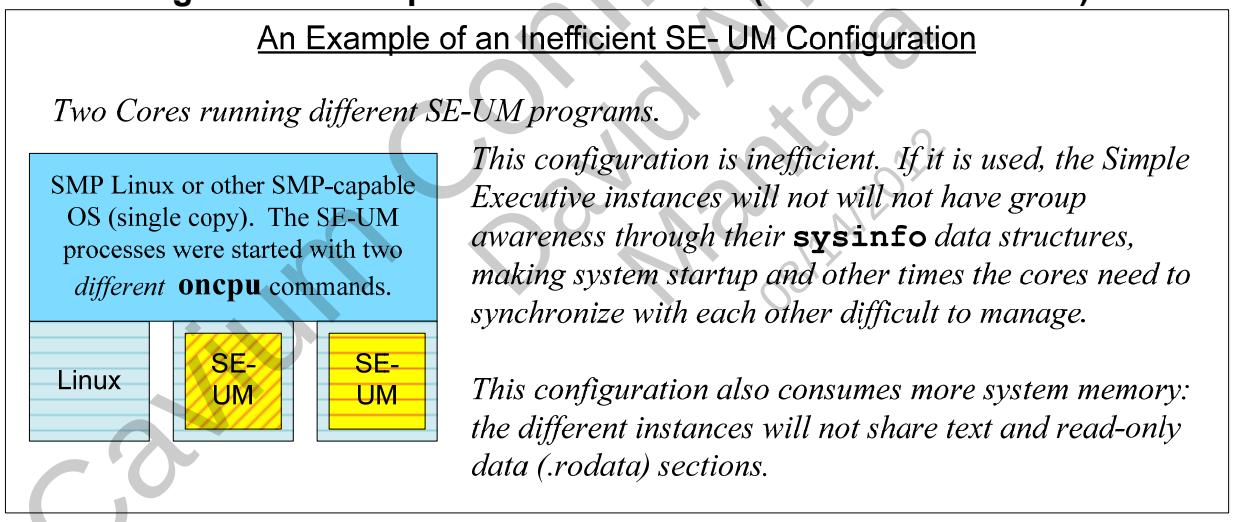
Figure 15: Multiple SE-S ELF Files (Not Recommended)



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)



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.

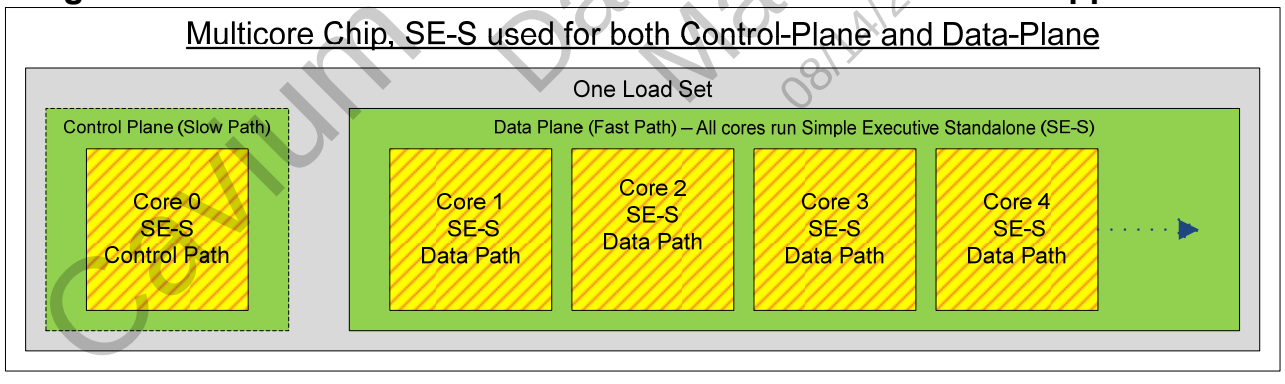
SE-S applications may be used for both control plane and data plane. SE-S applications provide the lowest overhead and highest potential for scaling. The next best solution (a typical solution) is SE-UM for control plane and SE-S for data plane.

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.***

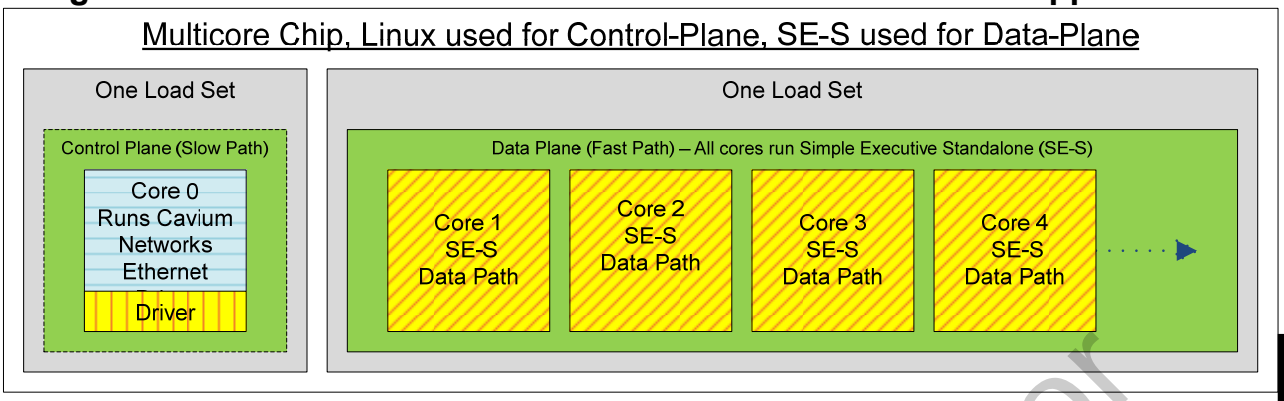
When running multicore applications, only one core does the initialization routine.

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



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`.

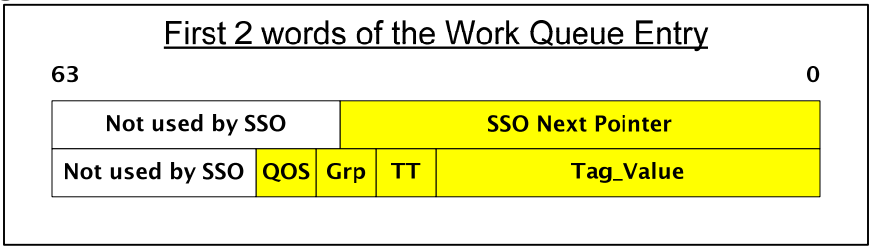
6.3 *Using Work Groups in Packet Processing*

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



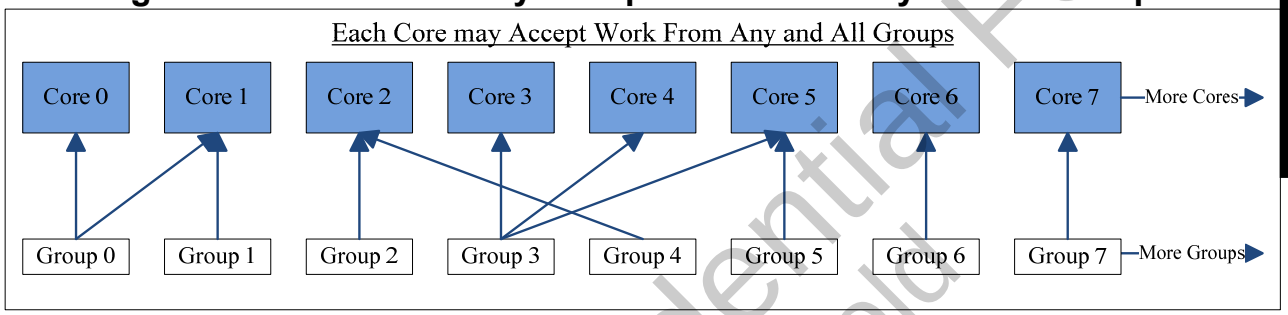
6.3.2 Configuring the Per-Core Group Mask in the SSO Scheduler

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 << 0. Group 15 is represented by 1 << 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

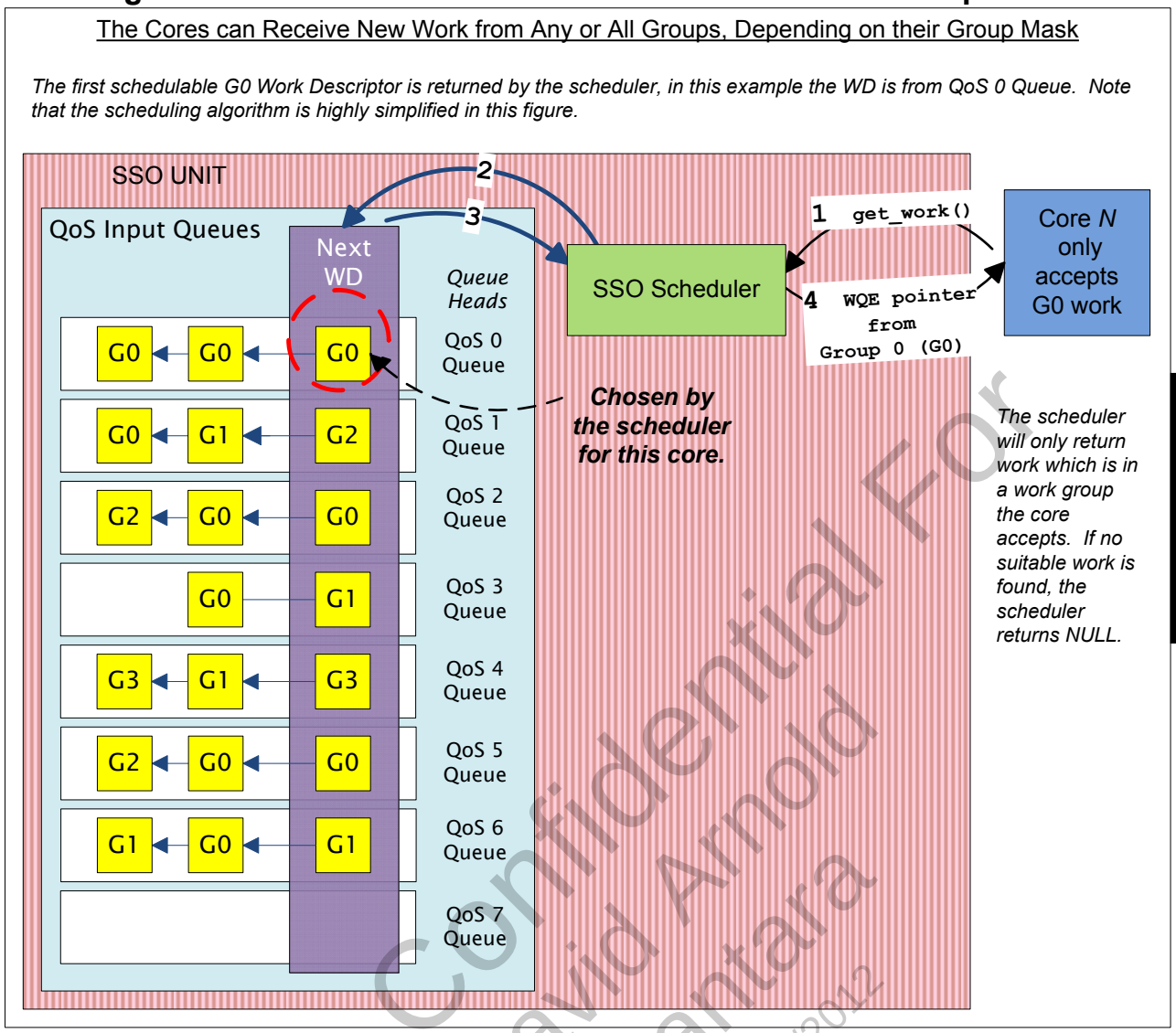
Group Mask [GRP_MSK)]																		Notes
Group		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Core...	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Only group 15 work is schedulable to this core.
	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Only groups 0 and 14 work is schedulable to this core.
	2																	
	3																	

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.

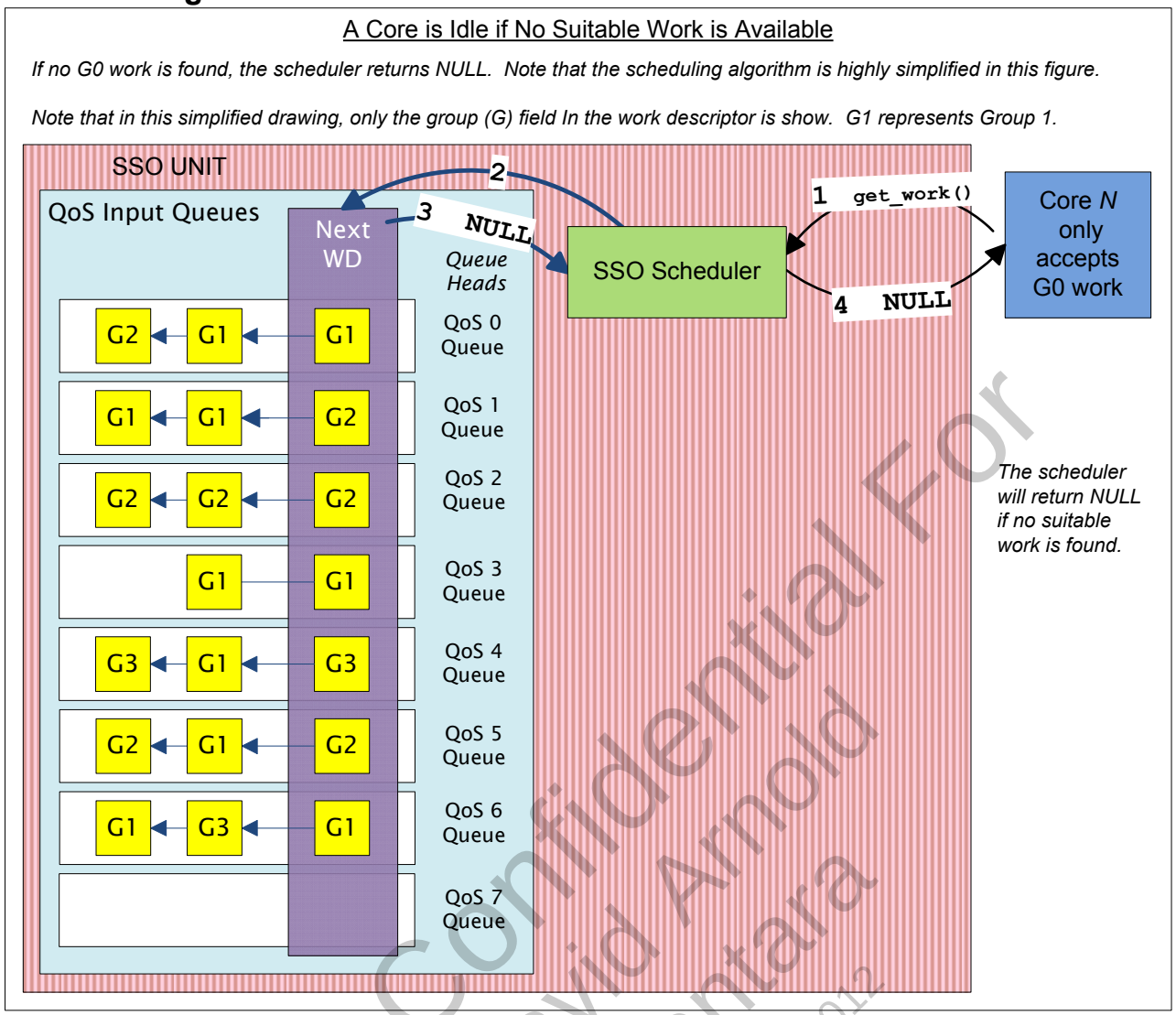
In the following figure, the core will receive the first schedulable group 0 work in from the highest priority QoS Input Queue (as viewed from the core's `QOS[N]_PRI` field). This is a highly simplified view of SSO scheduling based on groups.

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



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

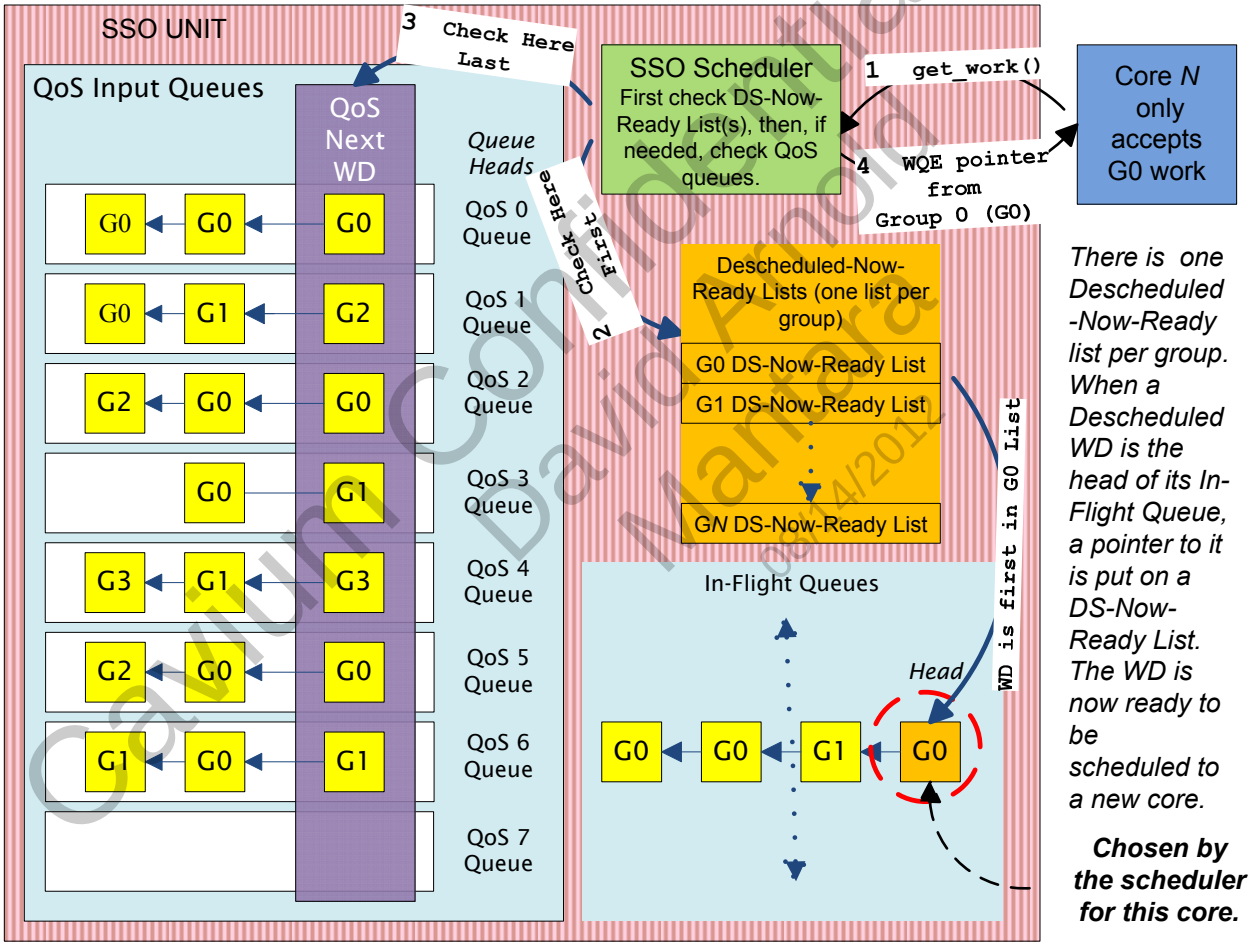
Scheduling Previously Descheduled Work

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:

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

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:

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

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

The available frame processing time, therefore, is:

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

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

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

And the CPU cycles available are:

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

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

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

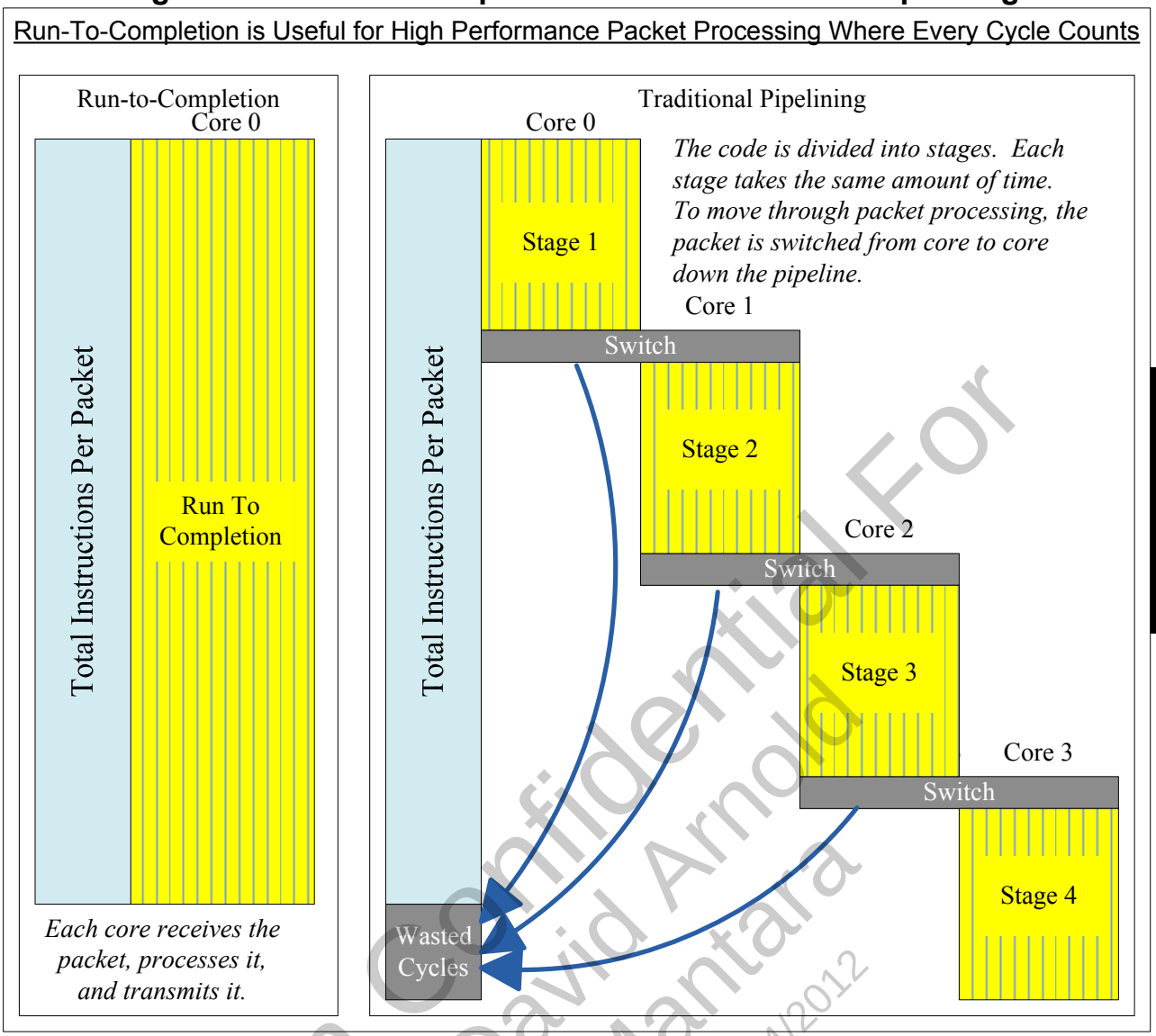
Note 1: The number of instructions executed per cycle may vary greatly depending on the application, compiler optimizations, cache sizes and cache utilization, and the locality of the code.

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

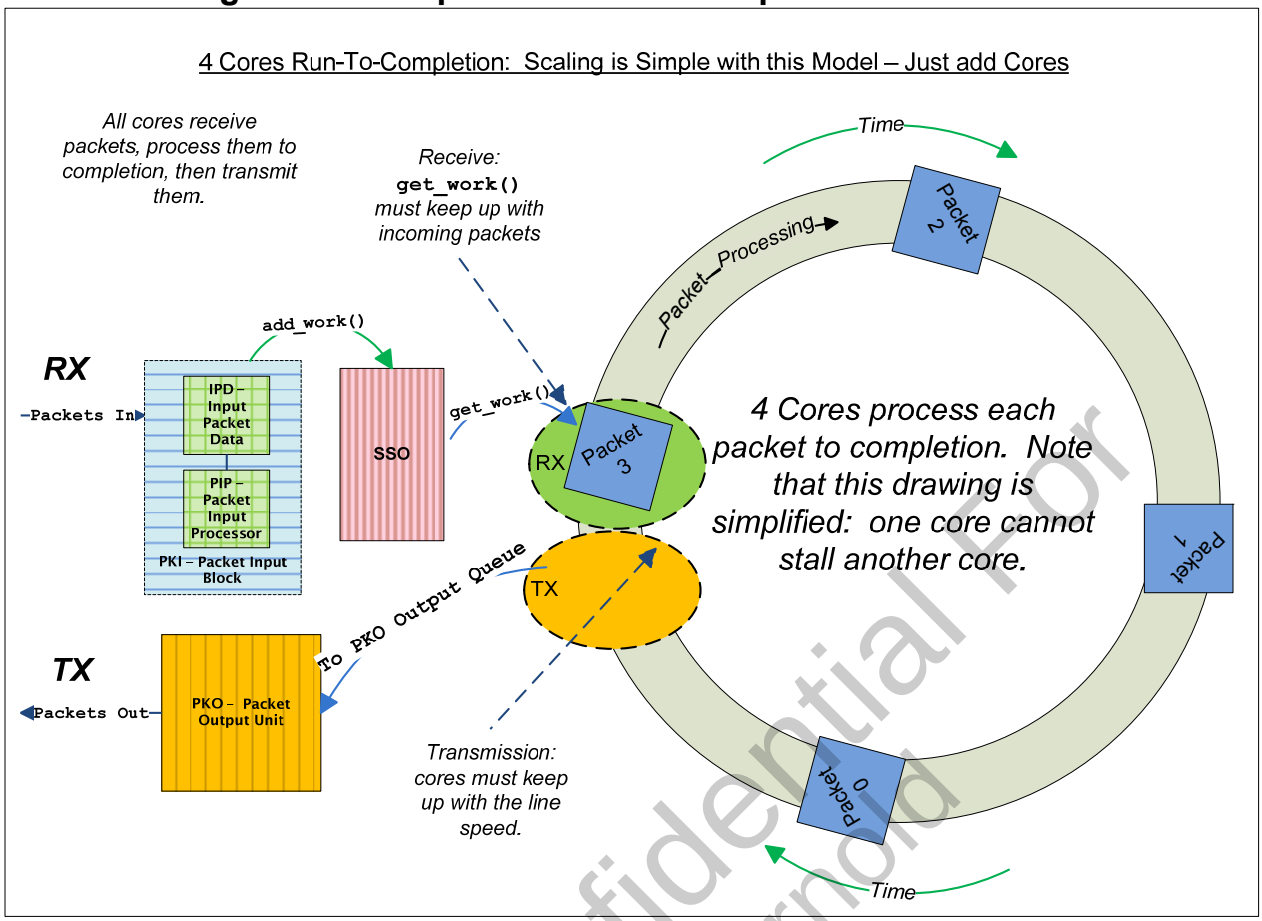


Note that as packet size increases, the packet rate drops dramatically. It is easier to keep up with the line rate when using larger packet sizes. There is a fixed per-packet overhead. Using a larger packet size will reduce the amount of overhead for the same data transfer.

6.4.3 Run-To-Completion

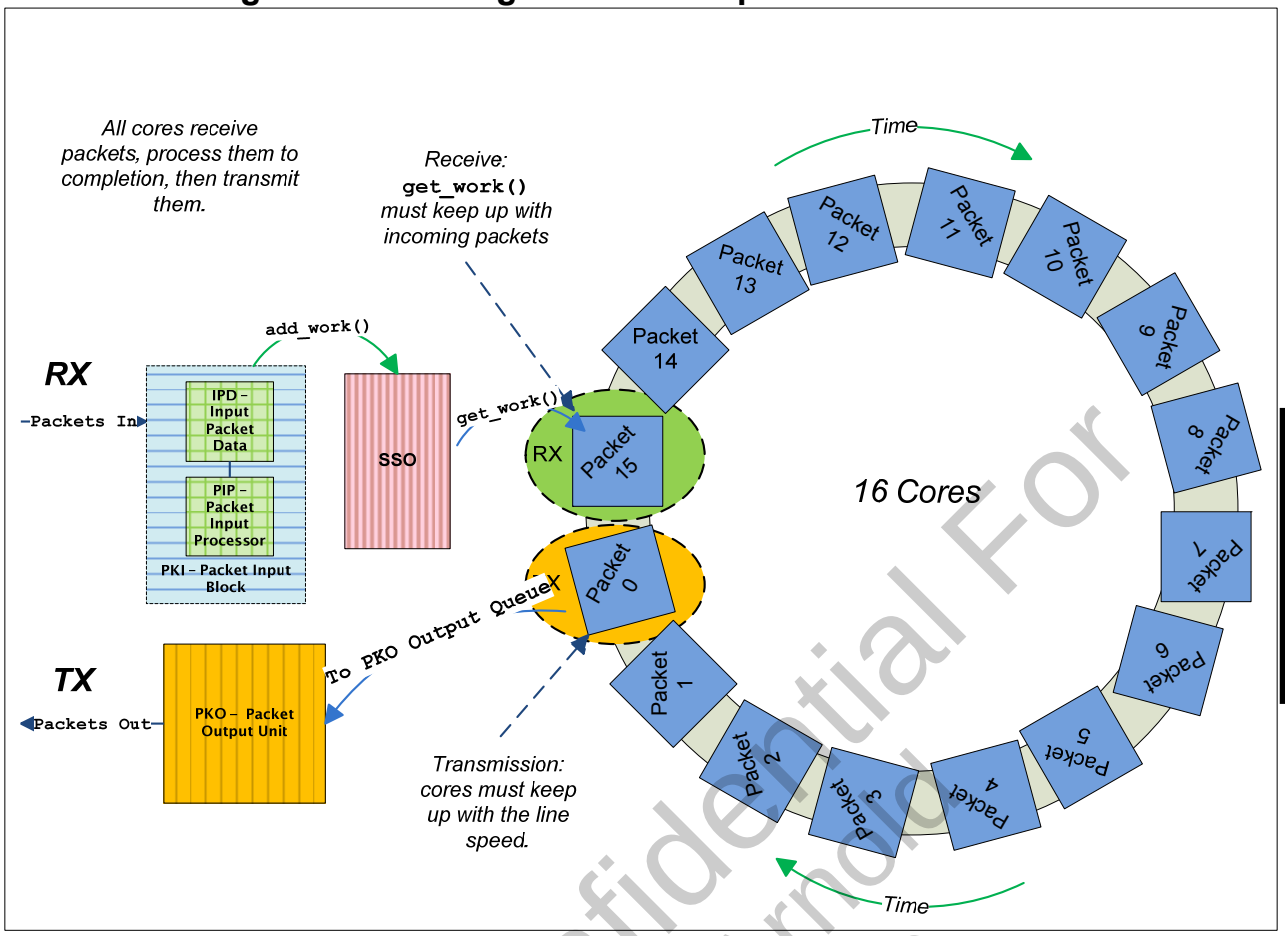
In run-to-completion architecture, each data-plane core runs the same application. Each core may receive new work and process it to completion. One core cannot stall packet processing for the system, only for the single packet involved.

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



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.

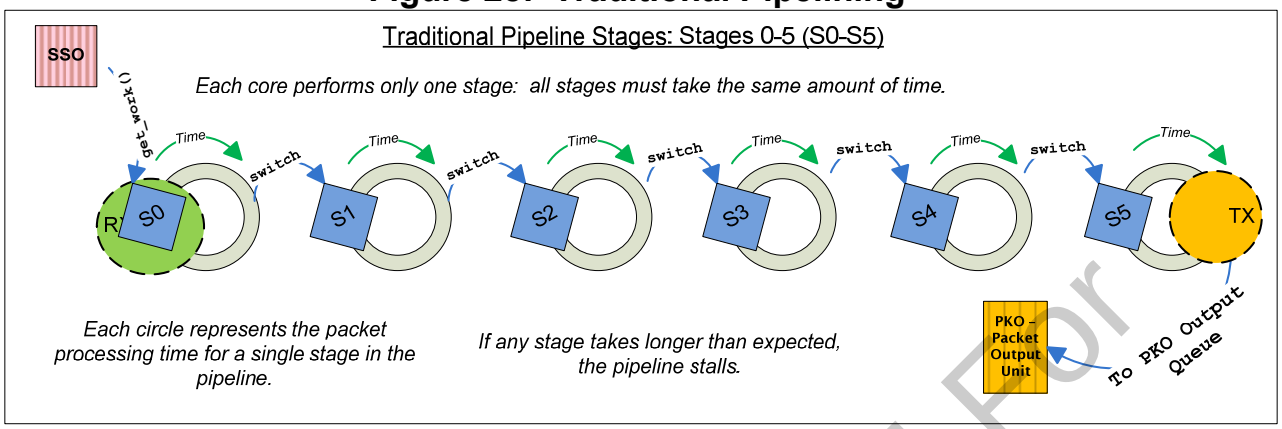
6.4.4 Traditional Pipelining

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.

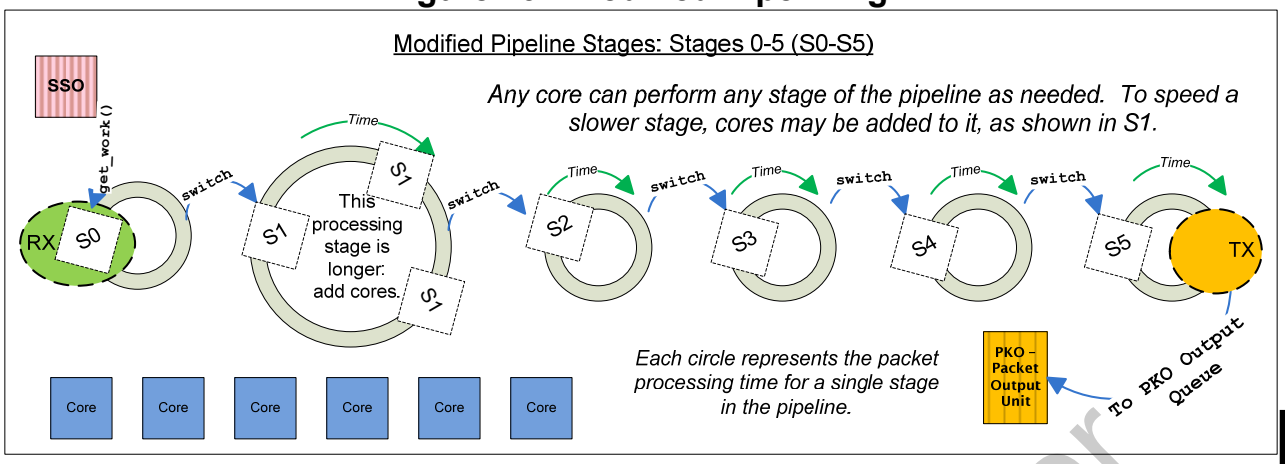
6.4.5 Modified Pipelining

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.

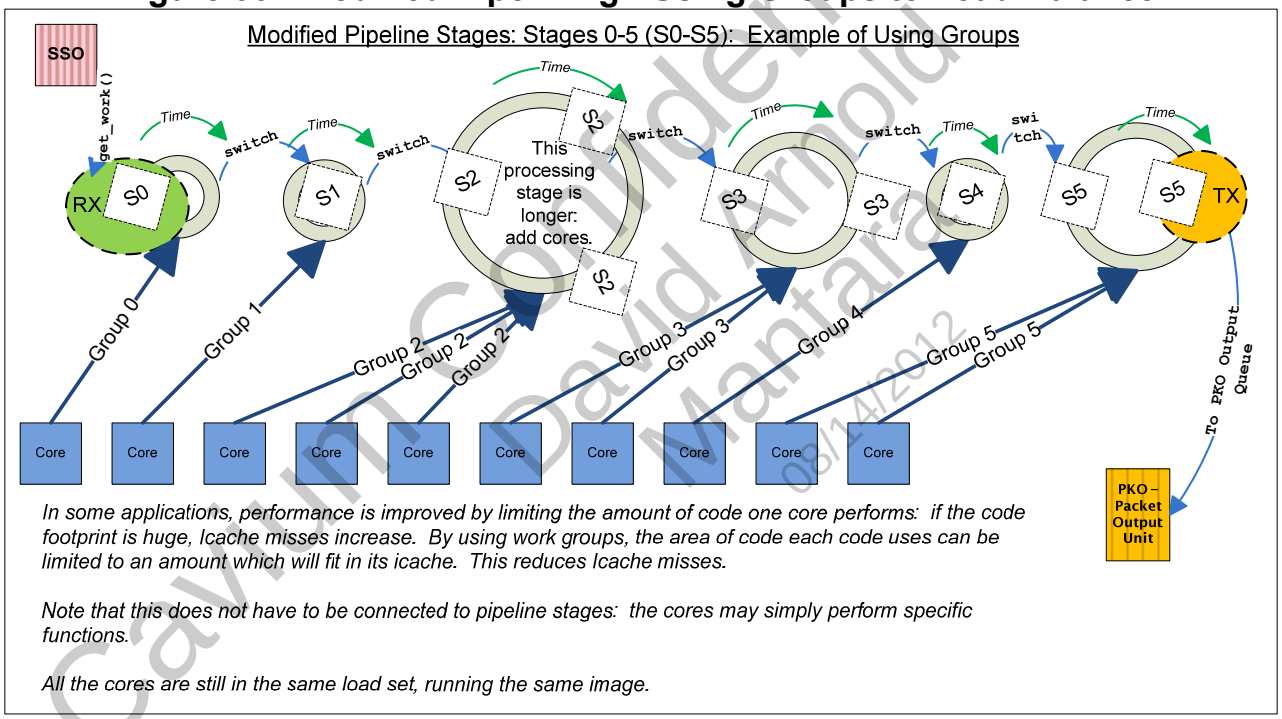
In this model, each core can run the same application. After the `get_work` operation returns a WQE pointer to the core, the core can execute the appropriate function based on the packet's work group.

Figure 29: Modified Pipelining



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.

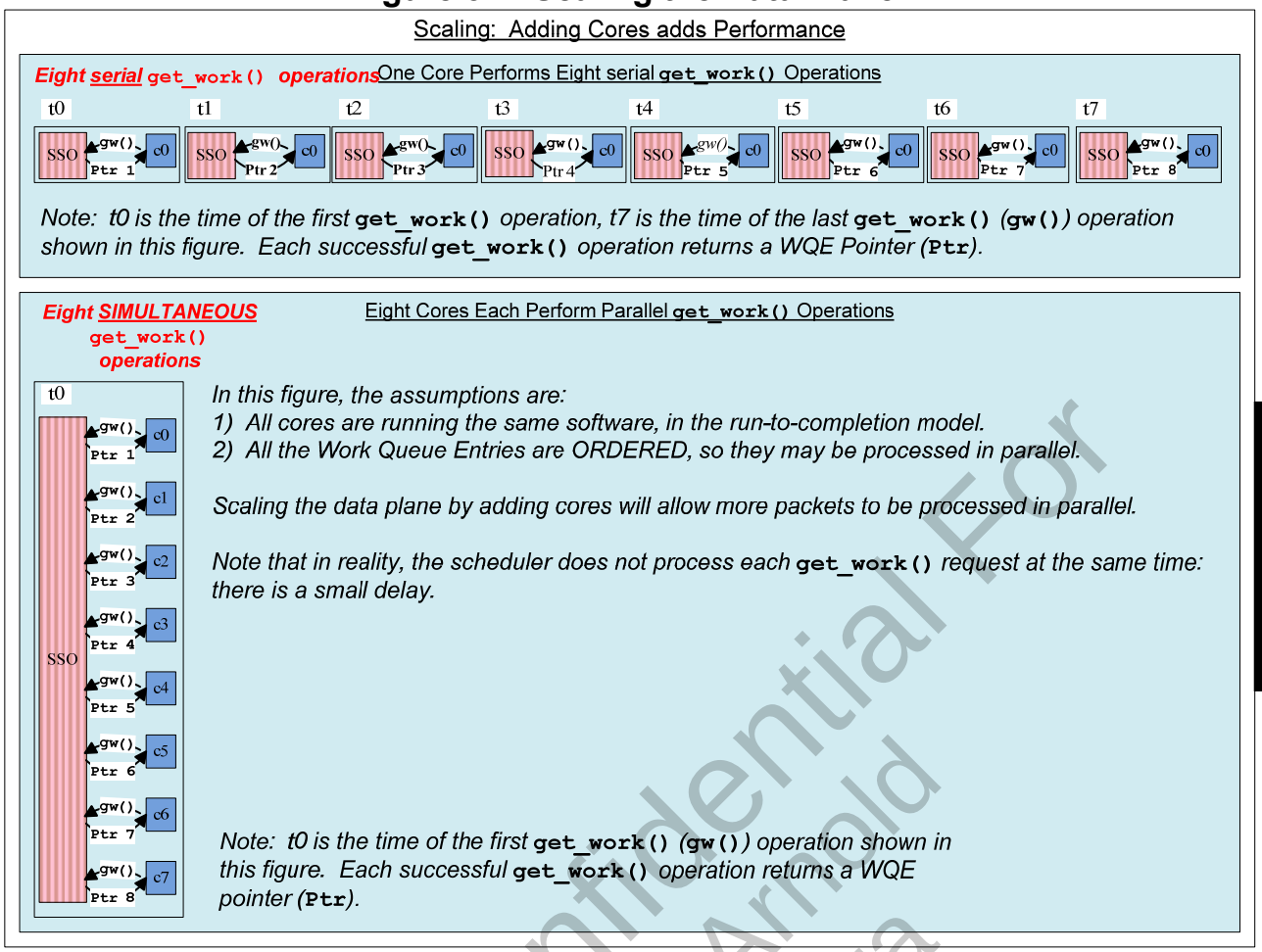
Key elements of a well-designed system:

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



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).

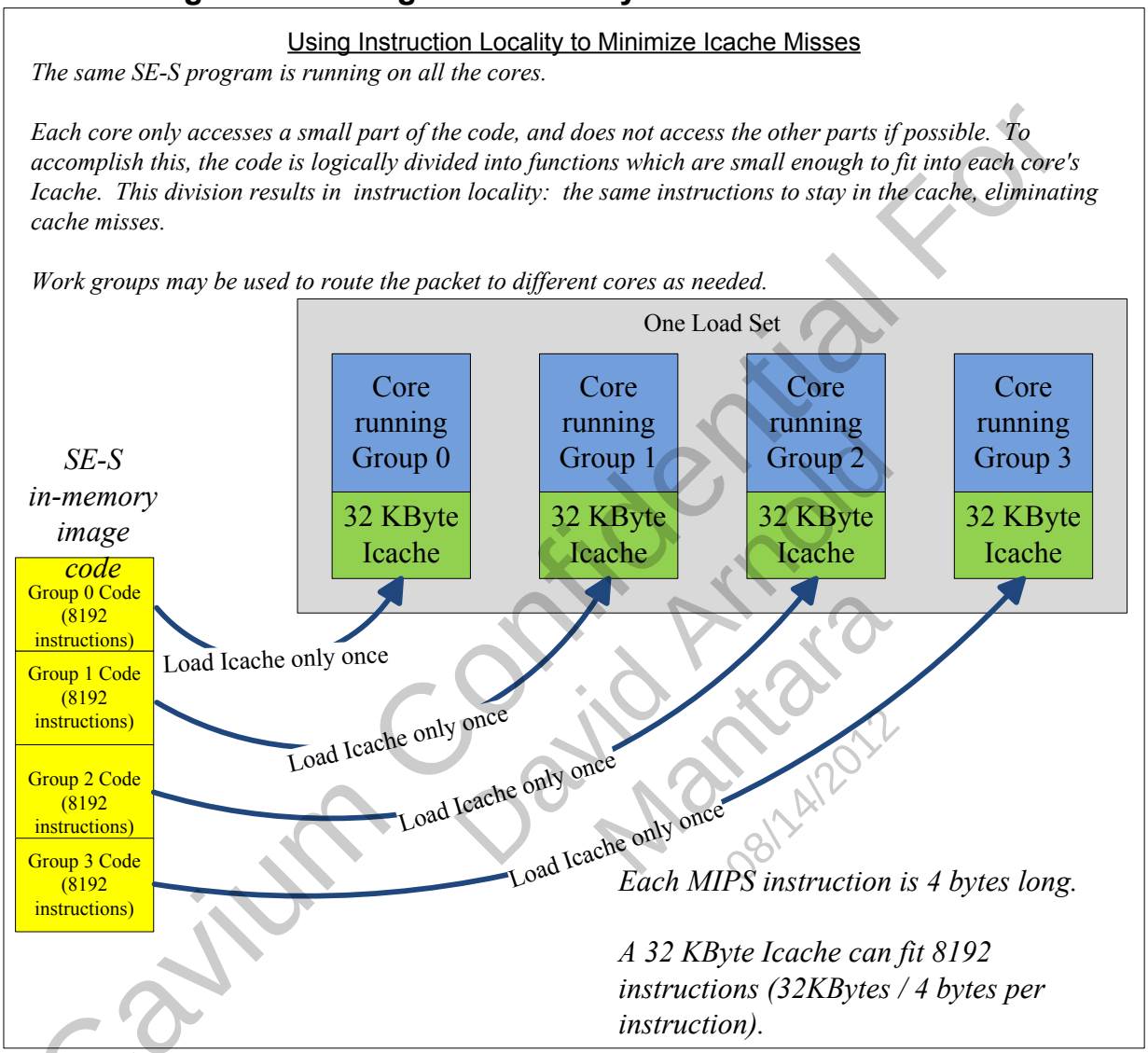
Once the core loads a set of instructions into Icache, if it continues to run only that set of instructions, the performance cost of Icache misses is 0.

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.

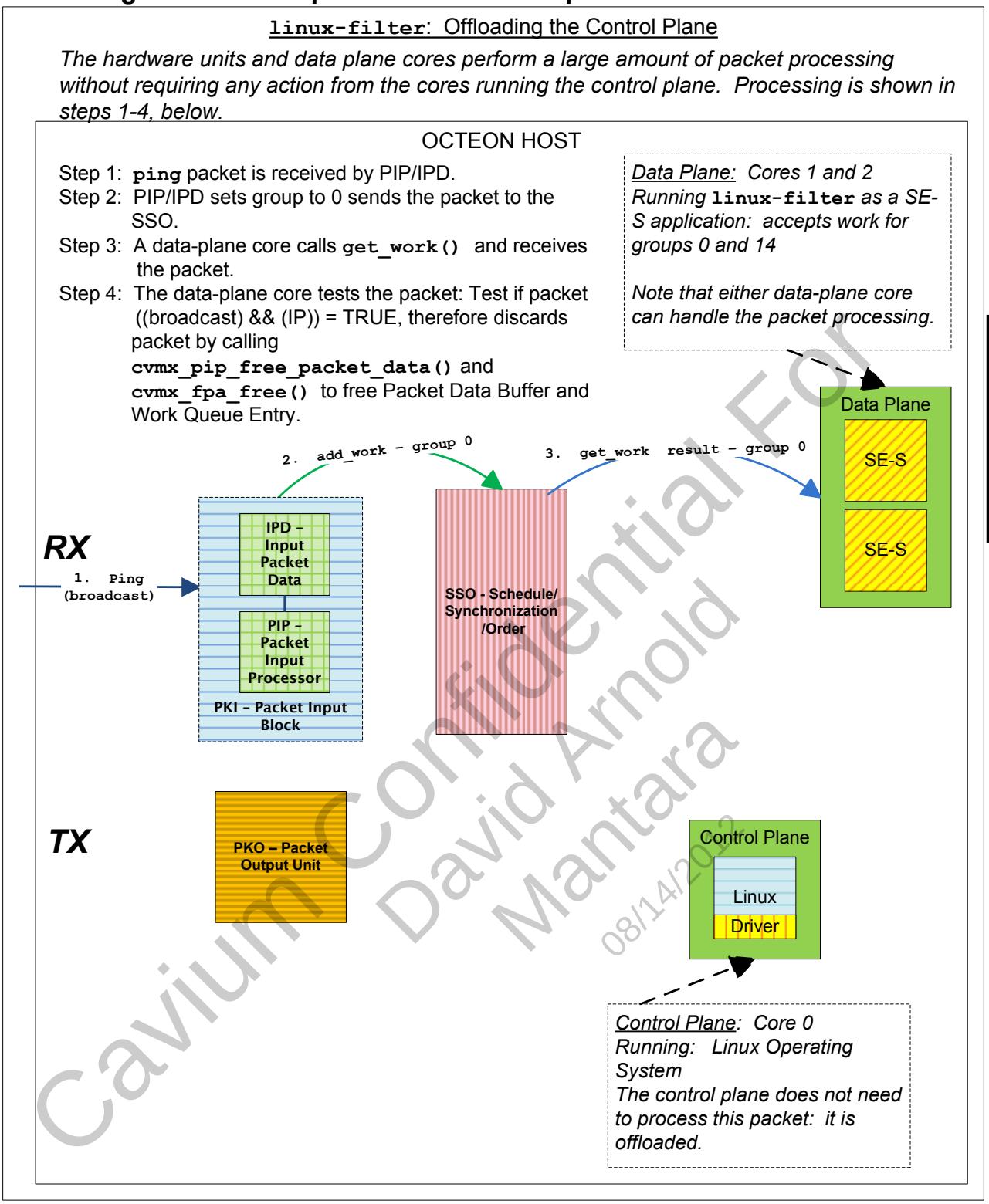
This program uses the idea of Work Groups to separate cores belonging to the fast path from those belonging to the slow path. Additionally, groups are used to identify the next processing phase. This example may be used as the base for an application which does similar processing.

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

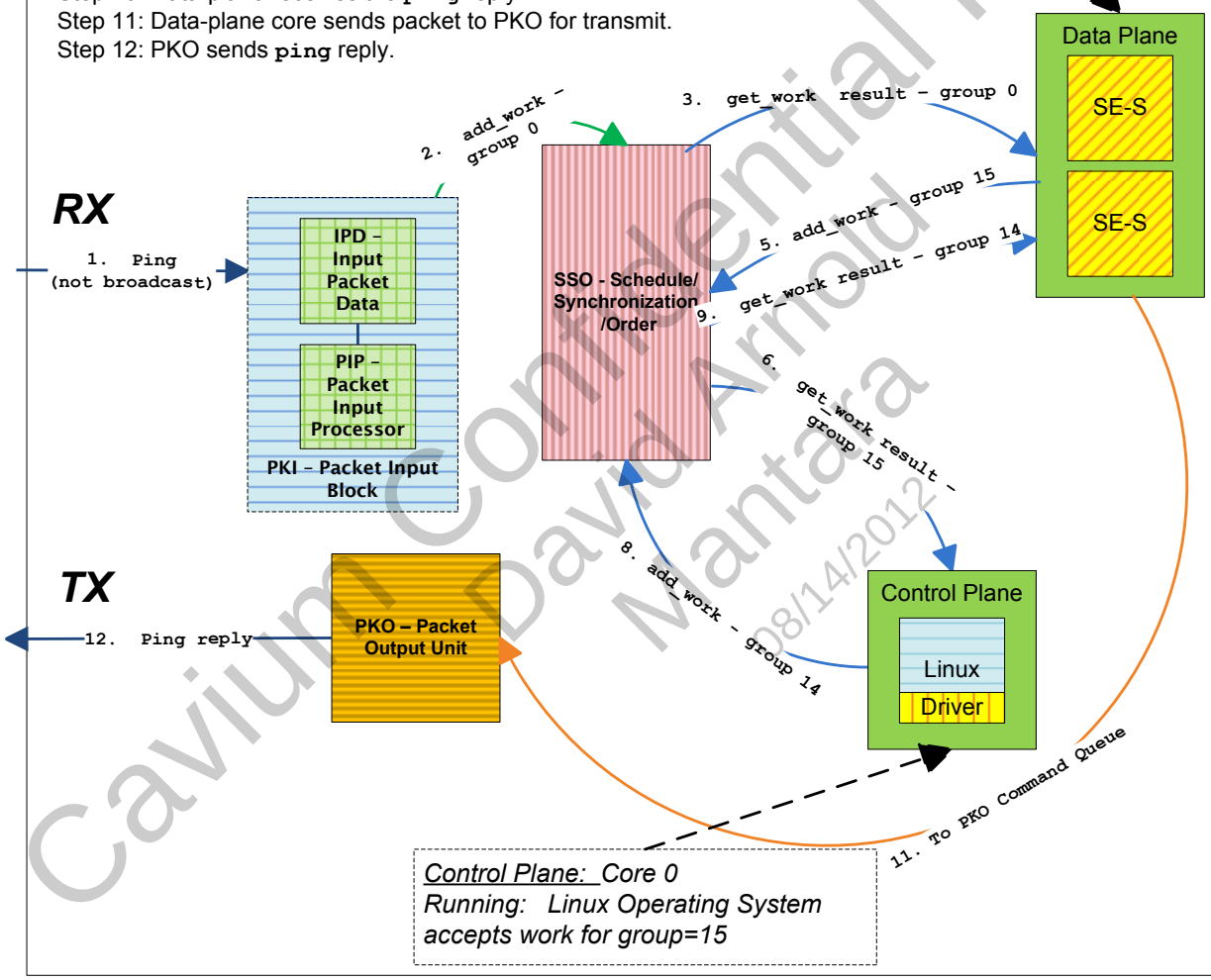
The hardware units and data plane cores perform a large amount of packet processing without requiring any action from the cores running the control plane. Processing is shown in steps 1-12, below.

OCTEON HOST

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

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

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



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

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

(the most useful values are highlighted)			
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

(the most useful values are highlighted)		
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-*`	
Note 1: Function calls are not ABI-conformant in this toolchain.		

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  Algn
  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  Algn
  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  Algn
  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*

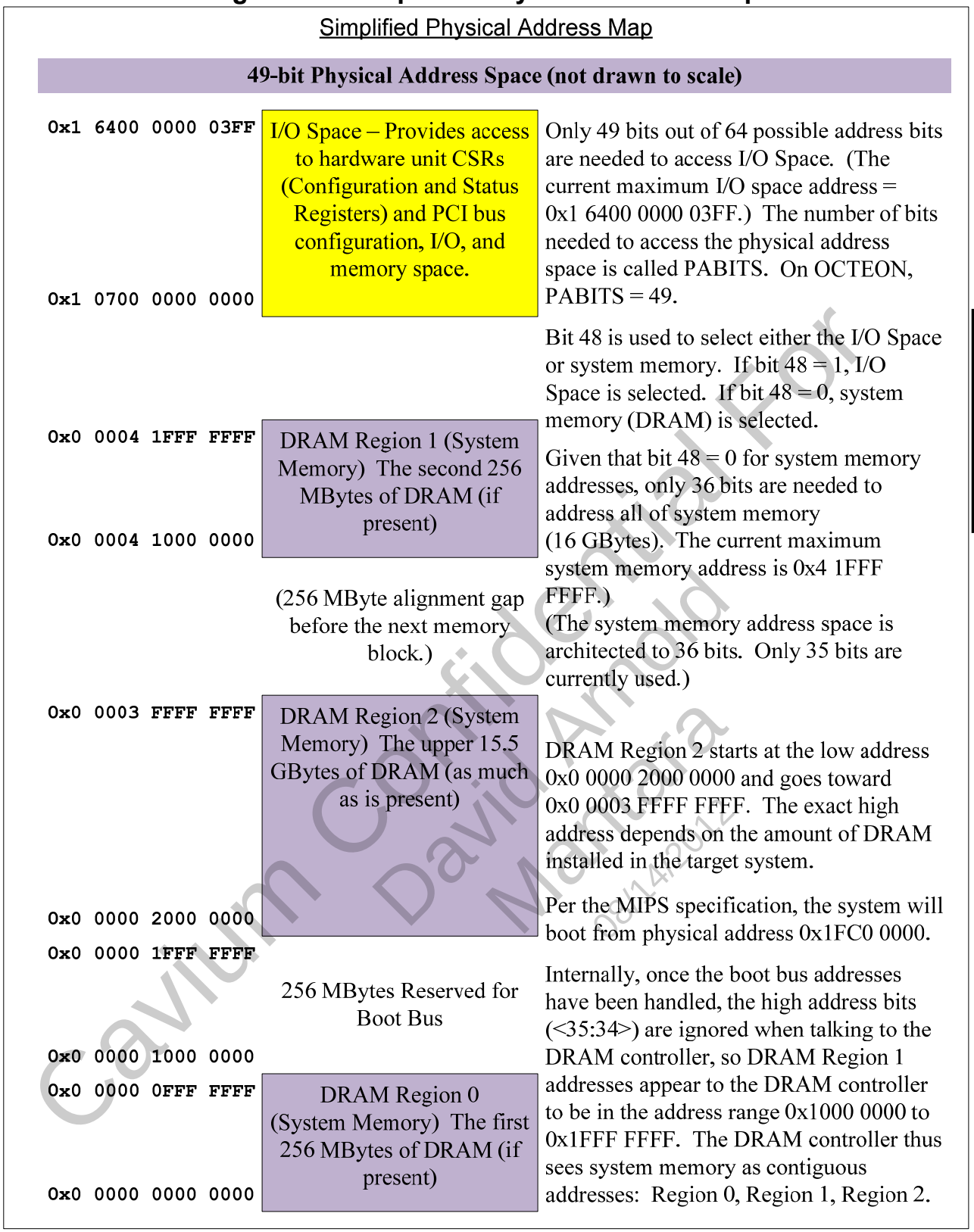
There are two key elements in the physical address map:

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 1E0F FFFF FFFF	PCI Bus Memory space
0x1 1A00 0000 0000	0x1 1A0F 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*

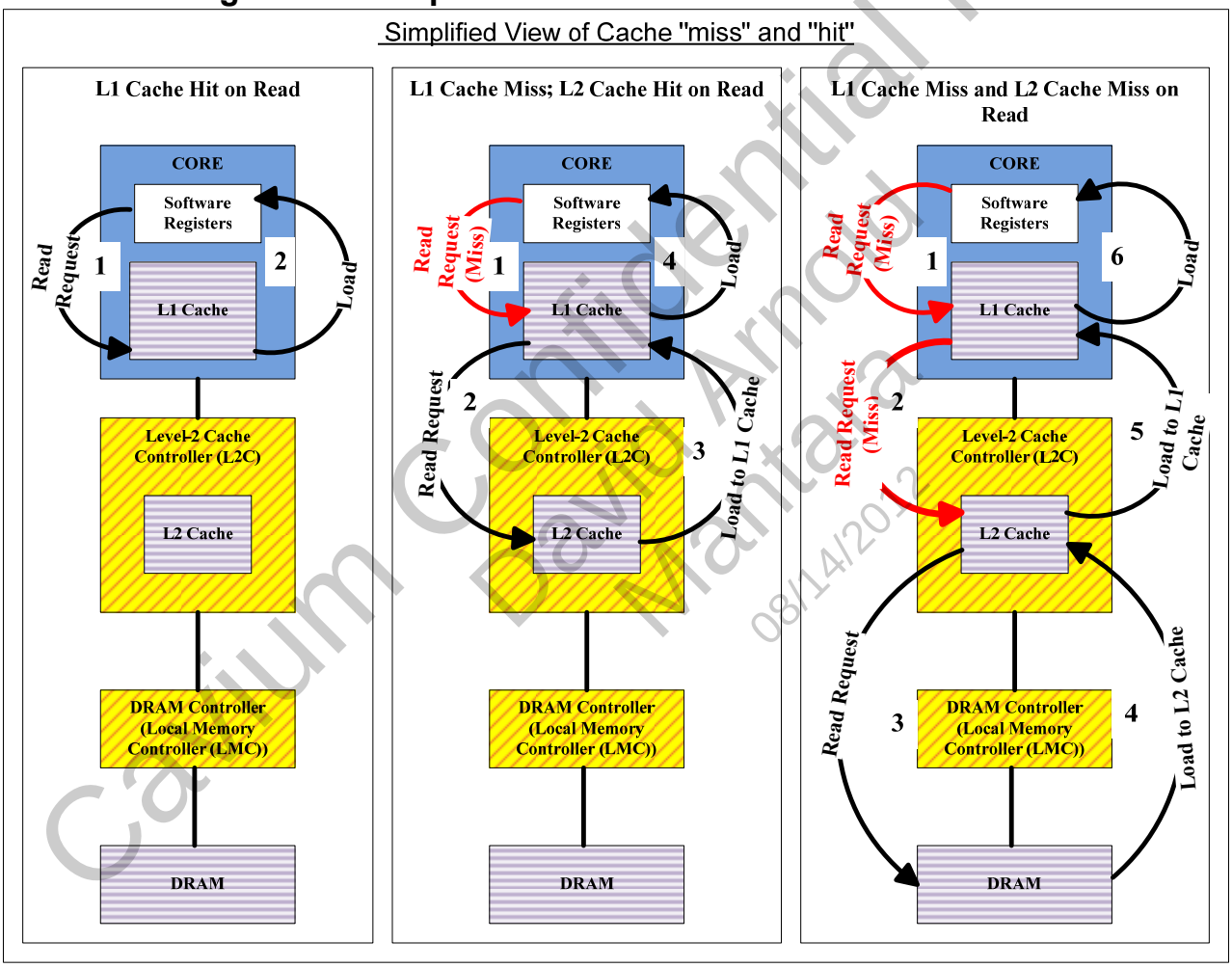
On the OCTEON processor, caching only applies to system memory (DRAM) accesses, not I/O space. Caching is used to improve system performance by providing core-local or chip-local fast memory which is used to cache (save a copy) of recently accessed data. This improves performance because accesses to 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"



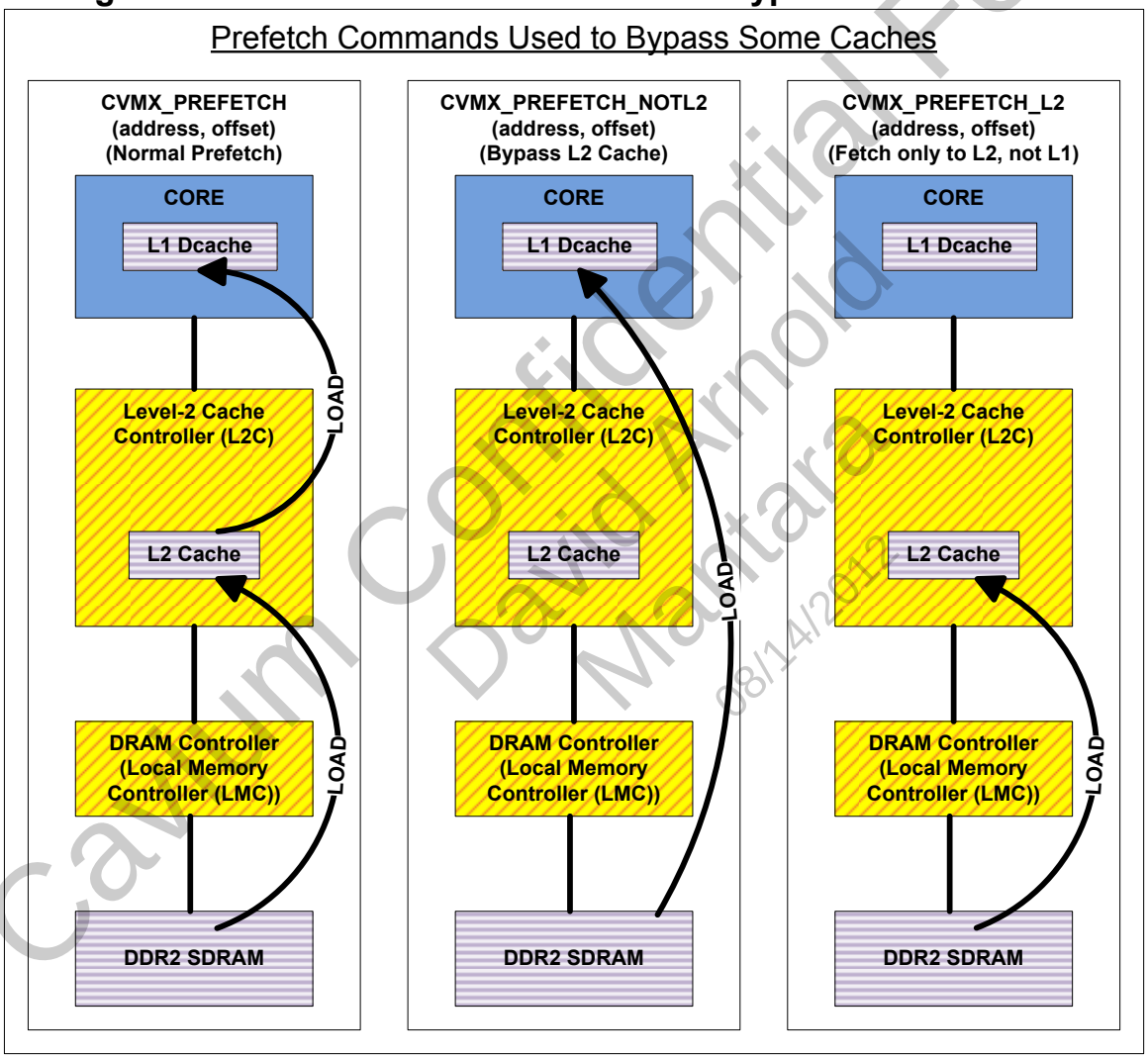
Note that the sizes of the L1 and L2 caches are limited. The specific sizes depend on the OCTEON model. The size of the L1 Dcache is also affected by the amount of Dcache set aside for *cvmseg*.

(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.

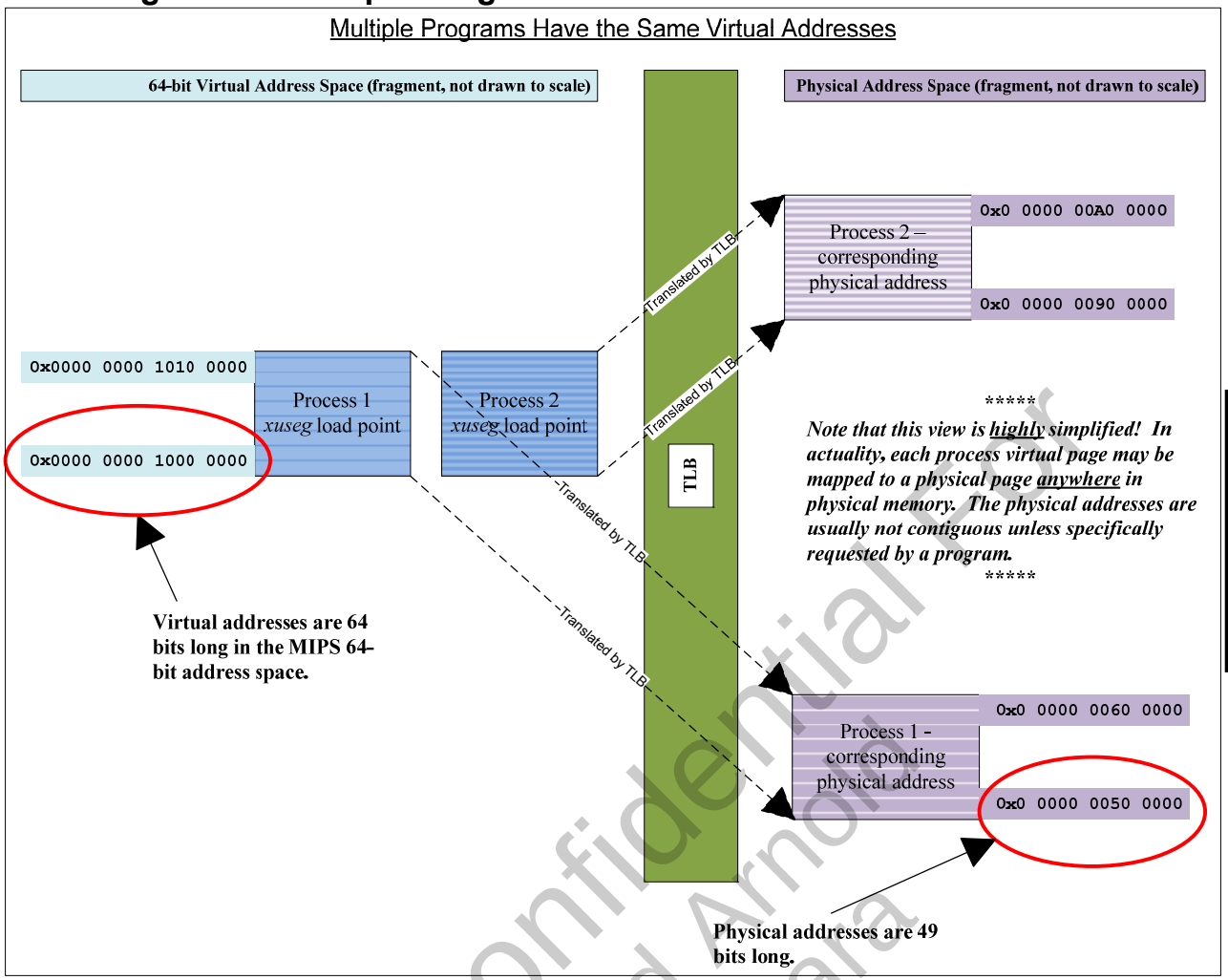
10 Virtual Memory

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



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.

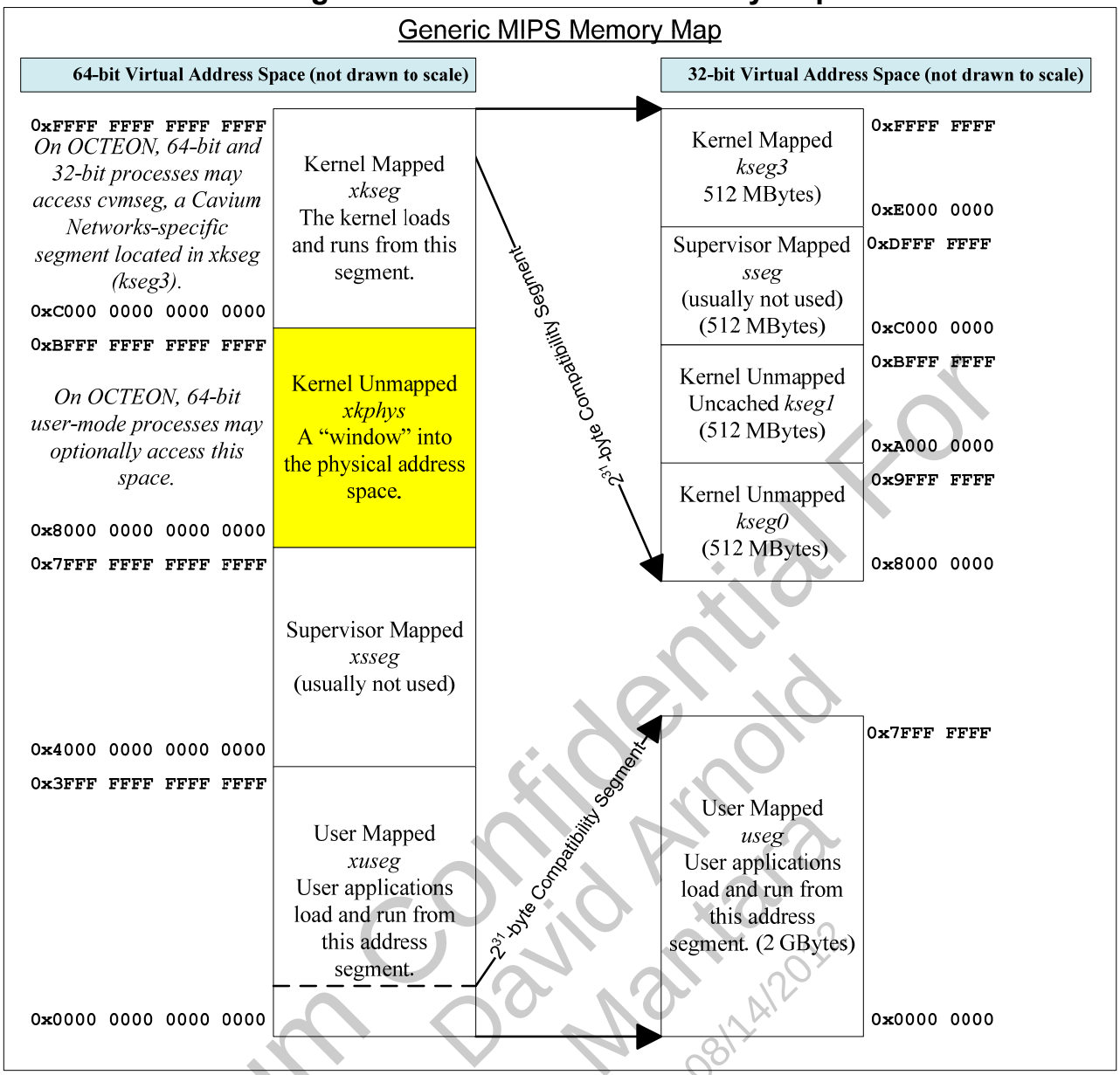
To determine the affect of wired TLB entries for the application, use profiling and performance tuning tools after the application has been written.

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 accesses 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.

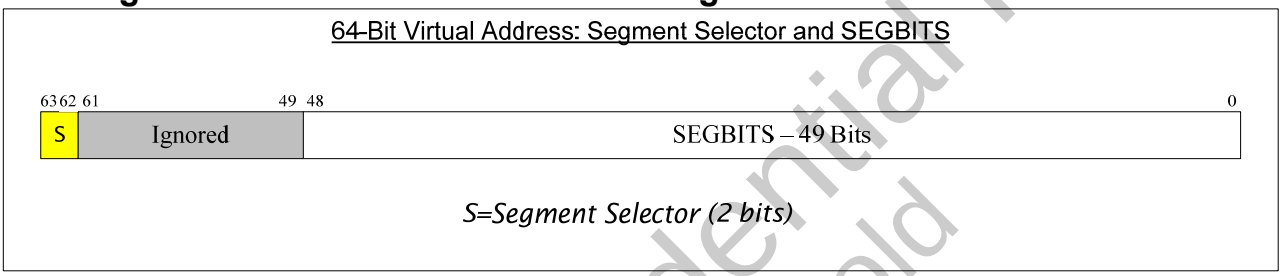
10.3.1.1 Segments: 64-Bit Virtual Address Map

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



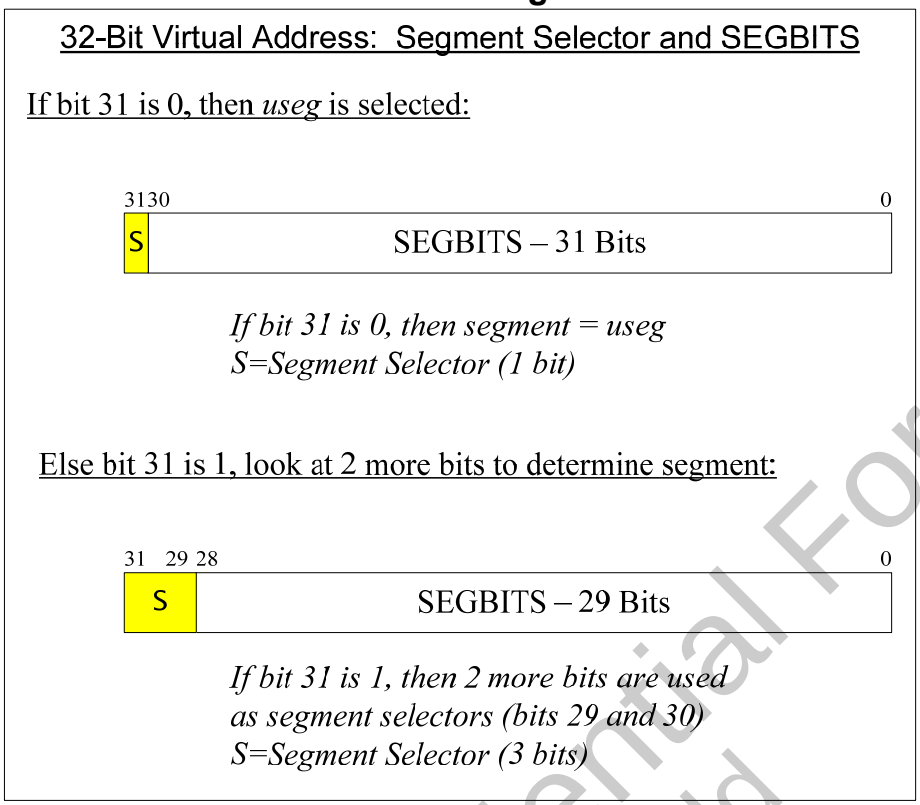
10.3.1.2 Segments: 32-Bit Virtual Address Map

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



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

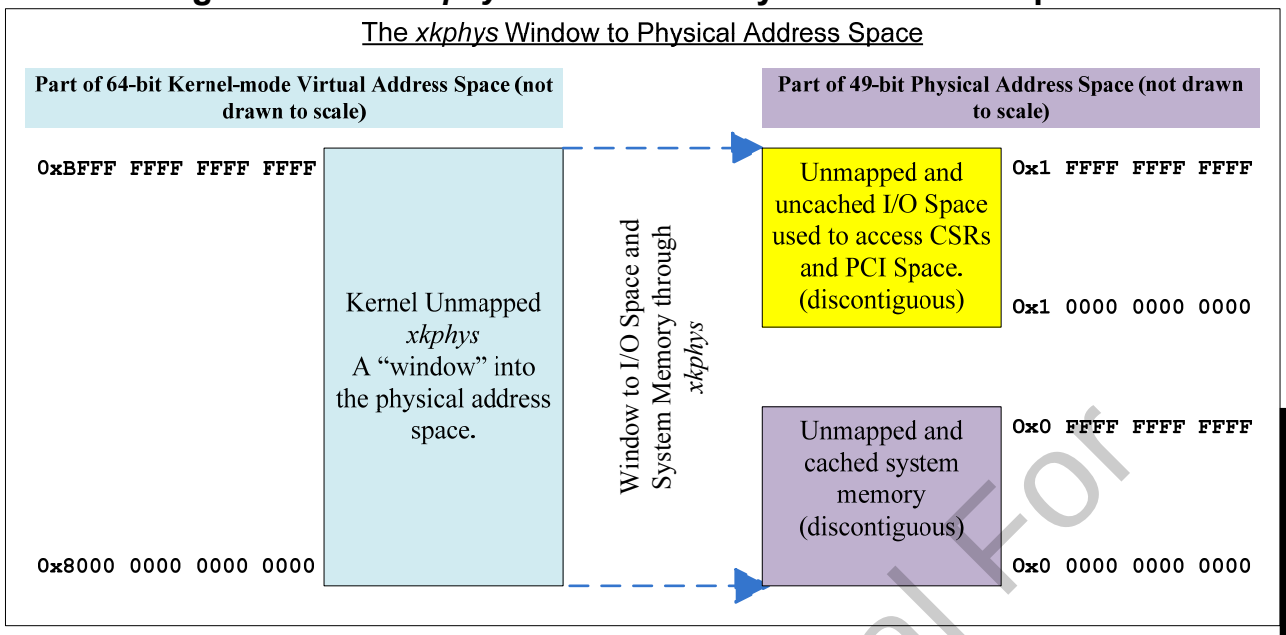
10.4.1.1 64-Bit Virtual Address Space: *xkphys*

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



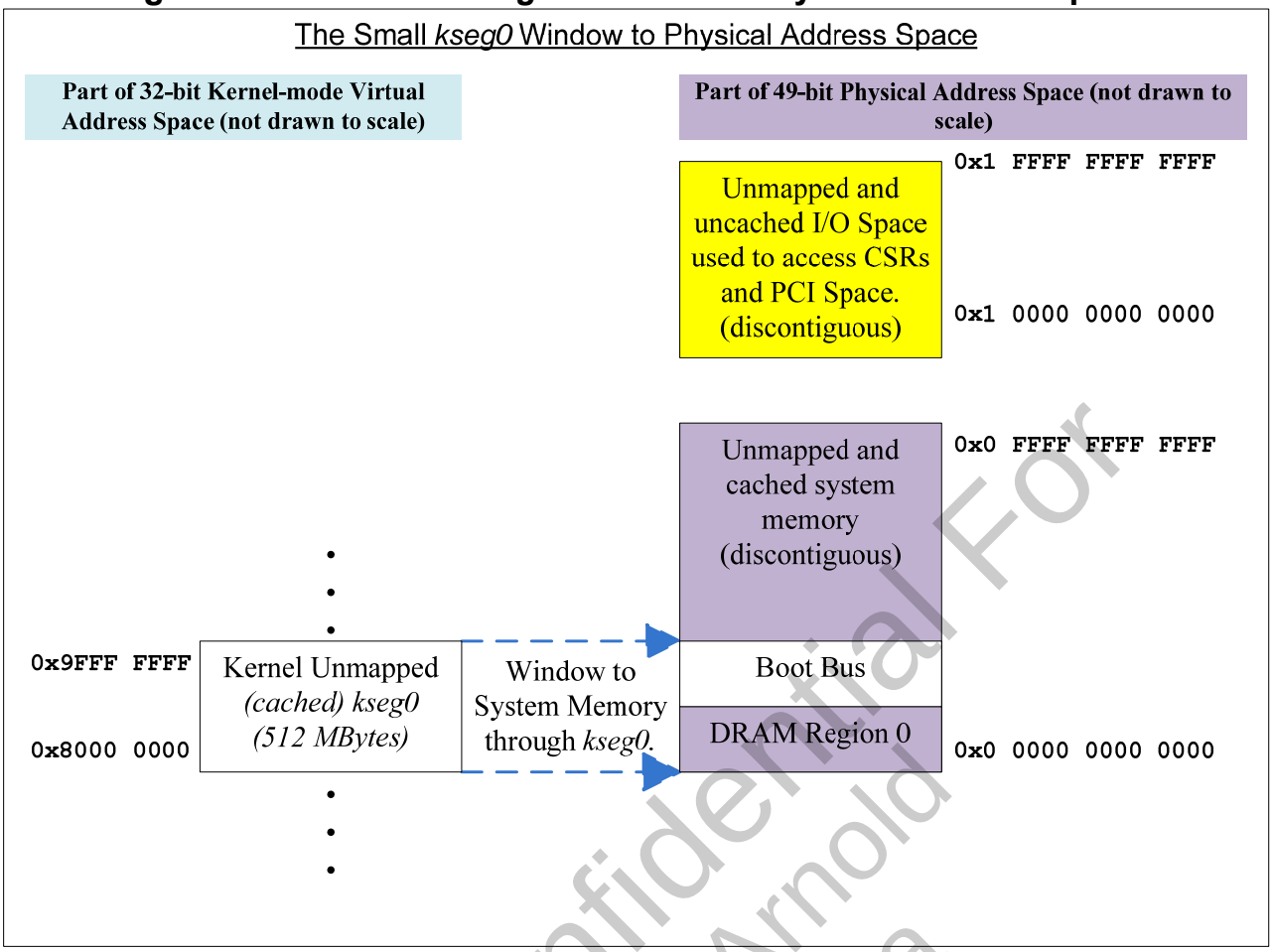
SW OVERVIEW

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).

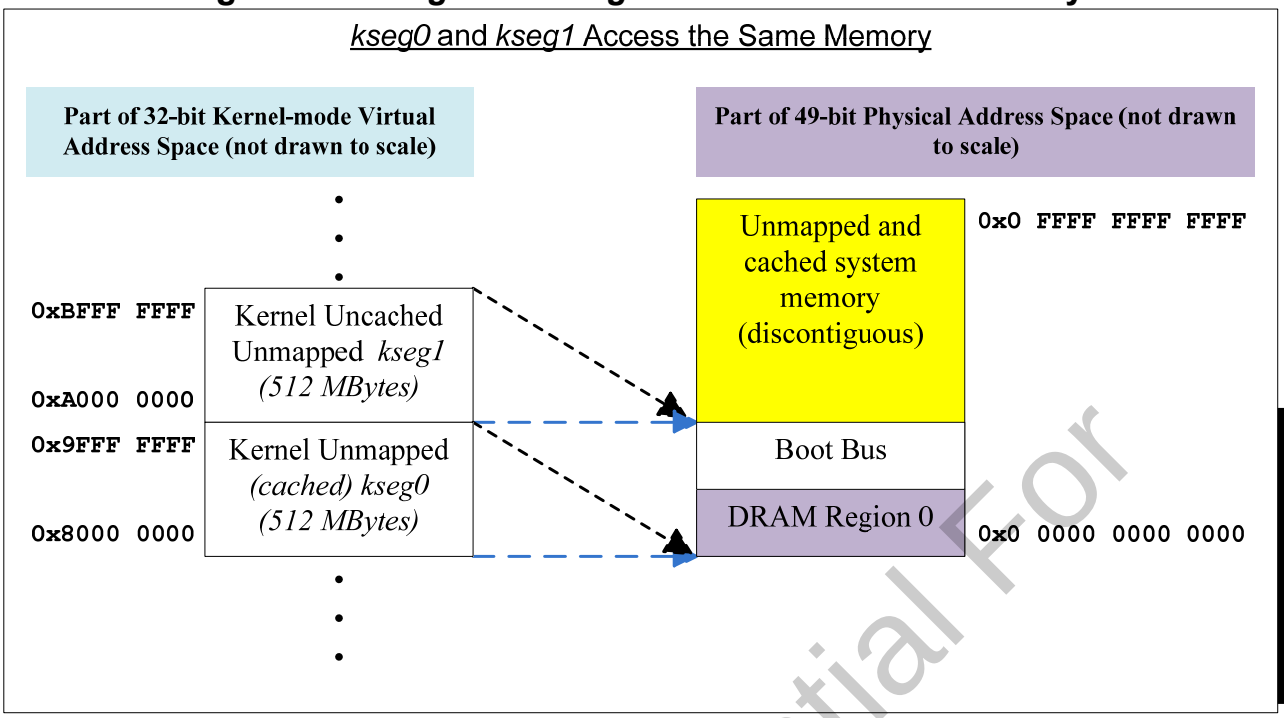
32-bit Simple Executive Standalone applications run in kernel mode and access physical memory through *kseg0* addresses.

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



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

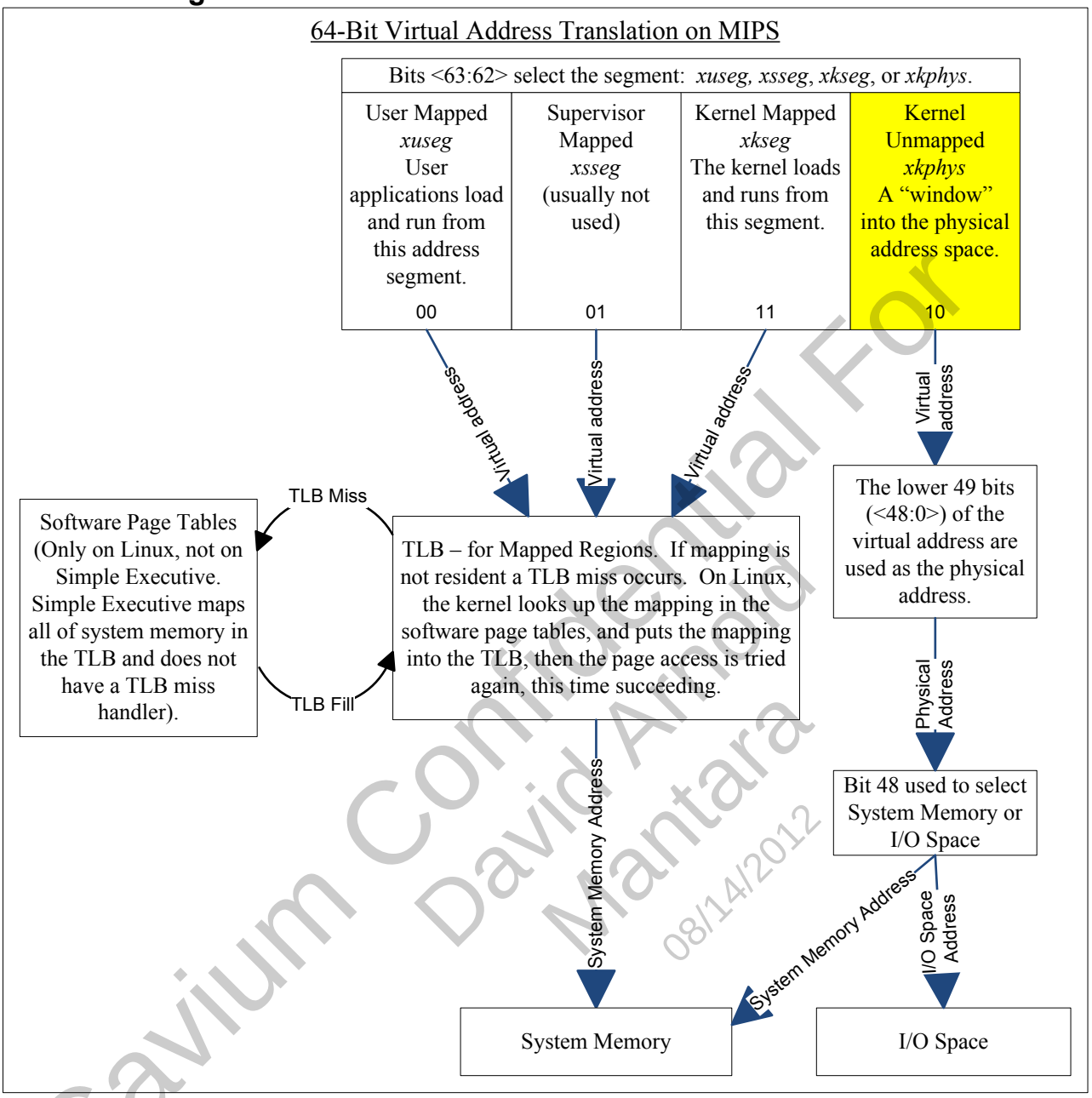
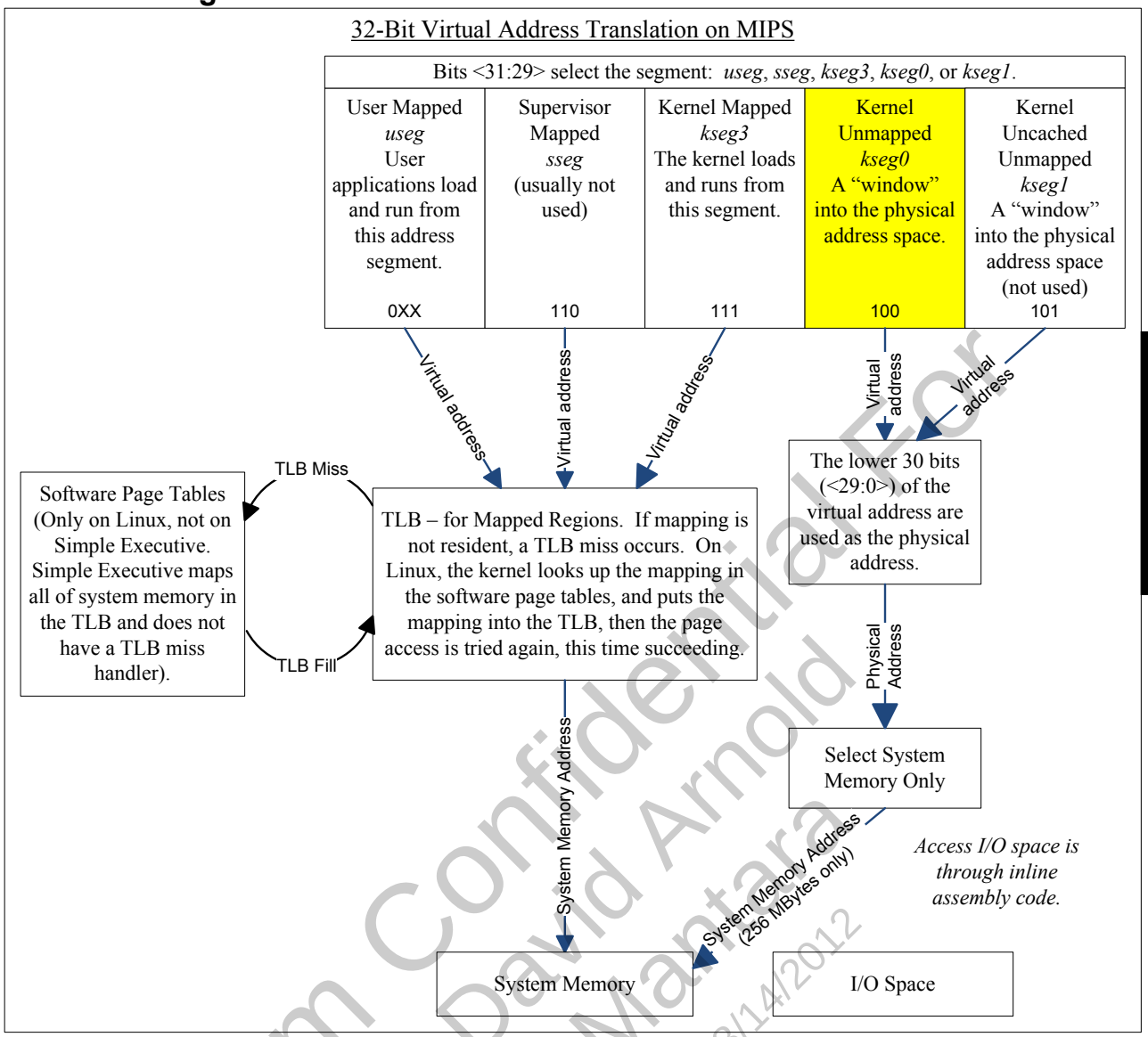


Figure 46: 32-Bit Virtual Address Translation on MIPS



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 onCavium 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	**Mapped Segments**
xuseg	The *xuseg* segment is the user address space (mapped).	SE-S 64-bit applications run in kernel-mode, but are mapped into *xuseg*.)
xsseg	The *xsseg* segment is the supervisor address space (usually not used) (mapped).	The *xsseg* segment is usually not used in OCTEON cnMIPS.
xkseg	The *xkseg* segment is in the kernel address space (mapped).	The *xkseg* segment contains the OCTEON-specific *cvmseg* segment. User-Mode access is allowed to *cvmseg*.
	Unmapped Segments	**Unmapped Segments**
xkphys	The *xkphys* 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 *xkphys* addresses. SE-S 64-bit applications always have access to *xkphys* addresses (they run in kernel-mode). Accesses to system memory are *always* cached. Accesses to I/O space are *never* cached.
Note: 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.		

Table 13: The 32-Bit Virtual Address Segments

Segment	Generic MIPS	OCTEON cnMIPS
	Mapped Segments	**Mapped Segments**
useg	The *useg* segment is the user address space (mapped).	OCTEON SE-S 32-bit applications run in kernel-mode, but are mapped into *useg*.
sseg	The *sseg* segment is the supervisor address space (usually not used) (mapped)	This segment is usually not used.
kseg3	The *kseg3* segment is in the kernel address space (mapped)	User-Mode access is allowed only to *cvmseg* in this segment.
	Unmapped Segments	**Unmapped Segments**
kseg0	The *kseg0* segment is in the kernel address space (unmapped, uncached)	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)	Accesses to this segment accesses system memory which is *always* cached on the OCTEON processor.
Note: The Linux kernel always runs in 64-bit mode. Relative to a SE-UM 32-bit virtual address space, cvmseg is in kseg3. 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

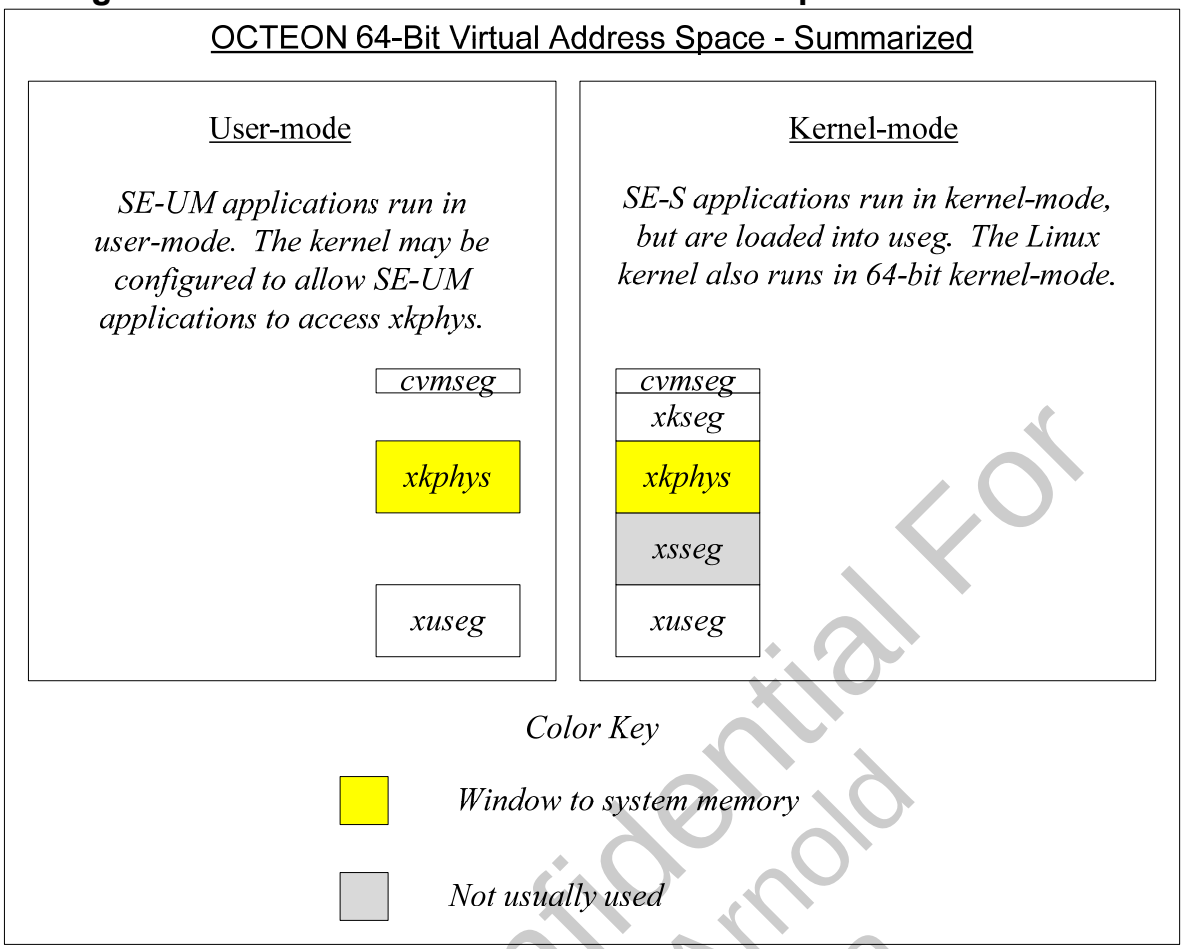
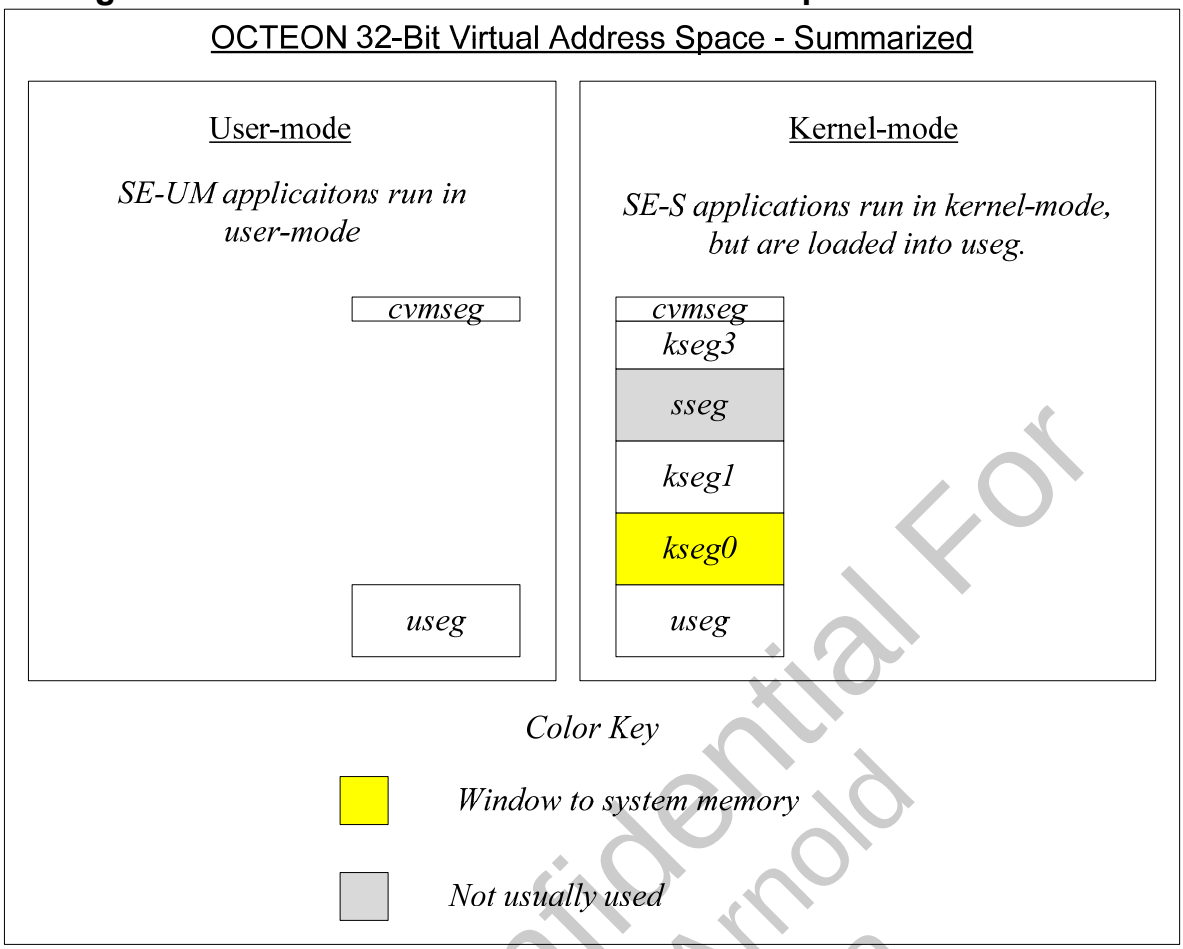


Figure 48: OCTEON 32-Bit Virtual Address Space - Summarized



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.

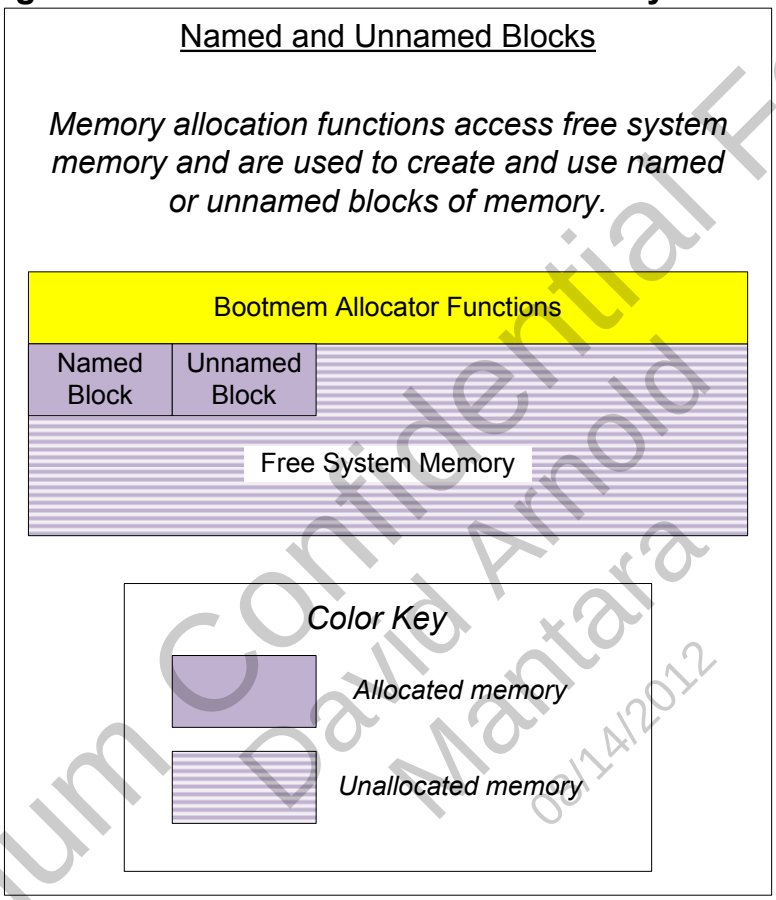
Table 14: Bootmem Allocator Functions in SDK 1.8

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 un-named 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

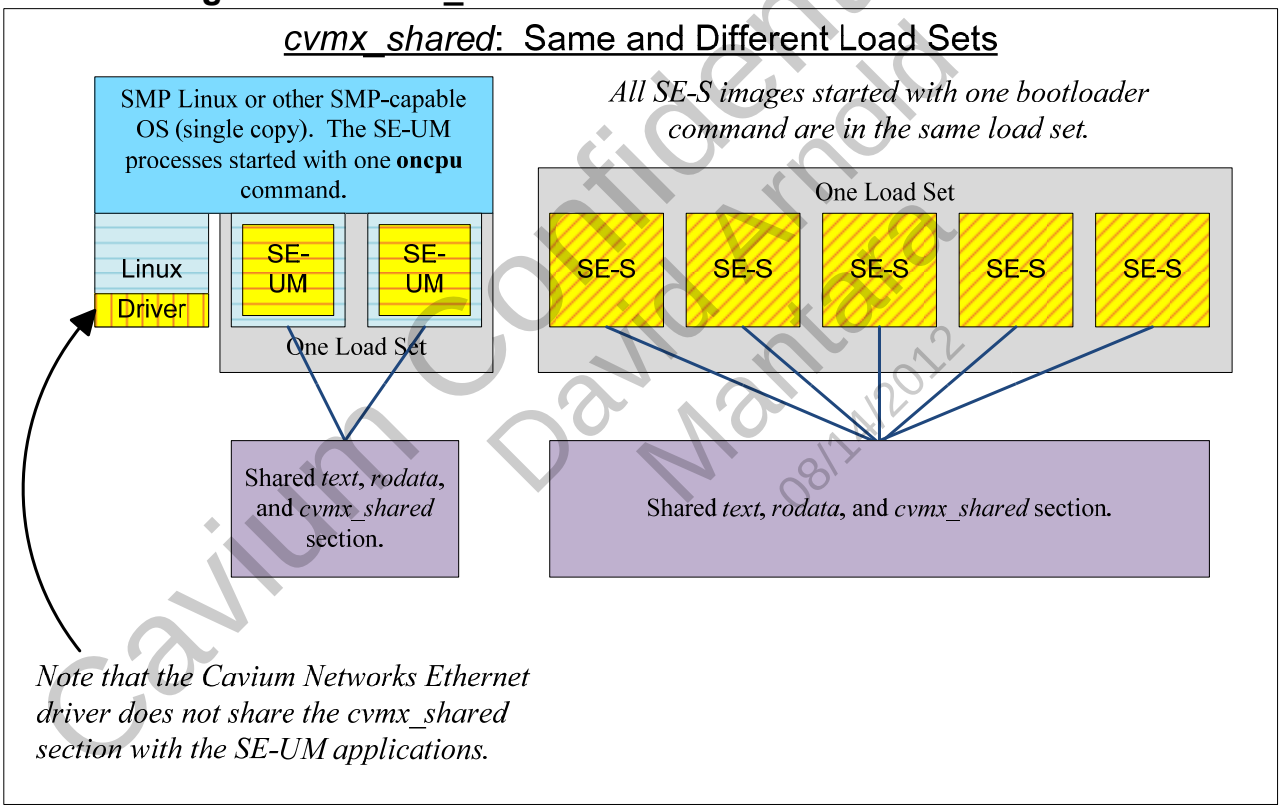
The *cvmx_shared* section provides shared memory between cores started with the *same* load command (the same load set):

- 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).

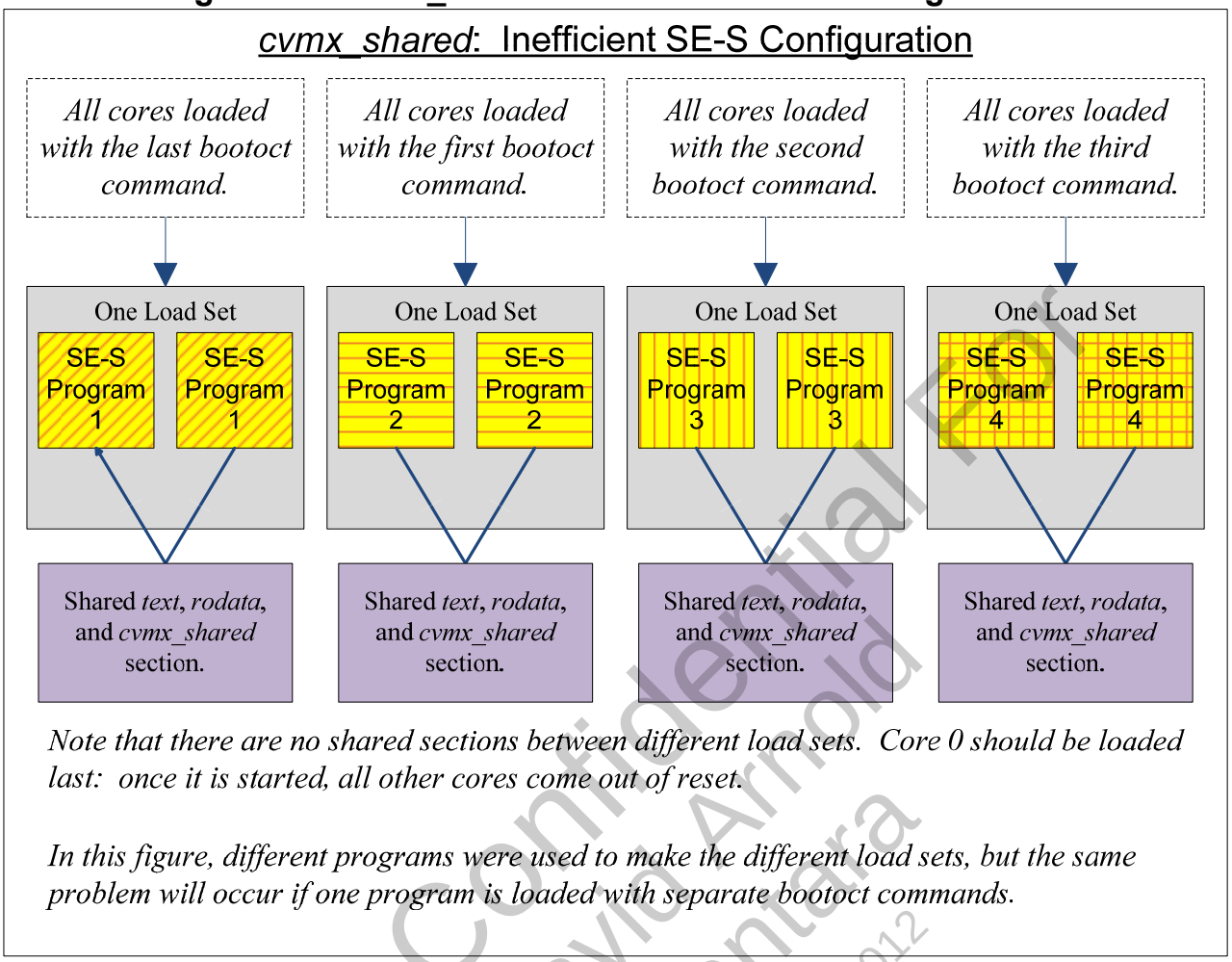
Figure 50: *cvmx_shared*: Same and Different Load Sets



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.

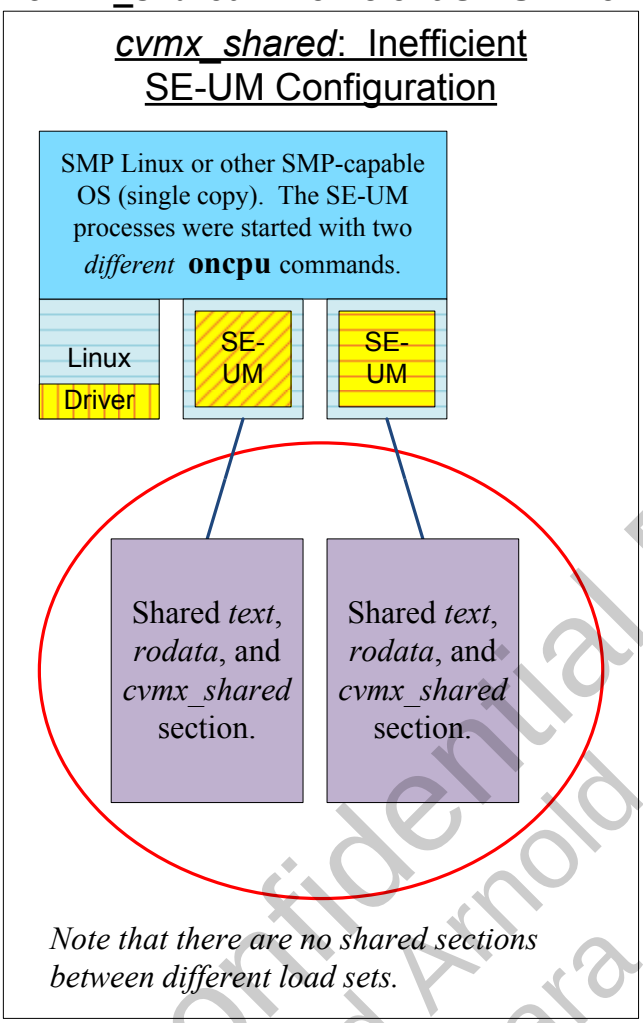
As shown in the following figure, it is inefficient to load different SE-S applications because they will not share common sections: they will be in different load sets.

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



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.

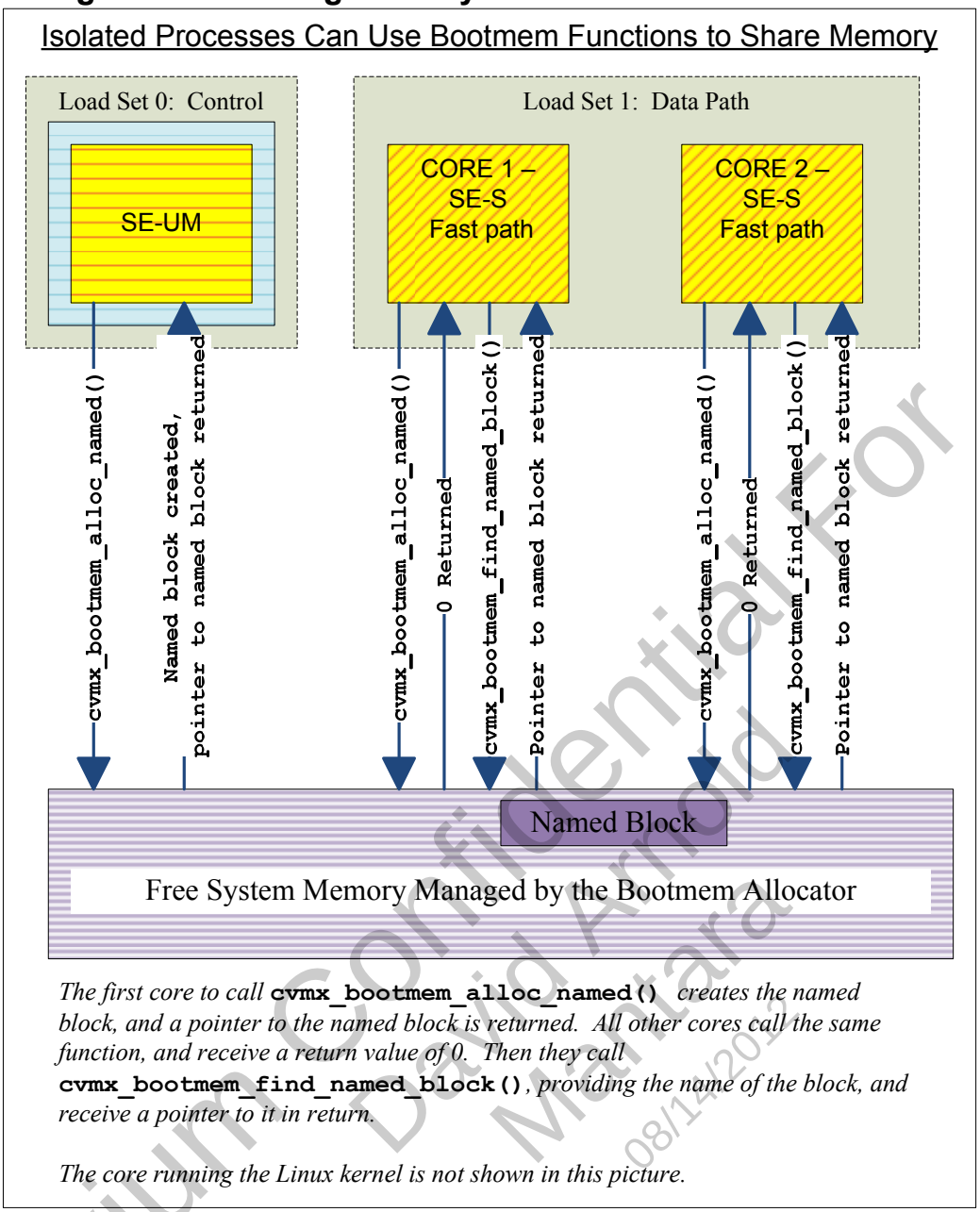
11.4 *Using Named Blocks to Share Memory Between Different Load Sets*

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 Sets



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	*xuseg (See Note 1)*	*xkphys*	*xkphys*	Preferred configuration - safest for porting.
SE-S: 64-bit with 1:1 mapping	64 bit	kernel mode	*xuseg (See Note 1)*	*xuseg*	*xkphys*	Using 1:1 mapping can result in porting problems if `cvmx_phys_to_ptr()` and `cvmx_ptr_to_phys()` are not used.
SE-S: 32-bit no 1:1 mapping	32 bit	kernel mode	*useg (See Note 1)*	*kseg0*	inline assembly code	This is the preferred configuration: safest for porting, but only 256 MBytes of memory are addressable through *kseg0*.
SE-S: 32-bit with 1:1 mapping	32 bit	kernel mode	*useg (See Note 1)*	*useg*	inline assembly code	Using 1:1 mapping can result in porting problems if `cvmx_phys_to_ptr()` and `cvmx_ptr_to_phys()` are not used.
Linux kernel and drivers	64 bit	kernel mode	*xkseg*	*xkphys*	*xkphys*	
Linux SE-UM: 64-bit	64 bit	user mode	*xuseg*	*xkphys*	*xkphys*	Kernel configuration option provides *xkphys* access to user-mode processes.
Linux SE-UM: 32-bit	32 bit	user mode	*useg*	*useg* (*reserve32*)	inline assembly code	A *reserve32* region is mapped into the address space of the process. Each application is limited to 2 GBytes of virtual address space.
Note 1: Although SE-S applications are run in kernel-mode, they use the xuseg or useg address space for application-private memory, depending on whether the application is 64-bit or 32-bit.						

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	*xuseg - "unlimited" (see Note 4)*	"unlimited"	*xkphys*	ALL DRAM (see Note 2)
SE-S 64-bit	1:1 mapping	*xuseg - "unlimited" (see Note 4)*	256 MBytes (squeezed by mapped memory)	*xuseg*	ALL DRAM (see Note 2)
SE-S 32-bit	NO 1:1 Mapping	*useg - 2 GBytes*	2 GBytes max (see Note 1, Note 3)	*kseg0*	256 MBytes (See Note 2)
SE-S 32-bit	1:1 mapping	*useg - 2 GBytes*	256 MBytes (squeezed by mapped memory)	*useg*	no more than 2 GBytes (useg limit) (see Note 2)
Linux SE-UM Applications					
SE-UM 64-bit	N/A	*xuseg - "unlimited" (see Note 4)*	"unlimited" (see Note 1, Note 3)	*xkphys*	ALL DRAM (see Note 2)
SE-UM 32-bit	reserve32 - not wired	*useg - 2 GBytes*	2 GBytes minus reserve32 size	*useg: reserve32*	Blocks of DRAM in power of 2. Application load size must be than 2 GBytes. (See Note 2)
SE-UM 32-bit	reserve32 - wired	*useg - 2 GBytes*	2 GBytes minus reserve32 size	*useg: reserve32*	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 xuseg, for practical purposes it is "unlimited".*					

12.5 *Using `cvmx_ptr_to_phys()` and `cmvx_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:

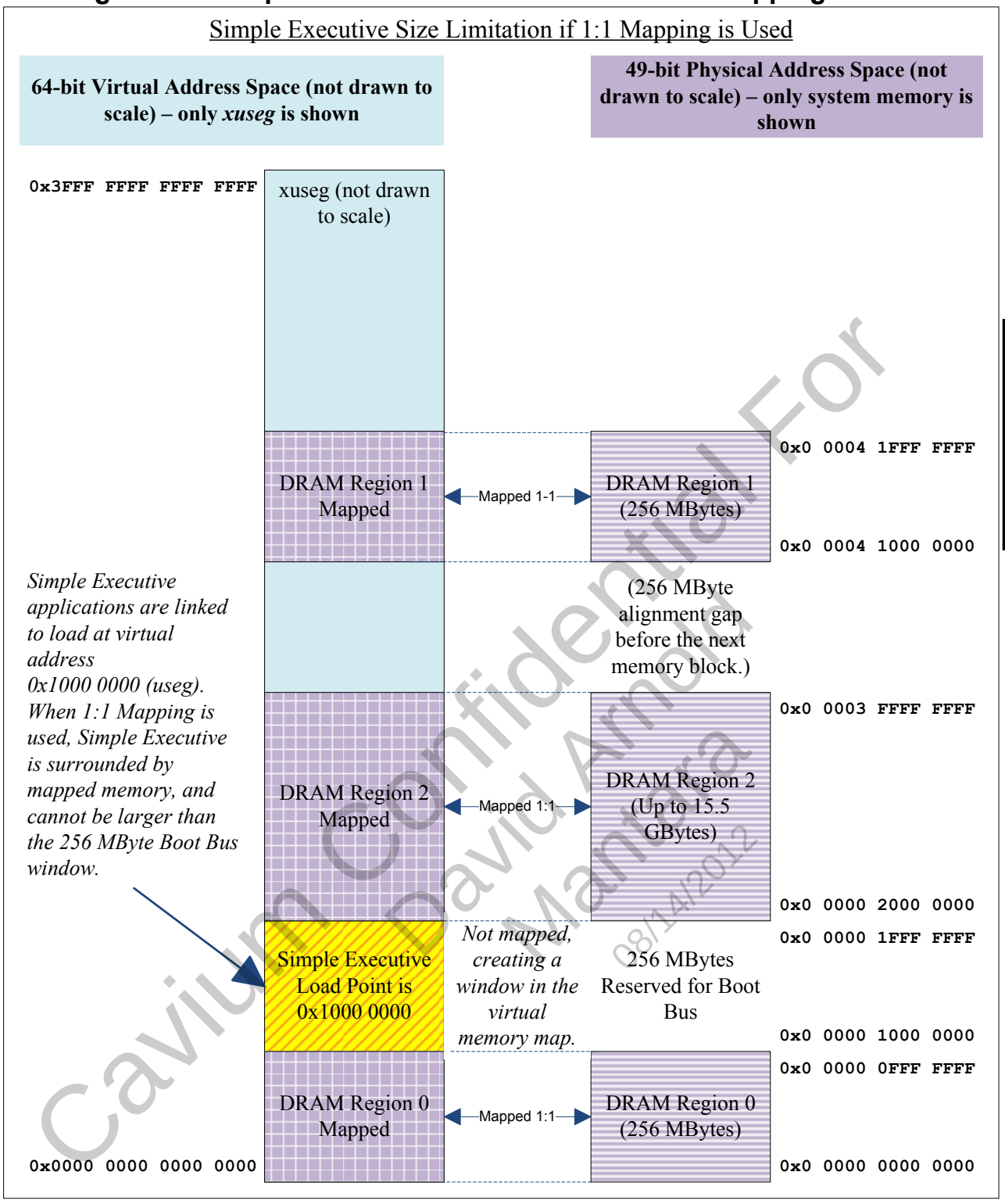
1. Porting problems occurred when code was not written to use the `cvmx_ptr_to_phys()` and `cvmx_phys_to_ptr()` functions. These functions hide pointer / address conversions, creating highly portable code.
2. The size of the Simple Executive application's runtime size was limited to 256 MBytes. (Note 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

SE-S 64-Bit Virtual Memory Map

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 0xFFFF FFFF FFFF A000	CVMSEG – IO (only valid address = 0xFFFF FFFF FFFF A200)	If CvmMemCtl[CVMK/S/U] is set, loads and stores to this address range are treated specially by the cnMIPS cores.
0xFFFF FFFF FFFF 9FFF ***kseg3*** 0xFFFF FFFF FFFF 8000	CVMSEG – LM (part of DCACHE)	This space is used for IOBDMA operations.
0x8001 6700 0000 03FF 0x8001 0000 0000 0000	Unmapped and uncached I/O Space (accessed through *xkphys*) (discontiguous where there is no matching I/O device)	Access to hardware unit CSRs (Configuration and Status Registers).
0x8000 0004 1FFF FFFF ***xkphys*** 0x8000 0000 0000 0000	Unmapped and uncached system memory (accessed through *xkphys*) (discontiguous where system memory is not present)	This access is used if CVMX_USE_1_TO_1_MAPPINGS is NOT defined to 1.
0x0000 0004 1FFF FFFF 0x0000 0004 1000 0000	Mapped and cached system memory: Second 256 MBytes of DRAM	This access is used if CVMX_USE_1_TO_1_MAPPINGS IS defined to 1.
0x0000 0003 FFFF FFFF 0x0000 0000 2000 0000	Mapped and cached system memory: Upper 15.5 GBytes of DRAM (as much as is present)	This access is used if CVMX_USE_1_TO_1_MAPPINGS IS defined to 1.
0x0000 0000 1FFF FFFF ***xuseg*** 0x0000 0000 1000 0000	Application Space (256 Mbytes) The size of the loaded application must not exceed 256 MBytes.	The application will only have access to the part of this space mapped in by the bootloader. This space is mapped and cached. To get more than 256 MBytes, define CVMX_USE_1_TO_1_MAPPINGS to 0. These addresses are the same in both 32-bit and 64-bit APIs.
0x0000 0000 0FFF FFFF 0x0000 0000 0000 0000	Mapped and cached system memory: First 256 MBytes of DRAM (the first MByte is unmapped)	This access is used if CVMX_USE_1_TO_1_MAPPINGS IS defined to 1. To map the first 1 MBytes, define CVMX_NULL_POINTER_PROTECT to 0.

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.

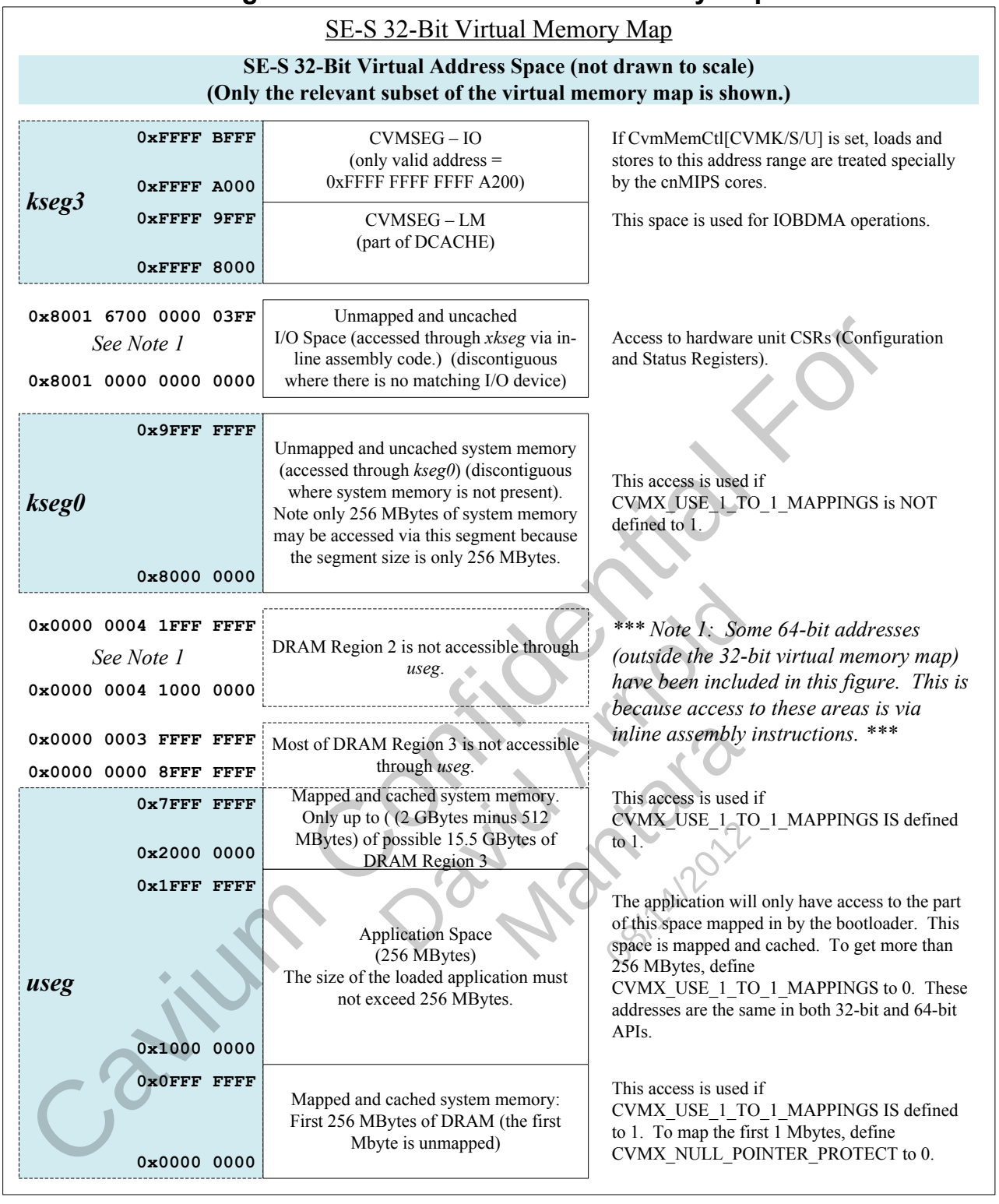
14.5 *SE-S 32-Bit Applications*

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 boomem 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 r*eserved32* 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 `autoconfig.h`	Default Value	Brief Descripton
Number of L1 cache lines reserved for CVMSEG memory.	CONFIG_CAVIUM_OCTEON_CVMSEG_SIZE	2	This memory is reserved for CVMSEG LM, the dcache lines set aside for IOBDMA operations.
Allow User space to access hardware IO directly.	CONFIG_CAVIUM_OCTEON_USER_IO	1 (yes)	64-bit applications can access the OCTEON I/O registers without switching to kernel mode.
Allow User space to access memory directly	CONFIG_CAVIUM_OCTEON_USER_MEM	1 (yes)	64-bit applications can access hardware buffers (such as FPA buffers) without switching to kernel mode.

Option	Variable in `autoconfig.h`	Default Value	Brief Descripton
Memory to reserve for user processes shared region (MB).	CONFIG_CAVIUM_RESERVE32	0 Mb	The number of MBytes to reserve so that 32-bit applications can use `cvmx_bootmem_alloc()` functions. Required for 32-bit applications to send and receive packets directly.
Use wired TLB entries to access the reserved memory region.	CONFIG_CAVIUM_RESERVE32_USE_WIRED_TLB	#undef	When this option is set, the *reserve32* region is globally mappped to all userspace programs using wired TLB entires. If CONFIG_CAVIUM_RESERVE32 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
```

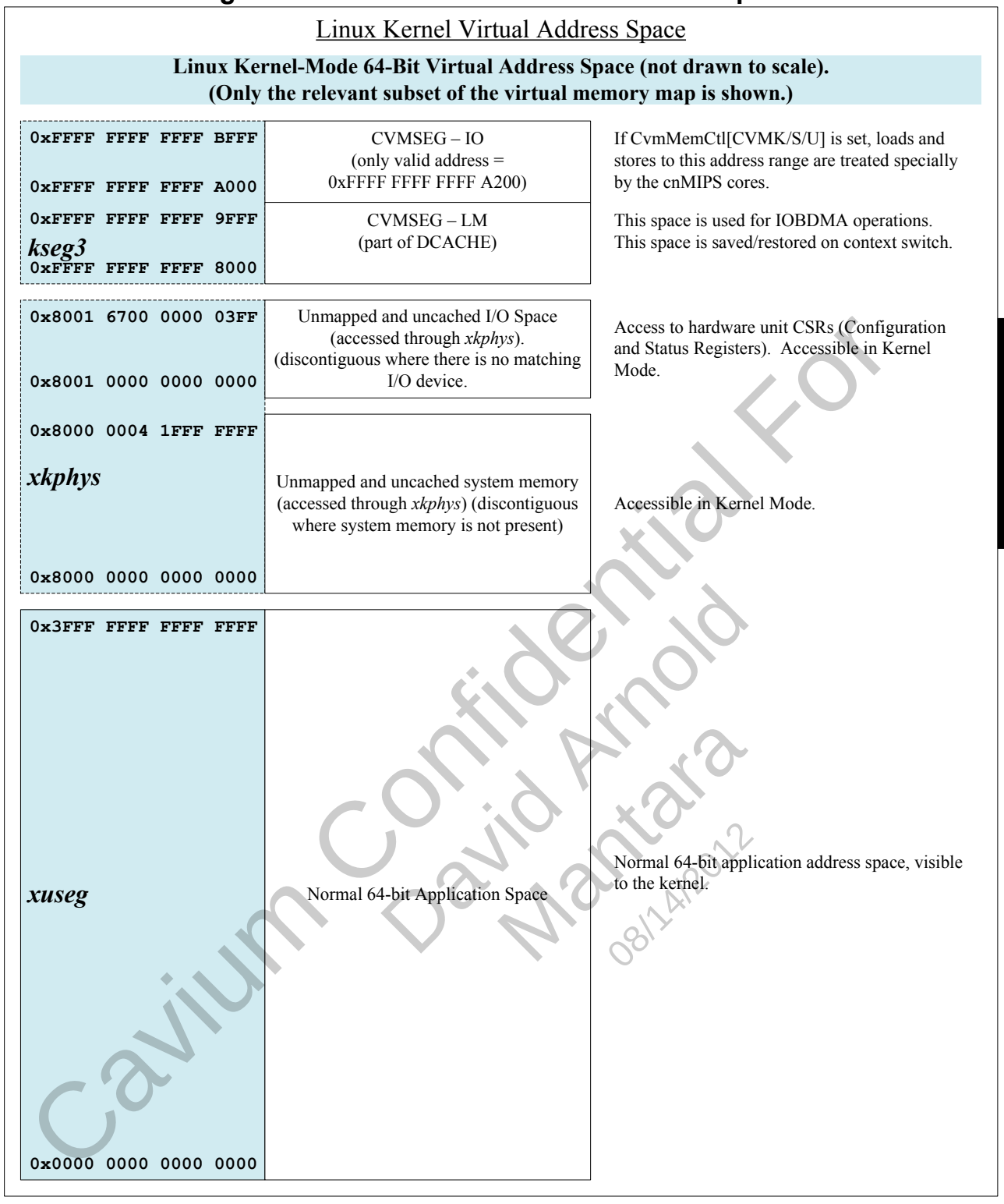
For a detailed discussion, see the SDK document "*Linux Userspace on the OCTEON*".

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.

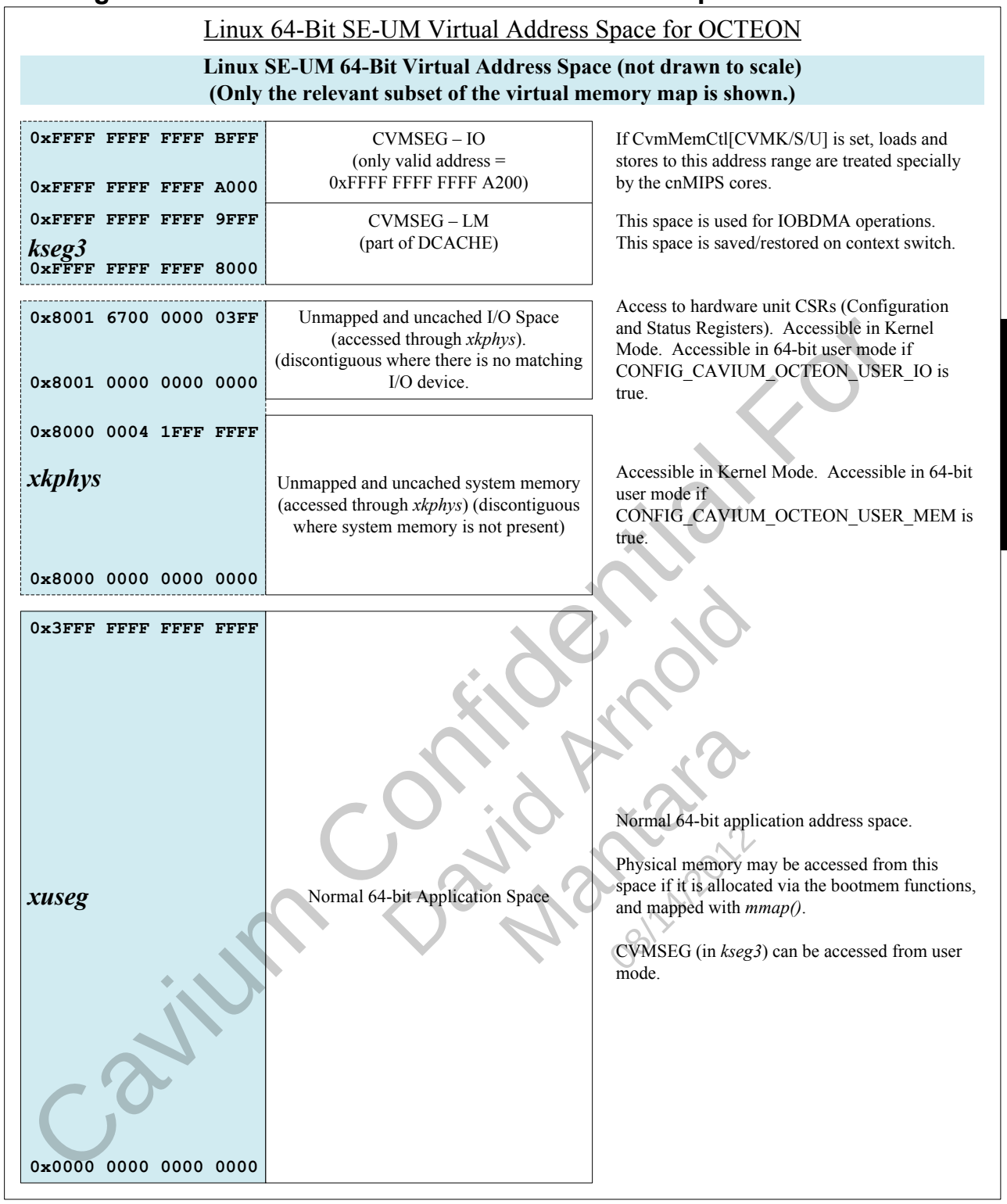
Figure 57: Linux Kernel Virtual Address Space



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



15.6 *Linux 32-Bit Virtual Address Space for OCTEON*

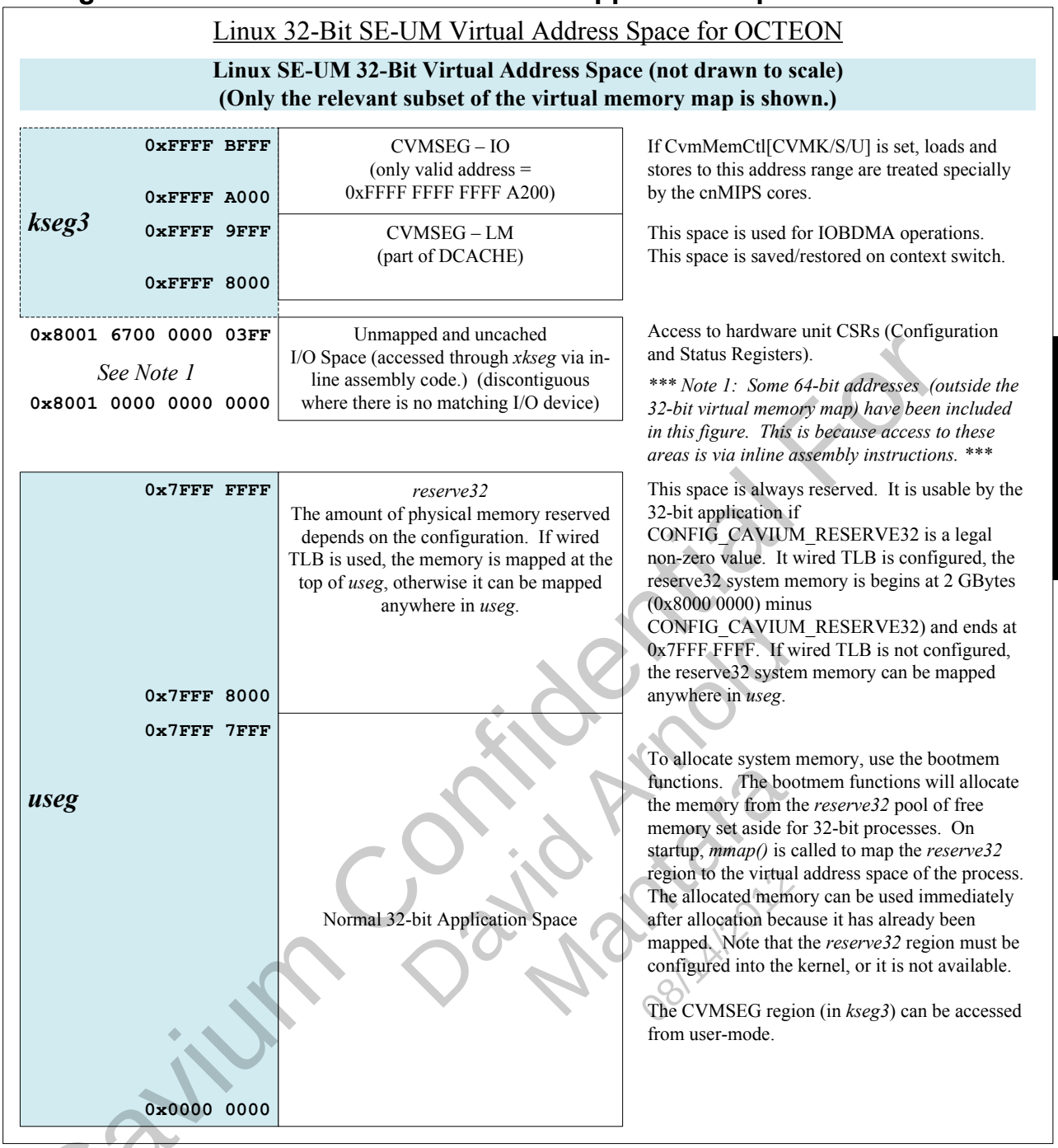
The following figure shows the Linux 32-bit Virtual Address Space for the OCTEON processor. The kernel always runs in 64-bit mode. The size of *cvmseg* is set during kernel configuration.

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.

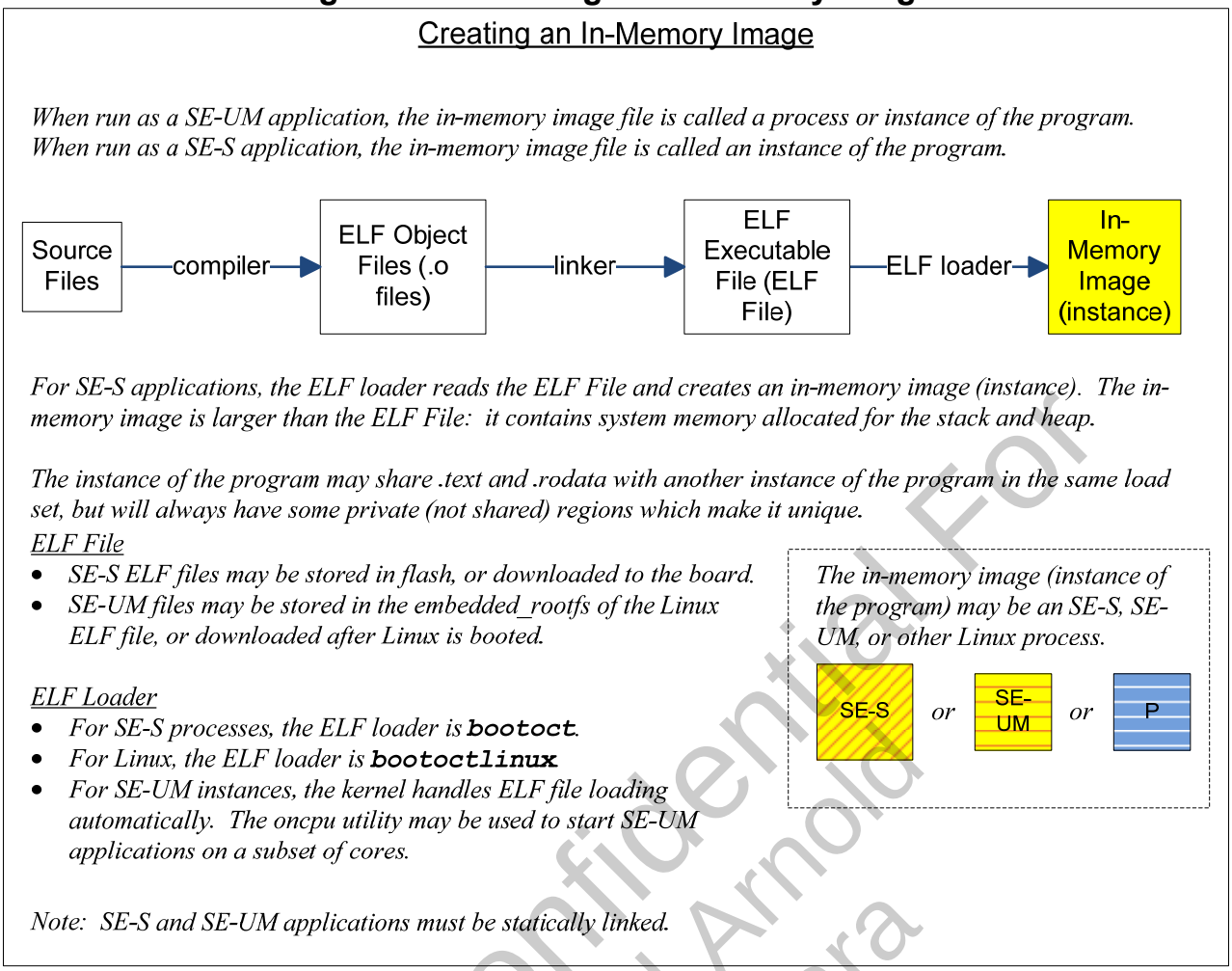
Figure 59: Linux 32-Bit SE-UM Virtual Application Space on OCTEON



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



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: 0x0000000000100000, size:
0x0000000008000000, index: 1
Name: __tmp_fpa_alloc_0, address: 0x000000000ffde800, size:
0x000000000001f400, index: 2
Name: __tmp_fpa_alloc_1, address: 0x000000000ffbe800, size:
0x0000000000020000, index: 3
Name: __tmp_fpa_alloc_2, address: 0x000000000fdca800, size:
0x00000000001f4000, index: 4
Name: cvmx_cmd_queues, address: 0x0000000008100000, size:
0x0000000000007800, index: 5
```

16.1.1 The Reserved Download Block

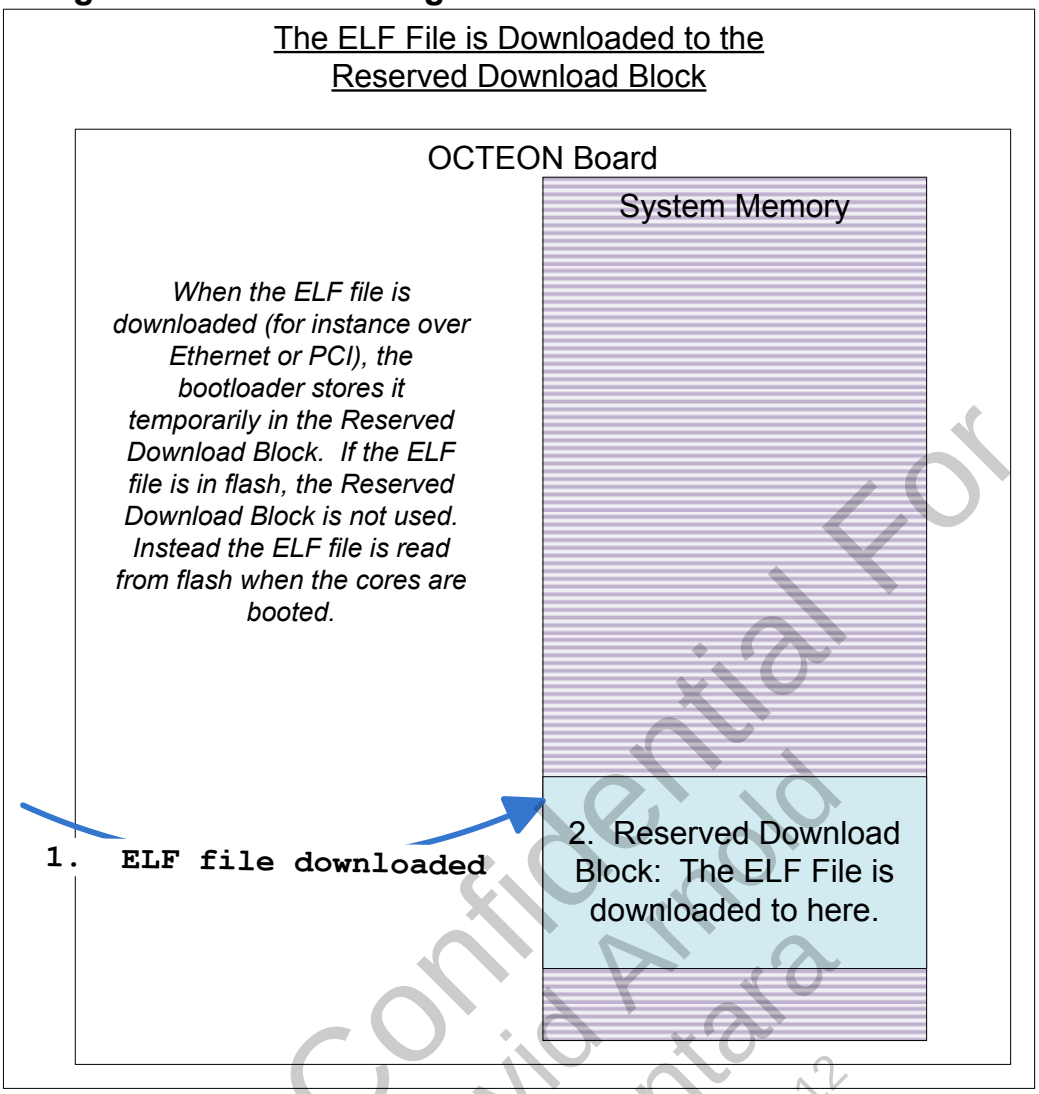
When downloading an ELF file, for instance over PCI, the ELF file is downloaded from the host and stored in memory in a temporary location: the Reserved Download Block.

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.

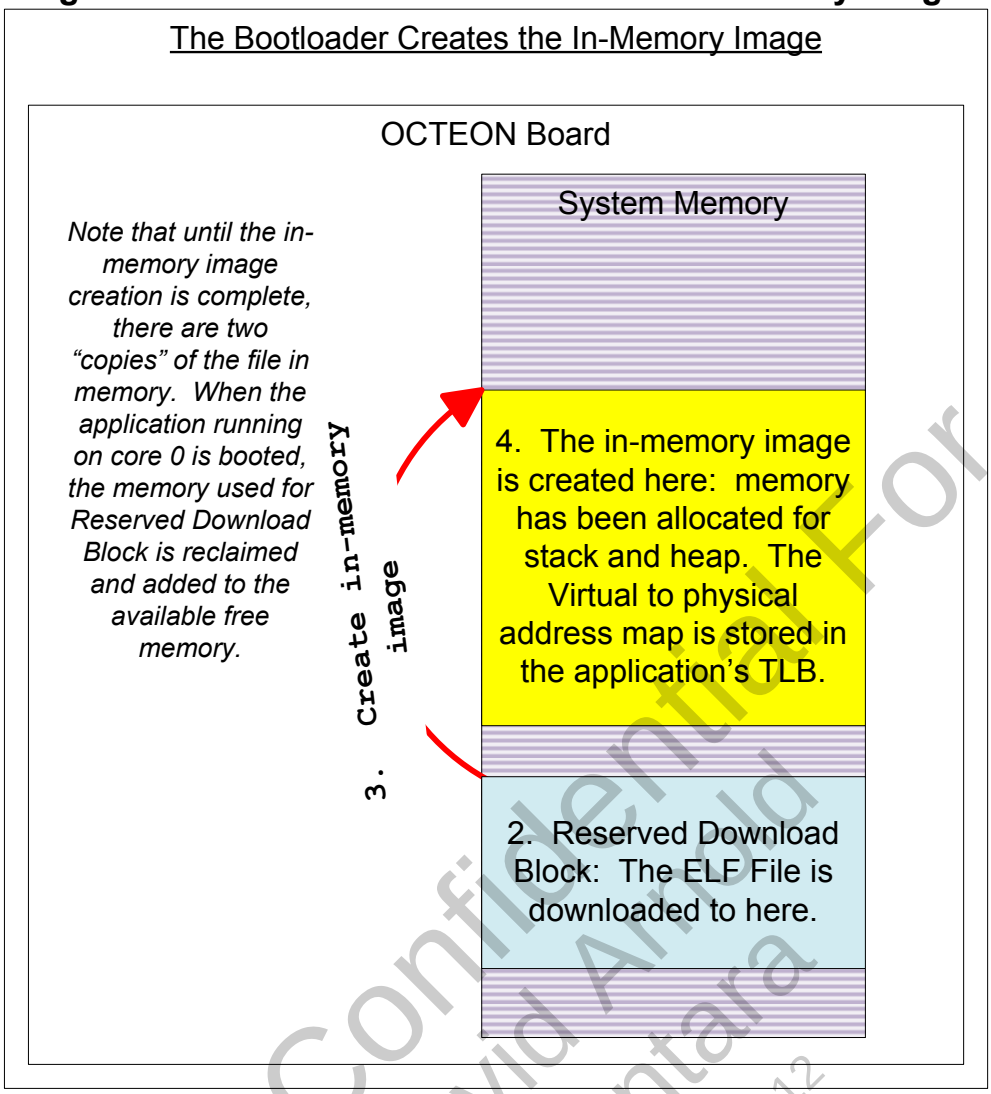
Figure 61: Downloading to the Reserved Download Block



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.

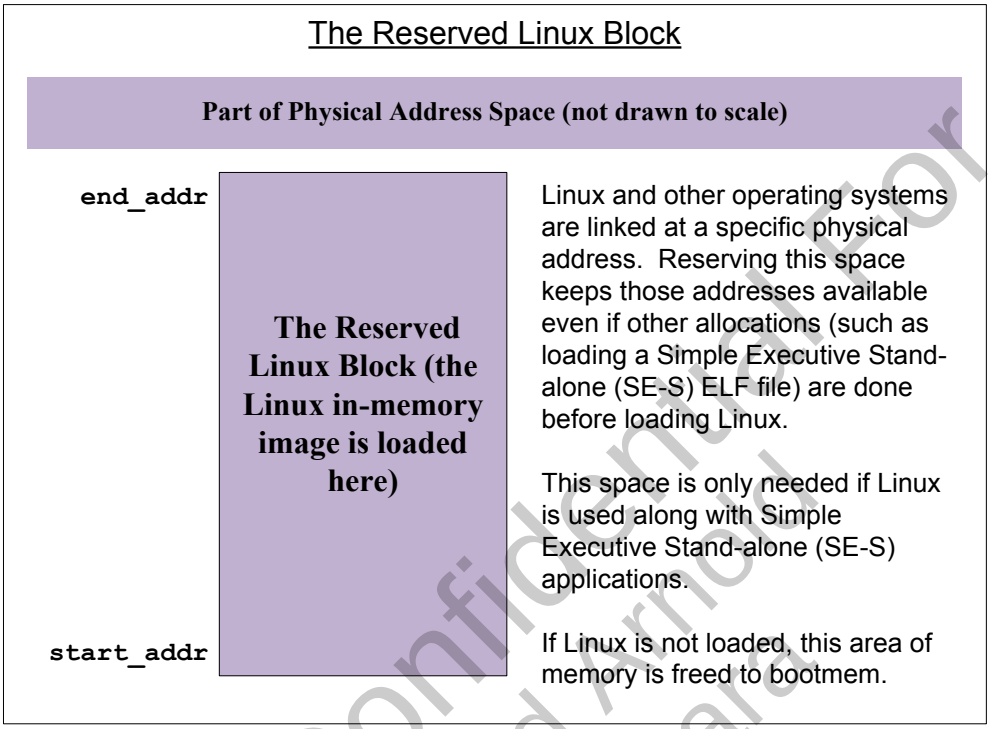
Once the application begins running (when the application running on core 0 is booted), the memory used for Reserved Download Block is reclaimed and added to the available free memory.

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:

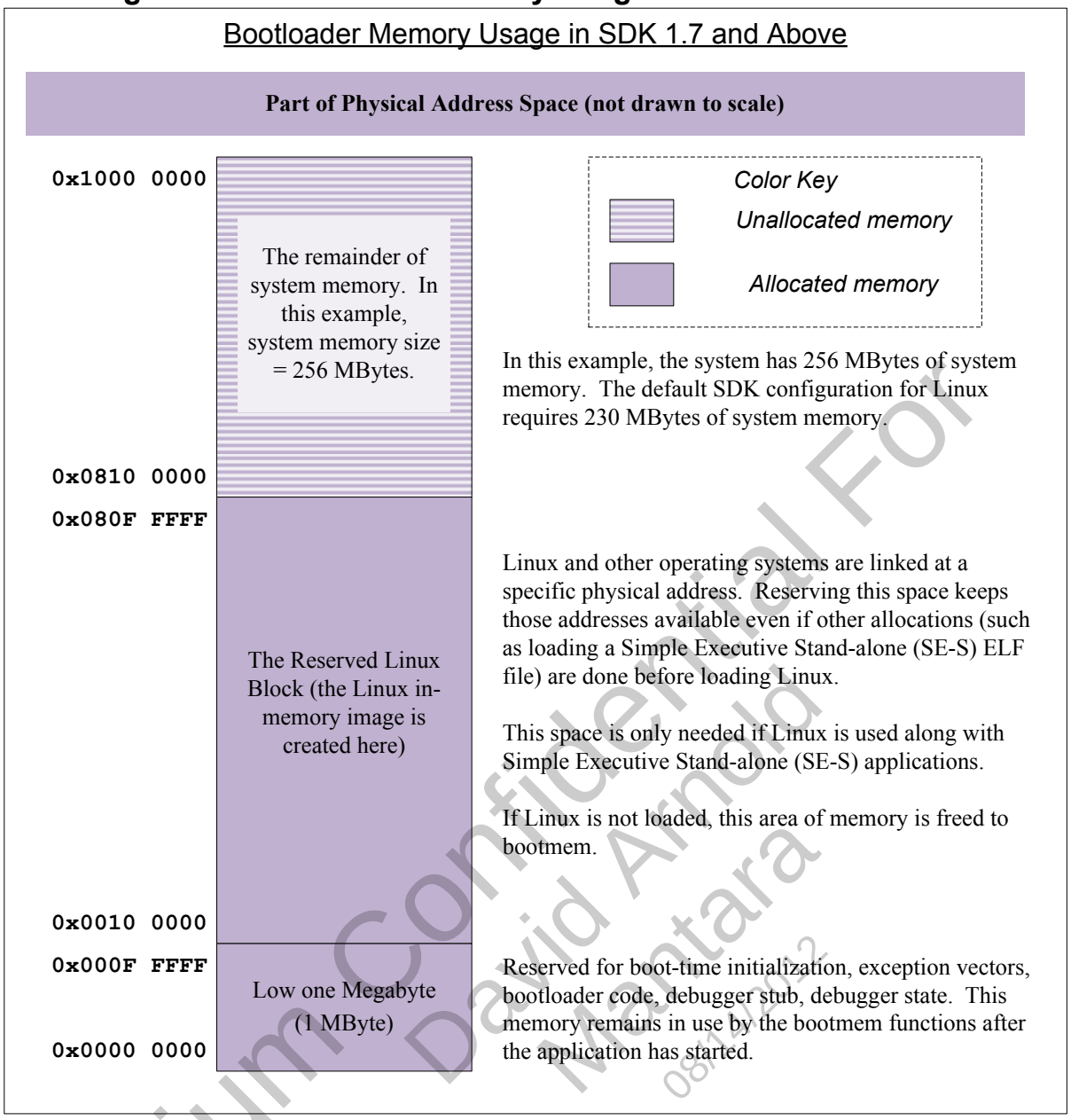
```
Name: __tmp_reserved_linux, address: 0x0000000000100000, size:
0x0000000008000000, index: 1
```

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

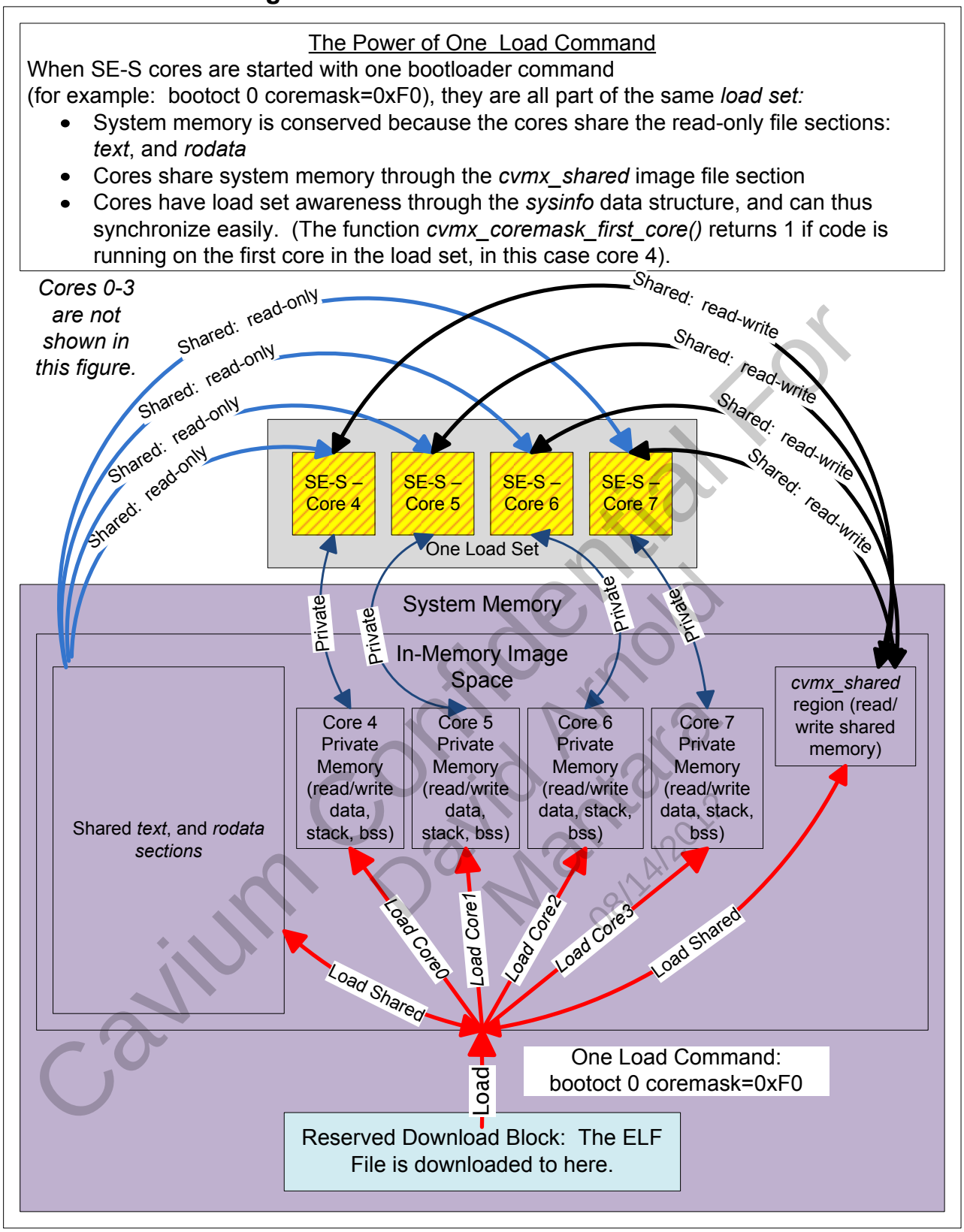


16.2 *Booting the Same SE-S ELF File on Multiple Cores*

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:

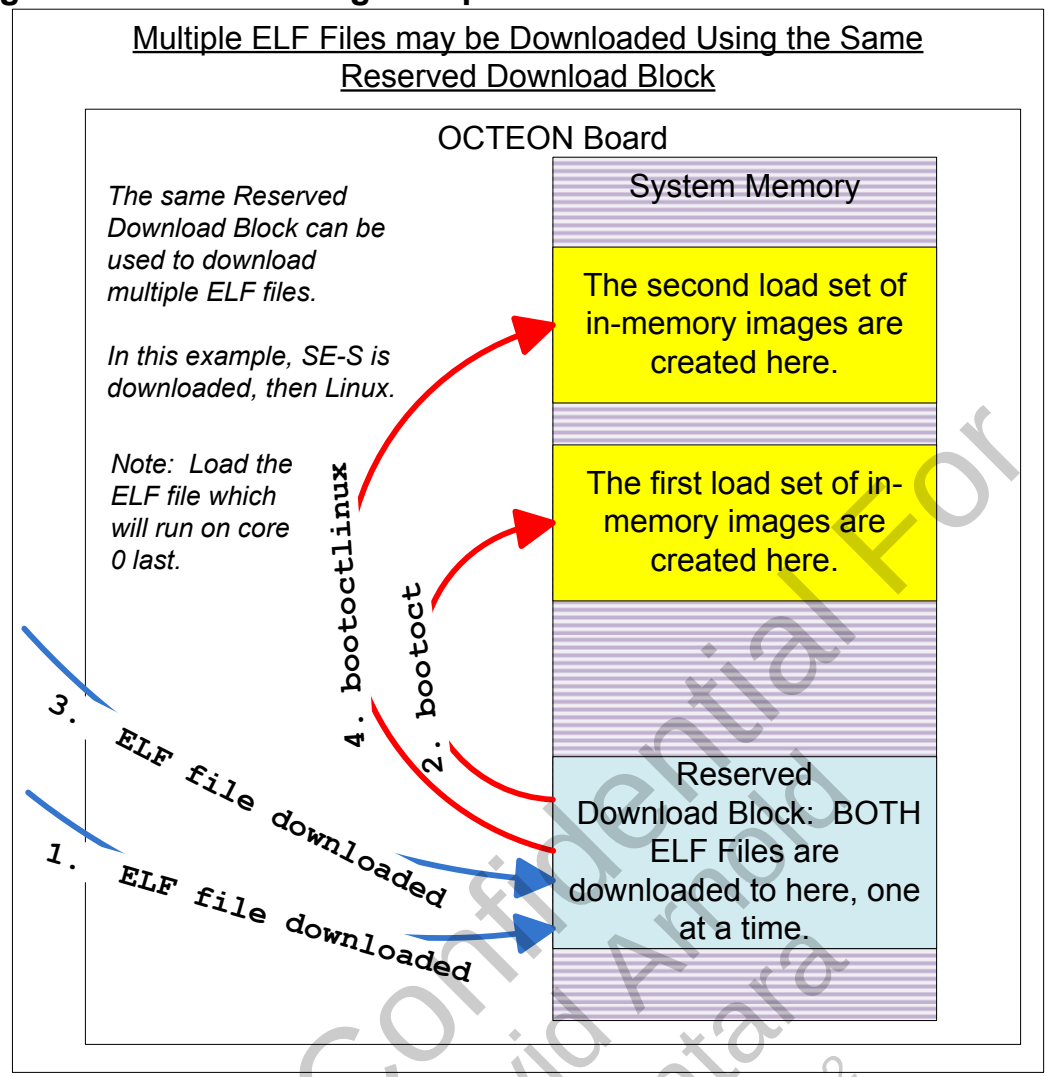
```
host$ oct-pci-load 0 testname/dl/vmlinux.64
host$ oct-pci-bootcmd "bootoctlinux 0 coremask=0xF0"
host$ oct-pci-load 0 testname/dl/linux-filter
host$ oct-pci-bootcmd "bootoct 0 coremask=0x0F"
```

On a Standalone Board:

```
target# dhcp
target# tftpboot 0 testname/dl/vmlinux.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
```

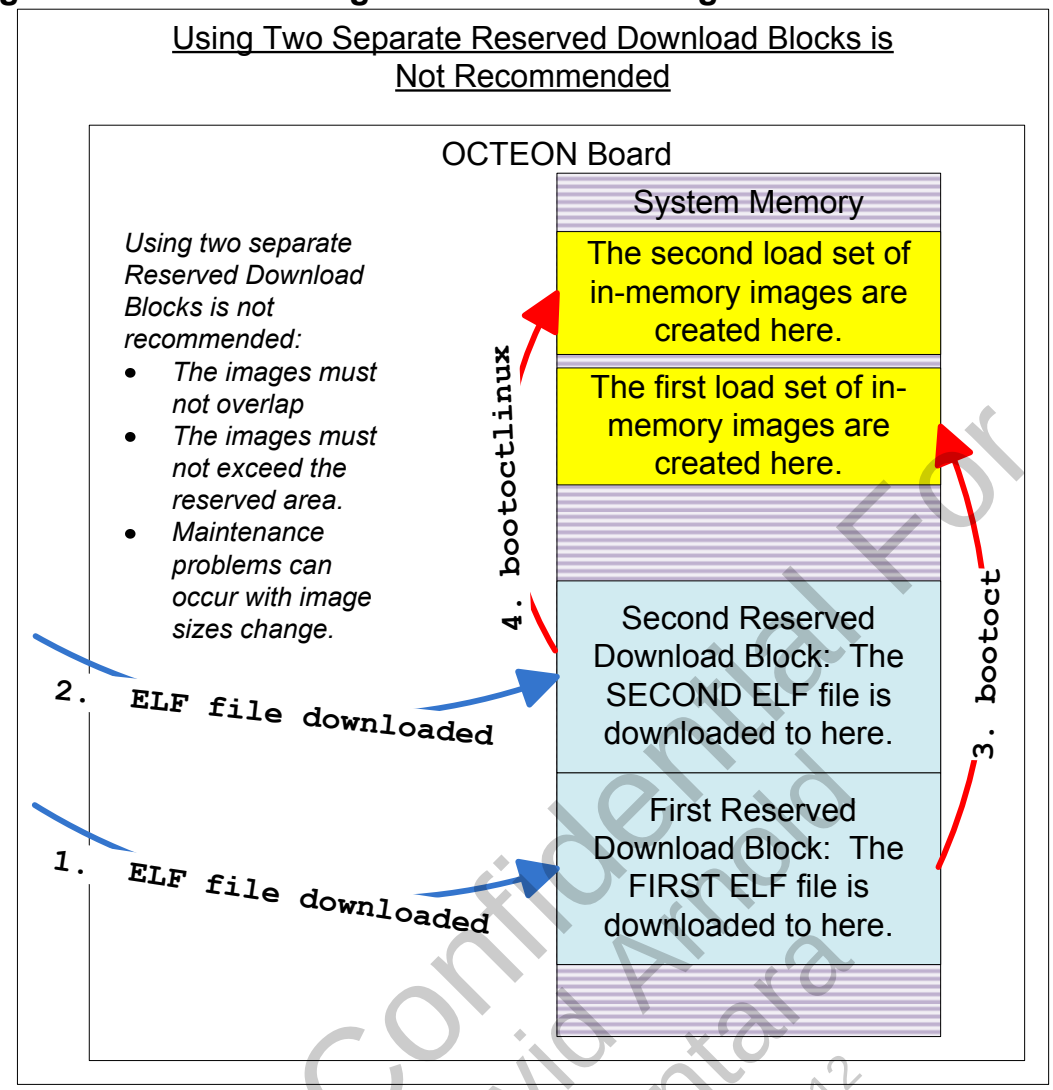
This will show you the Reserved Download Block: "`__tmp_load`". In this example, the start address is `0x20000000`, and the length is 96 MBytes (`0x6000000`).

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: 0x0000000006000000,
index:
0
Name: __tmp_reserved_linux, address: 0x0000000000100000, size:
0x0000000000800000
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         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_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: cvmx_fpa_available_t avail;	Field (N is one of pool 0-7) For example: avail.s.que_siz
FPA_CTL_STATUS	ENB	cvmx_fpa_ctl_status_t	s.enb
FPA_FPF*n*_SIZE	FPF_SIZ	cvmx_fpa_fpf0_size_t	s.fpf_siz
FPA_FPF*n*_MARKS	FPF_RD	cvmx_fpa_fpf_marks_t	s.fpf_rd
FPA_FPF*n*_MARKS	FPF_WR	cvmx_fpa_fpf_marks_t	s.fpf_wr
FPA_INT_ENB	FED0_SBE	cvmx_fpa_int_enb_t	s.fed0_sbe
FPA_INT_ENB	FED0_DBE	cvmx_fpa_int_enb_t	s.fed0_dbe
FPA_INT_ENB	FED1_SBE	cvmx_fpa_int_enb_t	s.fed1_sbe

Register	Field Name	Access from SDK: typedef (union)For example: cvmx_fpa_available_t avail;	Field (N is one of pool 0-7) For example: avail.s.que_siz
FPA_INT_ENB	FED1_DBE	cvmx_fpa_int_enb_t	s.fed1_dbe
FPA_INT_ENB	Q*n*_UND	cvmx_fpa_int_enb_t	s.q*N*_und
FPA_INT_ENB	Q*n*_COFF	cvmx_fpa_int_enb_t	s.q*N*_coff
FPA_INT_ENB	Q*n*_PERR	cvmx_fpa_int_enb_t	s.q*N*_perr
FPA_INT_SUM	FED0_SBE	cvmx_fpa_int_sum_t	s.fed0_sbe
FPA_INT_SUM	FED0_DBE	cvmx_fpa_int_sum_t	s.fed0_dbe
FPA_INT_SUM	FED1_SBE	cvmx_fpa_int_sum_t	s.fed1_sbe
FPA_INT_SUM	FED1_DBE	cvmx_fpa_int_sum_t	s.fed1_dbe
FPA_INT_SUM	Q*n*_UND	cvmx_fpa_int_sum_t	s.q*N*_und
FPA_INT_SUM	Q*n*_COFF	cvmx_fpa_int_sum_t	s.q*N*_coff
FPA_INT_SUM	Q*n*_PERR	cvmx_fpa_int_sum_t	s.q*N*_perr
FPA_QUE*n*_PAGES_AVAILABLE	QUE_SIZ	cvmx_fpa_quex_available_t	s.que_siz
FPA_QUE*n*_PAGE_INDEX	PG_NUM	cvmx_fpa_quex_page_index_t	s.pg_num
FPA_QUE_EXP	EXP_INDX	cvmx_fpa_que_exp_t	s.exp_indx
FPA_QUE_EXP	EXP_QUE	cvmx_fpa_que_exp_t	s.exp_que
FPA_QUE_ACT	ACT_INDX	cvmx_fpa_que_act_t	s.act_indx
FPA_QUE_ACT	ACT_QUE	cvmx_fpa_que_act_t	s.act_que

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:

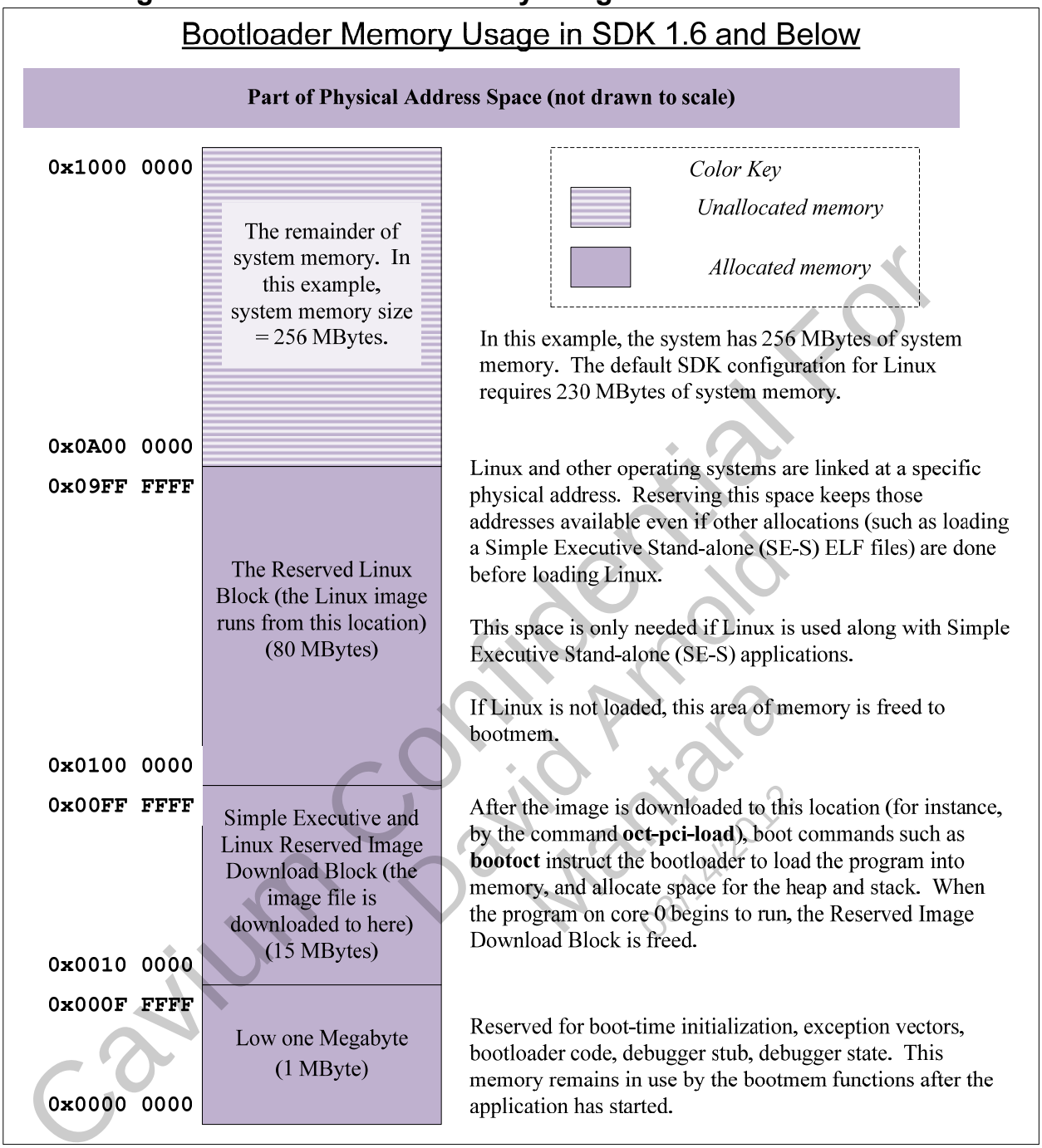
```
target# version
U-Boot 1.1.1 (U-boot build #: 194) (SDK version: 1.7.3-264) (Build time:
Jun 13)
```

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.