

# Using Multicore Navigator

CIV Application Team  
January 2012

# Agenda

## **1. Multicore Navigator Architecture Overview**

- a. Queue Manager Subsystem (QMSS)
- b. Packet DMA (PKTDMA)

## 2. Working Together

## 3. Configuration

## 4. LLD API

## 5. Examples

# What is Multicore Navigator?

- Multicore Navigator is a hardware mechanism that facilitates data movement and multicore co-working
- Supports multiple users (players)
  - Each core in a multicore system
  - High bit-rate peripherals including SRIO, AIF2, Network Coprocessor (NETCP), and PCIe
- Users can think of the Navigator as a mailbox mechanism with many additional and improved functions.
- Designed to be a “fire and forget” system; Load the data and the system handles the rest, without CPU intervention
  - Configuration effort is performed during initialization
  - Enables short and fast run-time operation

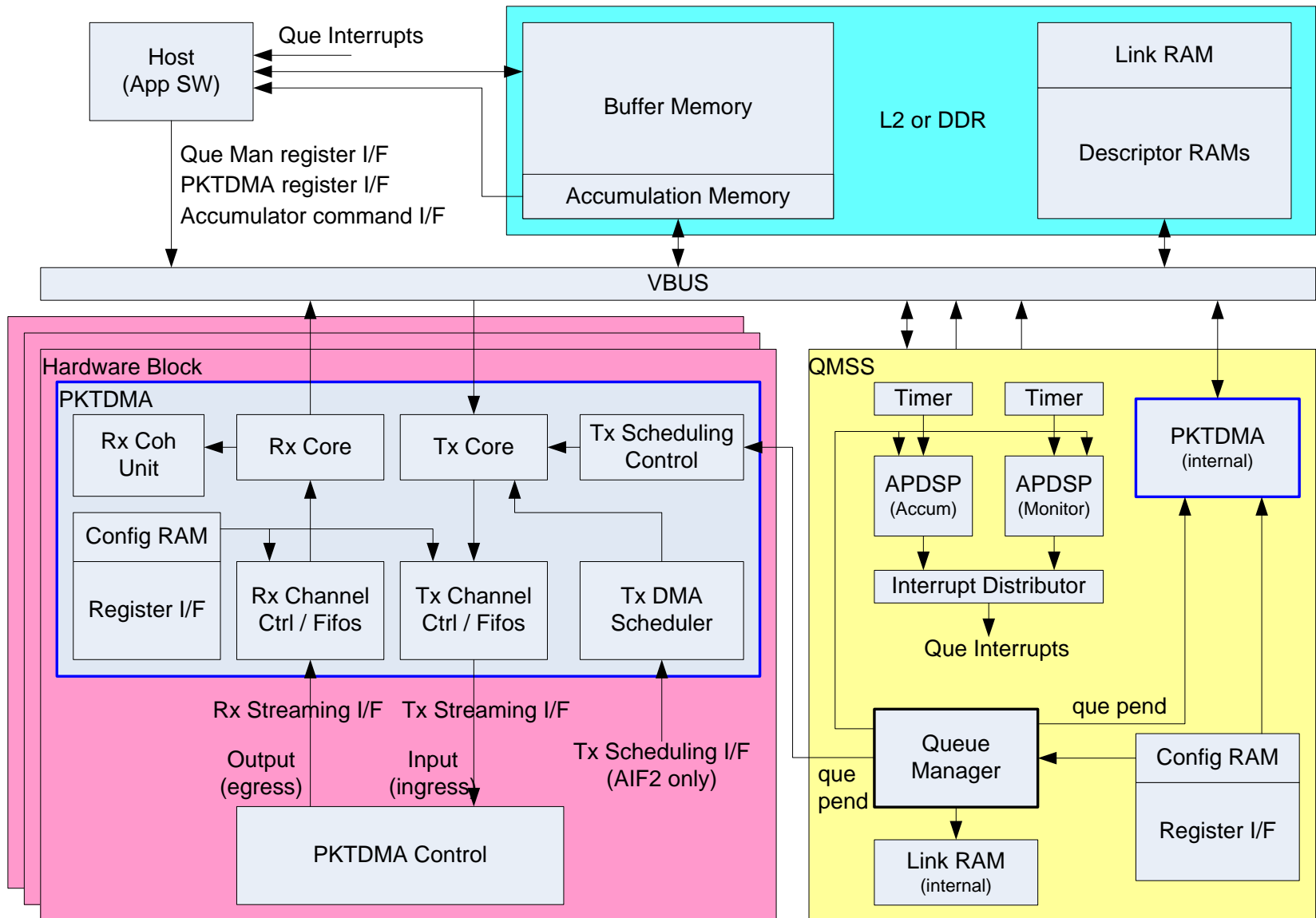
# Multicore Navigator: Typical Use Cases

- Exchanging messages between cores
  - Synchronize execution of multiple cores
  - Move parameters or arguments from one core to another
- Transferring data between cores
  - Output of one core as input to the second
  - Allocate memory in one core, free memory another, without leakage
- Sending data to peripherals
- Receiving data from peripherals
- Load Balancing and Traffic Shaping
  - Enable dynamic optimization of system performance

# Navigator Components

- Hardware-based Queue Manager Sub-System (QMSS)
  - One QMSS instance within the Multicore Navigator
  - One QMSS instance in each of the following modules: SRIO, AIF2, Network Coprocessor (NETCP), and PCIe
- Specialized Packet DMAs (PKTDMA)
  - Commonly referenced in commands and code as CPPI (Communication Peripheral Port Interface)
  - Multiple copies reside within each user (all cores, SRIO, PCIe, NETCP, AIF2, etc.)

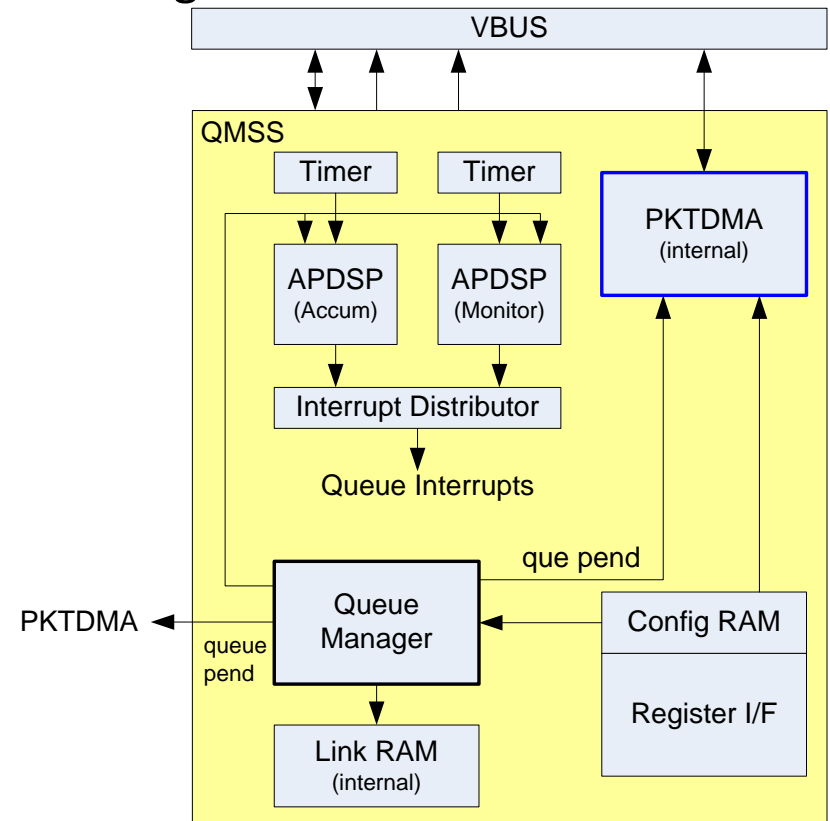
# Multicore Navigator Architecture



# QMSS: Components Overview

Major hardware components of the QMSS:

- Queue Manager
- Two PDSPs (Packed Data Structure Processors):
  - Descriptor Accumulation / Queue Monitoring
  - Load Balancing and Traffic Shaping
- Interrupt Distributor (INTD) module
- Two timers
- Internal RAM for descriptor memory
- PKTDMA that supports all cores



# QMSS: Queues

- Queues are like a mailbox. Descriptors are pushed and popped to and from queues.
- Navigator transactions typically involve two queues:
  - The TX queue of the source
  - The RX queue of the destination
- There are 8192 queues within the QMSS (see mapping on next slide)
- Each queue can be either general purpose queue or associated with functionality
- Queues associated with queue pending signals should not be used for general use, such as free descriptor queues (FDQs). Others can be used for any purpose.



# QMSS: Queue Mapping

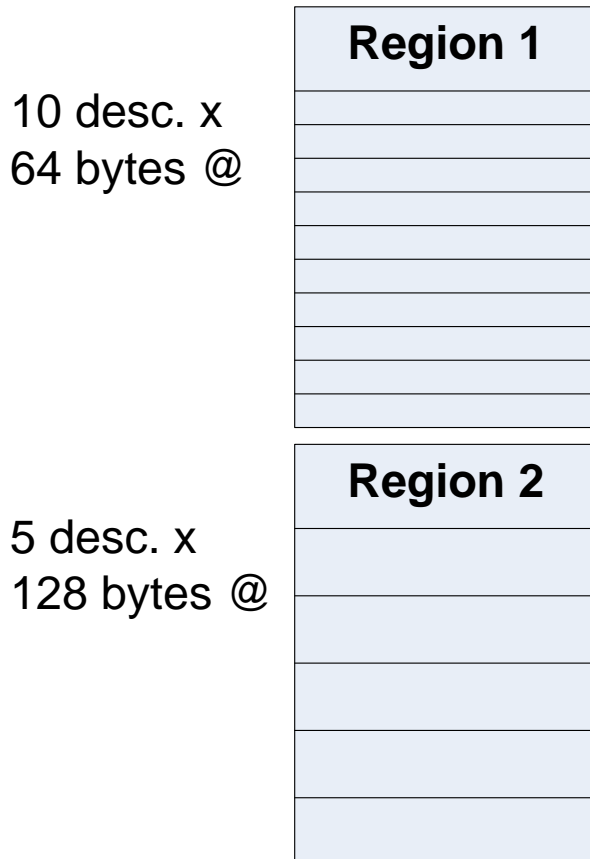
Queue Range	Count	Hardware Type	Purpose
0 to 511	512	pdsp/firmware	Low Priority Accumulation queues
512 to 639	128	queue pend	AIF2 Tx queues
640 to 651	12	queue pend	PA Tx queues (PA PKTDMA uses the first 9 only)
652 to 671	20	queue pend	CPintC0/intC1 auto-notification queues
672 to 687	16	queue pend	SRIO Tx queues
688 to 695	8	queue pend	FFTC_A and FFTC_B Tx queues (688..691 for FFTC_A)
696 to 703	8		General purpose
704 to 735	32	pdsp/firmware	High Priority Accumulation queues
736 to 799	64		Starvation counter queues
800 to 831	32	queue pend	QMSS Tx queues
832 to 863	32		Queues for traffic shaping (supported by specific firmware)
864 to 895	32	queue pend	vUSR queues for external chip connections
896 to 8191	7296		General Purpose

# QMSS: Descriptors

- Descriptors are messages that move between queues and carry information and data.
- Descriptors are allocated in the memory region (see next slide)
- 20 Memory regions for descriptor storage (LL2, MSMC, DDR)
- 1 or 2 link RAM that (link list) to index the descriptors (internal memory to QMSS or other memory )
- Up to 16K descriptors can be handled by internal Link RAM (Link RAM 0)
- Up to 512K descriptors can be supported in total.

# QMSS: Descriptor Memory Regions

- All Navigator descriptor memory regions are divided into *equal sized* descriptors. For example:

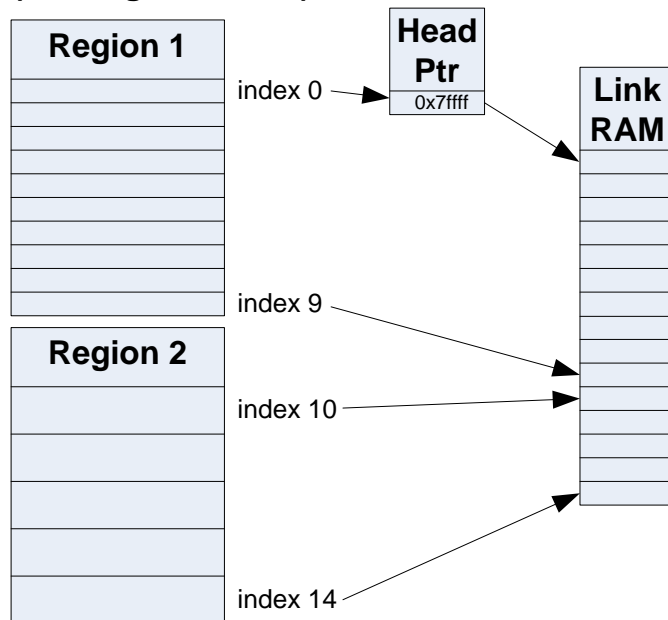


Memory regions are always aligned to 16-byte boundaries and descriptors are always multiples of 16 bytes.

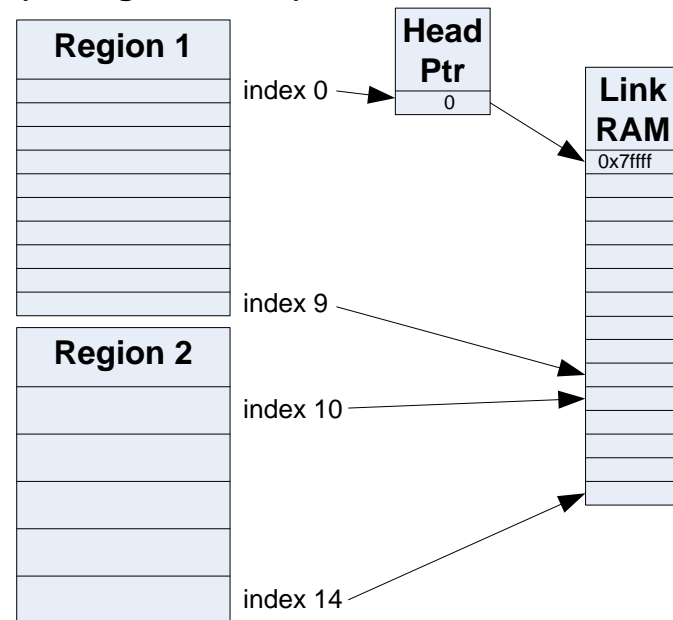
# QMSS: Descriptor Queuing

- The Queue Manager maintains a head pointer for each queue, which are initialized to be empty.

Push index 0 to an empty queue  
(starting condition)



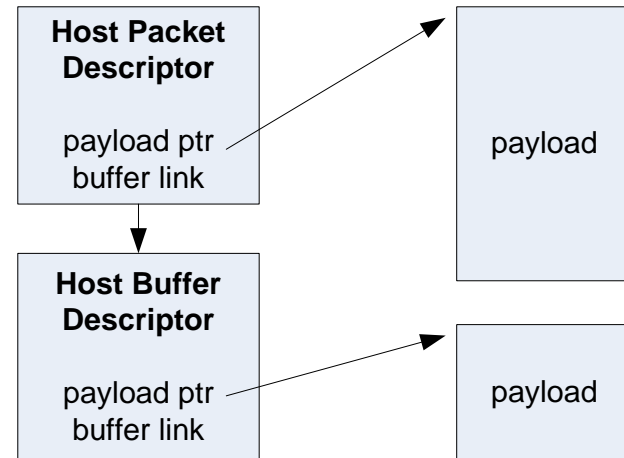
Push index 0 to an empty queue  
(ending condition)



- We actually do not push indexes; We push descriptor addresses. The QM converts addresses to indexes.

# QMSS: Descriptor Types

- Two descriptor types are used within Navigator:
  - **Host** type provide flexibility, but are more difficult to use
    - Contains a header with a pointer to the payload.
    - Can be linked together (packet length is the sum of payload (buffer) sizes).
  - **Monolithic** type are less flexible, but easier to use
    - Descriptor contains the header and payload.
    - Cannot be linked together.
    - All payload buffers are equally sized (per region).

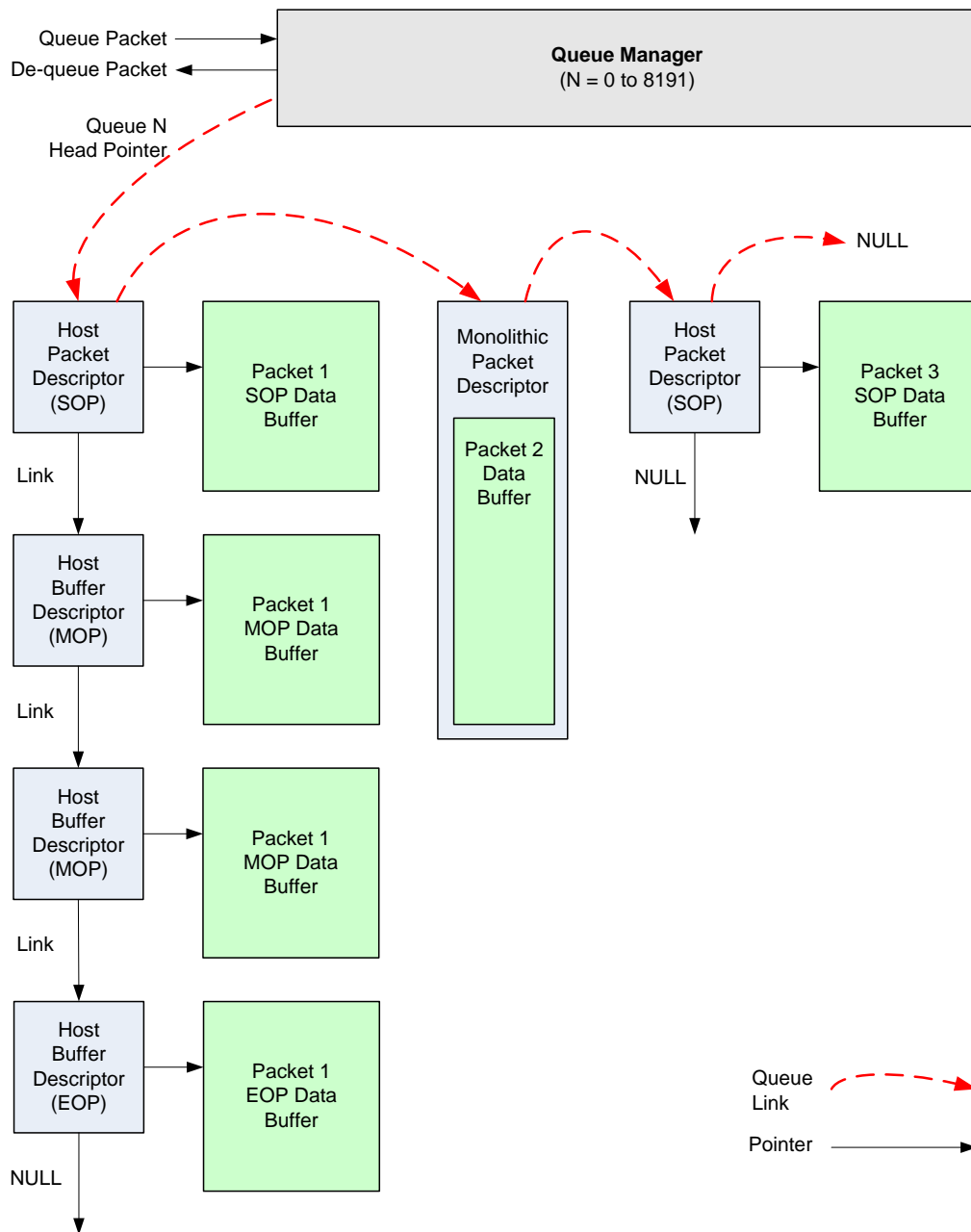


# Descriptor Queuing

This diagram shows several descriptors queued together.

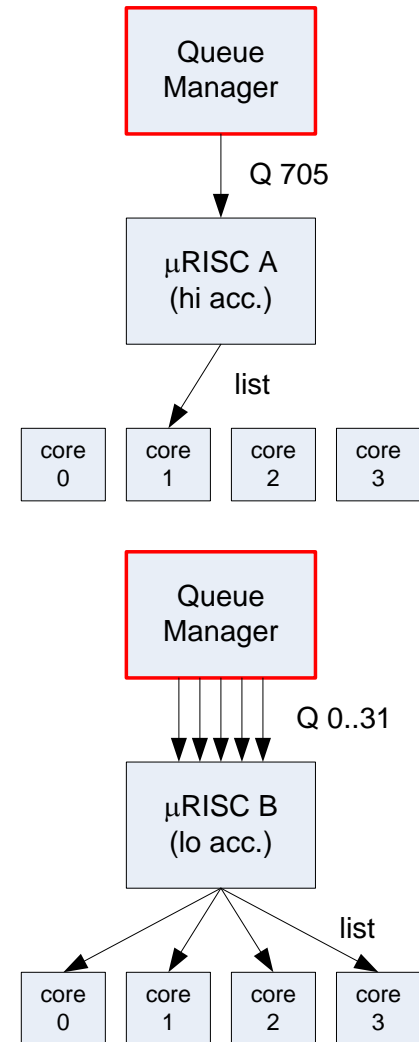
Things to note:

- Only the Host Packet is queued in a linked Host descriptor.
- A Host Packet is always used at SOP, followed by zero or more Host Buffer types.
- Multiple descriptor types may be queued together, though not commonly done in practice.

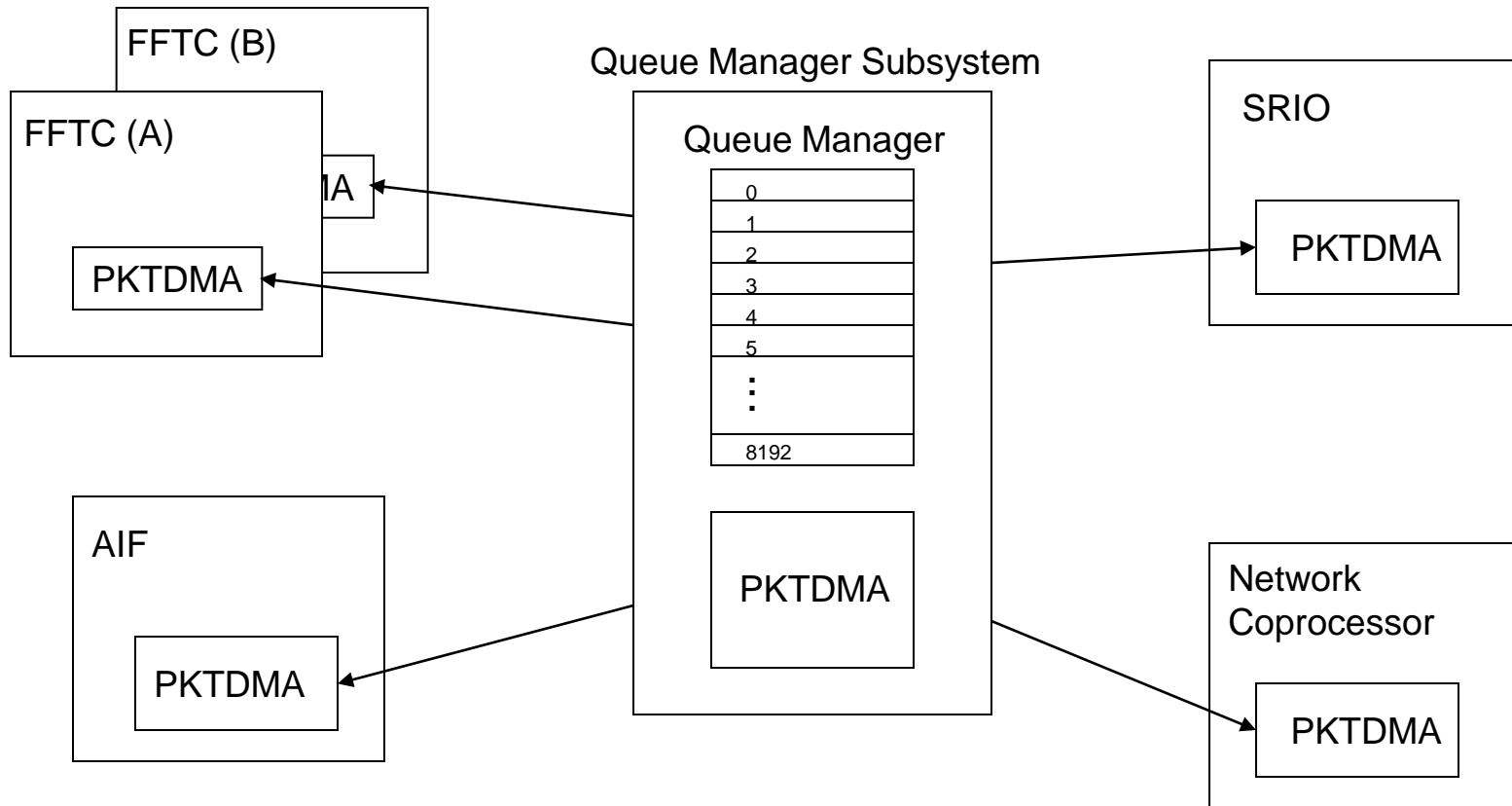


# Descriptor Accumulators

- Accumulators keep the cores from polling.
- Run in background, interrupts core with list of popped descriptor addresses.
- Core software must recycle.
- High Priority Accumulator:
  - 32 channels, one queue per channel
  - All channels scanned each timer tick (25us)
  - Each channel/event maps to 1 core
  - Programmable list size and options
- Low Priority Accumulator:
  - 16 channels, up to 32 queues per channel
  - 1 channel scanned each timer tick (25 us)
  - Each channel/event maps to all cores
  - Programmable list size and options



# Packet DMA Topology



Multiple Packet DMA instances in KeyStone devices:

- PA and SRIO instances for all KeyStone devices.
- AIF2 and FFTC (A and B) instances are only in KeyStone devices for wireless applications.



# Packet DMA (PKTDMA)

Major components for each instance:

- Multiple RX DMA channels
- Multiple TX DMA channels
- Multiple RX flow channels. RX flow defines behavior of the receive side of the navigator

# Packet DMA (PKTDMA) Features

- Independent Rx and Tx cores:
  - Tx Core:
    - Tx channel triggering via hardware qpend signals from QM.
    - Tx core control is programmed via descriptors.
    - 4 level priority (round robin) Tx Scheduler
      - Additional Tx Scheduler Interface for AIF2 (wireless applications only)
  - Rx Core:
    - Rx channel triggering via Rx Streaming I/F.
    - Rx core control is programmed via an “Rx Flow” (more later)
- 2x128 bit symmetrical Streaming I/F for Tx output and Rx input
  - These are wired together for loopback in QMSS’ PKTDMA.
  - Connects to peripheral’s matching streaming I/F (Tx->Rx, Rx->Tx)
- Packet based, so neither Rx or Tx cores care about payload format.

# Agenda

1. Overview architecture of the Multicore navigator
  - a. Overview of the queue manager
  - b. Overview of the packet DMA
- 2. Working together**
3. Item to configure
4. LLD API
5. Examples

# How does it works during run time?

For example, core A wants to send a message to core B

- Core A picks available descriptor (you can call it a message structure) that is partially or completely pre-built
  - If need core A adds missing information
- Core A pushes the descriptor into a queue
  - At this point core A is done
- The Navigator process the message and send it to a queue in the receive side of core B where it follows a set of pre-defined instructions (Rx flow) such as
  - Interrupt core B and tell it to process the message
  - Set a flag so core B can pull , change a flag value that core B synchronizes on
  - Move buffer into core B memory space and interrupt the core
- After usage, the navigator recycles the descriptors (and any buffer associated with) to prevent memory leak

# Agenda

1. Overview architecture of the Multicore navigator
  - a. Overview of the queue manager
  - b. Overview of the packet DMA
2. Working together
- 3. Item to configure**
4. LLD API
5. Examples

# What Needs to be Configured?

- **Link Ram** - Up to two LINK-RAM
  - one internal, Region 0, address 0x0008 0000, size up to 16K
  - One External, global memory, size up to 512K
- **Memory Regions** - Where Descriptors actually reside
  - Up to 20 regions, 16 bytes alignment
  - Descriptor size is multiple of 16 bytes, minimum 32
  - Descriptor count (per region) is power of 2, minimum 32
  - Configuration – base address, start index in the LINK RAM, size and number of descriptors
  - **The way the region is managed**
- **Loading PDSP firmware**

# What Needs to be Configured?

- **Descriptors**
  - Create and initialized
  - Allocated data buffers and associate them with descriptors
- **Queues**
  - Open transmit, receive, free and error queues
  - Define receive flows
  - Configure transmit and receive queues
- **PKTDMA**
  - Configure all PKTDMA in the system
  - Special configuration for PDKDMA that are used

# Agenda

1. Overview architecture of the Multicore navigator
  - a. Overview of the queue manager
  - b. Overview of the packet DMA
2. Working together
3. Item to configure
- 4. LLD API**
5. Examples



# QMSS Low Level Driver (LLD)

Provides an abstraction of register-level details.

- Provides two usage modes:
  - User manages/selects resources to be used
    - Generally faster
  - LLD manages/selects resources
    - Generally easier
- Allocates a minimal amount of memory for bookkeeping purposes.
- Built as two drivers:
  - QMSS LLD is a standalone driver for QM and Accumulators.
  - CPPI LLD is a driver for PKTDMA that requires the QMSS LLD.
- The following slides do not present the full API.

# QMSS LLD Initialization

- The following are one-time initialization routines to configure the LLD globally:
  - `Qmss_init(parms, queue_mapping);`
    - Configures Link RAM, # descriptors, queue mapping
    - May be called on one or all cores
  - `Qmss_exit();`
    - Deinitializes the QMSS LLD

# QMSS Configuration

- More QMSS configuration:
  - `Qmss_start( );`
    - Called once on every core to initialize config parms on those cores.
    - Must be called immediately following `Qmss_init()`
  - `Qmss_insertMemoryRegion(mem_parms);`
    - Configures a single memory region.
    - Should be called with protection so that no other tasks or cores could simultaneously create an overlapping region.

# QMSS LLD Queue Usage

- To allocate and release queues:
  - `queue_handle = Qmss_queueOpen(type, que, *flag);`
    - Once “open”, the DSP may push and pop to the queue.
      - type refers to an enum (tx queue, general purpose, etc.).
      - que refers to the requested queue number.
      - flag is returned true if the queue is already allocated.
  - `Qmss_queueClose(queue_handle);`
    - Releases the handle preventing further use of the queue

# Queue Push and Pop

- Queue management APIs:
  - `Qmss_queuePushDesc(queue_handle, desc_ptr);`
    - Pushes a descriptor address to the handle's queue.
    - Other APIs are available for pushing sideband info as well.
  - `desc_ptr = Qmss_queuePop(queue_handle);`
    - Pops a descriptor address from the handle's queue.
  - `count = Qmss_getQueueEntryCount(queue_handle);`
    - Returns the number of descriptors in the queue.

# QMSS Accumulator

- The following functions are available to program, enable, and disable an accumulator:
  - `Qmss_programAccumulator(type, *program);`
    - Programs/enables one accumulator channel (high or low)
    - Setup of the ISR is done outside the LLD using INTC
  - `Qmss_disableAccumulator(type, channel);`
    - Disables one accumulator channel (high or low)

# CPPI LLD Initialization

- The following are one-time initialization routines to configure the LLD globally:
  - `Cppi_init(pktdma_global_parms);`
    - Configures the LLD for one PKTDMA instance
    - May be called on one or all cores
    - Must be called once for each PKTDMA to be used
  - `Cppi_exit();`
    - Deinitializes the CPPI LLD

# CPPI LLD: PKTDMA Channel Setup

- More handles to manage in using the PKTDMA LLD
- To allocate a handle for a PKTDMA:
  - `pktdma_handle = CPPI_open(pktdma_parms);`
    - Returns a handle for one PKTDMA instance
    - Should be called once for each PKTDMA required.
- To allocate and release Rx channels:
  - `rx_handle = Cppi_rxChannelOpen(pktdma_handle, cfg, *flag);`
    - Once “open”, the DSP may use the Rx channel.
      - `cfg` refers to the Rx channel’s setup parameters
      - `flag` is returned true if the channel is already allocated
  - `Cppi_channelClose(rx_handle);`
    - Releases the handle preventing further use of the queue



# More Packet DMA Channel Setup

- To allocate and release Tx channels:
  - `tx_handle = Cppi_txChannelOpen(pktdma_handle, cfg, *flag);`
    - Same as the Rx counterpart
  - `Cppi_channelClose(tx_handle);`
    - Same as the Rx counterpart
- To configure/open an Rx Flow:
  - `flow_handle = Cppi_configureRxFlow(pktdma_handle, cfg, *flag);`
    - Similar to the Rx channel counterpart

# PKTDMA Channel Control

- APIs to control Rx and Tx channel use:
  - `Cppi_channelEnable(tx/rx_handle);`
    - Allows the channel to begin operation
  - `Cppi_channelDisable(tx/rx_handle);`
    - Allows for an immediate, hard stop.
    - Usually not recommended unless following a pause.
  - `Cppi_channelPause(tx/rx_handle);`
    - Allows for a graceful stop at next end-of-packet
  - `Cppi_channelTeardown(tx/rx_handle);`
    - Allows for a coordinated stop

# QMSS/CPPI LLD – Runtime Use

- Once initialization is finally complete, control is very simple:
  - `desc_ptr = Qmss_queuePop(queue_handle);`
    - Pop a descriptor address from a queue.
  - `Cppi_setData(type, *inbuf, *desc_ptr, len);`
    - Converts an “LLD format” descriptor to hardware format.
  - `Qmss_queuePushDesc(queue_handle, desc_ptr);`
    - Push the filled descriptor to a queue corresponding to a Tx DMA channel for processing.

# Agenda

1. Overview architecture of the Multicore navigator
  - a. Overview of the queue manager
  - b. Overview of the packet DMA
2. Working together
3. Item to configure
4. LLD API
- 5. Examples**

# Example

Part of PDK (Platform Development Kit) release is a set of examples for each of the peripherals

Several examples use the Navigator and can be used as starting point for development.

Location of the examples:

`pdk_C6678_1_0_0_18\packages\ti\drv\exampleProjects\`

Examples that use Navigator:

1. QM
2. CPPI
3. PA
4. SRIO

# For More Information

- For more information, refer to the to Multicore Navigator User Guide <http://www.ti.com/lit/SPRUGR9>
- For questions regarding topics covered in this training, visit the support forums at the [TI E2E Community](#) website.

# Backup

# Data Movement: Normal

- Peripheral input and output:
  - Drive data through IP block using QM and PKTDMA
  - Simple transmit is shown
- Infrastructure or core-to-core transfers:
  - Transfer payload from L2 to L2, or DDR

