TEXAS INSTRUMENTS
20450 Century Boulevard
Germantown, MD 20874
Fax: (301) 515-7954

CPPI/QMSS Low Level Driver

Software Design Specification (SDS)

Revision A

Apr 11, 2011

Revision Record	
Document Title: **Software Design Specification**	
Revision	Description of Change
A	1. Initial Release – Code Drop 1.0.0.0 2. OSAL API changes – Code Drop 1.0.0.1 3. Shared memory allocation considerations – Code Drop 1.0.0.2 4. Removed teardown descriptor. Added EOI and register C pop. Documented device specific layer – Code Drop 1.0.0.3 5. Changed on-chip to return push policy field in monolithic descriptor Documented intc pending queues – Code Drop 1.0.0.5 6. Cache Hooks – Code Drop 1.0.0.10 Checklist to avoid common configuration mistakes 7. Exported device specific – Code Drop 1.0.0.11 8. Updated checklist and qmss_device.c sample file

TABLE OF CONTENTS

1 Scope

Documents the CPPI and QMSS Low Level Driver design.

2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

No	Referenced Document	Control Number	Description
1	CPPI user guide	Version 0.5.7	CPPI User Guide
2	CPPI LLD API Documentation	Version 1.0.0.15	DOXYGEN generated API documentation located in the package under the "`docs`" directory in CHM format.
2	QMSS LLD API Documentation	Version 1.0.0.15	DOXYGEN generated API documentation located in the package under the "`docs`" directory in CHM format.

Table 1. Referenced Materials

3 Definitions

Acronym	Description
CPPI	Communications Port Programming Interface
LLD	Low Level Driver
QM	Queue Manager
QMSS	Queue Manager Sub System
PASS	Packet accelerator Sub System
SRIO	Serial Rapid Input Output
FFTC	Fast Fourier Transform Co-processor
IPC	Inter-process Communication
TCP3d	Turbo Co-processor decoder
LTE	Long Term Evolution
AIF	Antenna Interface
BD	Buffer Descriptor

Acronym	Description
PD	Packet Descriptor
EOI	End Of Interrupt
QoS	Quality of Service
INTD	Interrupt Distributer

Table 2. Definitions

4 Overview

This document describes the Communication Port programming Interface (CPPI) and Queue Mananger SubSystem (QMSS) low level driver design. This driver will potentially be used by FFTC, SRIO, AIF2, PASS, TCP3d, LTE stack.

The idea of providing a low level driver is to abstract the complexities of CPPI and QMSS while still allowing the drivers and applications to control and configure every aspect of CPPI and QMSS.

The low level driver is provided as two separate libraries. The QMSS library can be used as a stand alone library.

The CPPI library is also a user of the QMSS library.

5 Design

5.1 DESIGN GOALS

The design goals for the low level drivers are as follows:

1. Performance
   The low level drivers are designed with performance requirements at the forefront. Various levels of APIs are provided to quickly configure either the hardware or optional parameters.

2. OS independence
   The low level drivers are designed to be OS independent to ease porting from one OS to another.

3. Hardware abstraction
   The hardware details are identified and abstracted as much as possible to ease porting to another SoC.

4. Multicore awareness
   The low level drivers are designed to be used by multiple drivers, applications running on multiple cores.

5.2 DESIGN DETAILS

Figure 1 illustrates the high level CPPI and QMSS blocks and the interactions of application, driver or library with CPPI LLD and QMSS LLD.

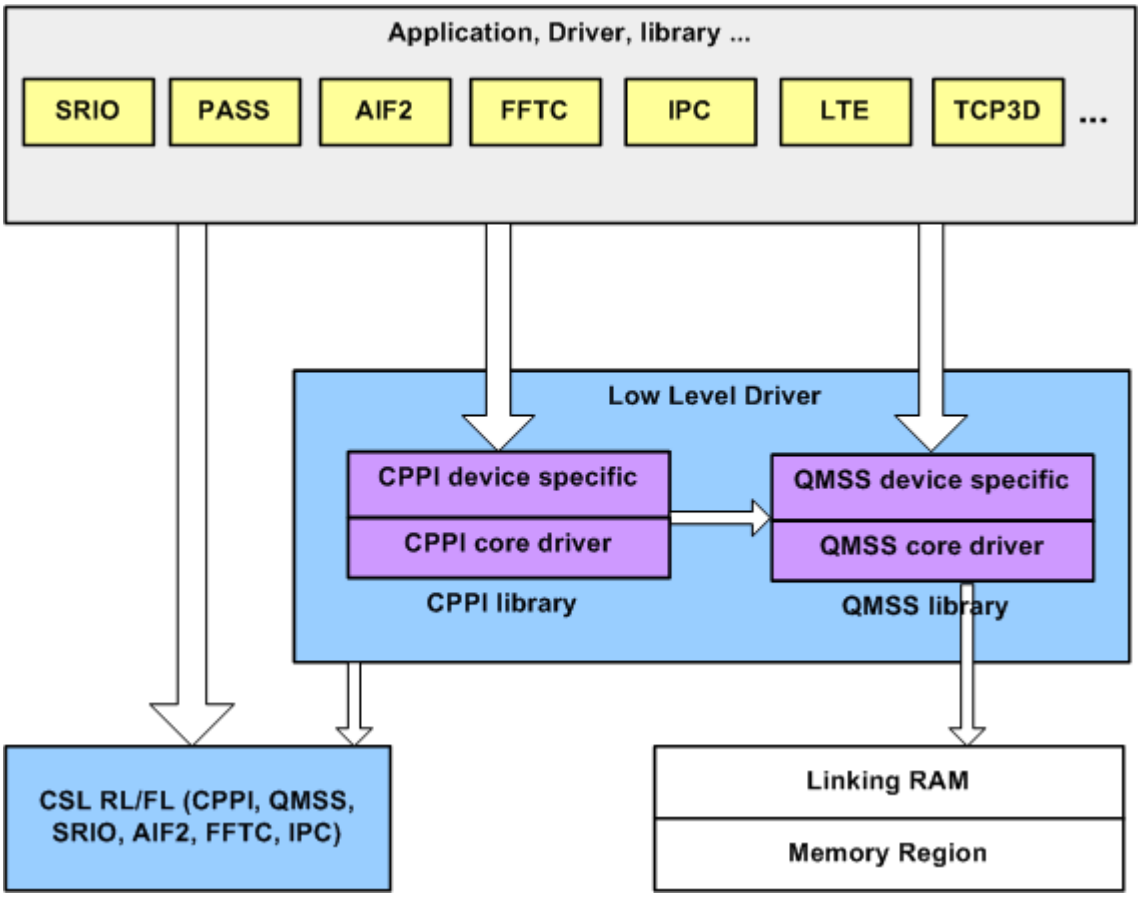

Figure 1 CPPI LLD and QMSS LLD Usage

5.3 QMSS DESIGN DETAILS

The Queue Manager provides high-performance descriptor management. A total of 512 K descriptors can be managed by the queue manager and these descriptors can be pooled into up to 8 K queues. To track the location of the descriptors in the queues, the queue manager uses 4x512 KB of memory. Up to 4x16 KB of this memory can be in the Linking RAM that is embedded inside the QMSS and the remaining memory can be located elsewhere inside or outside the device.

The descriptors that are managed by the queue manager can be clustered in up to 20 memory regions that can be independently located anywhere in the device's memory map. The queue manager is provided information about the location of these regions and the size and number of descriptors in each of these regions.

Each descriptor is referenced by its memory address pointer. A descriptor is added to a queue by writing the pointer to the queue's address and removed from a queue by reading the pointer from the queue's address.

QMSS low-level driver handles the queue manager subsystem.

Note: The LLD is designed to meet performance requirements on a local Keystone device. Using it to communicate with a remote keystone device is not supported. This is done so that the local device accesses don't have to pay the performance penalty required to differentiate between local and remote device accesses.

- It provides resource management for a variety of queue types listed below.

Queue Start Index	Queue count	Queue Type
0	512	Queues with low priority accumulation and interrupt support
512	128	AIF queues with hardware queue threshold status
640	32	PA_SS queues with hardware queue threshold status
672	16	SRIO queues with hardware queue threshold status
688	4	FFTC A queues with hardware queue threshold status
692	4	FFTC B queues with hardware queue threshold status
696	8	General purpose queues

704	32	Queues with high priority accumulation and interrupt support
736	64	Queues with starvation counters
800	32	Transmit Infrastructure hardwired with pending signals
832	32	Traffic Shaping
864	7328	General purpose queues

Table 3 Sample Queue Allocation Device Specific

- It provides descriptor allocation functionality.
- It provides enqueue and dequeue functions to queue descriptors onto logical queues.
- It provides accumulator functionality.
- It provides threshold configuration and status information on all of the 8K queues
- It can be used to configure infrastructure DMA mode.

5.3.1 QMSS LLD Initialization

This API is the entry point into QMSS low level driver. The QMSS LLD is initialized only once in the system using the *Qmss_init()* API. The function sets up the low-level driver with information pertaining to linking RAM, performs device-specific initialization such as number of supported queues and addresses for memory mapped registers. It also downloads the PDSP firmware if a valid firmware image is specified.

```
Qmss_Result Qmss_init (Qmss_InitCfg *initCfg , Qmss_GlobalConfigParams
*qmssGblCfgParams)
```

The inputs to this function are QMSS initialization parameters.

5.3.1.1 QMSS initialization parameters

The QMSS LLD provides a structure that should be filled with the required information.

- Linking RAM0 base address
- Linking RAM0 size
- Linking RAM1 base address
- Maximum number of descriptors required in the system
- PDSP firmware image

Note: The total linking RAM size (size0 and size1) must be at least equal to the total number of descriptors in all memory regions.
If Linking RAM1 size is zero, Linking RAM0 must be big enough to store all the descriptors.

The *Qmss_InitCfg* structure that needs to be filled and sent as an input parameter to the *Qmss_init()* function is listed below.

```
/**
 * @brief QMSS configuration structure
 */
typedef struct
{
    /** Base address of Linking RAM 0. LLD will configure linking RAM0 address
     * to internal linking RAM address if a value of zero is specified.
     */
    uint32_t          linkingRAM0Base;
    /** Linking RAM 0 Size. LLD will configure linking RAM0 size to maximum
     * internal linking RAM size if a value of zero is specified
     */
    uint32_t          linkingRAM0Size;
    /** Base address of Linking RAM 1. Depends on RAM 0 Size and total number
     * of descriptors. If linkingRAM1Base is zero then linkingRAM0Size
     * must be large enough to store all descriptors in the system
     */
```

```
    uint32_t          linkingRAM1Base;
    /** Maximum number of descriptors in the system. Should be equal to less
     * than the RAM0+RAM1 size */
    uint32_t          maxDescNum;
    /** PDSP firmware to download. If the firmware pointer is NULL, LLD will
     * not download the firmware */
    Qmss_PdspCfg    pdspFirmware[QMSS_MAX_PDSP];
}Qmss_InitCfg;
```

Table 4 QMSS Initialization Configuration Structure

5.3.2 QMSS Device-Specific Initialization

This section deals with initialization of parameters that define the device-specific characteristics of the queue manager sub system. For example, the number of various types of queues supported, the number of queues of each type, memory mapped register addresses etc. These parameters are portable across SoC making the LLD SoC independent.

The device specific initialization is performed as a part of *Qmss_init()* API.
The structure is as follows. Refer to the appendix for a sample configuration.

```
typedef struct
{
    /** Maximum number of queue Managers */
    uint32_t                                maxQueMgr;
    /** Maximum number of queues */
    uint32_t                                maxQue;
    /* Queue start index and maximum number of queues of each queue type */
    Qmss_QueueNumRange                      maxQueueNum[25];

    /** Base address for the CPDMA overlay registers */

    /** QM Global Config registers */
    CSL_Qm_configRegs                   *qmConfigReg;
    /** QM Descriptor Config registers */
    CSL_Qm_descriptor_region_configRegs *qmDescReg;
    /** QM queue Management registers */
    CSL_Qm_queue_managementRegs         *qmQueMgmtReg;
    /** QM queue Management Proxy registers */
    CSL_Qm_queue_managementRegs         *qmQueMgmtProxyReg;
    /** QM queue status registers */
    CSL_Qm_queue_status_configRegs      *qmQueStatReg;
    /** QM INTD registers */
    CSL_Qm_intdRegs                     *qmQueIntdReg;
    /** QM PDSP command register */
    volatile uint32_t                   *qmPdspCmdReg[QMSS_MAX_PDSP];
    /** QM PDSP control register */
    CSL_PdspRegs                        *qmPdspCtrlReg[QMSS_MAX_PDSP];
    /** QM PDSP IRAM register */
```

```
    volatile uint32_t                   *qmPdspIRamReg[QMSS_MAX_PDSP];
    /** QM Status RAM */
    CSL_Qm_Queue_Status                 *qmStatusRAM;
}Qmss_GlobalConfigParams;
```

Table 5 QMSS Global Configuration Parameters Structure

5.3.3 QMSS LLD Start

Configuration information passed during the initialization API is replicated in local memory on other cores when the *Qmss_start()*API is called. Every core has to call this API so the local data structures are initialized.

5.3.4 Memory Region Configuration

The queue manager supports 20 memory regions with each region storing up to 32K descriptors. The linking RAM stores the information for each descriptor in each memory region. E.g., the information for descriptor 0 in memory region 0 is stored at linking RAM location with offset zero from the linking RAM base address.

The figure below illustrates the relationship between memory regions and linking RAM. The queue manager computes the descriptor addresses by taking into account the base address of the region as well as the number of descriptors and size of each descriptor stored in the region The descriptor pointers that are pushed on a queue must have an address that matches the computed address based on the region base address, region index and descriptor size programmed in the region control registers.

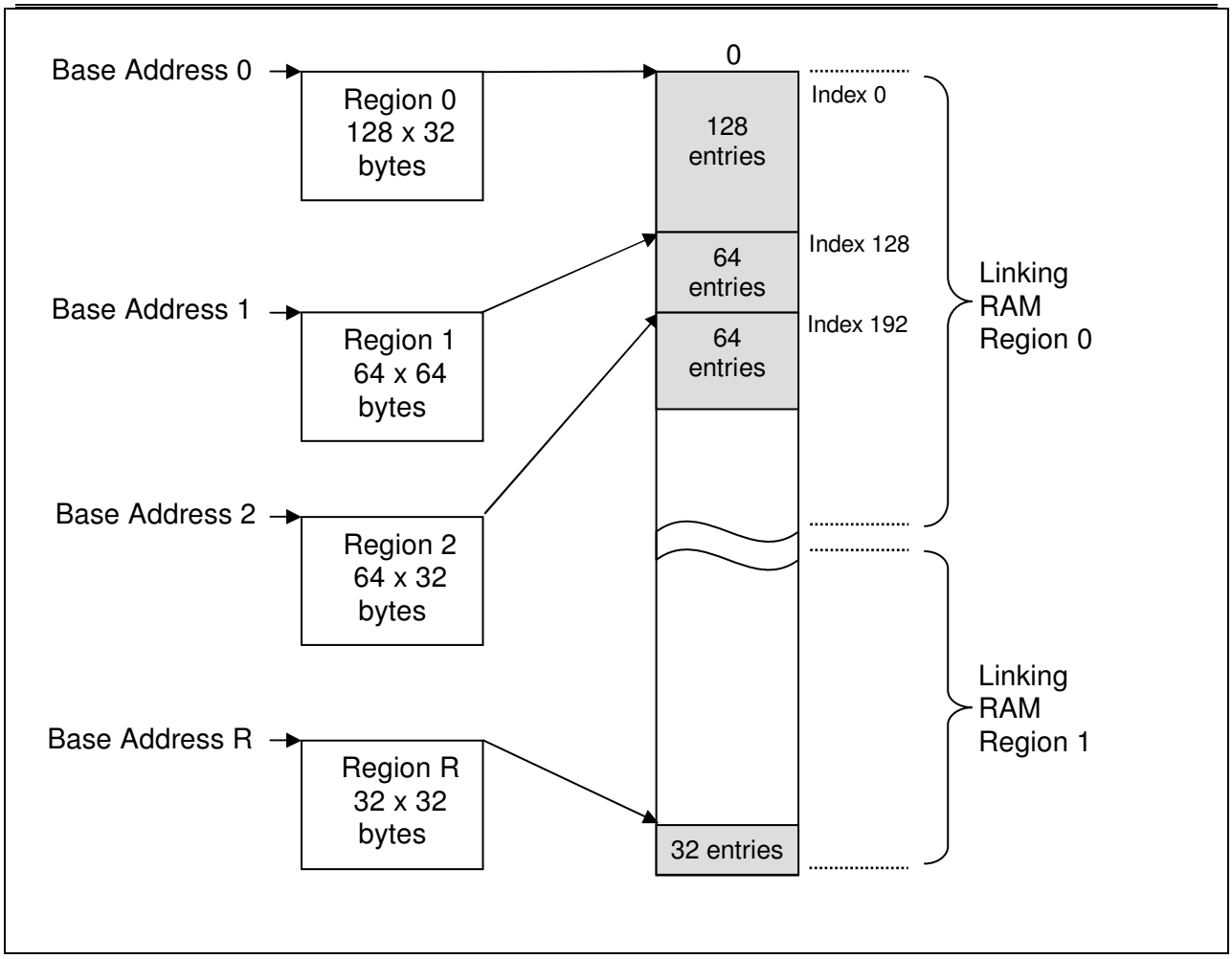

Figure 2 Linking RAM and Memory Region Mapping

As shown in Figure 2, the base addresses of the descriptor regions must be in ascending order, i.e., the lowest physical address needs to be in the lowest region. The hardware was designed such that the descriptor memory is allocated and the linking RAM configured at initialization time referred to as static configuration.

This can pose a problem when all descriptor needs are not known up front at init time and some runtime allocation schema might be required.

Software Solution:

We can leave logical holes of known sizes and insert memory regions with valid descriptor addresses in ascending address order.

Taking this solution a step further, 2 options provided to the users.

1) Static configuration schema for users who know descriptors requirements up front.
2) Dynamic memory region insertion for power users.

A single new API is provided to handle both the cases above:

```
Qmss_Result Qmss_insertMemoryRegion (Qmss_MemRegInfo *memRegCfg);
```

The input parameter structure *Qmss_MemRegInfo* is defined below:

```
typedef struct
{
    /** The base address of descriptor region. Note the
     * descriptor Base address must be specified in ascending memory order
     * */
    uint32_t          *descBase;

    /** Size of each descriptor in the memory region.
     * Must be a multiple of 16
     */
    uint32_t          descSize;

    /** Number of descriptors in the memory region.
     * Must be a minimum of 32.
     * Must be 2^(5 or greater)
     * Maximum supported value 2^20
     * */
     uint32_t          descNum;

    /** Memory Region corresponding to the descriptor.
     * At init time this field must have a valid memory region
     * index (0 to Maximum number of memory regions supported).
     *
     * At runtime this field is used to either
     *      * set to Qmss_MemRegion_MEMORY_REGION_NOT_SPECIFIED, in this case
     *      * the LLD will decide which memory region to use.
     *      * OR
     *      * specify the descriptor memory region, must be a valid memory
     *      * region index (0 to Maximum number of memory regions supported).
     */
    Qmss_MemRegion  memRegion;

    /** Flag control whether the descriptors are managed
     * by LLD or by the caller allocating descriptor memory
     */
    Qmss_ManageDesc manageDescFlag;

    /** Used to leave holes by configuring dummy regions which can be later
     * configured with actual values. Must be calculated and a correct
     * startIndex must be specified if memRegion value is
     * valid (0 to Maximum number of memory regions supported).
     */
    uint32_t          startIndex;
```

```
} Qmss_MemRegInfo;
```

Case1: Static Memory Region Configuration

This is to handle users who know the descriptor requirement up front and don't care about allocating descriptor memory at runtime.

Users can invoke the *Qmss_insertMemoryRegion()* API with the following parameters:

- descBase – Base memory address of the allocated descriptor pool in ascending order.
- descSize – Size of descriptors in this memory region.
- descNum – Number of descriptors in this memory region.
- memRegion – Should be set to **Qmss_MemRegion_MEMORY_REGION_NOT_SPECIFIED.**
- manageDescFlag

  If set to **Qmss_ManageDesc_MANAGE_DESCRIPTOR**, the LLD manages the resource; descriptors memory is chopped up into *descNum* number of descriptors of size *descSize*. The drivers/application can reclaim this memory by calling *Cppi_initDescriptor()* or *Qmss_initDescriptor()*.

  If set to **Qmss_ManageDesc_UNMANAGED_DESCRIPTOR**, descriptors are not managed by the LLD. It is the caller responsibility to manage the allocated descriptor memory.
- startIndex – Don't care.

The LLD will configure the next available memory region and compute the start index based on the previous configuration. This implies the configuration is sequential. The allocation starts with the lowest number memory region available.

The figure below illustrates a use case where the API is called in a loop to configure memory regions.

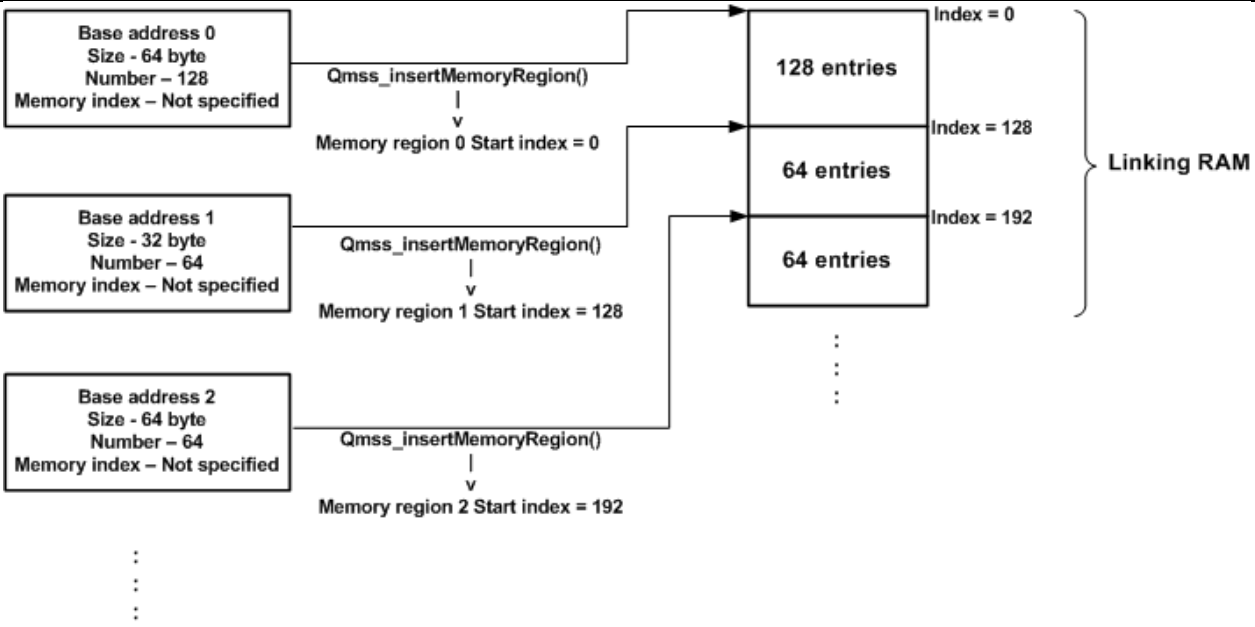

Figure 3 Static Memory Region Configuration

Case2: Dynamic Memory Region Configuration

This is to handle users who don't know the descriptor requirement up front and want to insert memory regions.

Users can invoke the Q*mss_insertMemoryRegion()* API with the following parameters:

- descBase – Base memory address of the allocated descriptor pool in ascending order.
- descSize – Size of descriptors in this memory region.
- descNum – Number of descriptors in this memory region.
- memRegion – Should be set to a valid memory region index from 0 to 19.
- manageDescFlag

  If set to **Qmss_ManageDesc_MANAGE_DESCRIPTOR**, the LLD manages the resource; descriptors memory is chopped up into *descNum* number of descriptors of size *descSize*. The drivers/application can reclaim this memory by calling C*ppi_initDescriptor()* or *Qmss_initDescriptor()*

  If set to **Qmss_ManageDesc_UNMANAGED_DESCRIPTOR**, descriptors are not managed by the LLD. It is the caller responsibility to manage the allocated descriptor memory.
- startIndex – Must be set to the location where the memory region must be inserted.

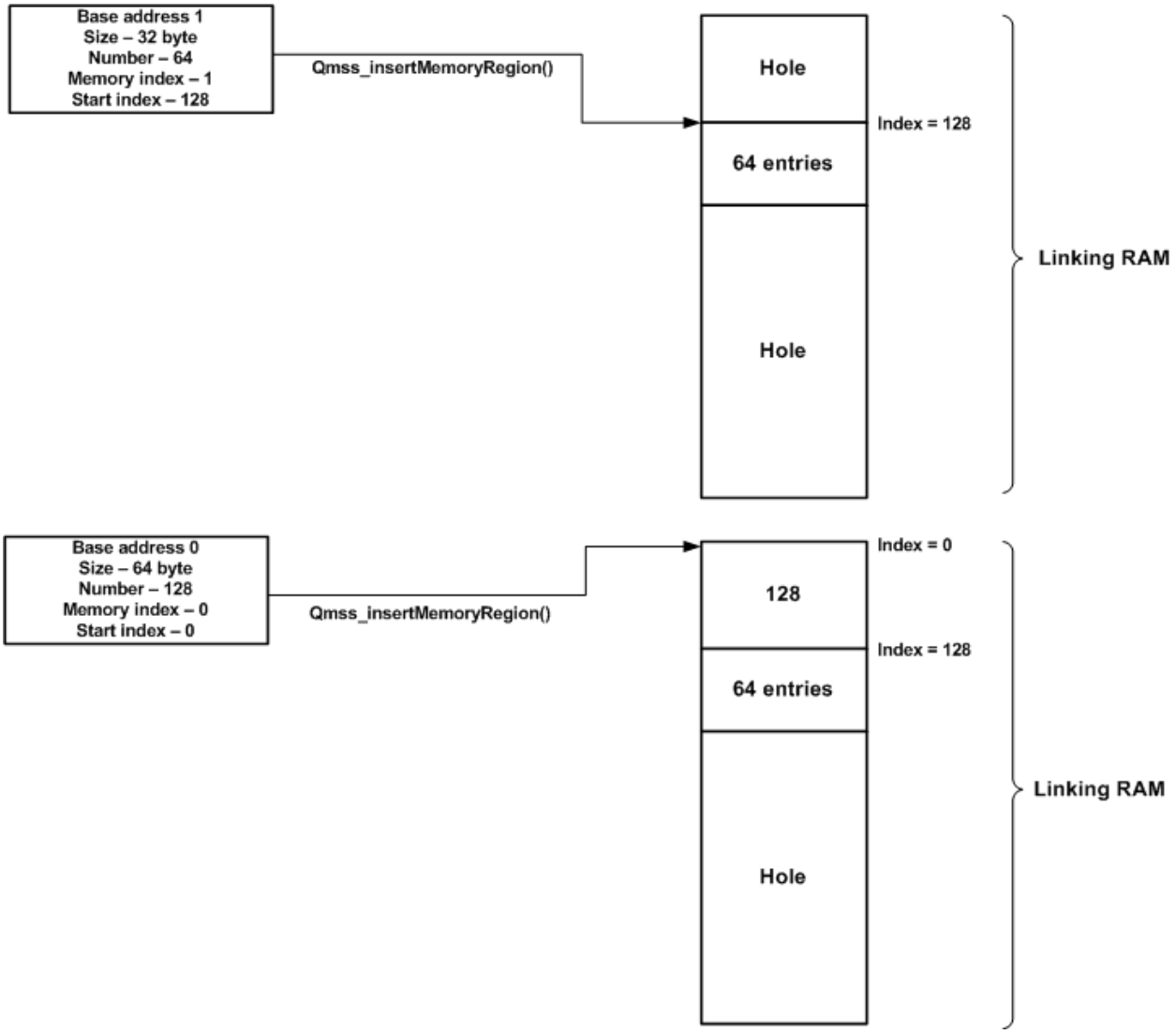

Figure 4 Dynamic Memory Region Configuration

LLD inserts the new memory region and configures the linking RAM.

It provides the following error checking:

- Prevent reconfiguring a memory region
- Overlapping start indexes
- If the hole is not big enough to accommodate specified number of descriptors
- Memory address is not in ascending order
- Maximum number of descriptors set up during queue manager initialization are already configured

If the memory region is successfully inserted, the API returns the inserted memory region index.

LLD provides a function to get the current memory region configuration so that the caller can make a decision on where to insert the new memory region.

```
Qmss_Result Qmss_getMemoryRegionCfg (Qmss_MemRegCfg *memRegInfo)
```

Note: The two calls above, Q*mss_getMemoryRegionCfg()* and Q*mss_insertMemoryRegion(),* must be protected with cross-core locks and critical sections to prevent another core or task from configuring the same memory region or configuring overlapping addresses.

The descriptors memory are chopped into descriptors of configured size and number and stored internally on general purpose queues, if the manage descriptor flag is set, until the driver can reclaim them using C*ppi_initDescriptor()* or Q*mss_initDescriptor()* APIs.

Note: The QMSS LLD uses 20 general purpose queues for internal use to correspond to 20 memory regions in order to store descriptors if the LLD is asked to do so when inserting a memory region.

5.3.5 QMSS resource manager

This section discusses the QMSS low-level driver resource allocation in detail.

5.3.5.1 Queue allocation

The QMSS LLD manages the allocation and freeing of queues. The maximum number of queues, the maximum number of queue managers and the division of queues into subcategories is configured in the LLD via the QMSS global configuration parameter structure (*Qmss_GlobalConfigParams*).

Users call the following API to obtain a queue:

```
Qmss_QueueHnd Qmss_queueOpen (Qmss_QueueType queType, int32_t queNum,
uint8_t_t *isAllocated)
```

```
Qmss_QueueHnd Qmss_queueOpenInRange (uint32_t startQueNum, uint32_t endQueNum,
uint8_t *isAllocated);
```

There are 2 ways to get a queue:

1. Request the queue number

The API allows the caller to specify a valid queue number via the input parameter *queNum*. Valid queue numbers are 0 to 8191.

If the queue is free, the LLD allocates the queue and marks it as used.

In this case the *queType* parameter is not used.

If the queue was already allocated (i.e., marked as used during a previous allocation call), an existing queue handle is returned.

2. LLD allocates the queue

The caller can have the LLD allocate the resource by setting the *queueNum* input parameter to *QMSS_PARAM_NOT_SPECIFIED*. In this case the *queType* parameter should be valid and be of the type *Qmss_QueueType*

The queue types can differ from SoC to SoC and are therefore defined in the CSL layer to make the LLD device independent.

Example for valid queue types:

```
File: csl_qm_queue.h

#define QMSS_FFTC_B_QUEUE_BASE                  692
#define QMSS_MAX_FFTC_B_QUEUE                   4
#define QMSS_HIGH_PRIORITY_QUEUE_BASE           704
#define QMSS_MAX_HIGH_PRIORITY_QUEUE            32
#define QMSS_STARVATION_COUNTER_QUEUE_BASE      736
#define QMSS_MAX_STARVATION_COUNTER_QUEUE       64
#define QMSS_INFRASTRUCTURE_QUEUE_BASE          800
#define QMSS_MAX_INFRASTRUCTURE_QUEUE           32
#define QMSS_TRAFFIC_SHAPING_QUEUE_BASE         832
#define QMSS_MAX_TRAFFIC_SHAPING_QUEUE          32
#define QMSS_GENERAL_PURPOSE_QUEUE_BASE         864
#define QMSS_MAX_GENERAL_PURPOSE_QUEUE          7328

/**
 * @brief Queue Type. Specifies different queue classifications
 */
typedef enum
{
    /** Low priority queue */
    Qmss_QueueType_LOW_PRIORITY_QUEUE = 0,
    /** AIF queue */
    Qmss_QueueType_AIF_QUEUE,
    /** PASS queue */
    Qmss_QueueType_PASS_QUEUE,
    /** INTC pending queue */
    Qmss_QueueType_INTC_QUEUE,
    /** SRIO queue */
```

```
    Qmss_QueueType_SRIO_QUEUE,
    /** FFTC queue A */
    Qmss_QueueType_FFTC_A_QUEUE,
    /** FFTC queue B */
    Qmss_QueueType_FFTC_B_QUEUE,
    /** High priority queue */
    Qmss_QueueType_HIGH_PRIORITY_QUEUE,
    /** starvation counter queue */
    Qmss_QueueType_STARVATION_COUNTER_QUEUE,
    /** Infrastructure queue */
    Qmss_QueueType_INFRASTRUCTURE_QUEUE,
    /** Traffic shaping queue */
    Qmss_QueueType_TRAFFIC_SHAPING_QUEUE,
    /** General purpose queue */
    Qmss_QueueType_GENERAL_PURPOSE_QUEUE
}Qmss_QueueType;
```

Table 6 Example Queue Types : Device dependent

The LLD allocates the next available queue and marks it as used. Allocation starts with the lowest queue number in the given queue type.

In both cases a handle to the opened queue is returned to the caller that must be used as an input parameter when operating on that queue.

You can also specify a start and end range using `Qmss_queueOpenInRange()` API. The LLD will allocate a free queue within the range if available.

The API also provides the caller with information on whether the handle returned for the requested queue number is due to a new queue allocation or not via the *isAllocated* output parameter. This parameter contains the reference count that indicates the number of times the queue was opened.

The LLD maintains a reference count per queue which is incremented every time the queue is opened using the *Qmss_queueOpen()* API and decremented when the queue is closed via the *Qmss_queueClose()* API. The queue is reallocated only when the reference count becomes zero.

5.3.5.2 Descriptors

The memory for descriptors that was allocated by the system and set up during the *Qmss_insertMemoryRegion()* API can be taken back by the drivers/applications via the *Qmss_initDescriptor()* API. The descriptor resources are managed by the LLD only if the manage descriptor flag *manageDescFlag* was set when the descriptor memory region was created, otherwise it is up to the caller to manage the descriptors.

```
Qmss_QueueHnd Qmss_initDescriptor (Qmss_DescCfg *descCfg, uint32_t
*numAllocated);
```

The caller needs to specify a few input parameters via the *Qmss_DescCfg* structure. The parameters are described below:

- Number of descriptors to be allocated
- Memory region to allocate from
- Destination queue where the descriptors are stored and returned to caller. Valid queue numbers are 0 to 8191. If queue number is not known then set to QMSS_PARAM_NOT_SPECIFIED and specify Destination queue type

OR

- Destination queue type

The input parameter structure *Qmss_DescCfg* is shown below:

```
typedef struct
{
    /** Memory Region corresponding to the descriptor */
    uint32_t          memRegion;
    /** Number of descriptors that should be allocated */
    uint32_t          descNum;
    /** Queue where the descriptor is stored. If destQueueNum is set to
        QMSS_PARAM_NOT_SPECIFIED then the next available queue of type
        Qmss_QueueType will be allocated */
    uint32_t          destQueueNum;
    /** If destQueueNum is set to QMSS_PARAM_NOT_SPECIFIED then the next
        available queue of type queueType will be allocated */
    Qmss_QueueType  queueType;
}Qmss_DescCfg;
```

Table 7 QMSS Descriptor Configuration Structure

The function informs the caller about the actual number of descriptors allocated from the specified memory region using the *numAllocated* output parameter. This is done to handle a case where the requested number of descriptors is not available; the function allocates what is available and informs the caller of the same.

The decision of which memory region to use is based on factors such as:

- Size of descriptor
- Type of memory the descriptor is allocated from e.g., L2, shared, DDR memory
- Reserved allocation for a particular IP block

Since the memory regions might be configured by the system during the initialization phase, the LLD provides an API that aids drivers and application to decide on the memory region to allocate from. The function *Qmss_getMemoryRegionCfg()* returns the memory region configuration parameters such as memory region index, descriptor base address, number and size of descriptors for all supported memory regions.

```
Qmss_Result Qmss_getMemoryRegionCfg (Qmss_MemRegCfg *memRegInfo)
```

Once the function determines descriptors are available in the requested memory region, they are allocated and pushed onto the destination queue specified in the input parameter. The destination queue allocation is discussed in section Queue allocation

The handle to the allocated destination queue is returned to the caller.

Note: The CPPI LLD provides a similar API, *Cppi_initDescriptor(),* that is used to allocate CPPI descriptors. The idea behind providing a descriptor allocate API in the QMSS LLD ((*Qmss_initDescriptor()*)) is to allow users access to non-CPPI descriptors.

5.3.6 QMSS En-queue

The packets are queued onto the logical queues by writing a burst of information. This burst contains the required pointer to the descriptor that is being added, optional control information, and optional descriptor size.

The LLD provides various APIs to add packets to a queue. This function writes a descriptor pointed to by *descAddr* onto a queue specified by the queue handle *hnd*.

5.3.6.1 Push Descriptor Only

This is a faster version of the API since only the descriptor address is written.
This function does not configure the optional parameters.
The API format is:

```
void Qmss_queuePushDesc (Qmss_QueueHnd hnd, void* descAddr)
```

5.3.6.2 Push Descriptor and Size

The size parameter is also used as a hint when pre-fetching the descriptor. Since the descriptors are 16 byte aligned, the lower 4 bits are used to indicate to the DMA the size of the descriptor. The DMA uses this information to control how large the initial packet descriptor fetch will be.

```
void Qmss_queuePushDescSize (Qmss_QueueHnd hnd, void* descAddr, uint32_t
descSize)
```

5.3.6.3 Push Descriptor and Optional Parameters

```
void Qmss_queuePush (Qmss_QueueHnd hnd, void* descAddr, uint32_t packetSize,
uint32_t descSize, Qmss_Location location)
```

The optional parameter *packetSize* is used to specify the size of packets and used during pop operation.
The optional parameter *location* is used to specify where the packet should be queued. The default behavior is to queue the packet to the tail of the queue. It can be overridden and the packet pushed to the head of the queue.

These functions make use of Queue Proxy so they are multicore safe. However critical sections to protect against preemption are required.

5.3.7 QMSS De-queue

The packets are de-queued from the logical queues by reading a descriptor pointer value from the queue. The LLD provides various APIs to pop a packet of a queue.

5.3.7.1 Pop Descriptor Only

Only the descriptor address is read. This API is multicore/task safe.

```
void* Qmss_queuePop (Qmss_QueueHnd hnd)
```

The function pops a descriptor from a queue specified by the queue handle *hnd*. It returns NULL if the queue is empty else a valid descriptor address.

The lower four bits of the descriptor address contain the size of the descriptor if the size was specified when pushing the descriptor onto the queue. The caller should mask the lower order four bits before using the descriptor.

5.3.7.2 Pop Descriptor and Packet Size

The descriptor address along with the packet size of the descriptor popped is read from a queue specified by the queue handle *hnd*. The packet size is available only if it was specified during the push operation. It returns NULL if the queue is empty.

This API is not multicore/task safe.

```
void Qmss_queuePopDescSize (Qmss_QueueHnd hnd, void* *descAddr, uint32_t
*packetSize)
```

It is possible that the descriptor is popped by another core/task between the time taken to read the packet size and the descriptor address by the first core/task. The caller should provide appropriate locks.

The packet size field is part of the descriptor and should be used to ensure correctness.

The lower four bits of the descriptor address contain the size of the descriptor if the size was specified when pushing the descriptor onto the queue. The caller should mask the lower order four bits before using the descriptor.

5.3.8 Emptying a Queue

The entire queue can be emptied in a single instruction. This operation must be used with caution because the contents of the cleared queue are irrecoverably lost.

```
void Qmss_queueEmpty (Qmss_QueueHnd hnd)
```

5.3.9 Diverting Queue Contents

The contents of an entire queue can be copied into another queue. Users can also specify whether the contents should be merged to the head or tail of the destination queue.

```
void Qmss_queueDivert (Qmss_QueueHnd srcQnum, Qmss_QueueHnd dstQnum,
Qmss_Location location)
```

5.3.10 Accumulation

The QMSS contains two PDSPs that allow for autonomous accumulation of descriptor pointers and subsequent notification of the host DSP or peripheral.

The 2 possible configurations are:

- 32 channel high priority accumulation PDSP1 & 16 channel low priority accumulation on PDSP2.
- 48 channel combined accumulation on PDSP1. QoS on PDSP2.

The accumulator uses the concept of channels that can be mapped to queues. Each channel can be individually enabled or disabled. Each channel can monitor a single or a group of 32 queues. If multiple queues are monitored, referred to as multi-mode, the queue group must be aligned to a 32 queue boundary. Individual queues within a queue group can be excluded from monitoring using an enable mask.

The host processor allocates a block of memory to hold a list of descriptor pointers. This memory block is divided into two pages referred to as ping/pong. Each channel can write packet information to a ping/pong descriptor list anywhere in memory independent of other channels.

Operation always begins on the first page (page 0). When an interrupt is fired for the first time on a particular channel, this indicates that page 0 is full and ready for use by the host processor. On each subsequent interrupt, the service page toggles, going from 1 and then back to 0. It is up to

the host processor to keep in synch with the accumulator PDSP based on the number of interrupts received for the accumulator channel.

The information accumulated by the channel can be one of the following:

Mode	Entry Byte Size	Entry Contents
"D"	4	Word 0 : Packet Descriptor Pointer
"C, D"	8	Word 0 : Packet Length (as reported by queue manager)
		Word 1 : Packet Descriptor Pointer
"A, B, C, D"	16	Word 0 : Packet Count on Queue (when read)
		Word 1 : Byte Count on Queue (when read)
		Word 2 : Packet Length (as reported by queue manager)
		Word 3 : Packet Descriptor Pointer

Table 8 Accumulator List Entry

For channels that are monitoring multiple queues, the upper 16 bits of the "C" field contain the index of the original source queue.

The number of entries in the list can be reported in one of two ways. In "NULL terminate" mode, the entry list always ends with an entry where the descriptor pointer is set to NULL. In the "entry count" mode, the first 32-bit word of the first entry in the list holds a count of the number of entries in the list (not including the count entry).

Note that in either case, there is room for one less list entry in a page than is actually specified by the host. In the "NULL terminate" mode, one entry is used at the end to act as the NULL terminator, and in the "entry count" mode, one entry is used at the start to specify entry count.

Each accumulator channel will always fire an interrupt when all the entries in the current buffer page are filled. It is also possible to fire interrupts more quickly by configuring the interrupt pacing mode. The interrupt pacing mode allows for interrupts to be generated on a partially filled page, based on configurable packet activity and a configurable amount of elapsed time. Note that the pacing is per channel, and not per interrupt, thus if two channels are using the same host interrupt, then the host interrupt can be fired as each channel independently requires it to.

The available interrupt pacing modes are based on one of the following events:

- Programmable delay since last interrupt to the host
- Programmable delay since first packet on new activity
- Programmable delay since last packet on new activity

The above event setting determines when the interrupt counter will be loaded and begin its countdown. An interrupt will fire only when both the timing conditions are met *and* there are packets available to forward to the host. If the timer expires with no packet activity, then the next incoming packet will fire an immediate interrupt.

The time delay from the configured event is programmable, ranging from 0 to 1.6 seconds in steps of configured timer value. A timer setting of 0 seconds is useful in cases where the user wishes to fire interrupts without delay based on any packet activity. Note that a delay of 0 seconds will always fire an immediate interrupt on the first received packet no matter which of the three configurable pacing events are used.

The Accumulator time "tick" is controlled by a local timer connected to the PDSP core. This timer has a programmable count based on the sub-system clock. When this count expires, a local "tick" is registered in the firmware. The tick is used when timing channel interrupts based on the "Timer Load Count" value supplied in the timer configuration.

The value of "Timer Constant" is the number of QM sub-system clocks divided by 2 that comprise a single "tick" in the accumulator firmware.
For example, if the QM sub-system clock is 350MHz, and the desired firmware "tick" is 20us, the proper Timer Constant for this command is computed as follows:

Timer Constant = (350,000,000 cycles/sec) * (0.000020 sec) / (2 cycles)
Timer Constant = 3,500

The firmware initializes with a default constant value of 4375. This corresponds to firmware tick of 25us.

The timer can be configured to 10, 20, 25 micro seconds using the following API:

```
Qmss_Result Qmss_configureAccTimer (Qmss_AccPriorityType type, uint16_t
timerConstant)
```

When a list buffer page is ready for processing, an interrupt is sent to the host CPU. The mapping between accumulator channel and host interrupt is fixed; however, each accumulator channel can be configured to any queue, or can be disabled, so there is a significant amount of flexibility in how queues can be mapped to host interrupts.

To enable a channel use the following API:

```
Qmss_Result Qmss_programAccumulator (Qmss_AccPriorityType type, Qmss_AccCmdCfg
*cfg)
```

When an accumulator channel is disabled, it may still potentially fire one last interrupt to the host processor. It does this to clear out all the descriptors that it is holding on its current page list at the time of being disabled. It does verify that all packets have been read out of the monitored queue.

To disable a channel use the following API:

```
Qmss_Result Qmss_disableAccumulator (Qmss_AccPriorityType type, uint16_t
timerConstant)
```

The host should acknowledge the interrupt and set the end of interrupt (EOI) vector to indicate to the PDSP it is done with processing the current page. The PDSP will then start writing to the freed page.

To acknowledge the interrupt use the following API:

```
Qmss_Result Qmss_ackInterrupt (uint8_t interruptNum, uint8_t value)
```

To set the EOI vector value use the following API:

```
Qmss_Result Qmss_setEoiVector (Qmss_IntdInterruptType type, uint8_t
interruptNum)
```

5.4 CPPI DESIGN DETAILS

The CPPI Low level driver supports the following CPDMAs:

- Serial Rapid IO
- Antenna Interface
- FFT Co-processor
- Packet Accelerator SubSystem
- Queue Manager SubSystem

It provides resource management for:

- Descriptors
- Receive channels
- Transmit channels
- Receive flows

It provides configuration of each of the CPPI blocks in each of the CPDMAs listed above.
It provides descriptor allocation, initialization and management functionality.

5.4.1 CPPI LLD Initialization

This api is the entry point into CPPI low level driver. The CPPI LLD is initialized only once in the system using *Cppi_init()* API.

```
Cppi_Result Cppi_init (Cppi_GlobalConfigParams *cppiGblCfgParams)
```

This API initializes the parameters that define the characteristics of the CPPI. For example, per CPDMA the number of receive channels, the number of transmit channels, number of flows that are supported, memory mapped register addresses etc. These parameters are portable across SoC making the LLD SoC independent. It is discussed further in the next section.

5.4.2 CPPI Device-Specific Initialization

The device-specific initialization is performed as a part of the *Cppi_init()* API.
The structure is as follows. Refer to the appendix for a sample configuration.

```
typedef struct
{
    /** CPDMA this configuration belongs to */
    Cppi_CpDma      dmaNum;
    /** Maximum supported Rx Channels */
    uint32_t          maxRxCh;
    /** Maximum supported Tx Channels */
    uint32_t          maxTxCh;
    /** Maximum supported Rx Flows */
    uint32_t          maxRxFlow;
    /** Priority for all Rx transactions of this CPDMA */
    uint8_t           rxPriority;
    /** Priority for all Tx transactions of this CPDMA */
    uint8_t           txPriority;

    /** Base address for the CPDMA overlay registers */

    /** Global Config registers */
    CSL_Cppidma_global_configRegs       *gblCfgRegs;
    /** Rx Channel Config registers */
    CSL_Cppidma_rx_channel_configRegs   *rxChRegs;
    /** Tx Channel Config registers */
    CSL_Cppidma_tx_channel_configRegs   *txChRegs;
    /** Tx Channel Scheduler registers */
    CSL_Cppidma_tx_scheduler_configRegs *txSchedRegs;
    /** Rx Flow Config registers */
```

```
    CSL_Cppidma_rx_flow_configRegs      *rxFlowRegs;
}Cppi_GlobalConfigParams;
```

Table 9 CPPI Global Configuration Parameters Structure

The function sets up the low-level driver with information pertaining to Rx priority, Tx priority, maximum supported Rx channels, maximum supported Tx channels, maximum supported Rx flows and memory-mapped register addresses to access the memory-mapped registers for each CPPI CPDMA.

To reinitialize, the CPPI LLD must first be closed using the *Cppi_exit()* API. The LLD is not de-initialized if any of the channels, flows or CPDMA instances are still enabled.

5.4.3 CPPI CPDMA Initialization

This function is called by the CPPI CPDMA driver or application to obtain a handle to the CPDMA instance. It also initializes the global configuration pertaining to each CPPI CPDMA that includes the Rx, Tx priority, write arbitration FIFO depth, receive starvation timeout and QM base addresses.

```
Cppi_Handle Cppi_open (Cppi_CpDmaInitCfg *initCfg)
```

The function returns a CPDMA object handle that should be used for all channel and flow management.

This function can be called any number of times but the CPDMA memory-mapped registers are configured once, when the function is called for the first time. Any further calls to this function returns the CPDMA object handle created during the first call.

To reinitialize, the CPPI CPDMA instance must first be closed using the *Cppi_close()* API. The CPDMA instance is not closed if any of the channels or flows belonging to the CPDMA are still enabled.

5.4.4 CPPI Resource Manager

This section discusses the CPPI low-level driver resource allocation in detail.

5.4.4.1 Descriptors

The memory for descriptors that was allocated by the system and set up during the *Qmss_init()* API can be taken back by the drivers/applications via the *Cppi_initDescriptor()* API. The descriptor resources are managed by the LLD only if the manage descriptor flag *manageDescFlag*

was set when the descriptor memory region was created else it is up to the caller to manage the descriptors.

```
Qmss_QueueHnd Cppi_initDescriptor (Cppi_DescCfg *descCfg, uint32_t
*numAllocated)
```

The caller needs to specify a few input parameters via the *Cppi_DescCfg* structure. The parameters are described below:

- Number of descriptors to be allocated
- Memory region to allocate from
- Destination queue where the descriptors are stored and returned to caller
  OR
- Destination queue type
- Descriptor type – host, monolithic
- Configuration values for fields in the descriptor
  - Return Policy
  - Return Push Policy
  - Return Queue number and Return Queue Manager
  - Protocol Specific data location

The input parameter structure for *Cppi_DescCfg* is shown below:

```
typedef struct
{
    /** Memory Region corresponding to the descriptor. */
    uint32_t          memRegion;
    /** Number of descriptors that should be configured with value below */
    uint32_t          descNum;
    /** Queue where the descriptor is stored. If destQueueNum is set to
        QMSS_PARAM_NOT_SPECIFIED then the next  available queue of type
        Qmss_QueueType will be allocated */
    int32_t           destQueueNum;
    /** If destQueueNum is set to QMSS_PARAM_NOT_SPECIFIED then the next
        available queue of type Qmss_QueueType will be allocated */
    Qmss_QueueType  queueType;

    /** Descriptor configuration parameters */
    /** Indicates if the descriptor should be initialized with parameters
     * listed below */
    Cppi_InitDesc           initDesc;

    /** Type of descriptor - Host or Monolithic */
    Cppi_DescType           descType;

    /** Indicates return Queue Manager and Queue Number. If both qMgr and qNum
     * in returnQueue is set to QMSS_PARAM_NOT_SPECIFIED then the destQueueNum
     * is configured in returnQueue of the descriptor
     */
```

```
    Qmss_Queue              returnQueue;

    /** Indicates presence of EPIB */
    Cppi_EPIB               epibPresent;

    /** Union contains configuration that should be initialized in for host or
     * monolithic descriptor. The configuration for host or monolithic
     * descriptor is choosen based on "descType" field.
     * The approriate structure fields must be specified if "initDesc" field
     * is set to CPPI_INIT_DESCRIPTOR.
     */
    union{
    /** Host descriptor configuration parameters */
    Cppi_HostDescCfg        host;

    /** Monolithic  descriptor configuration parameters */
    Cppi_MonolithicDescCfg  mono;
    }cfg;
}Cppi_DescCfg;
```

Table 10 CPPI Descriptor Configuration Structure

The function allocates the requested number of descriptors from the specified memory region.

It configures all the allocated descriptors based on the descriptor type with the specified configuration parameters if the *initDesc* flag is set. The idea is to have the descriptor fields that don't change filled in at init time rather than at runtime to save some cycles.

The function informs the caller about the actual number of descriptors allocated using the *numAllocated* output parameter. This is done to handle a case where the requested number of descriptors is not available; the function allocates what is available and informs the caller of the same.

The decision of which memory region to use is based on factors such as:

- Size of descriptor
- Type of memory the descriptor is allocated from e.g., L2, shared, DDR memory
- Reserved allocation for a particular IP block

Since the memory regions are configured by the system during the initialization phase, the LLD provides an API that aids drivers and application decide on the memory region to allocate from. The function *Qmss_getMemoryRegionCfg()* returns the memory region configuration parameters such as memory region index, descriptor base address, and number and size of descriptors for all supported memory regions.

```
Qmss_Result Qmss_getMemoryRegionCfg (Qmss_MemRegCfg *memRegInfo)
```

Once the function determines descriptors are available in the requested memory region, they are allocated and pushed onto the destination queue specified in the input parameter. The destination queue can be allocated as discussed in the queue allocation section.

The handle to the allocated destination queue is returned to the caller.

5.4.4.2 Receive Channels

The CPPI LLD manages the allocation and freeing of CPPI receive channels. The maximum number of receive channels supported per CPDMA is configured in the LLD via the CPPI global configuration parameter structure (*Cppi_GlobalConfigParams)*.

Users call the following API to obtain a channel:

```
Cppi_ChHnd Cppi_rxChannelOpen (Cppi_Handle hnd, Cppi_RxChInitCfg *cfg, uint8_t
*isAllocated)
```

There are 2 ways to open a receive channel:

1. Request the channel number

   The API allows the caller to specify a valid channel number via the *Cppi_RxChInitCfg* structure. If the channel is free, the LLD allocates the channel and marks it as used. The channel is configured via the channel's memory-mapped registers.

   If the channel was already allocated (i.e., marked as used during a previous allocation call), an existing channel handle is returned. The channel is not reconfigured.

   In other words the channel is configured only if it is a new channel allocation. To change the channel configuration the channel must first be closed and reopened with a new configuration.

2. LLD allocates the receive channel

   The caller has an option to let the CPPI LLD allocate the resource by setting the channel number field in the *Cppi_RxChInitCfg* structure to *CPPI_PARAM_NOT_SPECIFIED*. The LLD allocates the next available channel and marks it as used. Allocation starts with the lowest channel number.

   The channel is configured via the channel's memory-mapped registers.

In both cases a handle to the newly opened channel is returned to the caller that must be used as an input parameter when calling the channel management API for the allocated channel.

The API also provides the caller with information on whether or not the handle returned for the requested channel number is due to a new channel allocation via the *isAllocated* output parameter. This parameter returns the reference count that indicates the number of times the channel was opened.

The LLD maintains a reference count per receive channel which is incremented every time the channel is allocated and decremented when the channel is closed via the C*ppi_channelClose()* API. The channel is reallocated only when the reference count becomes zero.

5.4.4.3 Transmit Channels

The CPPI LLD manages the allocation and freeing of CPPI transmit channels. The maximum number of transmit channels supported per CPDMA is configured in the LLD via the CPPI global configuration parameter structure (*Cppi_GlobalConfigParams*).

Users call the following API to obtain a channel:

```
Cppi_ChHnd Cppi_txChannelOpen (Cppi_Handle hnd, Cppi_TxChInitCfg *cfg, uint8_t
*isAllocated)
```

There are 2 ways to open a transmit channel:

1. Request the channel number

   The API allows the caller to specify a valid channel number via the *Cppi_TxChInitCfg* structure. If the channel is free, the LLD allocates the channel and marks it as used. The channel is configured via the channel's memory-mapped registers.

   If the channel was already allocated (i.e., marked as used during a previous allocation call), an existing channel handle is returned. The channel is not reconfigured.

   In other words the channel is configured only if it is a new channel allocation. To change the channel configuration the channel must first be closed and reopened with a new configuration.

2. LLD allocates the channel

   The caller has an option to let the CPPI LLD allocate the resource by setting the channel number field in the *Cppi_TxChInitCfg* structure to *CPPI_PARAM_NOT_SPECIFIED*. The LLD allocates the next available channel and marks it as used. Allocation starts with the lowest channel number.
   The channel is configured via the channel's memory-mapped registers.

In both cases a handle to the newly opened channel is returned to the caller that must be used as an input parameter when calling the channel management API for the allocated channel.

The API also provides the caller with information on whether or note the handle returned for the requested channel number is due to a new channel allocation via the *isAllocated* output parameter. This parameter returns the reference count that indicates the number of times the channel was opened.

The LLD maintains a reference count per transmit channel which is incremented every time the channel is allocated and decremented when the channel is closed via the *Cppi_channelClose()* API. The channel is reallocated only when the reference count becomes zero.

5.4.4.4 Receive Flow

The CPPI LLD manages the allocation and freeing of CPPI receive flows. The maximum number of receive flows supported per CPDMA is configured in the LLD via the CPPI global configuration parameter structure (*Cppi_GlobalConfigParams*).

Users call the following API to obtain a flow:

```
Cppi_FlowHnd Cppi_configureRxFlow (Cppi_Handle hnd, Cppi_RxFlowCfg *cfg,
uint8_t *isAllocated)
```

There are 2 ways to open a receive flow:

1. Request the receive flow number

   The API allows the caller to specify a valid receive flow number *flowIdNum* in the *Cppi_RxFlowCfg* structure. If the flow is free, the LLD allocates the flow and marks it as used. The flow is configured via the flow's memory-mapped registers.

   If the flow was already allocated (i.e., marked as used during a previous allocation call), an existing flow handle is returned. The flow is not reconfigured.

   In other words the flow is configured only if it is a new flow allocation. To change the flow configuration the flow must first be closed and reopened with a new configuration.

   Note: The receive channels using the flow MUST be disabled or paused for the new configuration to take effect.

2. LLD allocates the receive flow

   The caller has an option to let the CPPI LLD allocate the resource by setting the *flowIdNum* field in the *Cppi_RxFlowCfg* structure to *CPPI_PARAM_NOT_SPECIFIED*. The LLD

allocates the next available flow and marks it as used. Allocation starts with the lowest flow number.
The flow is configured via the flow's memory-mapped registers.

Note: The receive channels using the flow MUST be disabled or paused for the new configuration to take effect.

In both cases a handle to the newly opened flow is returned to the caller that must be used as an input parameter when calling flow management API for the allocated flow.

The API also provides the caller with information on whether or not the handle returned for the requested flow number is due to a new flow allocation via the *isAllocated* output parameter. This parameter returns the reference count that indicates the number of times the flow was opened.

The LLD maintains a reference count per flow which is incremented every time the flow is allocated and decremented when the flow is closed via the *Cppi_closeRxFlow()* API. The flow is reallocated only when the reference count becomes zero.

5.4.5 Descriptor Management

The CPPI LLD defines 3 types of descriptor:

1. **Host Descriptor**.

The format is shown below:

Packet Info (12 bytes)
Buffer Info (8 bytes)
Linking Info (4 bytes)
Original Buffer Info (8 bytes)
Extended Packet Info Block (Optional) Includes Timestamp and Software Data (16 bytes)
Protocol Specific Data (Optional) (0 to M bytes where M is a multiple of 4)
Other SW Data (Optional and User Defined)

Table 11 CPPI Host Descriptor

The host descriptor data structure is as follows:

```
typedef struct {
    /** Descriptor type, packet type, protocol specific region location,
        packet length */
    uint32_t          descInfo;
    /** Source tag, Destination tag */
    uint32_t          tagInfo;
    /** EPIB present, PS valid word count, error flags, PS flags, return
        policy, return push policy, packet return QM number, packet return
        queue number */
    uint32_t          packetInfo;
    /** Number of valid data bytes in the buffer */
    uint32_t          buffLen;
    /** Byte aligned memory address of the buffer associated with this
        descriptor */
    uint32_t          buffPtr;
    /** 32-bit word aligned memory address of the next buffer descriptor */
    uint32_t          nextBDPtr;
    /** Completion tag, original buffer size */
    uint32_t          origBufferLen;
    /** Original buffer pointer */
    uint32_t          origBuffPtr;
    /** Optional EPIB word0 */
    uint32_t          timeStamp;
    /** Optional EPIB word1 */
    uint32_t          softwareInfo0;
    /** Optional EPIB word2 */
    uint32_t          softwareInfo1;
    /** Optional EPIB word3 */
    uint32_t          softwareInfo2;
    /** Optional protocol specific data */
    uint32_t          psData;
}Cppi_HostDesc;
```

Table 12 CPPI Host Descriptor Structure

2. Monolithic Descriptor

The format is shown below:

Packet Info (12 bytes)
Extended Packet Info Block (Optional) Includes PS Bits, Timestamp, and SW Data Words (20 bytes)
Protocol Specific Data (Optional)

(0 to M bytes where M is a multiple of 4)
Null Region (0 to 255 bytes)
Packet Data (0 to 64K – 1)
Other SW Data (Optional and User Defined)

Table 13 CPPI Monolithic Descriptor

The monolithic descriptor data structure is as follows:

```
typedef struct {
    /** Descriptor type, packet type, data offset, packet length */
    uint32_t          descInfo;
    /** Source tag, Destination tag */
    uint32_t          tagInfo;
    /** EPIB present, PS valid word count, error flags, PS flags, return push
        policy, packet return QM number, packet return queue number */
    uint32_t          packetInfo;
    /** NULL word to align the extended packet words to a 128 bit boundary */
    uint32_t          Reserved;
    /** Optional EPIB word0 */
    uint32_t          timeStamp;
    /** Optional EPIB word1 */
    uint32_t          softwareInfo0;
    /** Optional EPIB word2 */
    uint32_t          softwareInfo1;
    /** Optional EPIB word3 */
    uint32_t          softwareInfo2;
    /** Optional protocol specific data */
    uint32_t          psData;
}Cppi_MonolithicDesc;
```

Table 14 CPPI Monolithic Descriptor Structure

The CPPI LLD provides various inline functions to set and retrieve every field in host and monolithic descriptors.

5.4.6 Programming Considerations

This section discusses the various programming considerations that must be taken into account when programming CPPI and QMSS.

1. Multicore versus single core

   CPPI and QMSS low-level drivers are designed to be shared by various drivers and applications using different CPDMAs. They also manage resource allocation of different resources. If the drivers/applications are run on different cores, it implies that the global data structure access must be protected by cross-core locks. This also implies the global data structures be placed in shared memory. The LLD has callout functions that can be modified to suit these needs.
   If running on a single core, these data structures can be mapped to local memory.

2. Single thread versus multiple threads

   Resource allocation APIs and access to global data structures must be protected by critical sections to ensure intended results. The LLD has callout functions that can be modified to suit these needs.

3. Descriptor
   - Type of descriptor – The CPPI provides two types of descriptors: Host and Monolithic. The decision to use either or both is based on the driver/application requirement, size of data buffer, number of buffers dynamically linked, when and how the data arrives, ease of use, etc.
   - Size of descriptor – Depends on optional descriptor field usage. For monolithic descriptors it also depends on size of data. The descriptors must be a multiple of 16 bytes.
   - Number of descriptors – Depends on speed of the peripheral, burst modes, interrupt or polling mode, etc. It is necessary to ensure the host/DMAs are not starved before the descriptors are processed and recycled back to the queues. The number of descriptors in a memory region is specified as a power of 2, beginning with 2^5.
   - Alignment – Descriptors have to be aligned on the 16-byte boundary.

4. Memory Regions
   Memory regions can be set up either statically at initialization time if the descriptor requirement is known up front or dynamically at runtime.
   QMSS has 20 different memory regions and each region supports only one descriptor size. Also consider the memory where the descriptor is allocated from.

   Linking RAM must be aligned on the 16-byte boundary.

5. Queues
   - Number of queues – Allocating one or more queues for each operation. Queues will be required to store free transmit descriptors, to transmit a packet, and to receive a packet. Completion queues are required to recycle the packet if the recycling is done by host. It is possible to have the hardware recycle the descriptor by specifying the return queue manager/number in the descriptor. Queues are logical entities with low overheads. They can also be used to store intermediate data, as FIFOs, etc.
   - Type of queue – Deciding on which queue to allocate for what purpose. E.g., using queues with accumulation and interrupt support receive queues, using queues with starvation counters for free queues, etc.
   - Queue chaining – Queue chaining is a powerful concept where the output queue of one peripheral can be the input queue of another. This requires careful planning of queue use and recycling.

6. Receive Flows
   Flows are complicated but provide optimal use of buffer memory by specifying which free descriptor queue to use based on packet size. Refer to the CPPI user guide to obtain a description of configurable fields.

7. Interrupt versus polling mode
   The mode chosen can have significant impact on performance and latency. The host can determine the completion of data transfer using:
   a. Interrupts
      The application can program the accumulator and map it to the correct Interrupt Service Routine to be called when either the high priority or low priority of the accumulators have popped descriptors from a queue and placed the descriptor pointers into a host list buffer.
   b. Polling
      Reading one of the queue manager's status registers to check if any descriptors have been queued. This can be accomplished by either:
      - Popping the descriptor of a queue until it returns a non-NULL descriptor pointer using the *Qmss_queuePop()* API
      - Reading the queue's status register until it returns a non-zero count using the *Qmss_getQueueEntryCount()* API

5.4.7 Initialization Path

The CPPI LLD and QMSS LLD have to be initialized by the system once. Refer to the QMSS LLD initialization section and CPPI LLD initialization section for design details. Here we discuss the steps involved in initializing the LLDs from the caller's perspective.

- Step 1: QMSS low-level driver init
  - Fill in the linking RAM information
  - Decide on the number of linking RAMs required.
  - Whether the RAM is on-chip or off-chip
  - Decide on the maximum number of descriptors. This will determine the size of the linking RAM.
  - Linking RAM address must be a global memory address.
  - Download PDSP firmware if accumulator is used for interrupts.

  See Table 4 QMSS Initialization Configuration Structure for *Qmss_InitCfg* structure definition.
  See Table 5 QMSS Global Configuration Parameters Structure for *qmssGblCfgParams* structure definition.

- Step 2: CPPI low-level driver init.

  See Table 9 CPPI Global Configuration Parameters Structure for *cppiGblCfgParams* structure definition.

- Step 3: Creating memory regions
  - Create as many memory regions as required. Maximum of 20 are supported.
  - Fill in the memory region information.
  - Descriptor base address. This must be a global memory address.
  - Number of descriptor in this region
  - Size of descriptor
  - Whether descriptor should be managed by LLD
  - Memory region number to configure
    - If the memory region is allocated by the LLD, it is a sequential allocation. In this case the start index is computed by the LLD.
    - If a memory region to configure is specified by the user then a correct start index must be specified by the user.

- Step 4: Open a CPDMA instance
  - Fill in the CPDMA-related configuration parameters.

CPDMA number is one of the supported CPDMA listed below.
The CPDMA types can differ from SoC to SoC and are therefore defined in the CSL layer to make the LLD device independent. An example definition for a SOC.

```
File: csl_cppi.h

/** CPPI maximum number of CPDMAs */

#define CPPI_MAX_CPDMA                  6

typedef enum
{
   /** SRIO */
   SRIO_CPDMA = 0,
   /** AIF */
   AIF_CPDMA,
   /** FFTC A */
   FFTC_A_CPDMA,
   /** FFTC B */
   FFTC_B_CPDMA,
   /** PASS */
   PASS_CPDMA,
   /** QMSS */
   QMSS_CPDMA
}Cppi_CpDma;
```

- Step 5: Check if CPDMA is in loopback mode. Enable or disable loopback based on desired functionality. Refer to the CPPI user guide to check the normal operating mode for a given CPDMA.

- Step 6: Obtain the allocated descriptors
  - Memory region to allocate the descriptors from
  - Number of descriptors to be allocated
  - Queue where the descriptor is stored and returned to user
  - Indicates if the descriptor should be initialized
  - Fill in the initialization parameters for host or monolithic descriptor

5.4.8 Transmit

The figure illustrates the steps involved in transmitting a packet.

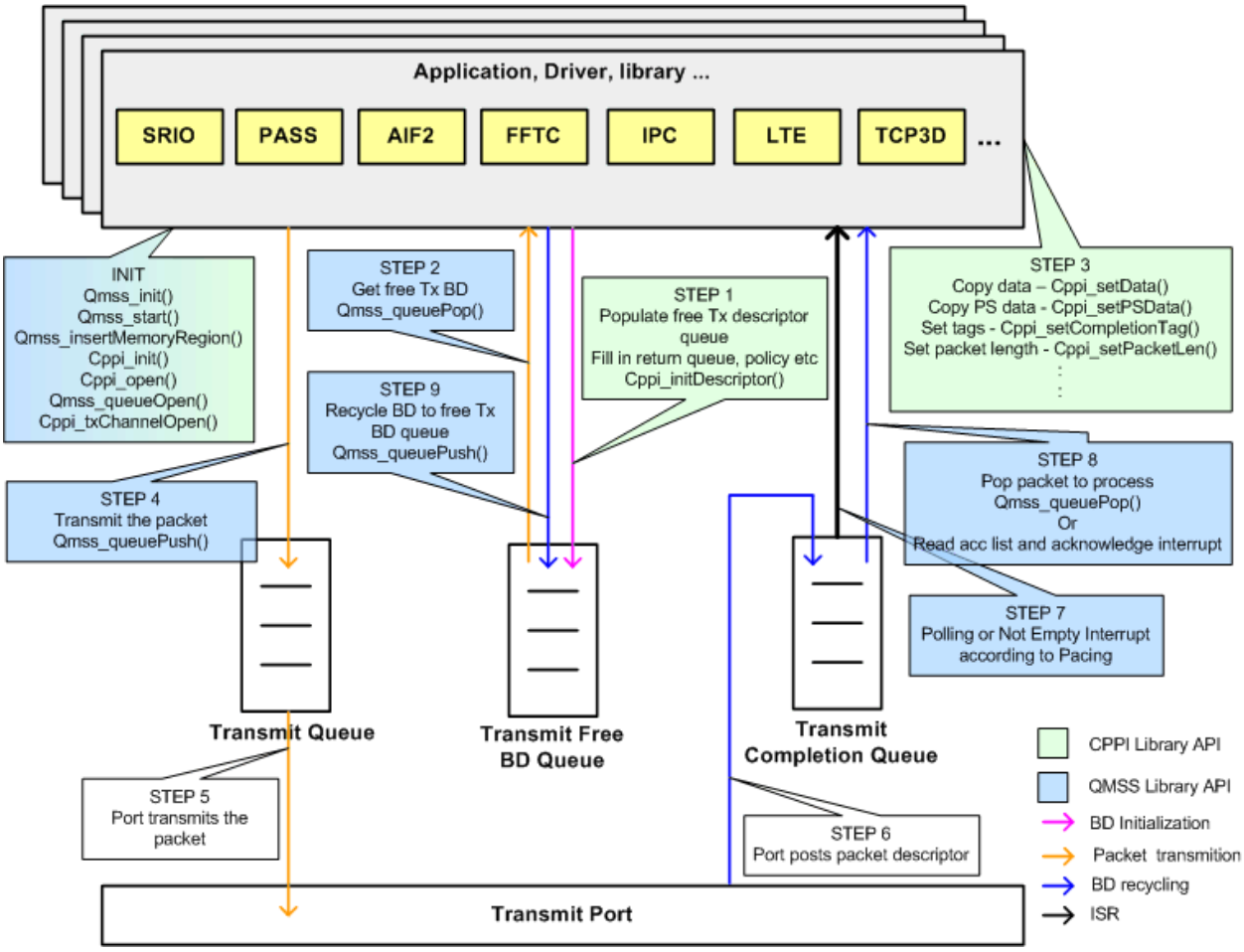

Figure 5 Transmit

5.4.8.1 Transmit Path – Detailed Steps

This section lists the steps involved in setting up the transmit channels, transmit queues, transmitting a packet, and post-processing.

Note: These steps might be part of the initialization sequence depending on the driver/application architecture.

- Transmit channel open

- Fill in the transmit channel configuration details.
- Specify the channel number or let LLD allocate it.
- Transmit scheduler priority for this channel.
- Channel can be enabled when opened
  OR
  Enable it later using the *Cppi_channelEnable()* API.

- Set up the transmit queue.
  - Recommendation is to use the queues with hardware threshold status. These queues are reserved for every CPDMA. Refer to Table 3 Sample Queue Allocation for details on queue allocation.
  - This queue will be used to queue transmit packets.

- Set up the transmit completion queue.
  - Recommendation is to use the queues with accumulation and interrupt support. There are low priority and high priority queues available. Refer to Table 3 Sample Queue Allocation for details on queue allocation.

Note: There need not be a separate transmit completion queue. The transmit completion queue can be the transmit free BD queue.

- Setup the transmit free queue with host/monolithic descriptors.
  - This queue will be used to get free descriptor for transmission. Recommendation is to use the queues with starvation counters. Refer to Table 3 Sample Queue Allocation for details on queue allocation.
  - Specify memory region to allocate descriptors from.
  - Number of descriptors.
  - Destination queue number to store the allocated descriptor as decided above.
  - Initialization parameters for host/monolithic descriptor
  - Return policy, return push policy, PS location for host descriptors
  - Data offset, return push policy for monolithic descriptors
  - Return queue number, EPIB present for both
    In this case the return queue is the transmit completion queue opened in step 3.

- Program the accumulator if interrupt mode is used for transmit completion notification. The list address provided for accumulation must be a global address.

- Configure the transmit queue threshold
  - Choose when the threshold should be asserted. If threshold is high the status is asserted when the size of the queue is at least equal to the threshold value. If threshold is low the status is asserted when the size of the queue is less than the threshold value. If threshold is set to zero, QM will not trigger the CDMA. The threshold value is encoded as 0'h3ff when it is ten or higher. It is (2^threshold-1) in the internal representation for other values.

- Enable the transmit channel if it is not enabled as part of step 1

- Preparing the packet to send
  - Get a free packet descriptor.
  - Copy data to transmit
  - Populate the descriptor fields
  - Set up any optional data.
    - Optional fields are timestamp, software information, and protocol specific data
  - Update packet length

- Send the packet.
  - Use the queue push functions to write to transmit queue

  Check the QMSS enqueue section for different versions of the push API.

- Recycling the descriptor.

  Upon receiving a transmit completion interrupt or polling the transmit completion queue, process the posted descriptors.

5.4.9 Receive

The figure illustrates the steps involved in receiving a packet.

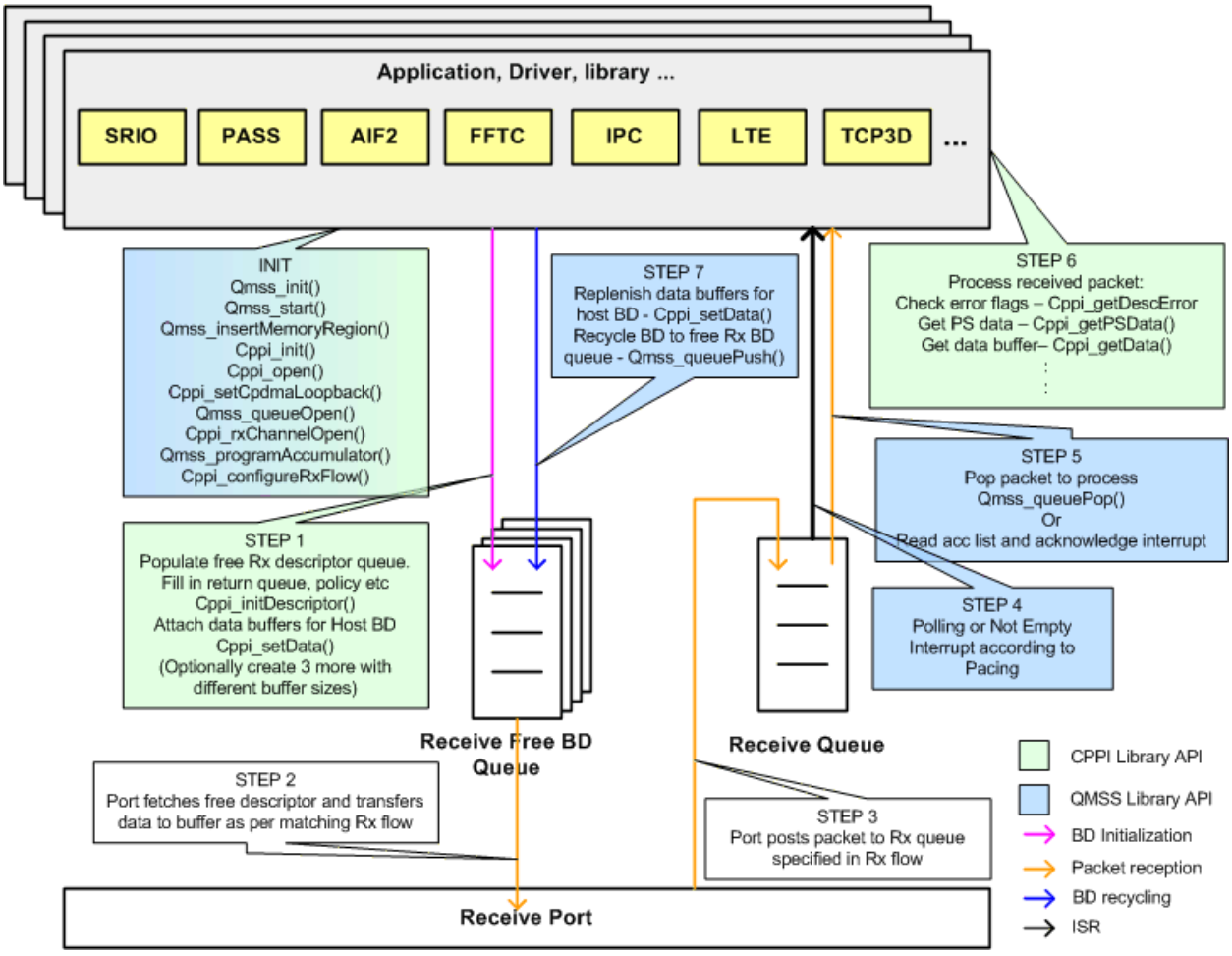

Figure 6 Receive

5.4.9.1 Receive Path – Detailed Steps

This section lists the steps involved in setting up the receive channels, receive queues, receiving a packet, and post-processing.

Note: These steps might be part of the initialization sequence depending on the driver/application architecture.

- Receive channel open
  - Fill in the receive channel configuration details.
  - Specify the channel number or let LLD allocate it.
  - Channel can be enabled when opened if the flow is already configured
    OR
    Enable it later using the C*ppi_channelEnable()* API.

- Setup the receive queue.
  - Recommendation is to use the queues with accumulation and interrupt support. There are low priority and high priority queues available. Refer to Table 3 Sample Queue Allocation for details on queue allocation.
  - This queue will be used by the hardware to queue received packets.

- Set up the receive free queue.
  - Recommendation is to use the queues with starvation counters. Refer to Table 3 Sample Queue Allocation for details on queue allocation.

- Set up the receive free queue with descriptors.
  - Specify memory region to allocate descriptors from.
  - Number of descriptors.
  - Destination queue number to store the allocated descriptor.
    The receive descriptors have to be populated with data buffers for host descriptors only. For this purpose we store them temporarily on a general purpose queue. The data buffer address must be a global memory address.
  - Initialization parameters for host/monolithic descriptor
  - Return policy, return push policy, PS location for host descriptors
  - Data offset, return push policy for monolithic descriptors
  - Return queue number, EPIB present for both types of descriptors
    In this case it is the receive free queue opened above.

- Attach data buffers to host descriptors. Pop descriptors off the general purpose queue, attach data buffers and then push them to the receive free queue.

- Program the accumulator if interrupt mode is used for receive notification. The list address provided for accumulation must be a global address.

- Configure receive flow.
  - If the receive channel using this flow is already enabled, disable or pause it.
  - Fill in the receive flow configuration details.
  - Specify flow id or have the LLD allocate it.
  - At a minimum, configure
    - Receive destination queue number
    - Free queue to get descriptors from
    - Descriptor type

- Data offset for monolithic descriptors

- Enable channel
- Receive a packet

  Upon receiving an interrupt or polling the receive queue, process the posted Rx packet.

- Replenish the receive data buffer
  - Allocate a new data buffer and attach to the current descriptor. This is done since the data buffer in the current descriptor might be used for further processing without having to copy the contents into another buffer.

5.4.10 Error Processing

5.4.10.1 Descriptor errors

```
static inline uint32_t Cppi_getDescError (Cppi_DescType descType, Cppi_Desc
*descAddr)
```

This function retrieves the error flags from the descriptor. API provided by the CPPI LLD.

5.4.10.2 Descriptor Starvation Errors

```
extern uint32_t Qmss_getStarvationCount (Qmss_QueueHnd hnd);
```

The starvation count is incremented every time the Free Descriptor/Buffer queue is read when empty. This function returns the starvation count of the queue. API provided by the QMSS LLD.

6 Memory Considerations

The queue manager and CPPI low-level driver manage shared resources. It is designed to be used by multiple drivers and applications running on multiple cores. This implies certain information related to resource management is placed in shared memory.

To handle these requirements the global data structures will be in shared memory.

Data sections are created for all global variables as shown below:

```
/* CPPI instance count to prevent reinitialization*/
#pragma DATA_SECTION (cppiInstance, ".cppi");
uint8_t        cppiInstance = 0;

/* CPPI object */
#pragma DATA_SECTION (cppiObject, ".cppi");
Cppi_Obj     cppiObject;
```

```
/* Maintain status of queues */
#pragma DATA_SECTION (queueFree, ".qmss");
uint8_t               queueFree[QMSS_MAX_QUEUES];

/* QMSS object */
Qmss_Obj             qmssGObj;
#pragma DATA_SECTION (qmssGObj, ".qmss");

/* QMSS instance count to prevent reinitialization */
#pragma DATA_SECTION (qmssInstance, ".qmss");
uint32_t              qmssInstance = 0;
```

These sections have to be placed in the shared memory via the linker.cmd file:

```
MEMORY
{
    L2SRAM (RWX) : org = 0x800000, len = 0x100000
    MSMCSRAM (RWX) : org = 0xc000000, len = 0x200000
}

SECTIONS
{
    .qmss: load >> MSMCSRAM
    .cppi: load >> MSMCSRAM
}
```

For memory allocations, the CPPI driver requires a heap to be created. An example of creation of heap in shared memory is provided in the "sample.cfg" file.

This heap is created in shared memory if CPPI resources are allocated from multiple cores. If a single core does all the resource allocation, the heap can be placed in local L2 memory.

When the heap is placed in shared memory it **MUST** be a separate heap used only by CPPI to avoid false sharing issues when caches are enabled.

The example code uses memory allocation APIs from IPC to allocate memory from shared memory to demonstrate channel and flow allocation from multiple cores.

Below is the code snippet used in the "example" to initialize shared heap.

```
/* Handle to CPPI heap in shared memory */
void myStartupFxn (void)
{
        MultiProc_setLocalId (CSL_chipReadReg (CSL_CHIP_DNUM));
}
/* Initialize heap in shared memory using IPC */
Ipc_start();
```

Note: If all the drivers and applications using the LLD are running on the same core, the global data structures can be placed in memory local to the core. The heap can be created in local memory as well.

Below is the code snippet used in the "test" to initialize local heap.

```
/* Handle to CPPI heap in local memory using BIOS XDC */
IHeap_Handle        cppiHeap;
static void cppiHeapInit ()
{
   cppiHeap = HeapMem_Handle_upCast (cppiLocalHeap);
}
```

7 Critical Sections

The resources of CPPI and QMSS low-level drivers are shared by various users. The drivers and applications could be running on different cores. They also manage resource allocation of different resources.

Resource allocation APIs and access to global data structures must be protected by critical sections to ensure intended results.

Critical sections should provide protection against preemption. If the drivers/applications are run on different cores, the critical sections should also define cross-core locks.

The LLD has callout functions that can be modified to suit these needs. Sample callouts are provided in the OSAL layer discussed in the next section.

8 OS Considerations

The CPPI and QMSS low-level drivers are OS-independent.
The LLDs have callouts to functions such as critical section enter and exit, malloc, free, log.

8.1 CPPI OSAL

The OSAL is the operating system abstraction layer which is used to port the CPPI driver to a specific OS. The OSAL callouts are implemented in the "`cppi_osal.h`" header file and need to be ported by the application developers to their specific operating system.

8.1.1 Memory Allocation

The CPPI driver allocates memory only in control path. There are no memory allocations done in the data path.

Internally the CPPI driver uses the *Cppi_osalMalloc* macro to perform all memory allocations. The OSAL adaptation layer ports this macro to the following API prototype:

```
void* Osal_cppiMalloc (uint32_t num_bytes)
```

The parameter *numBytes* reflects the total amount of memory that is requested.

8.1.2 Memory Cleanup

Internally the CPPI driver uses the *Cppi_osalFree* macro to perform all memory cleanups. The OSAL adaptation layer ports this macro to the following API prototype:

```
void Osal_cppiFree (void* ptr, uint32_t size)
```

The parameter *ptr* reflects the address of the memory block which needs to be cleaned up. The parameter *size* reflects the size of the memory which is being cleaned up.

8.1.3 Critical Section Enter

Internally the CPPI driver uses the *Cppi_osalCsEnter* macro to begin a critical section. The OSAL adaptation layer ports this macro to the following API prototype:

```
void* Osal_cppiCsEnter (void)
```

The return parameter is an opaque handle used to lock the critical section.
The function protects when accesses are performed from:

- Multiple cores
- Multiple threads on single core

8.1.4 Critical Section Exit

Internally the CPPI driver uses the *Cppi_osalCsExit* macro to end a critical section. The OSAL adaptation layer ports this macro to the following API prototype:

```
void Osal_cppiCsExit (void* CsHandle)
```

The parameter `CsHandle` is used to unlock the critical section.

8.1.5 Logging API

Internally the CPPI driver uses the *Cppi_osalLog* macro to perform all logging operations. The OSAL adaptation layer ports this macro to the following API prototype:

```
void Osal_cppiLog( String fmt, ... )
```

The parameter *fmt* is a *printf* style formatted string. This should only be defined and used for debugging purposes.

8.1.6 Memory Access Hooks

The CPPI LLD requires the data structures to be located in shared memory when it is used in a multi-core system. These data structures need to be synchronized to ensure that the contents of the cache and memory are always in sync with each other.

All cache coherency operations are performed only in control path.

Internally the CPPI LLD uses *Cppi_osalBeginMemAccess* macro to indicate that an access to the specified memory region is starting. The OSAL adaptation layer ports this macro to the following API prototype:

```
void  Osal_cppiBeginMemAccess(void* ptr, uint32_t size)
```

The function takes as parameters the address of the memory & number of bytes which are being accessed. The memory if cached needs to be invalidated so that the contents of the cache are reloaded back from the actual memory. This will ensure that there is no stale data in the cache.

Internally the CPPI LLD uses *Cppi_osalEndMemAccess* macro to indicate that an access to the specified memory region is ending. The OSAL adaptation layer ports this macro to the following API prototype:

```
void  Osal_cppiEndMemAccess(void* ptr, uint32_t size)
```

The function takes as parameters the address of the memory & number of bytes which are being accessed. The memory if cache needs to be written back so that the contents of the cache are synched up with the actual memory. This is true in the case of Write-back cache however if the caches are operating in Write through mode this API could be a NOP since the cache contents have already been written back to actual memory.

8.2 QMSS OSAL

The OSAL is the operating system abstraction layer which is used to port the QMSS driver to a specific OS. The OSAL callouts are implemented in the "`qmss_osal.h`" header file and need to be ported by the application developers to their specific operating system.

8.2.1 Memory Allocation

The QMSS driver allocates memory only in control path. There are no memory allocations done in the data path.

Internally the QMSS driver uses the *Qmss_osalMalloc* macro to perform all memory allocations. The OSAL adaptation layer ports this macro to the following API prototype:

```
void* Osal_qmssMalloc (uint32_t num_bytes)
```

The parameter *numBytes* reflects the total amount of memory that is requested.

8.2.2 Memory Cleanup

Internally the QMSS driver uses the *Qmss_osalFree* macro to perform all memory cleanups. The OSAL adaptation layer ports this macro to the following API prototype:

```
void Osal_qmssFree (void* ptr, uint32_t size)
```

The parameter *ptr* reflects the address of the memory block which needs to be cleaned up. The parameter *size* reflects the size of the memory which is being cleaned up.

8.2.3 Critical Section Enter

Internally the QMSS driver uses the *Qmss_osalCsEnter* macro to begin a critical section. The OSAL adaptation layer ports this macro to the following API prototype:

```
void* Osal_qmssCsEnter (void)
```

The return parameter is an opaque handle used to lock the critical section.
The function protects when accesses are performed from:

- Multiple cores
- Multiple threads on single core

8.2.4 Critical Section Exit

Internally the QMSS driver uses the *Qmss_osalCsExit* macro to end a critical section. The OSAL adaptation layer ports this macro to the following API prototype:

```
void Osal_qmssCsExit (void* CsHandle)
```

The parameter *CsHandle* is used to unlock the critical section.

8.2.5 Critical Section Enter

Internally the QMSS driver uses the *Qmss_osalMtCsEnter* macro to begin a critical section for protection against multiple threads when cross-core protection is not required. The OSAL adaptation layer ports this macro to the following API prototype:

```
void* Osal_qmssMtCsEnter (void)
```

The return parameter is an opaque handle used to lock the critical section.

8.2.6 Critical Section Exit

Internally the QMSS driver uses the *Qmss_osalMtCsExit* macro to end a critical section started using *Qmss_osalMtCsEnter* macro. The OSAL adaptation layer ports this macro to the following API prototype:

```
void Osal_qmssMtCsExit (void* CsHandle)
```

The parameter *CsHandle* is used to unlock the critical section.

8.2.7 Logging API

Internally the QMSS driver uses the *Qmss_osalLog* macro to perform all logging operations. The OSAL adaptation layer ports this macro to the following API prototype:

```
void Osal_qmssLog( char* fmt, ... )
```

The parameter *fmt* is a *printf* style formatted string. This should only be defined and used for debugging purposes.
Sample functions are provided in the “sample_osal.c” and “infrastructure_osal.c” files. These can be ported to any OS. The porting guidelines are mentioned in the integration section

8.2.8 Memory Access Hooks

The QMSS LLD requires the data structures to be located in shared memory when it is used in a multi-core system. These data structures need to be synchronized to ensure that the contents of the cache and memory are always in sync with each other.

All cache coherency operations are performed only in control path.

Internally the QMSS LLD uses *Qmss_osalBeginMemAccess* macro to indicate that an access to the specified memory region is starting. The OSAL adaptation layer ports this macro to the following API prototype:

```
void  Osal_qmssBeginMemAccess(void* ptr, uint32_t size)
```

The function takes as parameters the address of the memory & number of bytes which are being accessed. The memory if cached needs to be invalidated so that the contents of the cache are reloaded back from the actual memory. This will ensure that there is no stale data in the cache.

Internally the QMSS LLD uses *Qmss_osalEndMemAccess* macro to indicate that an access to the specified memory region is ending. The OSAL adaptation layer ports this macro to the following API prototype:

```
void  Osal_qmssEndMemAccess(void* ptr, uint32_t size)
```

The function takes as parameters the address of the memory & number of bytes which are being accessed. The memory if cache needs to be written back so that the contents of the cache are synched up with the actual memory. This is true in the case of Write-back cache however if the caches are operating in Write through mode this API could be a NOP since the cache contents have already been written back to actual memory.

9 Integration

The CPPI low-level driver depends on the following components:

- QMSS LLD
- CSL

The QMSS low-level driver depends on the following components:

- CSL

These dependent components have to be installed before the LLDs can be integrated. The CPPI and QMSS low-level drivers are released in source code and in pre-built library. Applications can decide how to use the low-level driver.

9.1 Pre-built Approach

In this approach the application developers decide to use the CPPI or QMSS driver pre-built libraries as is. The following steps need to be performed:

a. The application developers modify their application configuration file to use the CPPI or QMSS package.

```
var Cppi = xdc.loadPackage('ti.drv.cppi');
var Qmss = xdc.loadPackage('ti.drv.qmss');
```

b. Ensure that the XDCPATH is configured to have the path to the CPPI package.
c. Ensure that the XDCPATH is configured to have the path to the QMSS package.
d. This implies that XDC Configuration scripts will link the application using the CPPI/QMSS driver libraries (`Module.xs`).
e. The application authors need to provide an OSAL implementation file for CPPI and QMSS and ensure that this is linked with the application; failure to do so will result in linking errors.

   The OSAL source file should have the following CPPI OSAL functions implemented:

```
void* Osal_cppiMalloc (uint32_t num_bytes);
void Osal_cppiFree (void* ptr, uint32_t size);
void* Osal_cppiCsEnter (void);
void Osal_cppiCsExit (void* CsHandle);
void Osal_cppiLog ( char* fmt, ... );
void Osal_cppiBeginMemAccess (void *ptr, uint32_t size);
void Osal_cppiEndMemAccess (void *ptr, uint32_t size);
```

Table 15 CPPI OSAL Functions

The OSAL source file should have the following QMSS OSAL functions implemented:

```
void* Osal_qmssMalloc (uint32_t num_bytes);
void Osal_qmssFree (void* ptr, uint32_t size);
void* Osal_qmssCsEnter (void);
void Osal_qmssCsExit (void* CsHandle);
void* Osal_qmssMtCsEnter (void);
void Osal_qmssMtCsExit (void* CsHandle);
void Osal_qmssLog ( char* fmt, ... );
void Osal_qmssBeginMemAccess (void *ptr, uint32_t size);
void Osal_qmssEndMemAccess (void *ptr, uint32_t size);
```

Table 16 QMSS OSAL Functions

Note: Since the CPPI LLD depends on the QMSS LLD, the OSAL implementation for CPPI should also have the QMSS implementations.

If the application is not using XDC then replace steps (a) and (b) above with the following steps instead:

a. Append the include path to the top level CPPI package directory
b. Append the include path to the top-level QMSS package directory
c. Make sure the CPPI pre-built libraries are added to the application project and the library search path is configured correctly.
d. Make sure the QMSS pre-built libraries are added to the application project and the library search path is configured correctly.

This approach is highlighted in the CPPI and QMSS "`example`" projects.

9.2 Rebuild Library

In this approach the application developers decide to use the CPPI or QMSS driver source code and add these files to the application project to rebuild the CPPI or QMSS driver code base. The following steps need to be redone:

a. Application developers should port the file "`cppi_osal.h`" and "`qmss_osal.h`" to their operating system environment. *Developers are recommended to create a copy of this file and place it in their application directory.* They should use the file which is provided in the CPPI and QMSS installation only as a template. The goal here should be to map the `Cppi_osalXXX` macros and `Qmss_osalXXX` macros to the OS calls directly thus reducing the overhead of an API callout. For example:

```
#define Cppi_osalMalloc             Osal_biosMalloc
#define Qmss_osalMalloc             Osal_biosMalloc
```

b. Application developers should port the file "`cppi_types.h`" and "`qmss_types.h`"to the application environment. *Developers are recommended to create a copy of this file and place it in their application directory.*
c. Append the include path to the top-level CPPI package directory
d. Append the include path to the top-level QMSS package directory
e. Add the CPPI or QMSS driver files listed in the `src` directory to the application build files.

The approach above is highlighted in the CPPI and QMSS `test` directory.

10 Phase II Additions

11 CHECKLIST

The below app note is a checklist for using CPPI and QMSS IP. It is provided to help users avoid common mistakes made during configuration and help in debugging.

11.1 Setup related

- All addresses given to the hardware should be global addresses.
- The address should be aligned on a 16 byte boundary
- This includes External Linking RAM memory, any descriptor memory and accumulator list.

11.2 Linking RAM

- 16K descriptors can be tracked using the internal linking RAM.
- Use External linking RAM for additional descriptors.
    - Each entry MUST be 64 bit wide.
- Sum of internal and external memory MUST be equal to or greater than maximum number of descriptors used in the system.

11.3 Memory region configuration

- 20 memory regions available.
- Memory region have to be configured in ascending order of memory addresses.
- Number of descriptors in the memory region
    - MUST be a minimum of 32.
    - MUST be 2^(5 or greater)
    - Maximum supported value 2^20
- Size of each descriptor in the memory region MUST be a multiple of 16. Implies memory address MUST be aligned on 16 byte boundary
- No overlapping memory regions
- Linking RAM MUST be equal to or greater than the total number of descriptors in all memory region.
- LLD performs all the above checks when Qmss_insertMemoryRegion() API is called.

11.4 Push/pop

- Push is atomic only when descriptor address alone is written. i.e., Reg D is the only register written. Does not need critical sections to protect from multicores or multithreads pushing to the same queue.

- Use Queue Proxy when push descriptor address along with packet size and location (head/tail) of queue. i.e., Reg C and Reg D are written. Queue Proxy provides multicore protection. Critical sections to protect from multiple thread accesses are still required.
- Descriptor address must align on a size boundary that was configured in the memory region the descriptor belonged to.
- Writing a NULL address will empty all the entries queued on the queue.
- The lower nibble of descriptor address during push contains the "Hint Size". Hint size is the size of the descriptor. Does not include data size. Minimum size is 16 bytes. Maximum size is 256 bytes.

- Pop of descriptor address is atomic. i.e., Reg D is the only register read. Does not need critical sections to protect from multiple cores or multiple threads popping from the same queue.
- Popping the descriptor address along with packet size is not multicore/multithread safe. i.e., Reg C and Reg D are read. It is possible that the descriptor is popped by another core/task between the time taken to read the packet size and the descriptor address by the first core/task.
- The lower nibble of descriptor address will contain the "Hint Size" if hint size was specified during the push operation. Caller should clear the lower order 4 bits before using the descriptor.
- Popping a NULL descriptor when queue should have had descriptors means the linking RAM is corrupted.
- CPDMA lockup can be caused by
    - Setting the hint size too small to cover the entire descriptor (packet info, buffer info, extended info, ps words).
    - Setting the buffer sizes smaller than the packet size.
    - Zero byte buffers on middle of packet buffers

11.5 CPDMA configuration

- Configure the IP in correct operating mode. CPDMA Loopback mode is enabled by default. For QMSS this is the normal operating mode. For other IPs such as SRIO, PA, FFTC, AIF2, loopback mode must be disabled to send the descriptor out of CPDMA.
- QM base address in CPPI global register is configured. By default QM0 base address is configured. Configure QM1-3 if required.

11.6 Statistics

Queue statistics that are available are

- Number of entries queued on the queue
- Total number of bytes that are contained in all of the packets that are currently queued on the queue.
- Packet size of the packet queued at the head of the queue

- Starvation counter is incremented every time the Free Descriptor/Buffer queue is read by hardware when queue is empty.
- Threshold status of the queue.
    - The threshold bit is set for a queue if the number of element in a queue is above or below a certain threshold number of items configured using Queue status configuration Register D.
    - The threshold bit is set for a queue if there is atleast 1 element on the queue when the threshold is not set using Queue status configuration Register D.
- LLD provides APIs to retrieve above statistics.

CPPI related statistics are unique to each CPDMA. E.g, SRIO and QMSS do not provide any descriptor tx/rx statistics. FFTC does. Check individual IPS.

11.7 INTERRUPTS

- Interrupt MUST be acknowledged and EOI register written to before another interrupt is generated

12 APPENDIX

CPPI device-specific global configuration parameters - `cppi_device.c` file:

```
/**
 *   @file  cppi_device.c
 *
 *   @brief
 *       This file contains the device specific configuration and initialization routines
 *       for CPPI Low Level Driver.
 *
 *  \par
 *  ============================================================================
 *  @n   (C) Copyright 2009, Texas Instruments, Inc.
 *  @n   Use of this software is controlled by the terms and conditions found
 *  @n   in the license agreement under which this software has been supplied.
 *  ============================================================================
 *  \par
 */

/* CPPI LLD includes */
#include <cppi_types.h>

/* CSL RL includes */
#include <ti/csl/cslr_device.h>
#include <ti/csl/cslr_cppidma_global_config.h>
#include <ti/csl/cslr_cppidma_rx_channel_config.h>
#include <ti/csl/cslr_cppidma_rx_flow_config.h>
#include <ti/csl/cslr_cppidma_tx_channel_config.h>
#include <ti/csl/cslr_cppidma_tx_scheduler_config.h>
#include <ti/csl/csl_cppi.h>

/** @addtogroup CPPI_LLD_DATASTRUCT
@{
*/
/** @brief CPPI LLD initialization parameters */
/* TODO These values are according to Sim release. Not all are accurate */
Cppi_GlobalConfigParams cppiGblCfgParams[CPPI_MAX_CPDMA] =
{
    {
        /** CPDMA this configuration belongs to */
        Cppi_CpDma_SRIO_CPDMA,
        /** Maximum supported Rx Channels */
        (uint32_t) 16u,
        /** Maximum supported Tx Channels */
        (uint32_t) 16u,
        /** Maximum supported Rx Flows */
        (uint32_t) 20u,
        /** Priority for all Rx transactions of this CPDMA */
        (uint8_t) 0u,
        /** Priority for all Tx transactions of this CPDMA */
        (uint8_t) 0u,

        /** Base address for the CPDMA overlay registers */

        /** Global Config registers */
```

```
        (void*) CSL_SRIO_CONFIG_CPPI_DMA_GLOBAL_CFG_REGS,
        /** Tx Channel Config registers */
        (void*) CSL_SRIO_CONFIG_CPPI_DMA_TX_CFG_REGS,
        /** Rx Channel Config registers */
        (void*) CSL_SRIO_CONFIG_CPPI_DMA_RX_CFG_REGS,
        /** Tx Channel Scheduler registers */
        (void*) CSL_SRIO_CONFIG_CPPI_DMA_TX_SCHEDULER_CFG_REGS,
        /** Rx Flow Config registers */
        (void*) CSL_SRIO_CONFIG_CPPI_DMA_RX_FLOW_CFG_REGS,
        /** Queue Manager 0 base address register */
        (uint32_t) 0x02a20000,
        /** Queue Manager 1 base address register */
        (uint32_t) 0x02a30000,
        /** Queue Manager 2 base address register */
        (uint32_t) 0xFFFFFFFF,
        /** Queue Manager 3 base address register */
        (uint32_t) 0xFFFFFFFF,
    },
    {
        /** CPDMA this configuration belongs to */
        Cppi_CpDma_AIF_CPDMA,
        /** Maximum supported Rx Channels */
        (uint32_t) 129u,
        /** Maximum supported Tx Channels */
        (uint32_t) 129u,
        /** Maximum supported Rx Flows */
        (uint32_t) 129u,
        /** Priority for all Rx transactions of this CPDMA */
        (uint8_t) 0u,
        /** Priority for all Tx transactions of this CPDMA */
        (uint8_t_t) 0u,

        /** Base address for the CPDMA overlay registers */

        /** Global Config registers */
        (void*) CSL_AIF2_CFG_CPPI_DMA_GLOBAL_CFG_REGS,
        /** Tx Channel Config registers */
        (void*) CSL_AIF2_CFG_CPPI_DMA_TX_CFG_REGS,
        /** Rx Channel Config registers */
        (void*) CSL_AIF2_CFG_CPPI_DMA_RX_CFG_REGS,
        /** Tx Channel Scheduler registers */
        (void*) CSL_AIF2_CFG_CPPI_DMA_TX_SCHEDULER_CFG_REGS,
        /** Rx Flow Config registers */
        (void*) CSL_AIF2_CFG_CPPI_DMA_RX_FLOW_CFG_REGS,
        /** Queue Manager 0 base address register */
        (uint32_t) 0x02a20000,
        /** Queue Manager 1 base address register */
        (uint32_t) 0x02a30000,
        /** Queue Manager 2 base address register */
        (uint32_t) 0xFFFFFFFF,
        /** Queue Manager 3 base address register */
        (uint32_t) 0xFFFFFFFF,
```

```
    },
    {
        /** CPDMA this configuration belongs to */
        Cppi_CpDma_FFTC_A_CPDMA,
        /** Maximum supported Rx Channels */
        (uint32_t) 4u,
        /** Maximum supported Tx Channels */
        (uint32_t) 4u,
        /** Maximum supported Rx Flows */
        (uint32_t) 8u,
        /** Priority for all Rx transactions of this CPDMA */
        (uint8_t_t) 0u,
        /** Priority for all Tx transactions of this CPDMA */
        (uint8_t_t) 0u,

        /** Base address for the CPDMA overlay registers */

        /** Global Config registers */
        (void*) CSL_FFTC_A_CONFIG_CPPI_DMA_GLOBAL_CFG_REGS,
        /** Tx Channel Config registers */
        (void*) CSL_FFTC_A_CONFIG_CPPI_DMA_TX_CFG_REGS,
        /** Rx Channel Config registers */
        (void*) CSL_FFTC_A_CONFIG_CPPI_DMA_RX_CFG_REGS,
        /** Tx Channel Scheduler registers */
        (void*) CSL_FFTC_A_CONFIG_CPPI_DMA_TX_SCHEDULER_CFG_REGS,
        /** Rx Flow Config registers */
        (void*) CSL_FFTC_A_CONFIG_CPPI_DMA_RX_FLOW_CFG_REGS,
        /** Queue Manager 0 base address register */
        (uint32_t) 0x02a20000,
        /** Queue Manager 1 base address register */
        (uint32_t) 0x02a30000,
        /** Queue Manager 2 base address register */
        (uint32_t) 0xFFFFFFFF,
        /** Queue Manager 3 base address register */
        (uint32_t) 0xFFFFFFFF,
    },
    {
        /** CPDMA this configuration belongs to */
        Cppi_CpDma_FFTC_B_CPDMA,
        /** Maximum supported Rx Channels */
        (uint32_t) 4u,
        /** Maximum supported Tx Channels */
        (uint32_t) 4u,
        /** Maximum supported Rx Flows */
        (uint32_t) 8u,
        /** Priority for all Rx transactions of this CPDMA */
        (uint8_t_t) 0u,
        /** Priority for all Tx transactions of this CPDMA */
        (uint8_t_t) 0u,

        /** Base address for the CPDMA overlay registers */
```

```
        /** Global Config registers */
        (void*) CSL_FFTC_B_CONFIG_CPPI_DMA_GLOBAL_CFG_REGS,
        /** Tx Channel Config registers */
        (void*) CSL_FFTC_B_CONFIG_CPPI_DMA_TX_CFG_REGS,
        /** Rx Channel Config registers */
        (void*) CSL_FFTC_B_CONFIG_CPPI_DMA_RX_CFG_REGS,
        /** Tx Channel Scheduler registers */
        (void*) CSL_FFTC_B_CONFIG_CPPI_DMA_TX_SCHEDULER_CFG_REGS,
        /** Rx Flow Config registers */
        (void*) CSL_FFTC_B_CONFIG_CPPI_DMA_RX_FLOW_CFG_REGS,
        /** Queue Manager 0 base address register */
        (uint32_t) 0x02a20000,
        /** Queue Manager 1 base address register */
        (uint32_t) 0x02a30000,
        /** Queue Manager 2 base address register */
        (uint32_t) 0xFFFFFFFF,
        /** Queue Manager 3 base address register */
        (uint32_t) 0xFFFFFFFF,
    },
    {
        /** CPDMA this configuration belongs to */
        Cppi_CpDma_PASS_CPDMA,
        /** Maximum supported Rx Channels */
        (uint32_t) 24u,
        /** Maximum supported Tx Channels */
        (uint32_t) 9u,
        /** Maximum supported Rx Flows */
        (uint32_t) 32u,
        /** Priority for all Rx transactions of this CPDMA */
        (uint8_t_t) 0u,
        /** Priority for all Tx transactions of this CPDMA */
        (uint8_t_t) 0u,

        /** Base address for the CPDMA overlay registers */

        /** Global Config registers */
        (void*) CSL_PA_SS_CFG_CPPI_DMA_GLOBAL_CFG_REGS,
        /** Tx Channel Config registers */
        (void*) CSL_PA_SS_CFG_CPPI_DMA_TX_CFG_REGS,
        /** Rx Channel Config registers */
        (void*) CSL_PA_SS_CFG_CPPI_DMA_RX_CFG_REGS,
        /** Tx Channel Scheduler registers */
        (void*) CSL_PA_SS_CFG_CPPI_DMA_TX_SCHEDULER_CFG_REGS,
        /** Rx Flow Config registers */
        (void*) CSL_PA_SS_CFG_CPPI_DMA_RX_FLOW_CFG_REGS,
        /** Queue Manager 0 base address register */
        (uint32_t) 0x02a20000,
        /** Queue Manager 1 base address register */
        (uint32_t) 0x02a30000,
        /** Queue Manager 2 base address register */
        (uint32_t) 0xFFFFFFFF,
        /** Queue Manager 3 base address register */
```

```
        (uint32_t) 0xFFFFFFFF,
    },
    {
        /** CPDMA this configuration belongs to */
        Cppi_CpDma_QMSS_CPDMA,
        /** Maximum supported Rx Channels */
        (uint32_t) 32u,
        /** Maximum supported Tx Channels */
        (uint32_t) 32u,
        /** Maximum supported Rx Flows */
        (uint32_t) 64u,
        /** Priority for all Rx transactions of this CPDMA */
        (uint8_t_t) 0u,
        /** Priority for all Tx transactions of this CPDMA */
        (uint8_t_t) 0u,

        /** Base address for the CPDMA overlay registers */

        /** Global Config registers */
        (void*) CSL_QM_SS_CFG_CPPI_DMA_GLOBAL_CFG_REGS,
        /** Tx Channel Config registers */
        (void*) CSL_QM_SS_CFG_CPPI_DMA_TX_CFG_REGS,
        /** Rx Channel Config registers */
        (void*) CSL_QM_SS_CFG_CPPI_DMA_RX_CFG_REGS,
        /** Tx Channel Scheduler registers */
        (void*) CSL_QM_SS_CFG_CPPI_DMA_TX_SCHEDULER_CFG_REGS,
        /** Rx Flow Config registers */
        (void*) CSL_QM_SS_CFG_CPPI_DMA_RX_FLOW_CFG_REGS,
        /** Queue Manager 0 base address register */
        (uint32_t) 0x02a20000,
        /** Queue Manager 1 base address register */
        (uint32_t) 0x02a30000,
        /** Queue Manager 2 base address register */
        (uint32_t) 0xFFFFFFFF,
        /** Queue Manager 3 base address register */
        (uint32_t) 0xFFFFFFFF,
    },
};

/**
@}
*/
```

Table 17 Device-Specific CPPI Configuration File

QMSS device-specific global configuration parameters – `qmss_device.c` file:

```
/**
 *   @file  qmss_device.c
 *
 *   @brief
 *       This file contains the device specific configuration and initialization routines
 *       for QMSS Low Level Driver.
 *  \par
 *  ============================================================================
 *  @n   (C) Copyright 2009, Texas Instruments, Inc.
 *  @n   Use of this software is controlled by the terms and conditions found
 *  @n   in the license agreement under which this software has been supplied.
 *  ============================================================================
 *  \par
 */

/* QMSS LLD includes */
#include <qmss_types.h>

/* CSL RL includes */
#include <ti/csl/cslr_device.h>
#include <ti/csl/cslr_qm_config.h>
#include <ti/csl/cslr_qm_descriptor_region_config.h>
#include <ti/csl/cslr_qm_queue_management.h>
#include <ti/csl/cslr_qm_queue_status_config.h>
#include <ti/csl/cslr_qm_intd.h>
#include <ti/csl/cslr_pdsp.h>
#include <ti/csl/csl_qm_queue.h>

/** @addtogroup QMSS_LLD_DATASTRUCT
@{
*/
/** @brief QMSS LLD initialization parameters */
/* TODO These values are according to Sim release. Not all are accurate */
Qmss_GlobalConfigParams qmssGblCfgParams[] =
{
    /** Maximum number of queue Managers */
    (uint32_t) 2u,
    /** Maximum number of queues */
    (uint32_t) 8192u,
    {
    /** Base queue number and Maximum supported low priority queues */
    {QMSS_LOW_PRIORITY_QUEUE_BASE, QMSS_MAX_LOW_PRIORITY_QUEUE},
    /** Base queue number and Maximum supported AIF queues */
    {QMSS_AIF_QUEUE_BASE, QMSS_MAX_AIF_QUEUE},
    /** Base queue number and Maximum supported PASS queues */
    {QMSS_PASS_QUEUE_BASE, QMSS_MAX_PASS_QUEUE},
    /** Base queue number and Maximum supported Intc Pend queues */
    {QMSS_INTC_QUEUE_BASE, QMSS_MAX_INTC_QUEUE},
    /** Base queue number and Maximum supported SRIO queues */
    {QMSS_SRIO_QUEUE_BASE, QMSS_MAX_SRIO_QUEUE},
    /** Base queue number and Maximum supported FFTC A queues */
    {QMSS_FFTC_A_QUEUE_BASE, QMSS_MAX_FFTC_A_QUEUE},
```

```
    /** Base queue number and Maximum supported FFTC B queues */
    {QMSS_FFTC_B_QUEUE_BASE, QMSS_MAX_FFTC_B_QUEUE},
    /** Base queue number and Maximum supported high priority queues */
    {QMSS_HIGH_PRIORITY_QUEUE_BASE, QMSS_MAX_HIGH_PRIORITY_QUEUE},
    /** Base queue number and Maximum supported starvation counter queues */
    {QMSS_STARVATION_COUNTER_QUEUE_BASE, QMSS_MAX_STARVATION_COUNTER_QUEUE},
    /** Base queue number and Maximum supported infrastructure queues */
    {QMSS_INFRASTRUCTURE_QUEUE_BASE, QMSS_MAX_INFRASTRUCTURE_QUEUE},
    /** Base queue number and Maximum supported traffic shaping queues */
    {QMSS_TRAFFIC_SHAPING_QUEUE_BASE, QMSS_MAX_TRAFFIC_SHAPING_QUEUE},
    /** Base queue number and Maximum supported general purpose queues */
    {QMSS_GENERAL_PURPOSE_QUEUE_BASE, QMSS_MAX_GENERAL_PURPOSE_QUEUE},

    /* Unused */
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    {0u, 0u},
    },
    /** Base address for the CPDMA overlay registers */

    /** QM Global Config registers */
    (void *) CSL_QM_SS_CFG_CONFIG_STARVATION_COUNTER_REGS,
    /** QM Descriptor Config registers */
    (void *) CSL_QM_SS_CFG_DESCRIPTION_REGS,
    /** QM queue Management registers */
    (void *) CSL_QM_SS_CFG_QM_QUEUE_DEQUEUE_REGS,
    /** QM queue Management Proxy registers */
    (void *) CSL_QM_SS_CFG_PROXY_QUEUE_DEQUEUE_REGS,
    /** QM queue status registers */
    (void *) CSL_QM_SS_CFG_QUE_PEEK_REGS,
    /** QM INTD registers */
    (void *) CSL_QM_SS_CFG_INTD_REGS,
    /** QM PDSP1 command register */
    {
        (void *) CSL_QM_SS_CFG_SCRACH_RAM1_REGS,
        /** QM PDSP2 command register */
        (void *) CSL_QM_SS_CFG_SCRACH_RAM2_REGS,
    },
    /** QM PDSP 1 control register */
    {
        (void *) CSL_QM_SS_CFG_ADSP1_REGS,
        /** QM PDSP 2 control register */
```

```
        (void *) CSL_QM_SS_CFG_ADSP2_REGS,
    },
    /** QM PDSP 1 IRAM register */
    {
        (void *) CSL_QM_SS_CFG_APDSP1_RAM_REGS,
        /** QM PDSP 2 IRAM register */
        (void *) CSL_QM_SS_CFG_APDSP2_RAM_REGS,
    },
    /** QM Status RAM */
    (void *) CSL_QM_SS_CFG_QM_STATUS_RAM_REGS,
    /** QM Linking RAM register */
    (void *) CSL_QM_SS_CFG_LINKING_RAM_REGS,
    /** QM McDMA register */
    (void *) CSL_QM_SS_CFG_MCDMA_REGS,
    /** QM Timer16 register */
    {
        (void *) CSL_QM_SS_CFG_TIMER1_REGS,
        (void *) CSL_QM_SS_CFG_TIMER2_REGS,
    },
    /** QM queue Management registers, accessed via DMA port */
    (void *) CSL_QM_SS_DATA_QM_QUEUE_DEQUEUE_REGS,
    /** QM queue Management Proxy registers, accessed via DMA port */
    (void *) CSL_QM_SS_DATA_PROXY_QUEUE_DEQUEUE_REGS,
};
/**
@}
*/
```

Table 18 Device-Specific QMSS Configuration File

CPPI Error Codes:

```
/** CPPI Low level Driver return and Error Codes */

/** CPPI successful return code */
#define CPPI_SOK                            0
/** CPPI Error Base */
#define CPPI_LLD_EBASE                      (-128)
/** CPPI CPDMA not yet initialized */
#define CPPI_CPDMA_NOT_INITIALIZED          CPPI_LLD_EBASE-1
/** CPPI invalid parameter */
#define CPPI_INVALID_PARAM                  CPPI_LLD_EBASE-2
/** CPPI Rx/Tx channel not yet enabled */
#define CPPI_CHANNEL_NOT_OPEN               CPPI_LLD_EBASE-3
/** CPPI Rx flow not yet enabled */
#define CPPI_FLOW_NOT_OPEN                  CPPI_LLD_EBASE-4
/** CPPI Tx channels are still open. All Tx channels should be closed  before calling
Cppi_close */
#define CPPI_TX_CHANNELS_NOT_CLOSED         CPPI_LLD_EBASE-5
/** CPPI Rx channels are still open. All Rx channels should be closed before calling
Cppi_close */
#define CPPI_RX_CHANNELS_NOT_CLOSED         CPPI_LLD_EBASE-6
/** CPPI Rx flows are still open. All Rx flows should be closed before calling Cppi_close
*/
#define CPPI_RX_FLOWS_NOT_CLOSED            CPPI_LLD_EBASE-7
/** Queue Manager subsystem memory region not enabled */
#define CPPI_QMSS_MEMREGION_NOT_INITIALIZED CPPI_LLD_EBASE-8
/** Queue open error */
#define CPPI_QUEUE_OPEN_ERROR               CPPI_LLD_EBASE-9
/** CPPI extended packet information block not present in descriptor */
#define CPPI_EPIB_NOT_PRESENT               CPPI_LLD_EBASE-10
/** CPPI protocol specific data not present in descriptor */
#define CPPI_PSDATA_NOT_PRESENT             CPPI_LLD_EBASE-11
/** CPPI CPDMA instances are still open. All CPDMA instances should be closed  before
calling cppi_exit */
#define CPPI_CPDMA_NOT_CLOSED               CPPI_LLD_EBASE-12
```

Table 19 CPPI Error Codes

QMSS Error Codes:

```
/** QMSS Low level Driver return and Error Codes */
/** QMSS successful return code */
#define QMSS_SOK                            0
/** QMSS Error Base */
#define QMSS_LLD_EBASE                      (-128)
/** QMSS LLD invalid parameter */
#define QMSS_INVALID_PARAM                  QMSS_LLD_EBASE-1
/** QMSS LLD not initialized */
#define QMSS_NOT_INITIALIZED                QMSS_LLD_EBASE-2
/** QMSS LLD queue open error */
#define QMSS_QUEUE_OPEN_ERROR               QMSS_LLD_EBASE-3
/** QMSS memory region not initialized */
#define QMSS_MEMREGION_NOT_INITIALIZED      QMSS_LLD_EBASE-4
/** QMSS memory region already initialized */
#define QMSS_MEMREGION_ALREADY_INITIALIZED          QMSS_LLD_EBASE-5
/** QMSS memory region invalid parameter */
#define QMSS_MEMREGION_INVALID_PARAM                QMSS_LLD_EBASE-6
/** QMSS maximum number of allowed descriptor are already configured */
#define QMSS_MAX_DESCRIPTORS_CONFIGURED             QMSS_LLD_EBASE-7
/** QMSS Specified memory region index is invalid or no memory regions are available */
#define QMSS_MEMREGION_INVALID_INDEX                QMSS_LLD_EBASE-8
/** QMSS memory region overlap */
#define QMSS_MEMREGION_OVERLAP                      QMSS_LLD_EBASE-9
/** QMSS PDSP firmware download failure */
#define QMSS_FIRMWARE_DOWNLOAD_FAILED               QMSS_LLD_EBASE-10
```

Table 20 QMSS Error Codes

QMSS Accumulator Return Codes:

```
/** QMSS accumulator return and Error Codes */
/** QMSS accumulator idle return code */
#define QMSS_ACC_IDLE                               0
/** QMSS accumulator successful return code */
#define QMSS_ACC_SOK                                1
/** QMSS accumulator invalid command return code */
#define QMSS_ACC_INVALID_COMMAND                    2
/** QMSS accumulator invalid channel return code */
#define QMSS_ACC_INVALID_CHANNEL                    3
/** QMSS accumulator channel not active return code */
#define QMSS_ACC_CHANNEL_NOT_ACTIVE                 4
/** QMSS accumulator channel already active */
#define QMSS_ACC_CHANNEL_ALREADY_ACTIVE             5
/** QMSS accumulator invalid queue number */
#define QMSS_ACC_INVALID_QUEUE_NUMBER               6
```

Table 21 QMSS Accumulator Return Codes