Application Report
28 April 2020

Migrating Applications from NAVSS UDMA to DMSS using TI-RTOS

Embedded Processing, Processors Business Unit

ABSTRACT

AM64x SoC introduces an updated DMA subsystem – DMSS, which is a lighter-weight version of NAVSS UDMA. UDMA is the DMA engine used to do direct memory access (DMA) between different peripherals like McASP, SPI, UART and memory (DDR, L2, L3, MSMC) without CPU intervention in next generation TI SoCs like AM6x. DMSS is a new lighter version of DMA architecture and SW applications written for NAVSS UDMA need to be adapted to use the already existing UDMA API's. This application note is intended to help SW developers migrate their TI-RTOS based SW applications, device drivers from NAVSS UDMA based systems to DMSS based systems. The application note explains the differences between NAVSS UDMA and DMSS from a HW perspective. It then compares the SW API's provided by TI in Processor SDK RTOS for NAVSS UDMA / DMSS.

NOTE: It is expected that the reader knows about UDMA. For EDMA to UDMA migration refer to the application note 'Migrating Applications from EDMA to UDMA using TI-RTOS'.

IMPORANT NOTE: This application note provides a simplified description and comparison of HW/SW features so that SW users can effectively migrate typical applications from NAVSS UDMA to DMSS. For SoC specific details users should refer to SoC technical reference manual (TRM). The type of DMA EDMA/UDMA/DMSS varies depending on the SoC. SoC TRM will provide the details of the DMA infrastructure used.

Table of Contents

Figures

Tables

1 Introduction to DMSS

1.1 DMSS Overview

The primary goal of the Data Movement Subsystem (DMSS) is to ensure that data can be efficiently transferred from a producer to a consumer so that the real time requirements of the system can be met. The Data Movement architecture aims to facilitate Direct Memory Access (DMA) and to provide a consistent Application Programming Interface (API) to the host software. Data movement tasks are commonly offloaded from the host processor to peripheral hardware to increase system performance. Significant performance gains may result from careful design of the interface between the host software and the underlying acceleration hardware. In networking applications packet transmission and reception are critical tasks. In general purpose compute, ping pong buffer pre-fetch and store are critical tasks as are general misaligned block copy operations.

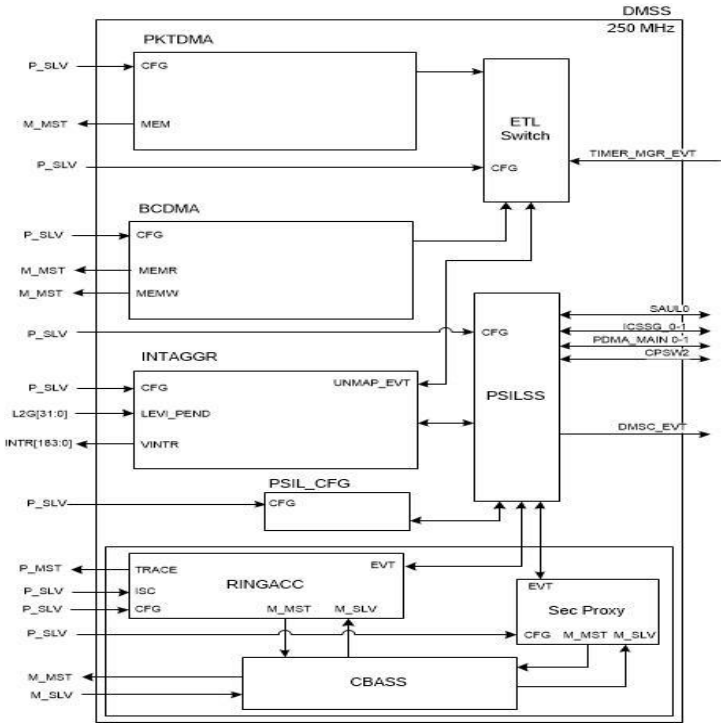

Figure 1. AM64x DMSS Top Level Block Diagram

The block diagram provides a high level picture of not only the 2 different interconnect fabrics but also some key standard data movement components that have been defined and placed in the various parts of the low cost compliant SoC. The following sections will provide a high level overview of Packet DMA (PKTDMA) and Block Copy DMA (BCDMA) which are the two instances of the DMSS specification serving different use cases as described below:

1.1.1 Packet DMA (PKTDMA)

The PKTDMA is intended to perform similar functions as the packet oriented DMA. The PKTDMA module supports the transmission and reception of various packet types. The PKTDMA is architected to facilitate the segmentation and reassembly of DMA data structure compliant packets to/from smaller data blocks that are natively compatible with the specific requirements of each connected peripheral. Multiple TX and RX channels are provided within the DMA which allow multiple segmentation or reassembly operations to be ongoing. The DMA controller maintains state information for each of the channels which allows packet segmentation and reassembly operations to be time division multiplexed between channels in order to share the underlying DMA hardware. An internal DMA scheduler is used to control the ordering and rate at which this multiplexing occurs for Transmit operations. The ordering and rate of Receive operations is indirectly controlled by the order in which blocks are pushed into the DMA on the RX PSI-L interface.

A block diagram of the PKTDMA Controller is shown below:

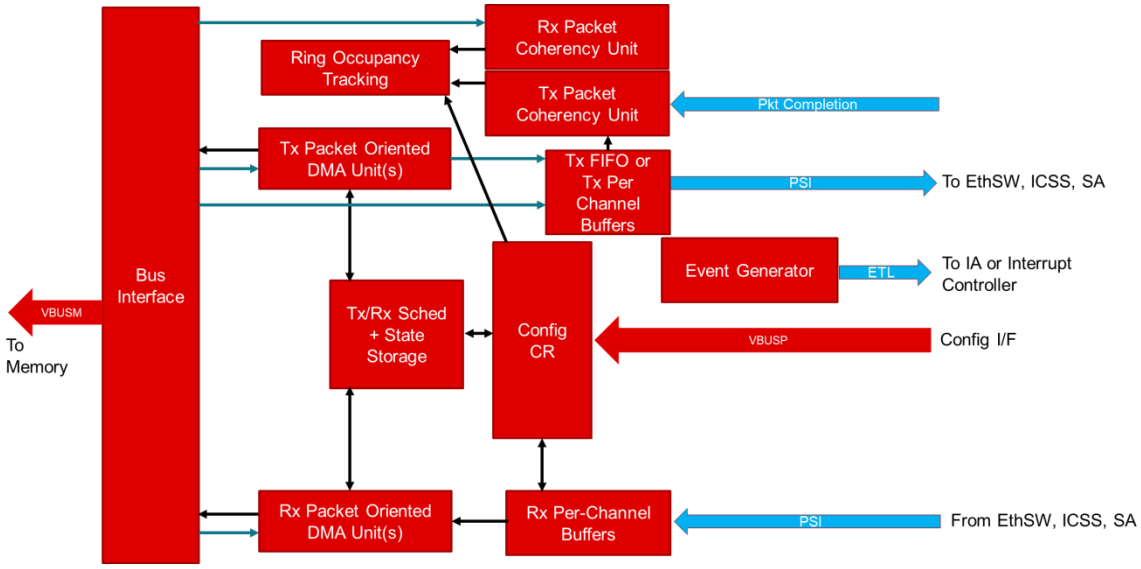

Figure 2. PKTDMA Block Diagram

1.1.2 Block Copy DMA (BCDMA)

The Block Copy DMA is intended to perform similar functions as the EDMA or the UDMA-P/UTC. The BCDMA module moves data from a memory mapped source address set to a corresponding memory mapped address set. The BCDMA maintains state information for each of the channels which allows data copy operations to be time division multiplexed between channels in order to share the underlying DMA hardware. An internal DMA scheduler is used to control the ordering and rate at which this multiplexing occurs.

A block diagram of the Block Copy DMA Controller is shown below:

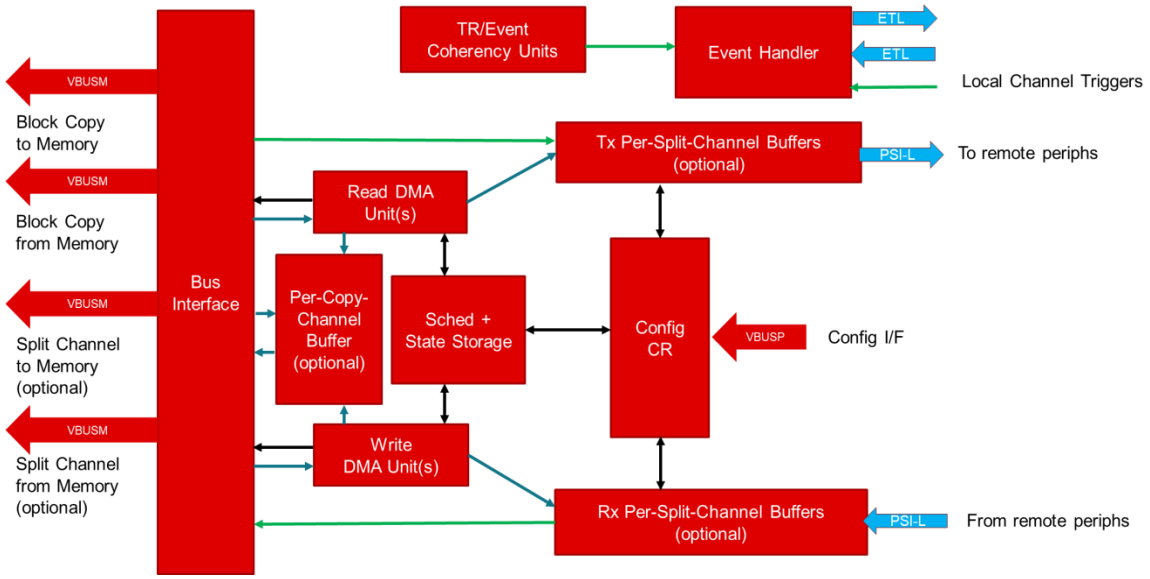

Figure 3. Block Copy DMA Block Diagram

1.2 Comparison of BCDMA and PKTDMA

This following table gives an overview of the comparison between BCDMA and PKTDMA, the two instances of the DMSS specification.

Table 1. Comparison between BCDMA and PKTDMA

	BCDMA	PKTDMA
Descriptors	The Block Copy DMA architecture provides for a single descriptor type: • Transfer Request Packet Descriptor (TRPD) All applications using TRPD must use BCDMA instance.	PKTDMA architecture provides for 2 basic types of descriptors: • Host Packet Descriptor (HPD) • Host Buffer Descriptor (HBD) All applications using HPD/HBD must use PKTDMA instance.
Channels	• Block Copy Channels • Split TR TX Channels • Split TR RX Channels *(Refer TRM for SoC specific details)*	• TX Channels • RX Channels *(See Section 2.2.1 for more details)*
Flows / Rings	Each flow is tied to a channel. There are no free / no extra flows. Only default flow is available.	No free flows. Extra flow(s) tied to channels. *(See Section 2.2.2 for more details)*

2 Comparison of NAVSS UDMA and DMSS

This section provides a top level overview of the comparison between NAVSS UDMA & DMSS and discuss in detail about the changes in PKTDMA compared to UDMA-P. It also discuss about the endpoint spanning restrictions in BCDMA & PKTDMA.

2.1 Top Level Overview

The following table gives an overview of hardware changes in DMSS compared to NAVSS UDMA, corresponding changes in UDMA LLD Driver and its impact to applications using UDMA SW API's.

Table 2. Comparison between DMSS and NAVSS UDMA

HW Changes	SW Changes	Impact on Applications based on how UDMA LLD abstracts the changes	Application Change Required
Breakout DMA into Block Copy DMA and Packet DMA	Two Instances in DMSS: BCDMA & PKTDMA	Low Impact. Instance Id must be modified to `UDMA_INST_ID_BCDMA_0/ UDMA_INST_ID_PKTDMA_0`	YES
Separate (free) ring accelerator is removed and support for only Exposed RING mode	Ring is tied to the channel used. Other ring modes not supported	Ring cannot be used for other general purposes.Other ring modes such as `MESSAGE, CREDENTIAL, QM` can't be used. API's return error.	YES – *if `MESSAGE/QM/ CREDENTIAL` Ring mode used* NO – *if Exposed `RING` mode used.*
Changes in PKTDMA compared to UDMA-P. *(See Section 2.2)*	Handled by UDMA driver or other drivers supported by PDK. *(See Section 2.3)*	No Impact / Low Impact. *(See Section 3.1.3)*	*Depends on Driver used.* YES – *if UDMA Driver alone is used.*
No Ring monitors	Ring Monitor API's not supported	Ring Monitor API's can't be used. API's return error	YES – *if Ring Monitor API's are used*
No non-secure Proxy	Proxy API's are not supported	Proxy API's can't be used. API's return error	YES – *if Proxy API's are used*
Single ring for both forward direction (FQ ring) and reverse direction(CQ ring)	Completion Queue (CQ) Ring points to Free Queue (FQ) Ring itself	No Impact	Optional -*but recommended to remove CQ ring memory from application to save memory*
No teardown descriptor is written back if a channel is disabled while DMA is active	`TEARDOWN_PACKET` event bypassed	No Impact on Teardown Event; `Udma_chDequeueTdResponse` API can't be used and will return error	Optional - *but recommended to remove Teardown ring memory from application to save memory*

Dedicated Block Copy Channels in BCDMA	TX Channel API's can be used for Block Copy Channels too. RX Channel API's bypassed in Block Copy mode.	No Impact	NO
DMA channel events moved into an OES table within the Interrupt Aggregator	Channel events from interrupt aggregator are handled in the same way as ring events.	No Impact	NO
No Interrupt Router as part of navigator	IR config is bypassed and IA events directly mapped to core interrupts	No Impact	NO

For detailed information Ref 'Section 3 - UDMA SW API for Applications'.

2.2 Changes in PKTDMA compared to UDMA-P

This section discuss in detail about the changes in PKTDMA compared to UDMA-P. This covers the changes in channels, flows/rings, descriptors and other major changes.

2.2.1 Channels in PKTDMA

In case of AM64x, PKTDMA instance has 42 TX channels and 29 RX channels. Among this some channels are hardwired to specific peripherals. i.e., there is no dynamic thread mapping.

The 42 TX Channels are grouped as given below:

- 16 Peripheral TX Channels
- 8 CPSW TX Channels
- 2 SAUL TX Channels
- 8 ICSSG 0 TX Channels
- 8 ICSSG 1 TX Channels

NOTE: The TX channels reserved for CPSW / SAUL / ICSSG 0 / ICSSG 1 will be referred as 'MAPPED TX' channels in UDMA LLD.

The 29 RX Channels are grouped as given below:

- 16 Peripheral RX Channels
- 1 CPSW RX Channels
- 4 SAUL RX Channels
- 4 ICSSG 0 RX Channels
- 4 ICSSG 1 RX Channels

NOTE: The RX channels reserved for CPSW / SAUL / ICSSG 0 / ICSSG 1 will be referred as 'MAPPED RX' channels in UDMA LLD.

NOTE: The above configurations may be different depending on the SoC .Refer TRM for details.

2.2.2 Flows / Rings in PKTDMA

DMSS eliminates all ring steering. The rings are now dedicated for a single TX or RX flow.

Henceforth the Flows specified in the DMSS specs are equivalent to Rings in UDMA LLD.

DMSS introduced the new feature 'TX-FLOW' for TX channels (similar to RX-FLOW for RX channels). It allows separate TX exposed rings to be multiplexed onto one TX channel. The multiplexing is done on packet / entire block copy work units. It does not allow interleaving of individual transfers within the DMA operation.

Compared to UDMA-P, PKTDMA has a reduced RX flow feature set. There is no multiple 'free pool' selection based on incoming packet size and no destination ring override. Destination ring is hard wired to the flow effectively. RX flows are hard-wired to the DMA channel. There is no longer a generic pool of flows that can be assigned ad hoc to any channel.

The following diagrams show how the flow(s) are tied to different channels in PKTMDA.

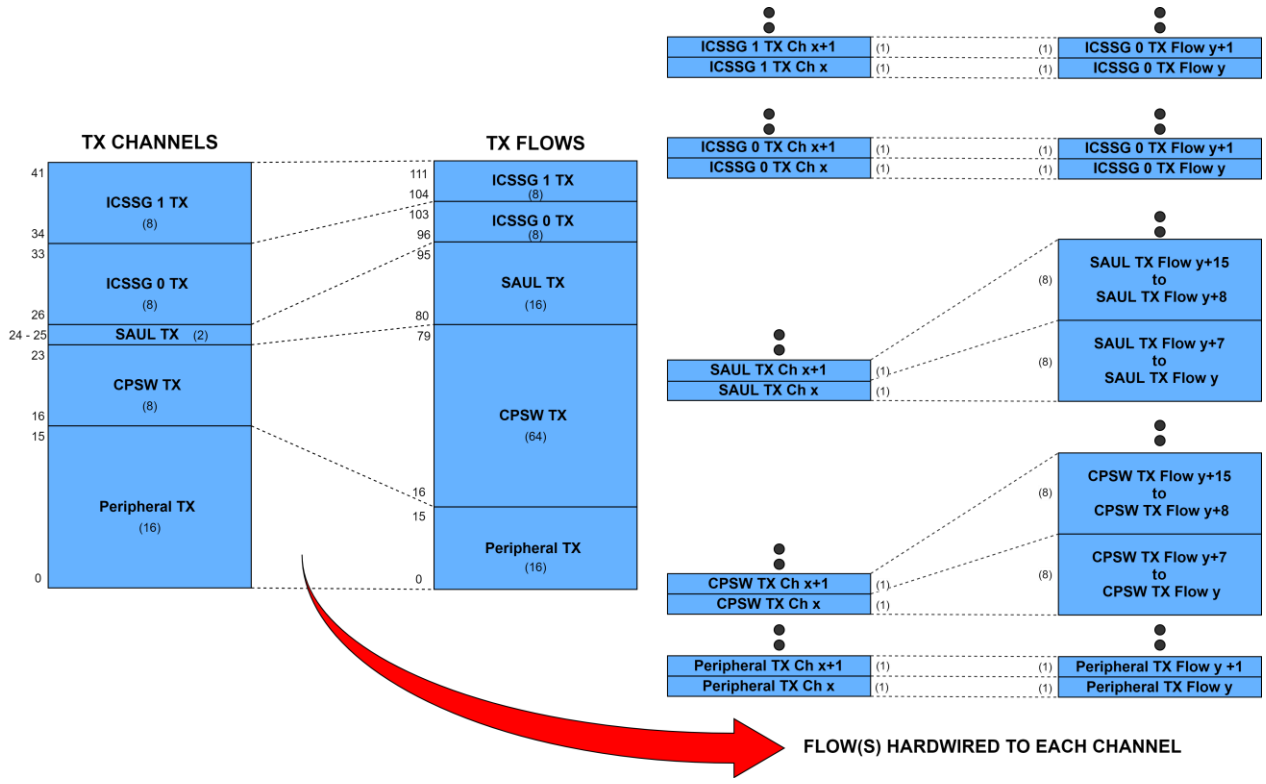

Figure 4. TX Channels and Flows in PKTMDA

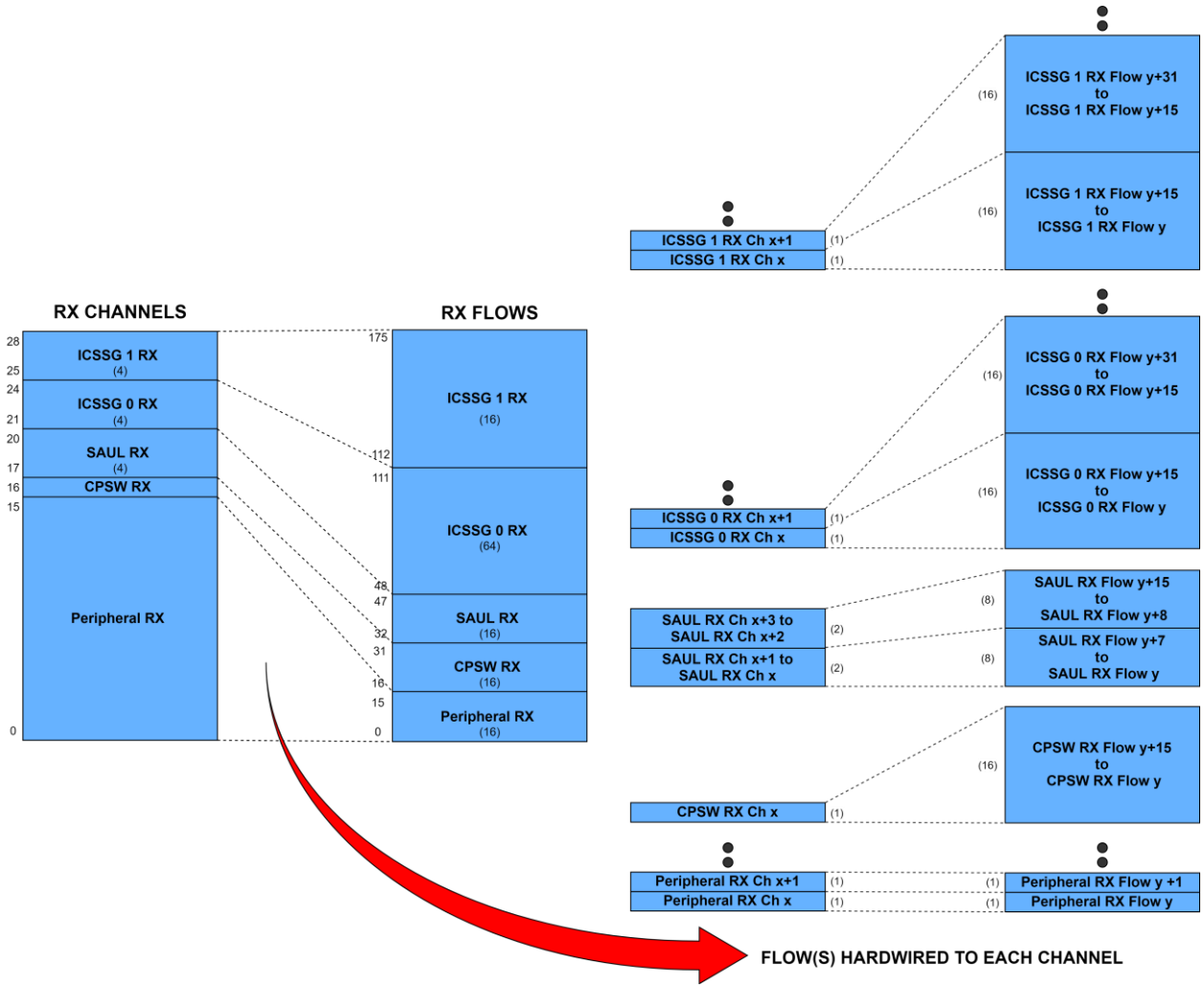

Figure 5. RX Channels and Flows in PKTDMA

2.2.3 Changes in PKTDMA Descriptors

The differences in descriptors of PKTDMA compared to UDMA-P include:

- Descriptor layout changes :
  - Fields no longer used are marked as reserved.
  - Protocol specific region must be in descriptor.
  - At start of packet option is no longer supported and the descriptor bit controlling this is now a reserved bit.
- No monolithic descriptor mode (only host mode for PKTDMA)
- No prefetching of descriptors. The control fetches are in-line with data transfers.

- No RX chaining of descriptors, but scatter is still supported.
- Less flexibility in packet return mode options.
    - Chained descriptors will be freed still chained.
    - No option to return descriptors to a specific ring.
    - Early return bit no longer used.
- PS words in descriptor only.

2.2.4 Other changes in PKTDMA

The key differences in PKTDMA compared to UDMA-P, other than that discussed above are:

- No automatic garbage collection on TX. There is no early completion or buffer return.
- Packet is received into pre-linked buffer chain provided by Host.
- No pass through of SECURE / PRIV/ PRIVID credentials with each packet. Each flow is locked to a single SECURE / PRIV/ PRIVID.
- Hardcodes output event indices.

2.3 Using DMSS with Processor SDK RTOS Drivers

If the usage of UDMA is intended for peripherals whose drivers are supported by PDK in Processor SDK RTOS, the configuration/setup is limited to the initialization of UDMA driver. In this case the change required to migrate to DMSS is to update the instance id before initialization.

For example in Packet DMA mode:

```
#include <ti/drv/udma/udma.h>

Udma_InitPrms   initPrms;
uint32_t        instId;

instId = UDMA_INST_ID_PKTDMA_0;

UdmaInitPrms_init(instId, &initPrms)

struct Udma_DrvObj      gUdmaDrvObj; /* MUST be global */
retVal = Udma_init(&gUdmaDrvObj, &initPrms);
```

This is because PDK drivers already configures and uses UDMA internally. The application would only need to perform the UDMA driver initialization. Please refer to the driver unit level tests and/or examples present in Processor SDK RTOS for sample usage.

2.4 Enpoint Spanning Restrictions in BCDMA & PKTDMA

Spanning endpoints means logical banks of the same memory or even between different memories. This section discusses in detail the restriction in spanning endpoints on BCDMA & PKTDMA.

2.4.1 BCDMA Endpoint Spanning Restrictions

The endpoint spanning restrictions for BCDMA are summarized below:

1. Descriptors:
    a) Descriptor reads for BCDMA cannot span multiple endpoints.
    b) Individual Descriptors can be at different endpoints.
    c) Descriptors and Data buffers can be at different endpoint.
2. BCDMA – block copy reads cannot cross end points. This is for a single TR regardless of dimensioning. The write phase of the block copy does not have this restriction.
3. All writes to memory by BCDMA (RX transfers) have no restrictions.
4. All reads to memory by BCDMA (TX transfers) cannot cross end points (straddle multiple OCSRAM banks).

2.4.2 PKTDMA Endpoint Spanning Restrictions

The endpoint spanning restrictions for PKTDMA are summarized below:

1. Descriptors:
    a) Descriptor reads for PKTDMA cannot span multiple endpoints.
    b) Individual Descriptors can be at different endpoints.
    c) Descriptors and Data buffers can be at different endpoint.
2. All writes to memory by PKTDMA (RX transfers) have no restrictions.
3. All reads to memory by PKTDMA (TX transfers) have no restrictions.
4. Chained buffers in the same packet can be at different endpoints
5. A buffer can span endpoints

3 UDMA SW API for Applications

This section provides an overview of the changes in UDMA SW API's, details on API's that are unsupported by DMSS and the behavioral changes in UDMA SW API's when used for DMSS as compared to that for NAVSS UDMA.

3.1 UDMA SW API changes

This section provides details about the changes in using UDMA SW API's for DMSS as compared to that for NAVSS UDMA.

3.1.1 BCDMA and PKTDMA instances

One of the key differences in DMSS as compared to NAVSS UDMA is the instances. The two instances that are present in DMSS are Packet DMA (PKTDMA) and Block Copy DMA (BCDMA). In contrast to this, the two instances in NAVSS UDMA were the MAIN NAVSS and the MCU NAVSS.

The Instance ID's in DMSS are:

1) `UDMA_INST_ID_BCDMA_0  /* For Block Copy DMA */`
2) `UDMA_INST_ID_PKTDMA_0 /* For Packet DMA */`

This change does impact UDMA SW API's. The application must be updated with the correct instance ID for DMSS.

```
instId = UDMA_INST_ID_BCDMA_0; /* or UDMA_INST_ID_PKTDMA_0*/
```

3.1.2 Unsupported Ring Modes

In DMSS the separate ring accelerator for DMA is removed and support for just Exposed RING mode is moved in with DMA. DMSS doesn't support other ring modes such as `MESSAGE`, `CREDENTIAL` and `QM`.

As a result of this, applications using DMSS must use only Exposed `RING` mode. If other ring modes are passed, the driver will return Error.

3.1.3 Changes in PKTDMA compared to UDMA-P

In PKTDMA there are mapped channels which are hardwired to specific peripherals and also the ring(s) are tied to each channel. The UDMA LLD has introduced new channel types and an additional parameter mapped group to configure and use these channels and rings with the already existing UDMA API's. This section describe in detail about the new channel types, mapped groups and how to use the UDMA channel / ring API's with minimal changes.

NOTE: The PKTDMA instance is majorly used by peripherals, whose drivers are supported by PDK in Processor SDK RTOS. So the changes in PKTDMA compared to UDMA-P may not impact the applications. For this, Ref section 2.3 - Using DMSS with Processor SDK RTOS Drivers.

3.1.3.1 New Channel Types

The new channel types that are introduced to configure and use the mapped channels are:

1. `UDMA_CH_TYPE_TX_MAPPED`
2. `UDMA_CH_TYPE_RX_MAPPED`

3.1.3.2 Mapped Groups

Mapped group is a new parameter that is introduced to UDMA LLD. The different types of mapped groups in DMSS are as follows:

1. UDMA_MAPPED_TX_GROUP_CPSW
2. UDMA_MAPPED_TX_GROUP_SAUL
3. UDMA_MAPPED_TX_GROUP_ICSSG_0
4. UDMA_MAPPED_TX_GROUP_ ICSSG_1
5. UDMA_MAPPED_RX_GROUP_CPSW
6. UDMA_MAPPED_RX_GROUP_SAUL
7. UDMA_MAPPED_RX_GROUP_ICSSG_0
8. UDMA_MAPPED_RX_GROUP_ ICSSG_1
9. UDMA_MAPPED_GROUP_INVALID

The new parameter in Udma_ChPrms named mappedChGrp and that in Udma_RingPrms named mappedRingGrp can be assigned with these values. By default the param mappedChGrp / mappedRingGrp is initialized to UDMA_MAPPED_GROUP_INVALID.

3.1.3.3 Usage of UDMA Channel / Ring API's

This section describes about the usage of UDMA API's to configure and use the mapped channels and its corresponding tied rings. With the addition of new channel types and additional parameters in Udma_ChPrms and Udma_RingPrms, these parameters should be properly set.

Before calling the UDMA channel API Udma_chOpen the following parameters in Udma_ChPrms should be set based on the use-case.

1. chType = UDMA_CH_TYPE_TX_MAPPED (for Mapped TX Channels)
   = UDMA_CH_TYPE_RX_MAPPED (for Mapped RX Channels)
2. mappedChGrp = UDMA_MAPPED_<TX/RX>_GROUP_<group> ,
   where group = CPSW / SAUL / ICSSG _0 / ICSGG_1

NOTE: When ring memory is provided while calling Udma_chOpen, then the UDMA driver allocates and configures the mapped channel and its corresponding tied ring. In this case, the UDMA driver itself will populate the new params in Udma_ringPrms.

For example to allocate and configure a CPSW TX channel and ring:

```
struct Udma_DrvObj      gUdmaDrvObj;
struct Udma_ChObj       gUdmaTxChObj;
```

```
uint32_t       chType;
Udma_ChPrms    chPrms;

chType = UDMA_CH_TYPE_TX_MAPPED;
UdmaChPrms_init(&chPrms, chType);
chPrms.mappedChGrp          = UDMA_MAPPED_TX_GROUP_CPSW;
chPrms.peerChNum            = UDMA_PSIL_CH_CPSW2_TX;
chPrms.fqRingPrms.ringMem   = &gTxFqRingMem[0U];

/* Open channel */
retVal = Udma_chOpen(&gUdmaDrvObj, &gUdmaTxChObj, chType, &chPrms);
```

If `Udma_ringAlloc` API is explicitly called to allocate and configure a ring after configuring the channel, then the following parameters in `Udma_RingPrms` should be properly set beforehand.

1. `mappedRingGrp` = `UDMA_MAPPED_<TX/RX>_GROUP_<group>` ,
   where group = CPSW / SAUL / ICSSG _0 / ICSGG_1
2. `mappedChNum` = Channel number of the already allocated mapped channel.

This is used to allocate the corresponding mapped ring for the particular channel.

For example to allocate and configure a ring for CPSW TX channel,
(where `txChHandle->txChNum` refers to the already allocated channel number)

```
struct Udma_RingObj  gUdmaRingObj;

Udma_RingPrms  fqRingPrms;
UdmaRingPrms_init(&fqRingPrms);

fqRingPrms.mappedRingGrp = UDMA_MAPPED_TX_GROUP_CPSW;
fqRingPrms.mappedChNum   = txChHandle->txChNum;

/* Allocate Ring */
retVal = Udma_ringAlloc(&gUdmaDrvObj,
                        &gUdmaRingObj,
                         UDMA_RING_ANY,
                         &fqRingPrms);
```

3.1.4 Absence of Free General Purpose Ring – No Completion Ring

In DMSS there is only a single ring (instead of 2) for both forward and reverse direction. So the common ring ID itself must be programmed in the TR Descriptor instead of the completion ring ID. UDMA SW API internally handles this by pointing the same ring ID to completion ring ID. That is, since there is no separate completion queue, completion queue ring handle points to forward queue ring itself. So programming the TR Descriptor either way will work fine.

Hence Completion queue ring params is not used, but even if the application sets this it will be ignored. But it's not required to be set. Here completion queue ring object is also not used.

This change doesn't impact the usage of UDMA SW API's, since it's handled inside the driver itself. So the existing application for NAVSS UDMA will work for DMSS without any issues. Hence the following lines of codes which allocate memory for completion ring are optional for DMSS (no effect):

```
static uint8_t gTxCompRingMem[UDMA_TEST_APP_RING_MEM_SIZE]
__attribute__((aligned(UDMA_CACHELINE_ALIGNMENT)));

chPrms.cqRingPrms.ringMem   = &gTxCompRingMem[0U];
chPrms.cqRingPrms.elemCnt   = UDMA_TEST_APP_RING_ENTRIES;
```

3.1.5 Teardown unsupported and absence of Teardown Ring

In DMSS, there is no teardown ring and no teardown descriptor is written back if a channel is disabled while DMA is active. So the `TEARDOWN_PACKET` event is not supported in DMSS.

Hence Teardown completion queue ring params is not used, but even if the application sets this it will be ignored. But it's not required to be set. Teardown completion queue ring object is also not used.

And there is no need to register teardown ring completion callback. But even if the application tries to register the teardown ring completion callback, the UDMA Driver internally bypass registering the `TEARDOWN_PACKET` event. This is to ensure backward compatibility with NAVSS UDMA.

As a result, this change doesn't impact the usage of UDMA SW API's. So the existing application for NAVSS UDMA will work for DMSS without any issues. Hence the following lines of codes are optional for DMSS (no effect):

1.
```
static uint8_t gTxTdCompRingMem[UDMA_TEST_APP_RING_MEM_SIZE]
__attribute__((aligned(UDMA_CACHELINE_ALIGNMENT)));

chPrms.tdCqRingPrms.ringMem = &gTxTdCompRingMem[0U];
chPrms.tdCqRingPrms.elemCnt = UDMA_TEST_APP_RING_ENTRIES;
```

2.
```
struct Udma_EventObj    gUdmaTdCqEventObj; /* MUST be global */
```

3.
```
/* Register teardown ring completion callback */
eventHandle = &gUdmaTdCqEventObj;
UdmaEventPrms_init(&eventPrms);
eventPrms.eventType         = UDMA_EVENT_TYPE_TEARDOWN_PACKET;
eventPrms.eventMode         = UDMA_EVENT_MODE_SHARED;
eventPrms.chHandle          = chHandle;
eventPrms.masterEventHandle = &gUdmaCqEventObj;
eventPrms.eventCb           = &App_udmaEventTdCb;
retVal = Udma_eventRegister(drvHandle, eventHandle, &eventPrms);
```

4.
```
eventHandle = &gUdmaTdCqEventObj;
retVal += Udma_eventUnRegister(eventHandle);
```

5.
```
static void App_udmaEventTdCb(Udma_EventHandle eventHandle,
                              uint32_t eventType,
                              void *appData)
{
   CSL_UdmapTdResponse tdResp;

   if(UDMA_EVENT_TYPE_TEARDOWN_PACKET == eventType)
   /* Response received in Teardown completion queue */
   Udma_chDequeueTdResponse(&gUdmaChObj, &tdResp);

   return;
}
```

3.1.6 Dedicated Block Copy Channels

DMSS introduced dedicated Block Copy channels in BCDMA apart from the Split TR TX and RX Channels. The main impact of this in UDMA SW API is that, for Block copy mode there is no need to configure RX channel (which was implicitly paired to TX channel in case of NAVSS UDMA for block copy mode). But even if the application tries to configure RX channel, the UDMA Driver internally bypass configuring the RX channel for block copy mode. i.e., the `Udma_chConfigRx()` API returns gracefully, without doing anything. This is to ensure backward compatibility with NAVSS UDMA.

UDMA Driver is designed such that, the new Block Copy channels uses the existing TX flavors of UDMA SW API's itself. i.e., when channel type is `UDMA_CH_TYPE_TR_BLK_COPY`, the UDMA TX channel API's make use of the Block Copy channels.

For example,

- `UdmaChTxPrms_init()` API, initialize the params for block copy channel
- `Udma_chConfigTx()` API, configures the block copy channel. (etc.)

As a result, this change doesn't impact the usage of UDMA SW API's. So the existing applications for NAVSS UDMA will work for DMSS without any issues. Hence the following lines of codes are optional for DMSS (no effect) [only for Block Copy Mode]

```
Udma_ChRxPrms       rxPrms;

/* Config RX channel - which is implicitly paired to TX channel in
block copy mode */
UdmaChRxPrms_init(&rxPrms, chType);
retVal = Udma_chConfigRx(chHandle, &rxPrms);
```

3.2 UDMA SW API Beaviour Changes

This section describes about the behavioral changes in UDMA SW API when used for DMSS, as compared to that for NAVSS UDMA.

3.2.1 Absence of Free General Purpose Ring – No completion Ring

Due to the absence of Completion Ring, there are some behavioral changes for the following API's when used for DMSS:

1. `Udma_chGetCqRingHandle()` - Returns the default common ring handle of the channel, since there is no separate completion queue ring.

2. `Udma_chGetCqRingNum()` - Returns the default common ring number to be programmed in descriptor, since there is no separate completion queue ring.

3.2.2 Teardown unsupported

In DMSS, since there is no need to register `TEARDOWN_PACKET` event and dequeue the teardown response, the following API's have some behavioral changes for DMSS:

1. `Udma_eventRegister()` *for event Type* = `UDMA_EVENT_TYPE_TEARDOWN_PACKET`

   - won't do any operation and will return gracefully without returning any error

2. `Udma_eventUnRegister()` *for event Type* = `UDMA_EVENT_TYPE_TEARDOWN_PACKET`

   - won't do any operation and will return gracefully without returning any error

3.2.3 Dedicated Block Copy Channels

Due to the presence of dedicated Block Copy Channels in BCDMA, there are some behavioral changes for the following API's when used for DMSS:

1. `Udma_chConfigRx()` *for event Type* = `UDMA_CH_FLAG_BLK_COPY`

   - won't do any operation and will return gracefully without returning any error

2. `Udma_chConfigTx()` *for event Type* = `UDMA_CH_FLAG_BLK_COPY`

   - configure Block Copy Channel instead to TX Channel

3.2.4 Channel Events moved to Interrupt Aggregator

In DMSS, all DMA channel events have been moved into an OES table within the IA peripheral. As a result, these channel events must be handled instead of the ring events.

In case of DMSS, while registering these event using `Udma_eventRegister()`, the UDMA Driver configures the channel events from IA instead of the ring events. It also unregisters these events during `Udma_eventUnRegister()` in a similar fashion. So the channel events from interrupt aggregator are handled the same way as ring events.

3.2.5 No Interrupt Router

In DMSS, there is no Interrupt Router as part of navigator. i.e., The DMSS IA has no IR's between itself and the destination cores. As a result there is no need to configure the IR's in case of DMSS.

In case of DMSS, UDMA Driver bypasses the configuration of IR's during `Udma_init()`, and the IA events are directly mapped to the corresponding core interrupts.

3.3 Unsupported UDMA SW API's for DMSS

Some of the UDMA SW API's is not supported by DMSS, due to unsupported ring modes, absence of various modules like Ring Monitor, Proxy, CLEC etc. These API's if called for DMSS, will return Error.

3.3.1 Unsupported Ring Modes

Since DMSS doesn't support ring modes such as `MESSAGE`, `CREDENTIAL` and `QM`, `Udma_chOpen()` will return error when the following modes are used for `Udma_RingPrms->mode`

1. `TISCI_MSG_VALUE_RM_RING_MODE_MESSAGE`
2. `TISCI_MSG_VALUE_RM_RING_MODE_CREDENTIALS`
3. `TISCI_MSG_VALUE_RM_RING_MODE_QM`

3.3.2 Absence of Ring Monitors

In DMSS, since there are no Ring Monitors present, the following ring monitor API's are unsupported for DMSS. If these API's are called, the driver will return error.

- `Udma_ringMonAlloc()`
- `Udma_ringMonFree()`
- `Udma_ringMonConfig()`
- `Udma_ringMonGetData()`
- `Udma_ringMonGetNum()` – will always return `UDMA_RING_MON_INVALID`

Even the event related to Ring Monitor is also not supported due to its absence and the following will return error in case of DMSS:

- `Udma_eventRegister()` *for event Type* = `UDMA_EVENT_TYPE_RING_MON`
- `Udma_eventUnRegister()` *for event Type* = `UDMA_EVENT_TYPE_RING_MON`

3.3.3 Absence of Proxy

Due to the absence Proxy in DMSS, the following proxy API's are unsupported for DMSS. If these API's are called, the driver will return error.

- `Udma_proxyAlloc()`
- `Udma_proxyFree()`
- `Udma_proxyConfig()`

3.3.4 Absence of Free General Purpose Ring – No completion Ring

For BCDMA instance, the UDMA ring allocation API, if passed with the parameter `UDMA_RING_ANY` will try to will allocate from free ring pool. In DMSS, there are no free general purpose rings. As a result, in case of BCDMA instance, `Udma_ringAlloc()` if used with `ringNum = UDMA_RING_ANY` will return error.

3.3.5 Teradown unsupported

In DMSS, No teardown descriptor is written back if a channel is disabled while DMA is active. Due to this, the UDMA SW API to dequeue teardown response `Udma_chDequeueTdResponse()` is unsupported and will return error if used.

4 Summary

This application note gave an overview of NAVSS UDMA and DMSS from a HW and SW perspective. It described and compared the programming model, SW API's of UDMA. Migrating applications from NAVSS UDMA to DMSS involves having a brief understanding the HW characteristics of DMSS as compared to NAVSS UDMA. An application writer should identify the sequence of UDMA SW API's used in his application during the various phases of driver usage like DMA driver initialization, DMA channel open, DMA transfer setup, DMA transfer trigger and wait. Users can then make the required changes to use the existing UDMA SW API's for DMSS as described in this app note. Finally refer to TRMs and API guide for more details about UDMA programming.

5 References

[1] AM6x Technical Reference Manual – NavSS / DMA controller chapter

[2] AM6x Processor SDK RTOS User Guide – UDMA LLD section

6 Revision History

Date	Revision	Author	Changes
28 Apr 2020	1.00	Don Dominic	First draft
18 May 2020	1.01	Don Dominic	Added updates for PKTDMA
27 May 2020	1.02	Don Dominic	Addressed review comments