Application Report
1 Aug 2018

Migrating Applications from EDMA to UDMA using TI-RTOS

Embedded Processing, Processors Business Unit

ABSTRACT

EDMA 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 TI SoCs AM3x, AM4x, AM5x, DRA7x, TDA2x, TDA3x. UDMA is the DMA engine that is included in next generation TI SoCs like AM6x for the same purpose. UDMA is a new DMA architecture and SW applications written for EDMA need to be adapted to use the UDMA APIs. This application note is intended to help SW developers migrate their TI-RTOS based SW applications, device drivers from EDMA based systems to UDMA based systems. The application note explains the differences between EDMA and UDMA from a HW perspective. It then compares the DMA programming model from a SW perspective. Finally it compares the SW APIs provided by TI in Processor SDK RTOS for UDMA with EDMA.

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 EDMA to UDMA. In some cases specific SoCs are referred to aid the description. For SoC specific details users should refer to SoC technical reference manual (TRM).

Table of Contents

Figures

Tables

1 Introduction to EDMA and UDMA

This section gives an overview of EDMA and UDMA from HW perspective. In order to use the DMA SW APIs effectively it is important to understand the DMA HW architecture at least at a conceptual level. A brief EDMA overview is presented to refresh EDMA concepts for users familiar with EDMA. A brief UDMA overview is presented with SW perspective in mind so that users can then map these concepts to the UDMA SW API.

NOTE: Please note that users who use drivers from TI Processor SDK RTOS need not delve in to the architectural and programming details of UDMA, instead need to use only the `Udma_init()` API provided by UDMA driver in their application. The TI Processor SDK RTOS drivers use UDMA driver internally greatly simplifying the migration of applications from EDMA to UDMA. See section 3.4 Using UDMA with Processor SDK RTOS Drivers.

1.1 EDMA Overview

The enhanced direct memory access module, also called EDMA, performs high-performance data transfers between two slave points, memories and peripheral devices without microprocessor unit (MPU) or digital signal processor (DSP) support during data transfer. EDMA transfer is programmed through a logical EDMA channel, which allows the transfer to be optimally tailored to the requirements of the application.

EDMA controller is based on two major principal blocks:

- EDMA third-party channel controller (TPCC)
- EDMA third-party transfer controller (TPTC)

In a given SoC, typically one or more TPTC are associated with a TPCC. A given SoC could have multiple EDMA controllers.

For example, there are two instances of the TPTC in the AM5x device and a single instance of EDMA controller at SoC level.

The TPCC is a highly flexible Channel Controller. It serves as the user interface and an event interface for the EDMA controller. The TPCC serves to prioritize incoming software requests or events from peripherals and submits transfer requests (TRs) to the transfer controller.

The TPTC performs read and write transfers by EDMA ports to the slave peripherals as programmed in the "Active" and "Pending" set of the registers. The transfer controllers are responsible for data movement and issue read/write commands to the source and destination addresses that are programmed for a given transfer in the TPCC.

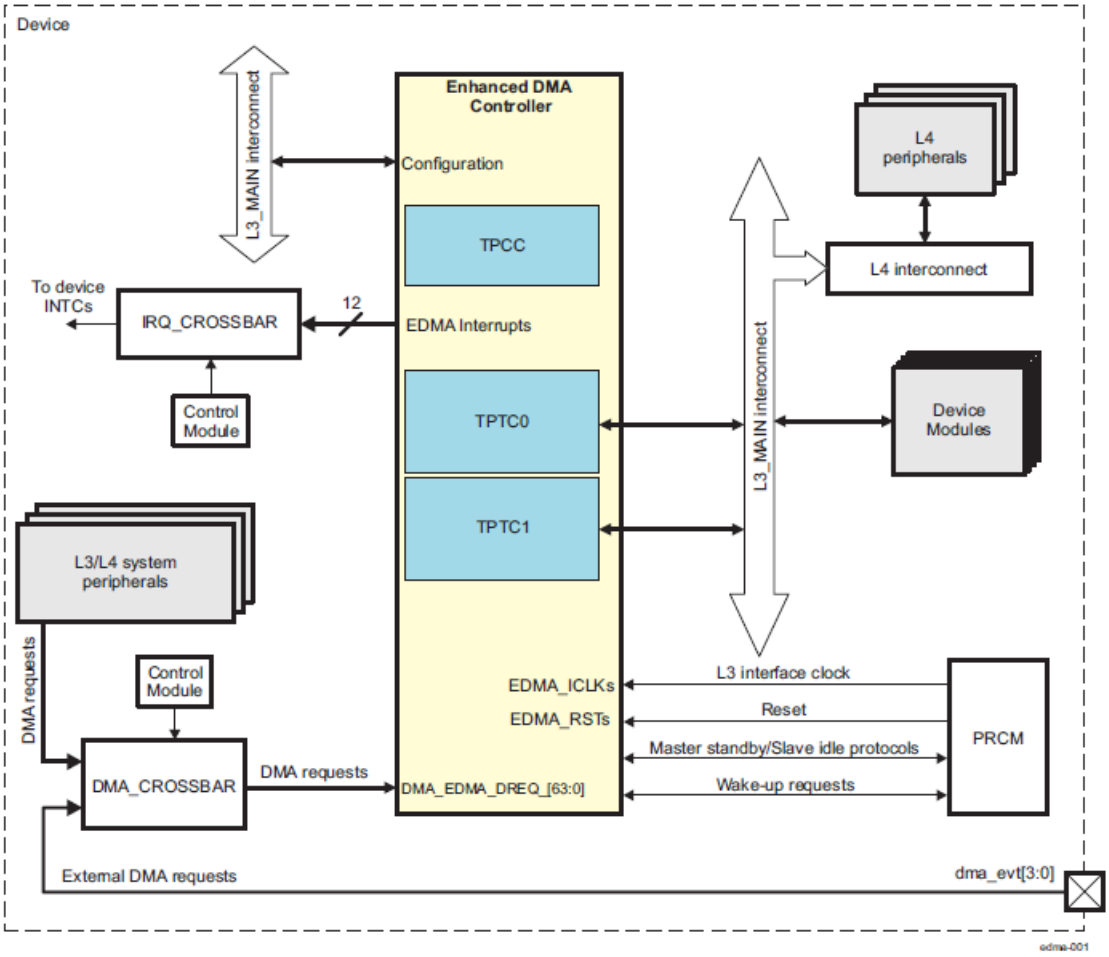

Figure 1. EDMA in AM5x SoC

NOTE: QDMA is not discussed in this application note since there isn't an equivalent of QDMA in UDMA architecture.

1.2 NavSS and UDMA Overview

1.2.1 NavSS Overview

The Navigator Subsystem (NAVSS) is a container which groups together components which are involved in providing DMA services in a SoC. Additional DMA components such as DRUs, UTC, and PDMAs exist outside the NAVSS but are controlled by UDMA controller provided in NAVSS. There can be multiple instances of NavSS and UDMA in a given SoC. For example, the AM6x SoC has two NAVSS's: one in main domain (Main NavSS) and one in MCU domain (MCU NavSS).

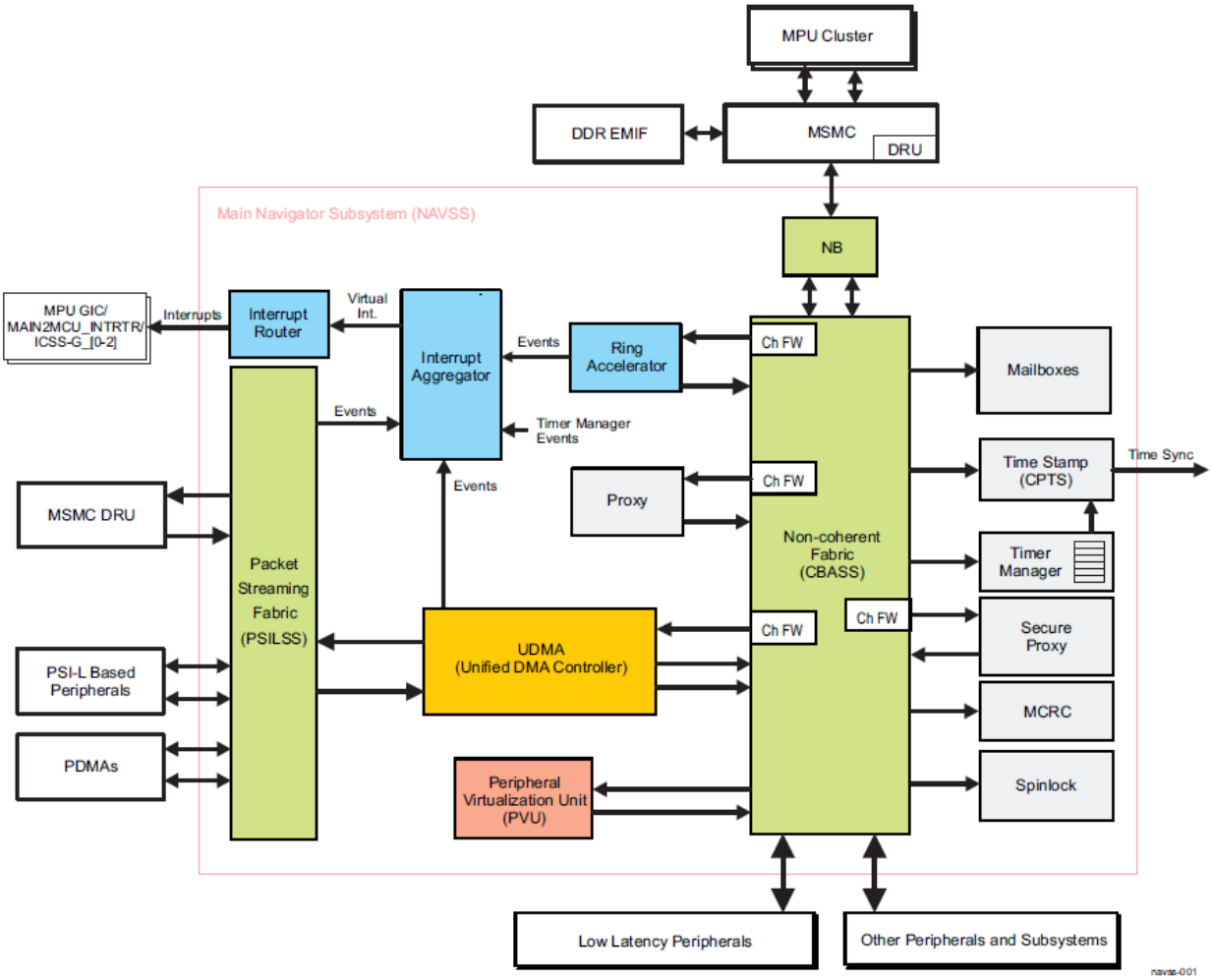

Figure 2. Main NavSS Block Diagram in AM6x SoC

1.2.2 UDMA Overview

The UDMA is intended to perform similar (but significantly upgraded) functions as the packet-oriented DMA used on some previous TI SoC devices. The UDMA module supports the transmission and reception of various packet types. The UDMA is architected to facilitate the segmentation and reassembly of SoC 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.

1.2.3 Unified Transfer Controller (UTC) Overview

The Unified Transfer Controller is intended to perform similar functions to the EDMA Transfer Controller engine used on previous devices. The UTC engine is generally classified as a third-party DMA. This designation comes from the fact that the engine is not actually the source or sink of the data which is being moved but is instead an intermediary 3rd party that performs the data move on behalf of the source and sink.

The UTC engine accepts Transfer Response messages from the UDMA via a PSI-L interface which provide instructions to copy data between a source read interface and a destination write interface. The sequence of operations that can be instructed includes up to 4-dimensional nested loops. Multiple types of Transfer Request messages are specified and each UTC instance in the system may support all types or any specified subset. When a Transfer Request has been completed, the UTC returns a Transfer Response message back to the originating UDMA.

The UTC has the ability to generate events at specified completion points when processing a Transfer Request. These events are sent back to the Interrupt Aggregator block.

1.2.4 Data Routing Unit (DRU) Overview

For purposes of the AM6x SoC Data Movement architecture, the DRU is a UTC which supports only the block copy mode subset of the Transfer Request message formats. The DRU typically will provide the highest performance block copy data movement capability of any DMA engine within the SoC.

1.2.5 PDMA Overview

The Peripheral DMA is a simple DMA which has been architected to specifically meet the data transfer needs of peripherals which perform data transfers using memory mapped registers accessed via a standard non-coherent bus fabric. The PDMA module is intended to be located close to one or more peripherals which require an external DMA for data movement and supporting only statically configured Transfer Request operations. The PDMA is only responsible for performing the data movement transactions which interact with the peripherals themselves. Data which is read from a given peripheral is packed by a PDMA source channel into a PSI-L data stream which is then sent to a remote peer UDMA-P destination channel which then performs the movement of the data into memory. Likewise, a remote UDMA-P source channel fetches data from memory and transfers it to a peer PDMA destination channel over PSI-L which then performs the writes to the peripheral.

The Peripheral DMA architecture is intentionally heterogeneous (UDMA + PDMA) to right size the data transfer complexity at each point in the system to match the requirements of whatever is being transferred to or from. Peripherals are typically FIFO based and do not require multi-dimensional transfers beyond their FIFO dimensioning requirements, so the PDMA transfer engines are kept simple with only a few dimensions (typically for sample size and FIFO depth), hardcoded address maps, and simple triggering capabilities.

Multiple source and destination channels are provided within the PDMA which allow multiple simultaneous transfer operations to be ongoing. The DMA controller maintains state information for each of the channels and employs round-robin scheduling between channels in order to share the underlying DMA hardware.

1.2.6 UDMA Block diagram : SW View

Below figures show how the various blocks of UDMA collaborate to perform a DMA transaction. Figure below shows as example of McASP DMA transfer in AM6x on using Main NavSS UDMA instance. The numbers in the figure show the sequence of steps that SW does for DMA configuration and the sequence of steps UDMA does to perform the DMA

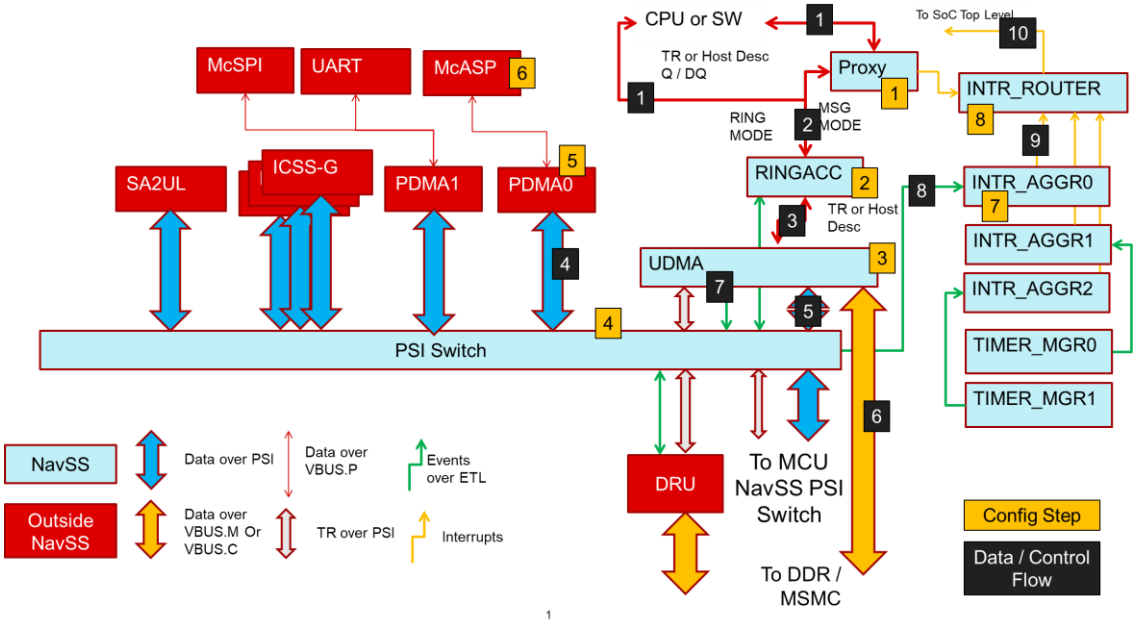

Figure 3. AM6x Main NavSS: SW view and McASP UDMA sequence

A brief description of the sequence of the configuration steps is given below

1. Setup “Proxy” to submit descriptors (this is an optional step)
2. Setup a RX free ring and RX completion within ring accelerator. A ring acts as HW FIFO to accept DMA transfer descriptors from the SW and pass on to the UDMA
3. Setup UDMA RX channel within UDMA
4. Pair the UDMA RX channel thread ID with McASP PDMA thread ID in the PSI-L (Packet Streaming Interface - Link) network
5. Setup static TR in PDMA channel for McASP
6. Setup McASP to receive data
7. Setup interrupt aggregator to receive RX packet completion event and convert to an interrupt
8. Setup interrupt router to route the interrupt to required CPU

A brief description of the sequence of steps as data flows through the SoC is given below

1. SW submits a DMA transfer descriptor to the RX free ring directly or via the “Proxy” HW
2. The descriptor is enqueue into the ring

3. The descriptor is forwarded to the UDMA channel so that when data is received from McASP the data is stored at the buffer pointed to by the descriptor
4. As McASP receives data the data is sent over the PSI-L network to the “paired” UDMA channel.
5. UDMA RX channel receives the data
6. The received data is written to the buffer pointed by the previously enqueued descriptor
7. Once the complete data packet is received an event is generated over the ETL (Event Transport Lane) bus
    a. The completed descriptor is written out to the RX completion queue (NOT shown in the figure)
8. The event is routed to a programmed interrupt within the interrupt aggregator
9. The interrupt aggregator route the event to the interrupt router
10. The interrupt router routes the event to the required CPU interrupt line

The figure below as example of DRU DMA transfer using UDMA

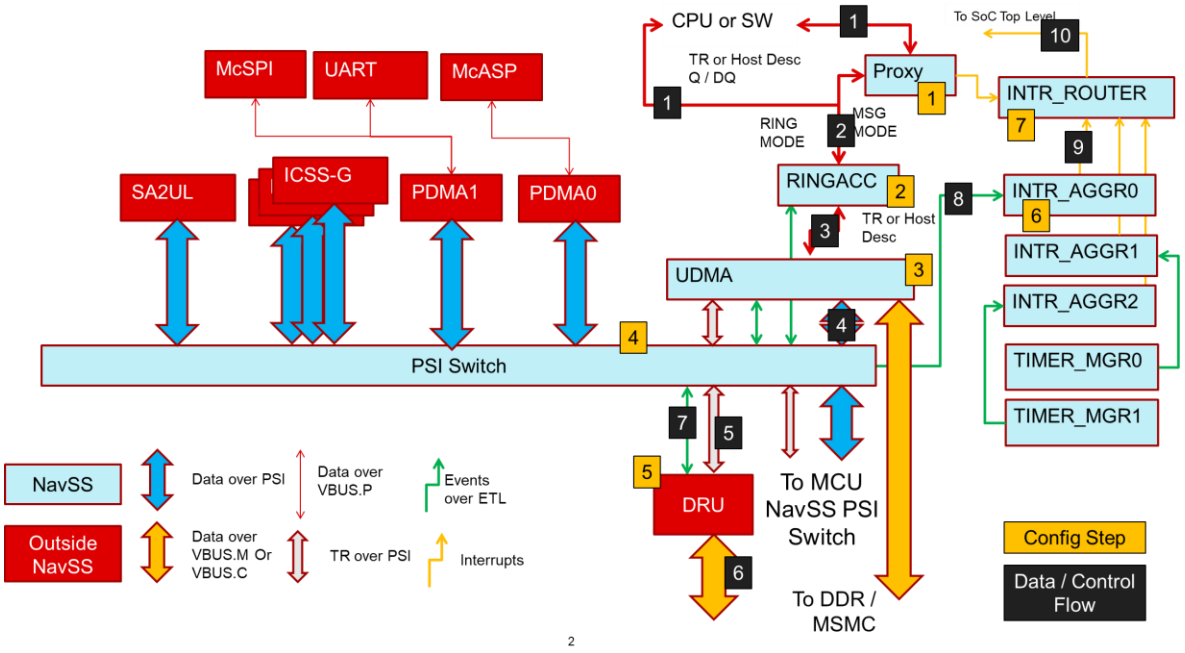

Figure 4. AM6x Main NavSS: SW view and DRU UDMA sequence

Here most of the steps are same as Figure 3 shown previously (config steps and data flow steps 1, 2, 3). Main difference is here the descriptor is forwarded by the UDMA to the DRU (UTC) peripheral (step 4, 5). Here the PSI-L network is used to transfer the descriptor. The DRU then does the DMA transfer without the intervention of the UDMA (step 6). The PSI-L network is not used for data transfer. When the DMA transfer completes the completion event is routed to the user via the ETL (step 7, 8, 9, 10).

1.3 Comparison of EDMA and UDMA

Below table shows a comparison of features in EDMA and UDMA.

Table 1. Comparison of EDMA and UDMA features

EDMA features	UDMA features
32b source and destination DMA buffer address	64b source and destination DMA buffer address
Three transfer dimensions (A, B, C)	Two modes of operations • Packet mode • Transfer Request (TR) mode Up to four transfer dimensions in TR mode (DIM0, DIM1, DIM2, DIM3)
• A-synchronized transfers: one-dimension serviced per event • AB-synchronized transfers: two-dimensions serviced per event	• DIM0 synchronized transfer: 1D • DIM1 synchronized transfer: 2D • DIM2 synchronized transfer: 3D • DIM3 synchronized transfer: 4D
Independent indexes on source and destination. Same ACNT, BCNT, CCNT on source and destination.	Independent indexes and count on source and destination for DIM0/1/2/3.
Increment or FIFO transfer addressing modes	Increment or FIFO transfer addressing modes
Linking mechanism allows multiple DMAs to be sequenced on the same EDMA Channel limited by number of PaRAM entries	TR Descriptor and ring accelerator allows multiple DMA transfers to be sequenced on the same UDMA channel limited only by available system memory space to store descriptors
Chaining allows multiple transfers to execute simultaneously on multiple EDMA channels with one event	Chaining allows multiple transfers to execute simultaneously on multiple UDMA channels with one event. Chaining achieved using event steering from event source to event sink on Event Transport Lane (ETL)
Interrupt generation for the following: • Transfer completion, intermediate transfer completion • Error conditions	Event generation for the following • Transfer completion, intermediate transfer completion • Error conditions Events can be converted to interrupts using Interrupt Aggregator and Interrupt Router

EDMA features	UDMA features
Debug visibility: • Queue water marking/threshold. • Error and status recording to facilitate debug.	Debug visibility: • Queue water marking/threshold. • Error and status recording to facilitate debug.
16 ~ 128 EDMA channels depending on the SoC Same channel can be used for RX or TX	• 120 TX channels (memory to peripheral) in AM6x Main NavSS • 150 RX channels (peripheral to memory) in AM6x Main NavSS • A pair of RX+TX channels is used for memory to memory DMA **NOTE:** the number of RX and TX channel can change from SoC to SoC and from Main NavSS to MCU NavSS. Refer to SoC TRM for exact numbers.
Synchronization based on • Event from peripheral. • Manual synchronization (CPU(s) write to event set registers EDMA_TPCC_ESR and EDMA_TPCC_ESRH). • Chain synchronization (completion of one transfer triggers another transfer).	Synchronization based on • Manual synchronization (CPU(s) write to UDMA channel trigger register). • Chain synchronization (completion of one transfer triggers another transfer (using events)).
Eight QDMA channels	32 external DMA channels in AM6x SoC to interface to DRU DRU used for high throughput memory to memory DMA
DMA operation described by a PaRAM set, up to 512 PaRAM set in a EDMA controller	DMA operation described by a packet descriptor or TR descriptor in system memory. Number of descriptors only limited by amount of system memory

2 DMA SW Programming Model

When programming a DMA there are multiple low level HW concepts, features like describing a DMA, DMA trigger, interrupt handling, that need to be understood by the SW user in order to use the SW API effectively. This section refreshes the EDMA programming model, followed by a detailed discussion on the UDMA programming model. Finally the EDMA and UDMA programming models are compared.

2.1 EDMA SW Programming model

2.1.1 Describing and Triggering DMA

In EDMA, a DMA transfer is described by programming a PaRAM set. A given EDMA CC has typically 32 to 512 PaRAM set MMRs located within the EDMA CC MMR memory map.

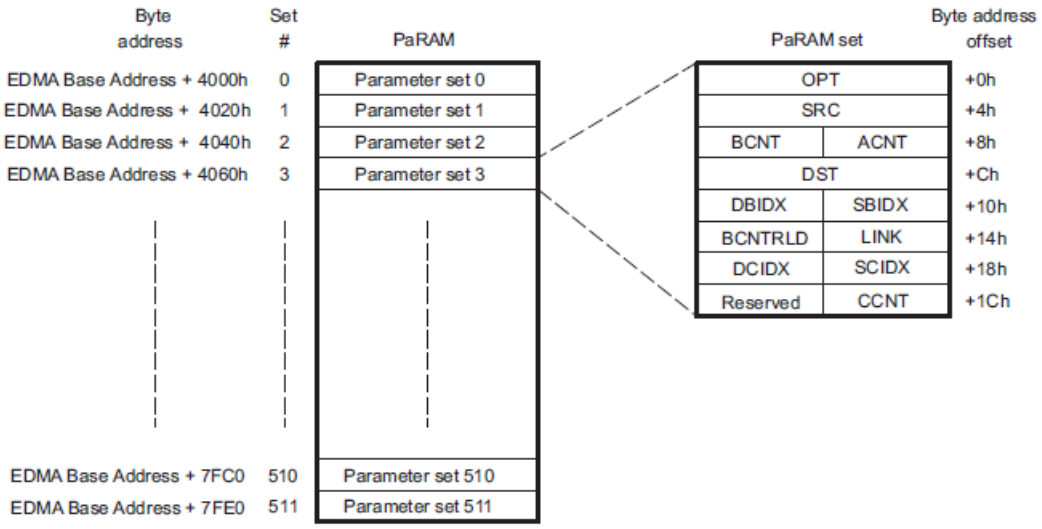

Figure 5. EDMA PaRAM set

A PaRAM set specifies

- Dimensions of the DMA transfer (3 dimensions ACNT, BCNT, CCNT)
- Source / destination 32b address
- DMA transfer flags

A EDMA is triggered in one of the ways below,

- **Event-triggered transfer request** (this is the typical usage of EDMA controller): A peripheral, system or externally-generated event triggers a transfer request.
- **Manually-triggered transfer request**: The CPU manually triggers a transfer by writing a 1 to the corresponding bit in the event set registers (EDMA_TPCC_ESR / EDMA_TPCC_ESRH).
- **Chain-triggered transfer request**: A transfer is triggered on the completion of another transfer or sub transfer.

2.1.2 Interfacing DMA to Peripherals

Peripherals are interfaced to EDMA via event signals. An event signal is asserted from a peripheral when it is ready to receive or transmit data to the EDMA. When an event is asserted from a peripheral or device pins, it gets latched in the corresponding bit of the event register (EDMA_TPCC_ER[31:0] En = 1). For more information about peripheral events to EDMA events mapping, refer to the device data manual.

2.1.3 Resource Management

In a typical SW architecture below modules in EDMA are considered as resources which need to be managed by a SW driver

- EDMA CC channels
- TCC for chaining of EDMA channels
- PaRAM sets
- Event Queues
- Number of regions (to allow same EDMA to be used from multiple CPUs within a SoC)

2.1.4 Interrupt Handling and Synchronization

The EDMA CC generates transfer completion interrupts to the CPU(s). The EDMA generates a single completion interrupt per shadow region. Usage of shadow regions allows multiple CPUs to handle completion interrupts from different DMA channels belonging to the same EDMA instance.

Interrupt generation can be enabled at either final transfer completion or intermediate transfer completion, or both.

Consider channel m as an example.

- If the final transfer interrupt (EDMA_TPCC_OPT_n[20] TCINTEN = 1) is enabled, the interrupt occurs after the last transfer request of channel m is either submitted or completed (depending on early or normal completion).
- If the intermediate transfer interrupt (EDMA_TPCC_OPT_n[21] ITCINTEN = 1) is enabled, the interrupt occurs after every transfer request, except the last TR of channel m is either submitted or completed (depending on early or normal completion).
- If both final and intermediate transfer completion interrupts (EDMA_TPCC_OPT_n[20] TCINTEN = 1, and EDMA_TPCC_OPT_n[21] ITCINTEN = 1) are enabled, then the interrupt occurs after every transfer request is submitted or completed (depending on early or normal completion).

2.2 UDMA SW Programming model

2.2.1 Describing and Triggering DMA

A UDMA DMA request is specified by user constructing a data structure in memory called “Descriptor”. Once a “Descriptor” is constructed by SW, it is submitted into a “Ring” to trigger the DMA operation,

There are various types of descriptors, the most commonly used descriptors are listed below

1. Host Packet Descriptor – these descriptors typically describe 1D “packet” buffers that need to be RX’ed or TX’ed with a peripheral or used in memory to memory copy transfer (Block Copy)
2. Transfer Request (TR) Descriptor – these descriptors typically describe an up to 4D data buffer that needs to be RX’ed or TX’ed with a peripheral or used in memory to memory copy transfer (Block Copy)

The descriptor that needs to be used depends on the peripheral that the DMA channel interfaces to. As an example, below table for AM6x SoC lists the peripherals and the descriptors types that are supported by each.

Table 2. UDMA supported peripherals in AM6x SoC

Peripheral	UDMA instance to use to submit descriptor (Main NavSS UDMA or MCU NavSS UDMA)	Peripheral Type	Recommended Descriptor type supported
SA2UL	Main NavSS UDMA	Native PSI-L	Host Packet Descriptor
ICSS-G	Main NavSS UDMA		Host Packet Descriptor
CPSW2g	MCU NavSS UDMA		Host Packet Descriptor
SPI	Main NavSS UDMA MCU NavSS UDMA	PDMA	Host Packet Descriptor
UART	Main NavSS UDMA MCU NavSS UDMA		Host Packet Descriptor
McASP	Main NavSS UDMA MCU NavSS UDMA		TR Descriptor
MCAN	MCU NavSS UDMA		Host Packet Descriptor
ADC	MCU NavSS UDMA		Host Packet Descriptor or TR Descriptor
FSS (OSPI, HyperFlash)	Main NavSS UDMA Or MCU NavSS UDMA	Block Copy	Host Packet Descriptor or TR Descriptor
MCRC	Main NavSS UDMA Or MCU NavSS UDMA		TR Descriptor

Peripheral	UDMA instance to use to submit descriptor (Main NavSS UDMA or MCU NavSS UDMA)	Peripheral Type	Recommended Descriptor type supported
PCIe	Main NavSS UDMA Or MCU NavSS UDMA		Host Packet Descriptor or TR Descriptor
Memory Copy	Main NavSS UDMA Or MCU NavSS UDMA		Host Packet Descriptor or TR Descriptor
DRU	Main NavSS UDMA		TR Descriptor

Figure 6 shows the TR Descriptor and TR data structure format. A TR Descriptor consists of a list of TR records. A TR record specifies up to 4D DMA transfer. A TR record data structure is shown in Figure 7.

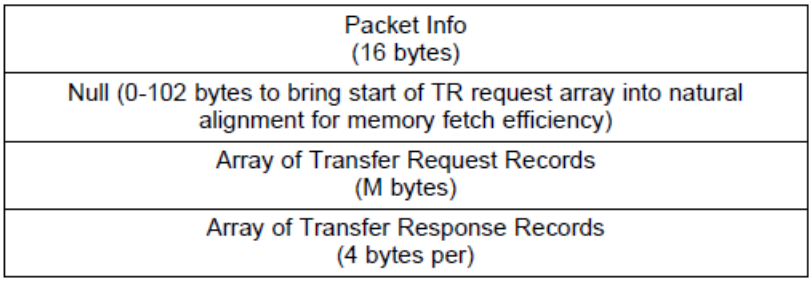

Figure 6. TR Descriptor

word 15		word 14		word 13		word 12
DICNT3	DICNT2	DICNT1	DICNT0	DDIM3		DDIM2
word 11		word 10		word 9		word 8
DADDR				DDIM1		FMTFLAGS
word 7		word 6		word 5		word 4
DIM3		DIM2		ICNT3	ICNT2	DIM1
word3		word 2		word 1		word0
ADDR				ICNT1	ICNT0	FLAGS

Figure 7. TR record

Figure 8 below shows the Host Packet Descriptor data structure format

Packet Info (16 bytes)
Linking Info (8 bytes)
Buffer Info (12 bytes)
Original Buffer Info (12 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)

Figure 8. HOST Packet Descriptor

2.2.2 Interfacing DMA to Peripherals

Peripherals are interfaced to UDMA using the PSI-L (Packet switching interface) bus. The peripheral to PSI-L bus connections are fixed in a SoC.

Some legacy peripherals like SPI, UART, McSPI do not support native PSI-L bus interface. Such peripherals are interfaced to UDMA via a PDMA peripheral. Here the legacy peripheral is connected to the PDMA peripheral and PDMA is connected to the PSI-L bus interface. The Peripheral to PDMA connection is fixed for a given SoC. For example see Figure 3 AM6x Main NavSS: SW view and McASP UDMA sequence.

Once a peripheral is connected to the PSI-L bus it appears on the bus with a unique “thread ID”. Each UDMA RX and TX channel is also visible on the bus as a unique “thread ID”. Connecting a peripheral to a UDMA channel is done by SW “pairing” of UDMA CH “thread ID” with the peripheral “thread ID”.

2.2.3 Resource Management

The following UDMA resources are limited in number and need to be resource managed by the SW.

1. UDMA Channels
    a. UDMA RX High Capacity channels
    b. UDMA TX High Capacity channels
    c. UDMA RX Normal Capacity channels
    d. UDMA TX Normal Capacity channels
    e. UDMA Block Copy Channels
    f. UDMA Free Flows

g. UDMA External Channels to interface with UTC's like DRU

2. Ring
   a. UDMA Ring – One Ring is associated with UDMA RX, TX, Block copy channels to submit descriptors – when a DMA channel is allocated the associated ring is also allocated by UDMA LLD. These rings are used for
      i. TX Ring for submitting TX packet, TX TR, or Block Copy TR
      ii. External TX Ring for submitting block copy to external channel
      iii. RX Ring for submitting RX TR
   b. Free Ring – Additional rings are used to receive completed DMA requests (completion rings). Any of the remaining rings can be used as free rings. These rings are used for,
      i. TX Completion Ring for packet or TR
      ii. RX Completion Ring for packet or TR
      iii. TX Completion for Teardown or External Channel Block Copy
      iv. RX Completion for Teardown
3. Events
   a. Free global events – global events are used to trigger interrupts via the interrupt aggregator. Any event ID from a range of event ID can be used by the SW to route a generated event to an interrupt
4. Interrupt Aggregator virtual interrupts, virtual interrupt bit position, interrupt router interrupt output
   a. Events are converted to interrupts by routing to an interrupt aggregator.

2.2.4 Interrupt Handling and Synchronization

In UDMA there is distinct mechanism of events and interrupts. An event is generated when some pre-defined condition occurs. SW can then use the event to either trigger another DMA channel (similar to chaining in EDMA) and/or convert the event to an interrupt.

In UDMA events can be generated for the below conditions

1. Descriptor RX or TX completed – this event is generated when a submitted descriptor is "fully consumed", i.e. all expected data is RX'ed or TX'ed. This condition is indicated to SW via a descriptor response in a completion ring. The event ID to generate is programmed within the completion ring in this case
2. Partial or full TR completion – this event generated when 1D, 2D,3D or 4D transfers of a TR are completed. This allows SW to sync on intermediate TR completion conditions. The event ID to generate for these conditions is programmed in the UDMA RX or TX channel (or DRU CH in case of DRU based DMA)
3. Events are also generated based on a ring status as well (ex, pushing to an empty ring).

Figure 9 shows the basic concept of event getting converted to an interrupt.

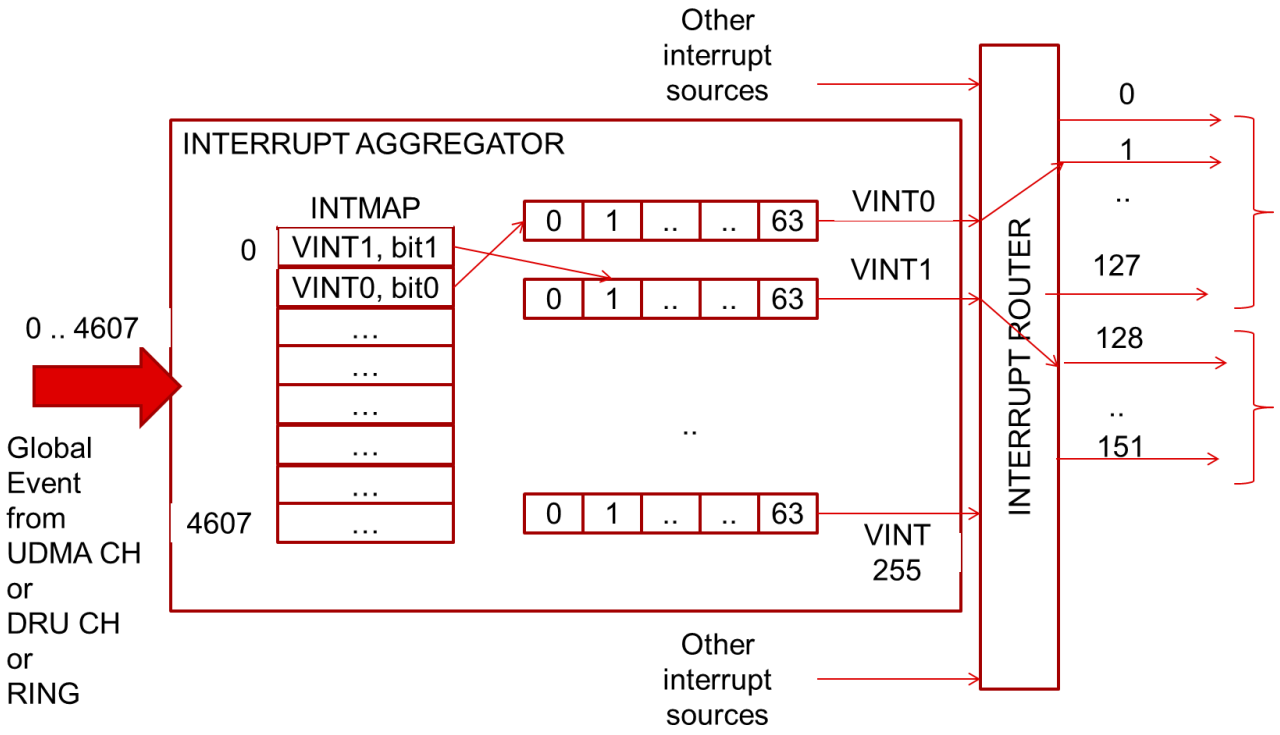

Figure 9. AM6x SoC Main NavSS UDMA Interrupt Aggregator and Interrupt Router

This shows example of AM6x Main NavSS UDMA interrupt routing from a global event to interrupt at NavSS boundary. Here global events from ID 0 to 4607 are routed to Main NavSS UDMA Interrupt aggregator. If an event ID in this range is programmed at any ring or UDMA CH or DRU CH or any other event generator source then it reaches this interrupt aggregator. The IMAP_j (j=0 .. 4607) register at the interrupt aggregator then maps this event to one of 256 “virtual interrupts (VINT)” and within a “virtual interrupt” to one of 64 “bitnum”. i.e. up to 64 events can be aggregated at a single interrupt aggregator VINT. SW can now poll on the VINT x, bitnum y MMR for interrupt arrival or SW can route this VINT further to the CPU via the interrupt router. The interrupt router is M:N mux. In above example the interrupt router takes 255 inputs from the interrupt aggregator and routes them to 151 interrupt outputs. The outputs of interrupt router are then further routed to respective CPUs based on the SoC architecture.

2.3 Comparison of SW Programming Model in EDMA and UDMA

The table below shows the comparison of programming model between EDMA and UDMA based on the concepts introduced in previous sections

Table 3. Programming model comparison between EDMA and UDMA

Programming Model	EDMA	UDMA
Describing DMA	PaRAM MMRs within EDMA	HOST packet descriptor (1D) or TR Descriptor (up to 4D) in any memory visible to UDMA, ex, DDR, MSMC, TCM, MCUSS MSRAM. **NOTE:** since memory could be cache SW should make sure cache coherency operations are performed for descriptors in memory
Triggering DMA	Setting channel enable bit	Queuing a descriptor into a ring
Linking DMA transfers	Using LINK field in PaRAM. Max DMA transfers that can be linked limited by number of PaRAM (typically 100's)	Multiple TRs within TR descriptors OR queueing multiple descriptors into a ring. Max DMA transfer than can be linked limited by available system memory (practically unlimited if descriptors in DDR).
Resource Management SW needs to manage these resources since they are limited in numbers	• EDMA channels • EDMA regions • PaRAMs • TCCs	• UDMA RX/TX/External channels • UDMA RX Flows • Free RINGs • Global event ID • Interrupt aggregator VINT and "bitnum" (0..63) • Interrupt router output interrupt

Programming Model	EDMA	UDMA
Interfacing Peripheral to DMA	By routing peripheral ready/busy signal to event input signal at EDMA	By paring peripheral PSI-L bus thread ID to UDMA CH thread ID
Interrupt handling	At DMA transfer completion and at intermediate DMA transfer completion (1D, 2D) Interrupts latched within one of N regions (typically 4-8) to allow multiple CPUs to use different EDMA channels of the same EDMA instance	At DMA transfer completion and at intermediate DMA transfer completion (1D, 2D, 3D) Interrupts latched by routing global events to interrupt aggregator. Multiple CPU can use different EDMA by routing different interrupt aggregator VINT to itself via interrupt router.
Chaining DMA channels	By setting TCC of EDMA channel to be chained within PaRAM	By setting event ID to a UDMA channel trigger event ID within the source UDMA CH CFG MMR

2.3.1 Comparision of EDMA PaRAM vs UDMA TR

A large part of user SW deals with setup of EDMA PaRAM fields to setup the DMA transfer and decide the SW sync points, next DMA transfer linking/chaining. In EDMA this is specified in PaRAM. This section does a field by field comparison of EDMA PaRAM with UDMA TR which is the closest equivalent in UDMA.

Table 4. Comparison of EDMA PaRAM and UDMA TR

EDMA PaRAM	UDMA TR (4D Block Copy TR)	Purpose
ACNT (16b)	ICNT0 (16b) DICNT0 (16b)	1st dimension loop count
BCNT (16b)	ICNT1 (16b) DICNT1 (16b)	2nd dimension loop count
CCNT (16b)	ICNT2 (16b) DICNT2 (16b)	3rd dimension loop count
NA	ICNT3 (16b) DICNT3 (16b)	4th dimension loop count

EDMA PaRAM	UDMA TR (4D Block Copy TR)	Purpose
SBIDX (signed 16b)	DIM1(signed 32b)	2nd dimension stride at source
DBIDX (signed 16b)	DDIM1 (signed 32b)	2nd dimension stride at destination
SCIDX (signed 16b)	DIM2 (signed 32b)	3rd dimension stride at source
DCIDX	DDIM2 (signed 32b)	3rd dimension stride at destination
NA	DIM3 (signed 32b) DDIM3 (signed 32b)	4th dimension stride at source and destination
SRC (32b)	ADDR (64b)	Source address
DST (32b)	DAADR (64b)	Destination address
LINK	Linking is done in UDMA by listing TRs one after another within a TR descriptor OR queuing multiple TR Descriptors at the same RING	Pointer to next DMA transfer to execute after current DMA transfer
BCNTRLD	NA	BCNT value to reload after BCNT decrements to zero.
OPT	FLAGS	DMA transfer options / flags see table below

Table 5. Comparison of EDMA.OPT and UDMA TR.FLAGS

EDMA.OPT	UDMA TR.FLAGS	Purpose
PRIV	Specified via ISC registers in Ring that is used to submit the TR OR inherited from the CPU host the writes to the ring when ring is used in credentials mode	Privilege level User / supervisor
PRIVID	Specified via ISC registers in Ring that is used to submit the TR OR inherited from the programming CPU host when ring is used in credentials mode	Privilege ID for the external host/cpu/dma that programmed this DMA transfer

EDMA.OPT	UDMA TR.FLAGS	Purpose
ITCCHEN TCCHEN TCC	In UDMA this is specified by programing the destination UDMA Block Copy channel event ID in current UDMA channel event ID MMR	Chaining condition and EDMA channel to chain after current EDMA channel DMA transfer completes. In UDMA after event EVENT_SIZE completion an event is generated. If the event ID maps to another UDMA channel then that triggers the UDMA channel (when trigger type of the chained channel is “global” in TRIGGER_TYPEx)
ITCINTEN TCINTEN	EVENT_SIZE	DMA transfer interrupt/event condition, 1D, 2D, 3D, or 4D (In EDMA SYNCDIM controls both the sync condition and the intermediate event/interrupt condition. In UDMA the sync and event/condition can be specified independently in EVENT_SIZE and TRIGGER_TYPEx respectively)
SYNCDIM	TRIGGERx_TYPE (x=0,1)	DMA trigger type, A-sync, AB-sync mode. (A-sync mode in EDMA has a different dimension skip pattern vs AB-sync, however in UDMA 1D, 2D, 3D, 4D sync modes all have the same dimension skip pattern, that is similar to AB-sync in EDMA) UDMA support two trigger event and sync type can be specified on each trigger event.
FWID DAM SAM	NA	Used for FIFO mode of access in EDMA, typically to read/write data to peripheral FIFOs In UDMA, the PDMA typically reads the peripheral FIFO so this mode is not applicable for UDMA
WIMODE TCCMODE STATIC	NA	No equivalent feature in UDMA

3 DMA SW API for Applications

This section provides an overview of EDMA and UDMA SW APIs and describes how an application can migrate from EDMA SW APIs to UDMA SW APIs.

3.1 EDMA SW API Overview

EDMA LLD is the SW API for EDMA on TI-RTOS. It is included as a package within TI Processor SDK. This section gives brief overview of EDMA LLD APIs using "edma3_lld_02_12_00_20" package as the reference. Specifically it gives an overview of the EDMA LLD driver module within this package (edma3_lld_02_12_00_20\packages\ti\sdo\edma3\drv)

3.1.1 Steps to initialize and deinitialize EDMA driver

1. Create a EDMA3 driver handle.
   a. EDMA instance ID and global EDMA configuration is provided as input.

```
edma3Result = EDMA3_DRV_create (
                        pObj->edma3InstanceId,
                        pObj->pGblCfgParams ,
                        (void *)&miscParam);
```

   b. Global params among other things defines the capabilities of the EDMA controller like total number of channels, total number of PaRAM entries, EDMA CC/TC MMR base address, and so on. These parameters are different for different SoCs
2. Open a EDMA3 driver instance handle
   a. EDMA instance ID and initialization configuration is provided as input

```
EDMA3_DRV_InitConfig initCfg;
initCfg.drvSemHandle = Osal_createSemaphore;
initCfg.isMaster = TRUE;
/* Choose shadow region unique to current CPU */
initCfg.regionId = (EDMA3_RM_RegionId)pObj->regionId;
/* Driver instance specific config NULL */
initCfg.drvInstInitConfig = pObj->pInstInitConfig;
initCfg.gblerrCb          = NULL;
initCfg.gblerrData        = NULL;

pObj->hEdma = EDMA3_DRV_open (
           pObj->edma3InstanceId,
           (void *) &initCfg,
           &edma3Result);
```

   b. Initialization configuration specifies among other things, the EDMA channels reserved for current CPU, the region ID current CPU will use and so on. This allows the same EDMA controller instance to be shared among multiple CPU in the same SoC.

3. Register interrupt handler based on CPU specific architecture. The below code snippet shows an example of ISR registration on ARM M4 for AM5x SoC

```
/* Do interrupt cross bar connection if any */
IntXbar_connect(
        pObj->pIntrConfig->ccXferCompCtrlModXbarIndex,
        pObj->pIntrConfig->ccXferCompXbarInt
        );

pObj->hwiCCXferCompInt = Osal_registerIntr(
               pObj->pIntrConfig->ccXferCompCpuInt, /* CPU interrupt
number associated with EDMA completion for a specific EDMA region */
               &lisrEdma3ComplHandler0, /* defined by EDMA driver */
               (void*)pObj->edma3InstanceId);
```

4. To de-initialize EDMA driver SW uses below sequence
    a. Deregister CPU interrupts (EDMA driver API not needed for this)
    b. Close EDMA driver instance handle, delete EDMA driver

```
EDMA3_DRV_close (pObj->hEdma, NULL);
EDMA3_DRV_delete (pObj->edma3InstanceId, NULL);
```

3.1.2 Steps to setup and use EDMA channels

There many different ways to use program and use an EDMA channel and PaRAM. This section shows the typical sequence used for memory to memory copy using EDMA.

1. Firstly a EDMA channel is allocated using below API.
    a. The EDMA driver instance handle create earlier is used as input.
    b. In the below example, SW requests for any available EDMA channel and TCC ID.
    c. SW also registers a callback to be called when EDMA transfer is complete. If NULL is specified then SW wants to use the channel in polling mode

```
pObj->tccId         = EDMA3_DRV_TCC_ANY;
pObj->edmaChId     = EDMA3_DRV_DMA_CHANNEL_ANY;
tccCb = NULL;
if(pObj->enableIntCb)
    tccCb = (EDMA3_RM_TccCallback)&Utils_dmaCallback;
edma3Result = EDMA3_DRV_requestChannel(
                    pObj->hEdma,
                    (uint32_t*)&pObj->edmaChId,
                    (uint32_t*)&pObj->tccId,
                    (EDMA3_RM_EventQueue)pObj->eventQ,
                    tccCb, (void *)pObj);
```

2. Reset EDMA channel error state to clear any pending error condition on the channel.

```
EDMA3_DRV_clearErrorBits( pObj->hEdma, pObj->edmaChId);
```

3. When a channel is allocated, one PaRAM is automatically allocated for that channel. User can allocate additional PaRAMs for linking using below API

```
paramID = EDMA3_DRV_LINK_CHANNEL;
tccId    = EDMA3_DRV_TCC_ANY;
edma3Result = EDMA3_DRV_requestChannel (
                          pObj->hEdma,
                          &paramId,
                          &tccId,
                          (EDMA3_RM_EventQueue)pObj->eventQ,
                          NULL,
                          NULL);
```

4. Next, the PaRAM associated with the EDMA channel and its associated linked PaRAMs are programmed.

   a. Below API is called to get the PaRAM address pointer.

   b. After get the pointer to the PaRAM, the PaRAM fields are set based on the Figure 5 EDMA PaRAM set

```
edma3Result = EDMA3_DRV_getPaRAMPhyAddr(
                        pObj->hEdma,
                        edmaChId,
                        &paramPhyAddr);
```

5. After the PaRAM fields are programmed EDMA channel is triggered as shown below

   a. Previous EDMA state is cleared, if any

```
edma3Result = EDMA3_DRV_checkAndClearTcc(pObj->hEdma,
                                         pObj->tccId,
                                         &tccStatus);
```

   b. Trigger EDMA. Here the manual mode DMA trigger example is shown

```
edma3Result = EDMA3_DRV_enableTransfer (
                pObj->hEdma,pObj->edmaChId,
                EDMA3_DRV_TRIG_MODE_MANUAL);
```

6. After a EDMA channel is triggered SW can wait for the previously registered callback to get called on DMA channel completion and/or DMA intermediate channel completion (depending on the bits set in PaRAM) OR SW can poll on DMA completion as shown below

```
EDMA3_DRV_waitAndClearTcc(pObj->hEdma,pObj->tccId);
EDMA3_DRV_clearErrorBits (pObj->hEdma,pObj->edmaChId);
```

7. After a DMA channel completes, SW can program the same PaRAM again and trigger another DMA channel

8. After SW is done using the DMA channel it can free the DMA channel (and linked PaRAMs) as shown below

```
EDMA3_DRV_freeChannel(pObj->hEdma, pObj->edmaChId);
```

3.2 UDMA SW API Overview

UDMA LLD is the SW API for UDMA on TI-RTOS. It is included as a package within TI Processor SDK. This section gives brief overview of EDMA LLD APIs using "pdk_xx_xx_xx_xx" package as the reference. Specifically it gives an overview of the UDMA LLD driver module within this package (pdk_xx_xx_xx_xx\packages\ti\drv\udma)

3.2.1 Steps to initialize and deinitialize UDMA driver

1. Initialize the UDMA driver with default resource allocation as shown below
   a. Initialize default parameters. When default parameters are used UDMA resources are divided among different CPUs by a TI defined scheme. Users can override the default by modify the fields within initPrms. rmInitPrms after calling UdmaInitPrms. Users must to careful to make sure overlapping resources are not assigned to different CPUs.

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

    Udma_InitPrms   initPrms;
    uint32_t        instId;

    instId = UDMA_INST_ID_MAIN_0; /* or UDMA_INST_ID_MCU_0 */
    UdmaInitPrms_init(instId, &initPrms);
```

   b. Initialize the UDMA instance

```
struct Udma_DrvObj      gUdmaDrvObj; /* MUST be global */

Udma_DrvHandle  drvHandle = &gUdmaDrvObj;
    retVal = Udma_init(drvHandle, &initPrms);
```

2. To de-initialize the UDMA driver call the below API

```
Udma_deinit(drvHandle);
```

3.2.2 Steps to setup and use UDMA channels

There many different ways to use program and use an UDMA channels. This section shows the typical sequence used for memory to memory copy using UDMA. For more examples users are encouraged to browse the code at (pdk_xx_xx_xx_xx\packages\ti\drv\udma\examples). The below sequence uses the "examples\ udma_memcpy_test" as reference

3.2.2.1 Open a UDMA channel

1. UDMA driver handle created previous is used as input
2. A UDMA channel handle is returned as output
3. UDMA channel type is block copy for memory to memory copy DMA transfer
4. UDMA channel parameters can be set to default, however user has to give additional memory for the UDMA ring memory (submit ring, completion ring, teardown ring). In this example only one UDMA descriptor is queued at time and hence ring element count can be one.

5. The ring memory MUST be cache line aligned. The ring memory can be from a cache region of the CPU

```
/** \brief Number of ring entries - we can prime this much memcpy
operations */
#define UDMA_TEST_APP_RING_ENTRIES      (1U)

/** \brief Size (in bytes) of each ring entry (Size of pointer - 64-
bit) */
#define UDMA_TEST_APP_RING_ENTRY_SIZE   (sizeof(uint64_t))

/** \brief Total ring memory */
#define UDMA_TEST_APP_RING_MEM_SIZE     (UDMA_TEST_APP_RING_ENTRIES *
UDMA_TEST_APP_RING_ENTRY_SIZE)

/* MUST be global */
static uint8_t gTxRingMem[UDMA_TEST_APP_RING_MEM_SIZE]
__attribute__((aligned(UDMA_CACHELINE_ALIGNMENT)));
static uint8_t gTxCompRingMem[UDMA_TEST_APP_RING_MEM_SIZE]
__attribute__((aligned(UDMA_CACHELINE_ALIGNMENT)));
static uint8_t gTxTdCompRingMem[UDMA_TEST_APP_RING_MEM_SIZE]
__attribute__((aligned(UDMA_CACHELINE_ALIGNMENT)));

struct Udma_ChObj       gUdmaChObj; /* MUST be global */

Udma_ChHandle   chHandle = &gUdmaChObj;

/* channel type for memory to memory DMA copy */
chType = UDMA_CH_TYPE_TR_BLK_COPY;
UdmaChPrms_init(&chPrms, chType);
/* ring memory for descriptors and number of elements in the ring */
chPrms.fqRingPrms.ringMem   = &gTxRingMem[0U];
chPrms.cqRingPrms.ringMem   = &gTxCompRingMem[0U];
chPrms.tdCqRingPrms.ringMem = &gTxTdCompRingMem[0U];
chPrms.fqRingPrms.elemCnt   = UDMA_TEST_APP_RING_ENTRIES;
chPrms.cqRingPrms.elemCnt   = UDMA_TEST_APP_RING_ENTRIES;
chPrms.tdCqRingPrms.elemCnt = UDMA_TEST_APP_RING_ENTRIES;
/* Open channel for block copy */
retVal = Udma_chOpen(drvHandle, chHandle, chType, &chPrms);
```

3.2.2.2 Setup UDMA channel for DMA transfer

1. A UDMA block copy involves setup of a pair of UDMA TX and UDMA RX channels. This detail is handle by the SW internally and users can simple initialize the RX and TX portions of the UDMA channel with default parameters as shown below

```
Udma_ChTxPrms       txPrms;
Udma_ChRxPrms       rxPrms;

/* Config TX channel */
UdmaChTxPrms_init(&txPrms, chType);
retVal = Udma_chConfigTx(chHandle, &txPrms);

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

2. Enable events and register callbacks. For polling mode operation, users can set the callback function pointer to NULL but still use the same sequence of steps below. Here two interrupts are enabled, a DMA transfer completion via ring response and ring tear down completion callback. Both events are registered on the same interrupt line using the “shared interrupt” property as shown below.

```
struct Udma_EventObj    gUdmaCqEventObj; /* MUST be global */
struct Udma_EventObj    gUdmaTdCqEventObj; /* MUST be global */

Udma_EventHandle    eventHandle;
Udma_EventPrms      eventPrms;

/* Register ring completion callback */
eventHandle = &gUdmaCqEventObj;
UdmaEventPrms_init(&eventPrms);
eventPrms.eventType         = UDMA_EVENT_TYPE_DMA_COMPLETION;
eventPrms.eventMode         = UDMA_EVENT_MODE_SHARED;
eventPrms.chHandle          = chHandle;
eventPrms.masterEventHandle = NULL;
eventPrms.eventCb           = &App_udmaEventDmaCb;
retVal = Udma_eventRegister(drvHandle, eventHandle, &eventPrms);

/* 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);
```

3. After interrupts are registered, UDMA channel is enabled. Note, this does NOT start any DMA transfer, this just keeps the channel ready for DMA transfer. DMA transfer is trigger when a descriptor is enqueued into the submit ring later.

```
retVal = Udma_chEnable(chHandle);
```

3.2.2.3 Program descriptor

1. Reserve memory for TR Descriptor. Make sure it is cache line aligned

```
/**
 *  \brief UDMA TR packet descriptor memory.
 *  This contains the CSL_UdmapCppi5TRPD + Padding to
sizeof(CSL_UdmapTR15) +
 *  one Type_15 TR (CSL_UdmapTR15) + one TR response of 4 bytes.
 *  Since CSL_UdmapCppi5TRPD is less than CSL_UdmapTR15, size is just
two times
 *  CSL_UdmapTR15 for alignment.
 */
#define UDMA_TEST_APP_TRPD_SIZE         ((sizeof(CSL_UdmapTR15) * 2U) +
4U)

static uint8_t gUdmaTprdMem[UDMA_TEST_APP_TRPD_SIZE]
__attribute__((aligned(UDMA_CACHELINE_ALIGNMENT)));
```

2. Program TR Descriptor header. Most of the fields can be programed to default as shown below. The completion ring ID is programmed in the TR Descriptor as shown below

```
static void App_udmaTrpdInit(Udma_ChHandle chHandle,
                             uint8_t *pTrpdMem,
                             const void *destBuf,
                             const void *srcBuf,
                             uint32_t length)
{
    CSL_UdmapCppi5TRPD *pTrpd = (CSL_UdmapCppi5TRPD *) pTrpdMem;
    CSL_UdmapTR15 *pTr = (CSL_UdmapTR15 *)(pTrpdMem +
sizeof(CSL_UdmapTR15));
    uint32_t *pTrResp = (uint32_t *) (pTrpdMem + (sizeof(CSL_UdmapTR15)
* 2U));
    uint32_t cqRingNum = Udma_chGetCqRingNum(chHandle);
    uint32_t descType = CSL_UDMAP_CPPI5_PD_DESCINFO_DTYPE_VAL_TR;

    /* Setup descriptor */
    CSL_udmapCppi5SetDescType(pTrpd, descType);
    CSL_udmapCppi5TrSetReload(pTrpd, 0U, 0U);
    CSL_udmapCppi5SetPktLen(pTrpd, descType, 1U);       /* Only one TR
in TRPD */
    CSL_udmapCppi5SetIds(pTrpd, descType, 0U, 0x3FFFU); /* Flow ID and
Packet ID */
    CSL_udmapCppi5SetSrcTag(pTrpd, 0x0000);     /* Not used */
    CSL_udmapCppi5SetDstTag(pTrpd, 0x0000);     /* Not used */
    CSL_udmapCppi5TrSetEntryStride(
        pTrpd, CSL_UDMAP_CPPI5_TRPD_PKTINFO_RECSIZE_VAL_64B);
    CSL_udmapCppi5SetReturnPolicy(
        pTrpd,
        descType,
        CSL_UDMAP_CPPI5_PD_PKTINFO2_RETPOLICY_VAL_ENTIRE_PKT,   /*
Don't
care for TR */
        CSL_UDMAP_CPPI5_PD_PKTINFO2_EARLYRET_VAL_NO,
        CSL_UDMAP_CPPI5_PD_PKTINFO2_RETPUSHPOLICY_VAL_TO_TAIL,
        cqRingNum);
```

3. Program the TR as shown below

```
/* Setup TR */
pTr->flags    = CSL_FMK(UDMAP_TR_FLAGS_TYPE, 15)
                CSL_FMK(UDMAP_TR_FLAGS_STATIC, 0U)        |
                CSL_FMK(UDMAP_TR_FLAGS_EOL, 0U)           | /* NA */
                CSL_FMK(UDMAP_TR_FLAGS_EVENT_SIZE,
CSL_UDMAP_TR_FLAGS_EVENT_SIZE_COMPLETION)                 |
                CSL_FMK(UDMAP_TR_FLAGS_TRIGGER0,
CSL_UDMAP_TR_FLAGS_TRIGGER_NONE)                          |
                CSL_FMK(UDMAP_TR_FLAGS_TRIGGER0_TYPE,
CSL_UDMAP_TR_FLAGS_TRIGGER_TYPE_ALL)                      |
                CSL_FMK(UDMAP_TR_FLAGS_TRIGGER1,
CSL_UDMAP_TR_FLAGS_TRIGGER_NONE)                          |
                CSL_FMK(UDMAP_TR_FLAGS_TRIGGER1_TYPE,
CSL_UDMAP_TR_FLAGS_TRIGGER_TYPE_ALL)                      |
                CSL_FMK(UDMAP_TR_FLAGS_CMD_ID, 0x25U)     | /* This
will come back in TR response */
                CSL_FMK(UDMAP_TR_FLAGS_SA_INDIRECT, 0U)   |
                CSL_FMK(UDMAP_TR_FLAGS_DA_INDIRECT, 0U)   |
                CSL_FMK(UDMAP_TR_FLAGS_EOP, 1U);
    pTr->icnt0    = length;
    pTr->icnt1    = 1U;
    pTr->icnt2    = 1U;
    pTr->icnt3    = 1U;
    pTr->dim1     = pTr->icnt0;
    pTr->dim2     = (pTr->icnt0 * pTr->icnt1);
    pTr->dim3     = (pTr->icnt0 * pTr->icnt1 * pTr->icnt2);
    pTr->addr     = (uint64_t) srcBuf;
    pTr->fmtflags = 0x00000000U;/*Linear addressing, 1 byte per elem*/
    pTr->dicnt0   = length;
    pTr->dicnt1   = 1U;
    pTr->dicnt2   = 1U;
    pTr->dicnt3   = 1U;
    pTr->ddim1    = pTr->dicnt0;
    pTr->ddim2    = (pTr->dicnt0 * pTr->dicnt1);
    pTr->ddim3    = (pTr->dicnt0 * pTr->dicnt1 * pTr->dicnt2);
    pTr->daddr    = (uint64_t) destBuf;
```

4. Clear TR response memory

```
*pTrResp = 0xFFFFFFFFU;
```

5. Cache Write back the descriptor since the descriptor could be cache in the CPU

```
CacheP_wb(pTrpdMem, UDMA_TEST_APP_TRPD_SIZE);
```

3.2.2.4 Trigger and wait for DMA transfer

1. Submit TR descriptor to submit ring

```
retVal = Udma_ringQueueRaw(
            Udma_chGetFqRingHandle(chHandle), (uint64_t) tprdMem);
```

2. Wait for ring completion. Application waits on a semaphore which is posted from the application registered callback

```
/* Wait for return descriptor in completion ring - this marks the
 * transfer completion */
SemaphoreP_pend(gUdmaAppDoneSem, SemaphoreP_WAIT_FOREVER);
```

3. The application callback is shown below

```
static void App_udmaEventDmaCb(Udma_EventHandle eventHandle,
                               uint32_t eventType,
                               void *appData)
{
    if(UDMA_EVENT_TYPE_DMA_COMPLETION == eventType)
        SemaphoreP_post(gUdmaAppDoneSem);
}
```

4. After a ring completion callback is received, dequeue the completed TR descriptor as shown below

```
uint64_t    pDesc = 0;

/* Completed TR descriptor received in completion queue */
retVal = Udma_ringDequeueRaw(Udma_chGetCqRingHandle(chHandle), &pDesc);
```

5. Check if DMA completed successfully by checking the TR response field as shown below. Note, since the TR descriptor can be cached user should invalidate cache line before touching the TR descriptor fields

```
uint32_t   *pTrResp, trRespStatus;

/* Invalidate cache */
CacheP_Inv((void*)pDesc, UDMA_TEST_APP_TRPD_SIZE);

/* check TR response status */
pTrResp = (uint32_t *) (pDesc + (sizeof(CSL_UdmapTR15) * 2U));
trRespStatus = CSL_FEXT(*pTrResp, UDMAP_TR_RESPONSE_STATUS_TYPE);
if(trRespStatus != CSL_UDMAP_TR_RESPONSE_STATUS_COMPLETE)
    retVal = UDMA_EFAIL;
```

6. Once a TR descriptor is received, the same TR descriptor can be updated with new DMA transfer parameters and submitted again. Also while a DMA transfer is active another TR descriptor can be submitted into the DMA channel asynchronously.

3.2.2.5 Close a UDMA channel

Once all DMA transfers are done from application point of view, DMA channel can be disabled and closed as shown below

1. Disable channel

```
retVal = Udma_chDisable(chHandle, UDMA_DEFAULT_CH_DISABLE_TIMEOUT);
```

2. If a channel is disabled while DMA is active a teardown descriptor is written to the teardown ring. This descriptor MUST be dequeued. In the example a teardown callback is registered to handle this condition as shown below

```
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. Unregister interrupts

```
Udma_EventHandle    eventHandle;

/* Unregister master event at the end */
eventHandle = &gUdmaTdCqEventObj;
retVal += Udma_eventUnRegister(eventHandle);
eventHandle = &gUdmaCqEventObj;
retVal += Udma_eventUnRegister(eventHandle);
```

4. Flush any pending requests. These are descriptors that are submitted but DMA channel is disabled before UDMA got a chance to execute these descriptors.

```
uint64_t            pDesc;

/* Flush any pending request from the free queue */
while(1)    {
    tempRetVal = Udma_ringFlushRaw(
                    Udma_chGetFqRingHandle(chHandle), &pDesc);
    if(UDMA_ETIMEOUT == tempRetVal)
        break;
}
```

5. Close DMA channel handle

```
retVal += Udma_chClose(chHandle);
```

3.3 Migration from EDMA LLD to UDMA LLD

This section compares the SW APIs between EDMA and UDMA to aid in migrating applications from EDMA to UDMA. When migrating applications from EDMA to UDMA, users are recommended to refer to EDMA sequence documented in 3.1 EDMA SW API Overview and replace the EDMA sequence of APIs in their application with corresponding sequence of UDMA APIs described in 3.2 UDMA SW API Overview.

Table 6. Comparison of EDMA and UDMA SW API

	EDMA SW API	UDMA SW API
Package	edma3_lld_xx_xx_xx_xx\packages\ti\sdo\edma3\	pdk_xx_xx_xx_xx\packages\ti\drv\udma
Interface header files	#include <ti/sdo/edma3/drv/edma3_drv.h> #include <ti/sdo/edma3/rm/edma3_rm.h> #include <ti/sdo/edma3/rm/src/edma3resmgr.h>	#include <ti/drv/udma/udma.h>

Programming Step	EDMA SW API		UDMA SW API	
	Data Structure	Function	Data structure	Function
DMA driver init	EDMA3_DRV_InitConfig EDMA3_RM_MiscParam EDMA3_DRV_GblConfigParams EDMA3_DRV_InstanceInitConfig EDMA3_DRV_Handle	EDMA3_DRV_create EDMA3_DRV_open	Udma_DrvObj Udma_DrvHandle Udma_InitPrms	UdmaInitPrms_init Udma_init
DMA channel open	EDMA3_DRV_Handle	EDMA3_DRV_requestChannel	Udma_ChObj Udma_ChHandle Udma_ChPrms	UdmaChPrms_init Udma_chOpen
DMA channel setup	EDMA3_DRV_Handle EDMA3_DRV_PaRAMRegs	EDMA3_DRV_clearErrorBits EDMA3_DRV_getPaRAMPhyAddr	Udma_ChHandle Udma_ChTxPrms Udma_ChRxPrms	UdmaChTxPrms_init UdmaChRxPrms_init Udma_chConfigTx Udma_chConfigRx Udma_chEnable
DMA interrupt register	Done by application based on the CPU on which the SW is running		Udma_DrvHandle Udma_ChHandle Udma_EventObj Udma_EventHandle Udma_EventPrms	UdmaEventPrms_init Udma_eventRegister
DMA transfer descriptor setup	Set PaRAM fields in EDMA MMRs		Set TR Descriptor header and TR in memory	
DMA trigger	EDMA3_DRV_Handle	EDMA3_DRV_enableTransfer	Udma_ChHandle	Udma_ringQueueRaw Udma_chGetFqRingHandle
DMA wait	EDMA3_DRV_Handle	EDMA3_DRV_waitAndClearTcc EDMA3_DRV_checkAndClearTcc	Udma_ChHandle	Udma_ringDequeueRaw Udma_chGetFqRingHandle
DMA channel close	EDMA3_DRV_Handle	EDMA3_DRV_freeChannel	Udma_ChHandle Udma_EventHandle	Udma_chDisable Udma_eventUnRegister Udma_chClose
DMA driver deinit	EDMA3_DRV_Handle	EDMA3_DRV_close EDMA3_DRV_delete	Udma_DrvHandle	Udma_deinit

3.4 Using UDMA 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. This is because PDK drivers already configures and uses UDMA internally. The application would only need to perform the UDMA driver initialization as described in 3.2.1 Steps to initialize and deinitialize UDMA driver. Please refer to the driver unit level tests and/or examples present in Processor SDK RTOS for sample usage.

4 Summary

This application note gave an overview of EDMA and UDMA from a HW and SW perspective. It described and compared the programming model, SW API of EDMA and UDMA. Migrating applications from EDMA to UDMA involves having a brief understanding the HW characteristics of UDMA as compared to EDMA. An application writer should identify the sequence of EDMA SW APIs 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 replace these with equivalent sequence of APIs from UDMA driver as described in this app note. Finally refer to TRMs and API guide for more details about UDMA programming.

5 References

[1] AM5x Technical Reference Manual – EDMA / DMA controller chapter

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

[3] AM5x Processor SDK RTOS User Guide – EDMA LLD section

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

6 Revision History

Date	Revision	Changes
28 Jun 2018	1.00	First draft
1 Aug 2018	1.01	Updated Figure 2, Figure 3, section 2.2