

Nithyaa Shri R B, Sunil Kumar M S

Abstract

This application note describes the Complex Device Driver – Direct Memory Access (CDD DMA) module for Texas Instruments' SMPU devices. This document covers the fundamental concepts of DMA, the types and modes of DMA transfers supported, the key hardware components (RINGACC, PSIL, Descriptors, and Interrupt Aggregator), the software architecture of the CDD DMA driver, integration into AUTOSAR environment, and practical use-cases such as DMA-accelerated Flash read and UART transmission operations.

Table of Contents

1 Introduction	2
1.1 Understanding the Purpose of DMA	2
1.2 How does DMA work?	2
1.3 Why CDD DMA?	2
1.4 CDD Module Placement in the AUTOSAR BSW Stack	3
2 Types of DMA	5
2.1 Block Copy DMA (BCDMA)	5
2.2 Packet DMA – Peripheral TX (PKTDMA-TX)	7
2.3 Packet DMA – Peripheral RX (PKTDMA-RX)	7
2.4 Polling Mode Versus Interrupt Mode	9
2.5 DMA Channel Chaining	9
3 Hardware Architecture: Inside the DMA Factory	11
3.1 Ring Accelerator (RINGACC)	11
3.2 Packet Streaming Interface Link (PSIL)	13
3.3 Descriptors: [TRPD and HPD]	14
3.4 Interrupt Aggregator	15
4 Software Overview	18
4.1 Three-Layer Driver Architecture	18
4.2 Memory Requirements	19
4.3 AUTOSAR Integration	19
5 Use Cases	23
5.1 Flash Copy with DMA	23
5.2 Use Case: Amplifier Data Transmission Using CDD UART With DMA	25
6 Step-by-Step Guide to Write a CDD DMA Application	27
6.1 Step 1 - Configure EB Tresos	27
6.2 Step 2 - Allocate Memory (128-Byte Aligned, in DDR)	27
6.3 Step 3 - Initialize the Driver	27
6.4 Step 4 - Open and Enable the Channel	27
6.5 Step 5 - Submit the Transfer	27
6.6 Step 6 - Wait for Completion	27
6.7 Step 7 - Clean Up	27
6.8 Common Mistakes to Avoid	27
7 Summary	28
8 References	29
8.1 Example Applications (in MCAL)	29

Trademarks

Sitara™ is a trademark of Texas Instruments.

All trademarks are the property of their respective owners.

1 Introduction

Direct Memory Access (DMA) is a hardware mechanism that allows peripherals or memory subsystems to transfer data directly to/from system memory without involving the CPU for every byte moved.

Data movement tasks are commonly offloaded from the host processor to peripheral hardware to increase system performance. The CPU simply sets up the transfer, and the DMA hardware handles the rest - freeing the CPU to execute application code in parallel.

1.1 Understanding the Purpose of DMA

Interactive Example 1:

Think of DMA as your neighborhood's smart package delivery service. Instead of you carrying every package from point A to point B (which would be the CPU doing data transfers), you have a more efficient system:

1. You ring the doorbell (trigger a DMA transfer request)
2. You leave instructions (provide source, destination, and transfer details in a descriptor)
3. You go about your day (CPU is free to handle other tasks)
4. The delivery service handles everything (DMA manages the actual data transfer)
5. You get notified when done (interrupt or polling for completion notification)

The following sections detail how this is achieved technically.

1.2 How does DMA work?

In a traditional CPU-driven transfer, the processor fetches data from a source (for example, a peripheral or flash memory) and writes the data to a destination (for example, a RAM buffer) one word at a time. For large transfers, this wastes valuable CPU cycles.

With DMA, the transfer is described once in a small data structure called a descriptor and handed to the DMA controller. The DMA controller executes the transfer autonomously and notifies the CPU only on completion (via interrupt) or when the CPU polls a status register.

Table 1-1. CPU Transfer Versus DMA Transfer

Without DMA	With DMA
CPU reads and writes every byte	CPU writes a descriptor and waits
CPU is busy during the entire transfer	CPU is free to run other tasks
Latency increases with transfer size	Latency is deterministic
No parallel jobs possible	CPU and DMA work in parallel

1.3 Why CDD DMA?

Complex Device Driver – Direct Memory Access (CDD DMA) is the AUTOSAR-compliant DMA driver used in Texas Instruments' MCAL software. CDD DMA abstracts the TI Sitara™ series Unified DMA (UDMA) hardware and provides a clean, portable API for:

- Memory-to-memory block copies (BCDMA)
- Peripheral TX/RX data movement (PKTDMA) (for example, UART, MCSPI, and OSPI)
- DMA-accelerated Flash reads using the FLS driver
- Multi-stage automated pipelines using channel chaining

CDD DMA is a complex device driver (CDD) because CDD DMA provides functionality beyond the standard AUTOSAR modules while still following AUTOSAR coding and configuration conventions.

1.4 CDD Module Placement in the AUTOSAR BSW Stack

As defined in the AUTOSAR Layered Software Architecture specification, a CDD is a software entity that is not standardized by AUTOSAR, but can access or be accessed through AUTOSAR interfaces and Basic Software Module APIs. A CDD is a specific module located in the Complex Drivers Layer of the Basic Software (BSW) and is permitted to interact with standard BSW modules or the Runtime Environment (RTE). The following interaction patterns are defined for a CDD:

- A CDD can need to interface with modules of the AUTOSAR layered software architecture
- A module of the layered software architecture can need to interface with a CDD
- A CDD can need to interface with Software Components (SW-Cs) through the RTE

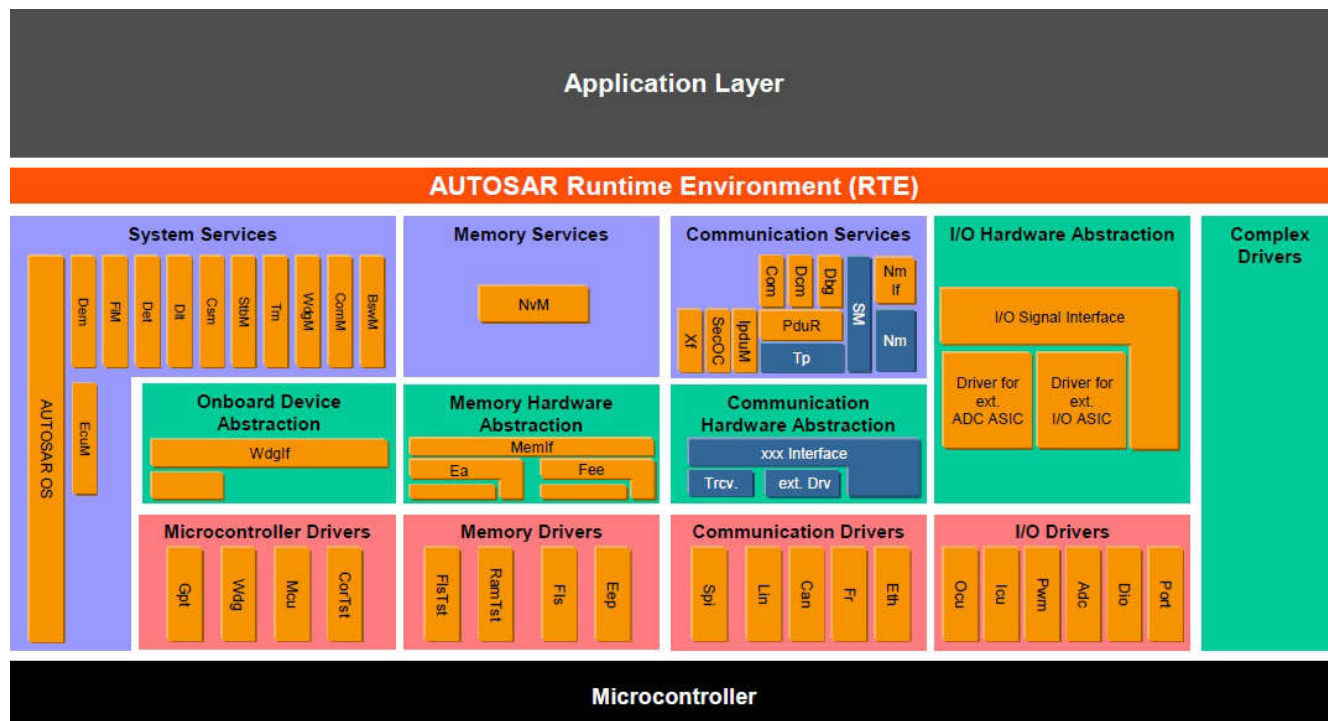


Figure 1-1. AUTOSAR STACK

The DMA controller on TI AM6x and AM275x devices does not correspond to any standardized AUTOSAR MCAL module; no Software Specification (SWS) exists for a DMA driver within the AUTOSAR standard. Consequently, the CDD classification is the architecturally correct choice: CDD DMA provides a hardware-specific interface to the DMA subsystem (data movement subsystem, or DMSS) that cannot be represented through any existing AUTOSAR-standardized module interface.

Within the AUTOSAR BSW stack, CDD DMA (Module ID: 255, Vendor ID: 44) is positioned in the Complex Drivers Layer. CDD DMA resides below the AUTOSAR RTE and above the hardware abstraction registers (CSL or hardware include layer). Higher-level MCAL drivers, specifically the SPI driver and the Flash (FLS) driver, interface with CDD DMA as BSW clients, invoking its channel management APIs (Cdd_Dma_ChOpen, Cdd_Dma_ChEnable, Cdd_Dma_StartTransfer) to perform DMA-accelerated data movement. [Figure 1-2](#) shows this layer:

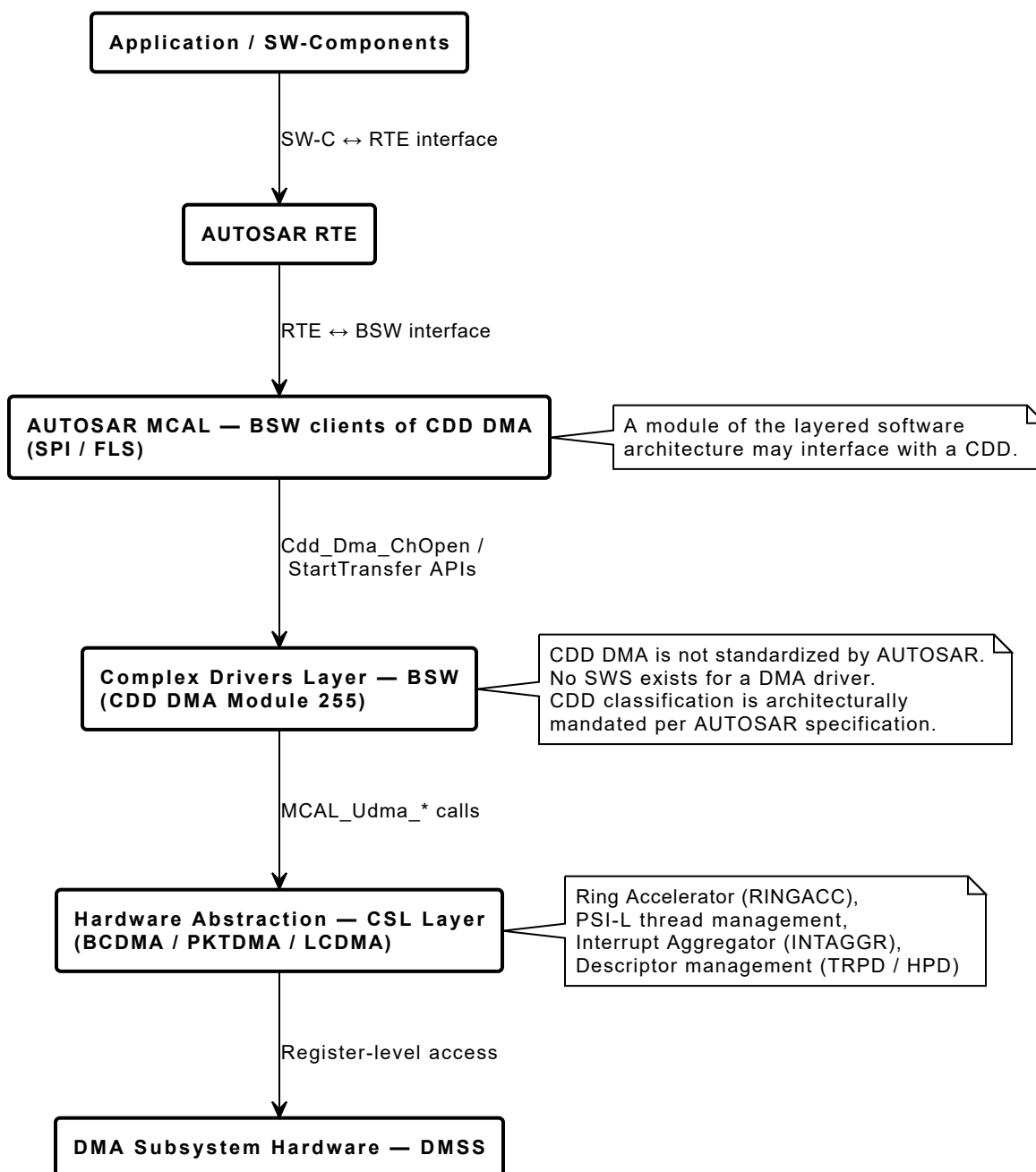


Figure 1-2. CDD DMA Placement in AUTOSAR STACK

CDD DMA centralizes all DMA resource management – including ring accelerator allocation, channel life cycle control, descriptor (TRPD/HPD) management, PSIL thread pairing, and interrupt aggregator routing – into a single BSW module. This design makes sure that multiple MCAL drivers sharing the same DMA subsystem do not independently manage hardware resources, in accordance with the AUTOSAR principle of single-point resource ownership within the BSW.

2 Types of DMA

The CDD DMA driver supports three transfer types, controlled by the EB Tresos parameter `Cdd_Dma_TransType`. Each type targets a different data-movement scenario.

A DMA system can:

- Handle different types of transfers (memory-to-memory, peripheral-to-memory, and so forth)
- Automatically chain operations: when one transfer completes, the next one starts
- Notify the CPU through interrupts, based on configuration
- Manage multiple concurrent channels efficiently

2.1 Block Copy DMA (BCDMA)

Block Copy DMA performs memory-to-memory transfers. Data moves directly from one memory location to another (for example, DDR to DDR). This is the simplest DMA mode and is practical for large buffer copies, data reshuffling.

Table 2-1. BCDMA Attributes

Transfer direction	Memory → Memory
Descriptor used	Transfer Request Packet Descriptor (TRPD)
Peripheral involved	OSPI
EB Tresos TransType	CDD_DMA_BC_MODE/ CDD_DMA_PERIPH_TX_MODE(OSPI)
DMA engine	BCDMA
Typical use	Copying large RAM buffers, OSPI DAC read
Channel chaining	Supported

2.1.1 BCDMA Software Flow

CDD DMA BCDMA Flow

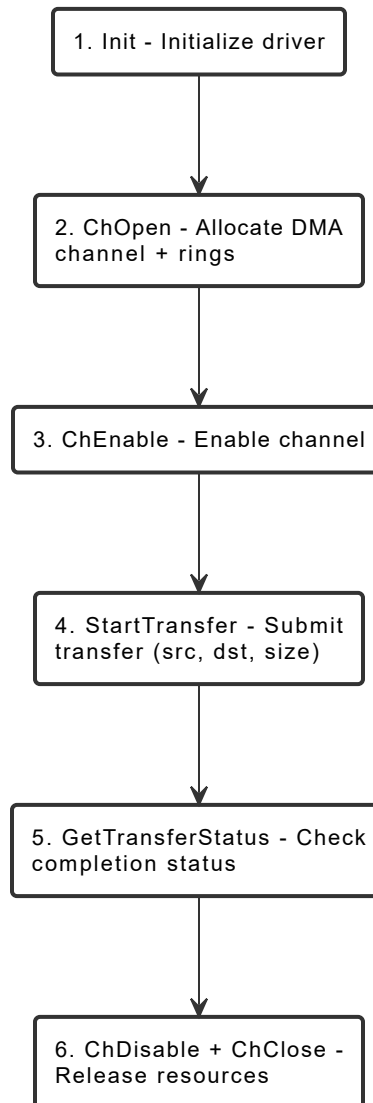


Figure 2-1. Block Copy DMA Transfer Flow

2.2 Packet DMA – Peripheral TX (PKTDMA-TX)

Peripheral TX mode transfers data from system memory to a peripheral device (for example, UART transmit, OSPI write). The DMA controller uses the Packet Streaming Interface Link (PSIL) to connect to the hardware FIFO of the peripheral.

Table 2-2. Peripheral TX DMA Attributes

Attribute	Value
Transfer direction	Memory → Peripheral
Descriptor used	Host Packet Descriptor (HPD) driver managed
Peripheral involved	UART, SPI
EB Tresos TransType	CDD_DMA_PERIPH_TX_MODE
DMA engine	Packet DMA (PKTDMA)
Typical use	UART transmit, (as TX DMA)
PSIL thread	Required (peripheral TX thread)

2.3 Packet DMA – Peripheral RX (PKTDMA-RX)

Peripheral RX mode transfers data from a peripheral device into system memory. The DMA hardware reads incoming data from the peripheral FIFO and writes the data to the buffer of the application, notifying completion through interrupt or status polling.

Table 2-3. Peripheral RX DMA Attributes

Attribute	Value
Transfer direction	Peripheral → Memory
Descriptor used	Host Packet Descriptor (HPD) driver managed
Peripheral involved	UART, SPI
EB Tresos TransType	CDD_DMA_PERIPH_RX_MODE
DMA engine	Packet DMA (PKTDMA)
Typical use	UART receive, SPI RX
PSIL thread	Required (peripheral RX thread)

2.3.1 PKTDMA Software Flow

CDD DMA PKTDMA Flow

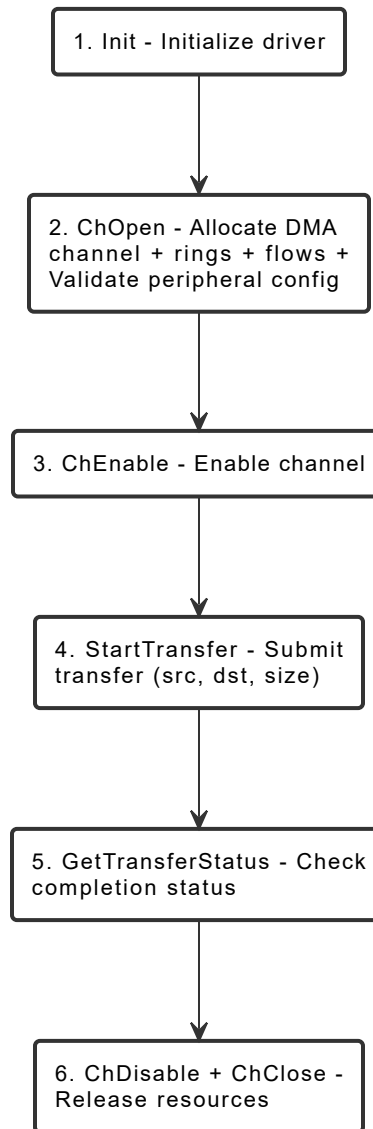


Figure 2-2. Packet DMA Transfer Flow

Note

The application does *not* call CDD DMA APIs directly for peripheral TX/RX. The peripheral driver (for example, UART) calls CDD DMA API internally. The application only calls Dma_Init before initializing the peripheral driver.

2.4 Polling Mode Versus Interrupt Mode

Regardless of transfer type, CDD DMA supports two completion-notification modes. The mode is selected in EB Tresos through the `Cdd_Dma_Mode` parameter.

Table 2-4. Polling Mode Versus Interrupt Mode

	Polling Mode	Interrupt Mode
How completion is detected	Application calls <code>GetTransferStatus</code> with a defined timeout	ISR fires when DMA completes; callback is invoked
CPU usage during transfer	CPU spends cycles polling	CPU is free; ISR notifies on completion
Latency	Can add loop overhead	Minimal - reaction on hardware event
When to use	Simple, non-time-critical transfers	High-performance or real-time applications
EB Tresos setting	<code>CDD_DMA_MODE_POLLING</code>	<code>CDD_DMA_MODE_INTERRUPT</code>

2.5 DMA Channel Chaining

Channel Chaining allows more than one DMA transfer to execute sequentially and automatically. When the first channel (trigger channel) completes, it triggers the second channel (chained channel) based on the trigger event configured - with zero CPU intervention. This is powerful for multi-stage pipelines.

CDD DMA Chaining Flow

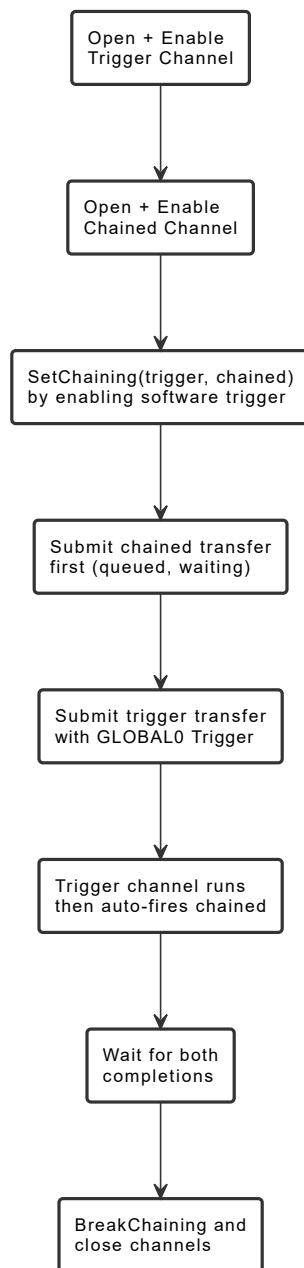


Figure 2-3. DMA Channel Chaining Flow

Note

Only TRIGGER_GLOBAL0 is currently supported and tested in CDD DMA.

3 Hardware Architecture: Inside the DMA Factory

Let us take a guided tour inside the CDD DMA hardware to understand how it works:

The CDD DMA driver manages four key hardware components of the AM62x DMSS. Understanding these components helps explain how a DMA transfer is set up, executed, and signaled.

3.1 Ring Accelerator (RINGACC)

The Ring Accelerator (RINGACC) is a hardware module that manages circular queues (rings) in system memory.

The RINGACC removes the burden of pointer arithmetic from software - instead, software simply pushes a descriptor pointer into the ring using the Forward Doorbell Register and pops from ring using the Reverse Doorbell register, and the hardware maintains all head and tail pointers and occupancy counts automatically.

Every DMA channel has two rings:

Table 3-1. Free Queue and Completion Queue Rings

Ring	Full Name	Direction	Purpose
FQ Ring	Free Queue Ring	Software → DMA	Application places descriptor pointers here to kick off a transfer
CQ Ring	Completion Queue Ring	DMA → Software	DMA hardware places completed descriptor pointers here when a transfer finish

3.1.1 Key Characteristics

- **Dual Queue System:** Each DMA channel manages two logical queues
- **Free Queue (FQ):** Contains pointers to descriptors ready for work
- **Completion Queue (CQ):** Contains pointers to completed descriptors with results
- **Automatic Management:** Hardware handles queue pointer updates - no CPU intervention needed for basic operations
- **Memory Efficient:** Queue elements are simply 64-bit pointers to descriptor memory
- **Cache Optimized:** All ring memory must be 128-byte cache line aligned for best performance

The typical ring interaction for a Block Copy transfer is shown in [Figure 3-2](#):

CDD DMA Ring / Descriptor Flow

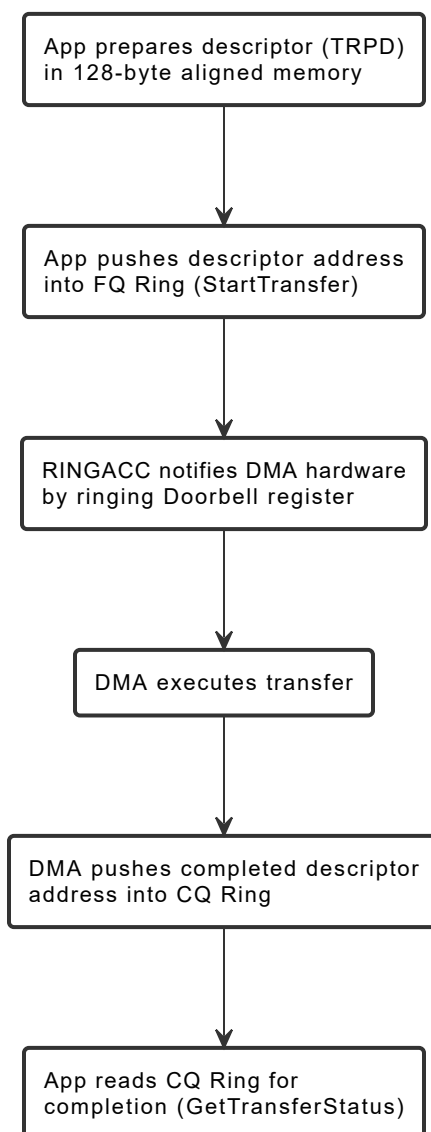


Figure 3-1. Ring Accelerator Interaction Flow

Note

Ring memory must be allocated in DDR and *must* be 128-byte cache-line aligned. Misalignment causes silent data corruption. Each ring element is eight bytes (a 64-bit descriptor pointer).

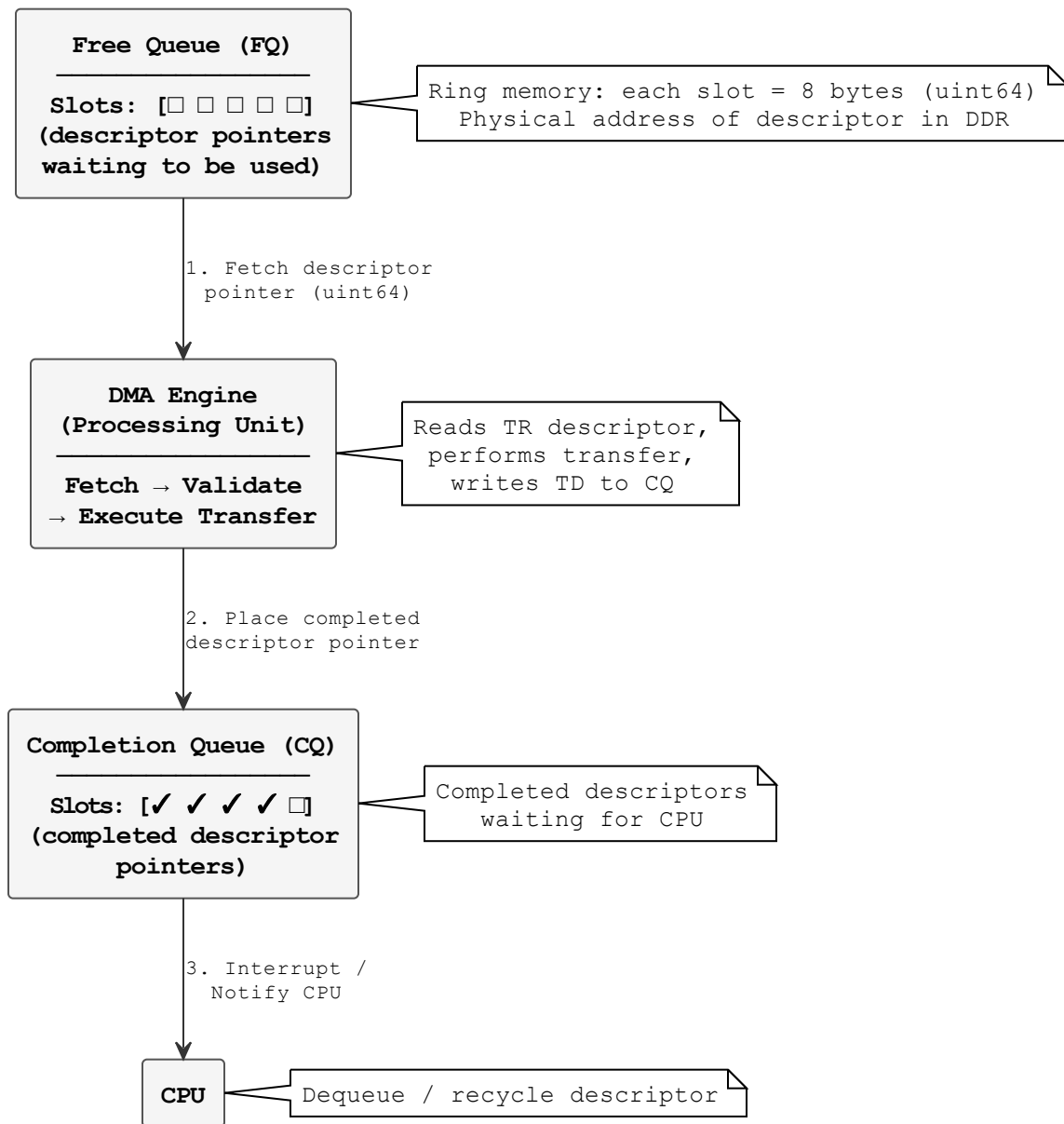


Figure 3-2. Ring Accelerator (RINGACC) Descriptor Flow Between Free Queue, DMA Engine, and Completion Queue

3.2 Packet Streaming Interface Link (PSIL)

"This is where DMA connects to peripherals like UART, SPI."

PSIL is a packet-based, point-to-point interconnect fabric inside the SoC that connects DMA channels to peripheral modules (UART, SPI). Think of PSIL as a dedicated data highway between the DMA engine and the peripheral FIFO.

Every PSIL endpoint is identified by a Thread ID. Before any data can flow, a TX thread (DMA side) must be paired with a peripheral RX/TX thread. This pairing is done using Sciclient and is handled automatically by CDD DMA during ChOpen.

Table 3-2. PSIL Thread Concepts

Concept	Explanation
Source Thread (0x0000 – 0x7FFF)	Sends data; used for DMA TX channels
Destination Thread (0x8000 – 0x8FFF)	Receives data; used for DMA RX channels
Thread Pairing	Links a DMA TX thread to a peripheral RX thread (or vice versa)

3.2.1 Key Characteristics

- Thread Pairing System:
 - Each peripheral device connects to DMA via a unique TX/RX thread pair
 - Peripheral drivers calculate the "peer channel number" (thread offset + channel ID)
- Bidirectional Support: Separate logical paths for transmit (memory → peripheral) and receive (peripheral → memory)
- Dynamic Connection: Channels can be connected/disconnected on-demand through ChEnable/ChDisable

Note

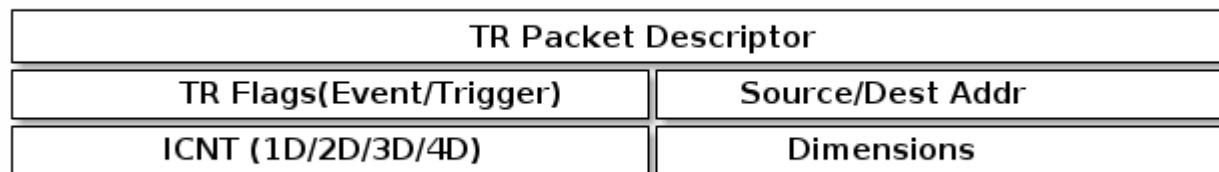
For Block Copy (BCDMA) mode, PSIL is NOT used. The peerChNum parameter must be set to INVALID VALUE(0xFFFF0000U).

3.3 Descriptors: [TRPD and HPD]

A descriptor is a small data structure in RAM that tells the DMA hardware what to do: source address, destination address, how many bytes to transfer, and how to signal completion. CDD DMA uses two types of descriptors depending on the transfer mode.

3.3.1 Transfer Request Packet Descriptor (TRPD) - For Memory-to-Memory Moves

TRPD is used exclusively in Block Copy (BCDMA) mode. TRPD describes a multi-dimensional memory transfer using nested loop counts (ICNT0 to ICNT3). The application is responsible for allocating TRPD memory and passing it to StartTransfer API.

**Figure 3-3. TRPD Structure**

The responsibility of the application in BCDMA mode is to provide a TRPD memory that must be 128-byte cache-line aligned, which contains all the information needed for the transfer [Cdd_Dma_ParamsType]; that is, source and destination buffer addresses and transfer count and dimensions (1D line to 4D hypercube). The application can trigger settings for chaining operations and configure flags for interrupt generation and so forth.

Note

Maximum transfer size per TRPD: $ICNT0 \times ICNT1 \times ICNT2 \times ICNT3 \leq 65,535$ bytes. For transfers larger than 64KB, split into multiple StartTransfer() calls or use multiple ring elements. TRPD memory must be 128-byte cache-line aligned. ICNT is configured via EB Tresos.

3.3.2 Host Packet Descriptor (HPD) - For Peripheral Communications

HPD is used in Peripheral TX and RX (PKTDMA) mode. Unlike TRPD, the driver initializes the HPD internally. The application provides only the data buffer and the length of the buffer. The driver fills in source and destination pointers and all other fields automatically.

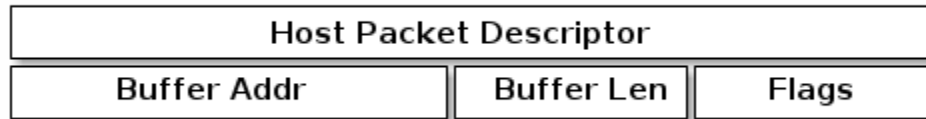


Figure 3-4. HPD Structure

The driver manages HPD in peripheral mode, which is 128-byte cache-line aligned. It contains buffer pointers, packet information, and transfer length.

Table 3-3. TRPD Versus HPD Comparison

	TRPD	HPD
Used in	Block copy (BCDMA)	Peripheral TX/RX (PKTDMA)
Who initializes	Application (Allocates memories)	CDD DMA driver
Transfer endpoint	DDR ↔ DDR	DDR ↔ Peripheral FIFO
Max transfer size	64KB per descriptor	Up to 4MB per descriptor

3.4 Interrupt Aggregator

The Interrupt Aggregator (INTAGGR) is the hardware block that converts DMA completion events into CPU interrupts. Without the INTAGGR, every DMA channel needs a dedicated CPU interrupt line, which can quickly exhaust the available interrupt lines on the SoC.

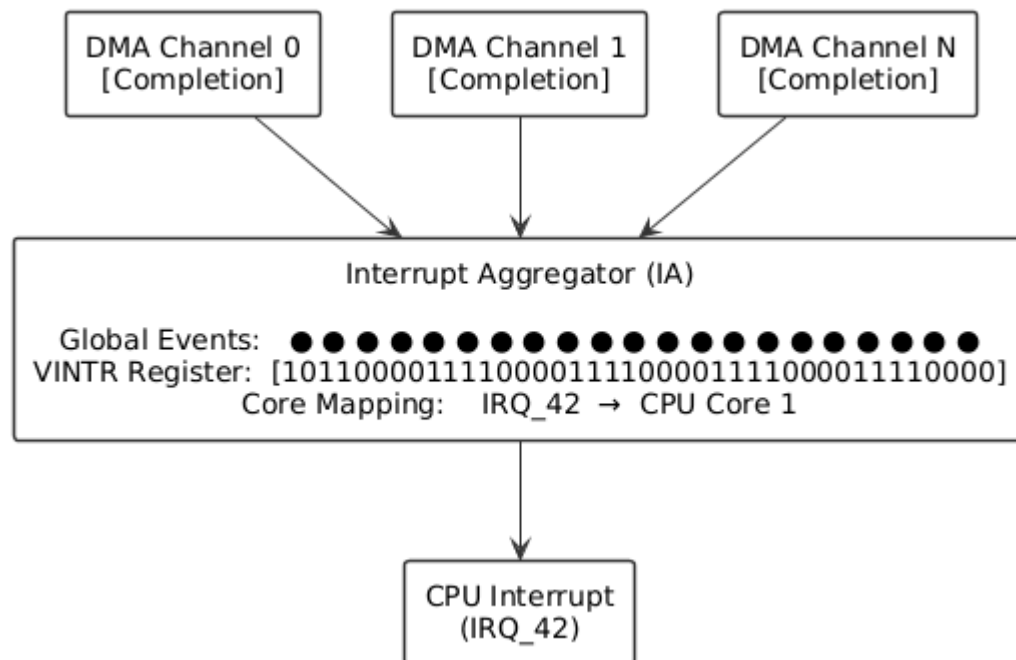


Figure 3-5. Interrupt Aggregator Flow

Table 3-4. Interrupt Aggregator Concepts

IA Concept	Explanation
Global Event	A virtual event generated by a DMA channel when its transfer completes. Identified by a global event number.
VINTR (Virtual Interrupt)	A 64-bit status register. Each bit corresponds to one global event. Up to 64 events can share one VINTR.
Core Interrupt	The actual hardware interrupt delivered to the CPU. One core interrupt can service all 64 events in a single VINTR register.
Controller Event	Owns the VINTR register and core interrupt. ISR is registered against the controller event.
Peripheral Event	Shares the controller's VINTR register. Each peripheral occupies a unique bit in the 64-bit VINTR.
Shared Event Mode	Multiple DMA channels share one VINTR. The ISR reads the VINTR, identifies which bit is set, and dispatches the right callback.

Figure 3-6 shows the event flow from DMA completion to CPU interrupt:

CDD DMA Interrupt Flow

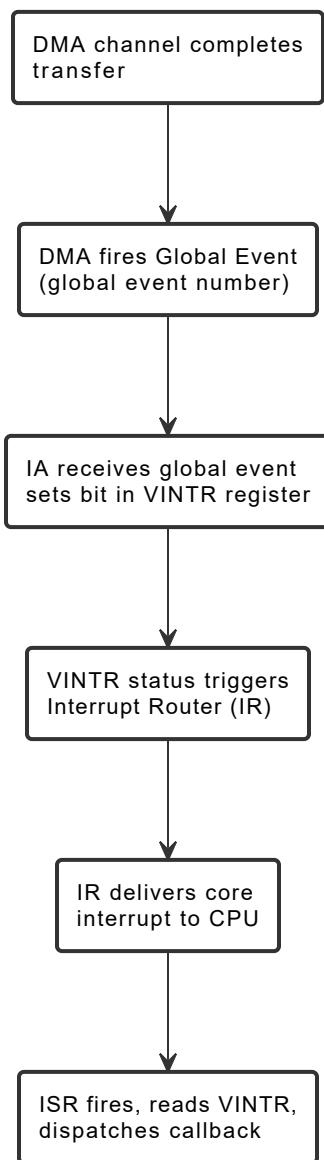


Figure 3-6. DMA Event-to-Interrupt Flow

3.4.1 Key Characteristics

- **Event Consolidation:** Many DMA completion events → Fewer CPU interrupts
- **Virtual Interrupts (VINTRs):** Software-interpreted bits that map to hardware interrupts
- **Shared Notification Efficiency:** Multiple channels can share one VINTR register
- **Controller-Peripheral Relationships:**
 - **Controller Events:** Own VINTR bits and core interrupt mappings
 - **Peripheral Events:** Linked to masters, occupy unique VINTR bits only
- **Interrupt Reduction:** Instead of N separate interrupts for N channels, you might need only fewer

4 Software Overview

This section describes the CDD DMA software architecture, driver objects, public API, EB Tresos configuration parameters, and memory requirements.

4.1 Three-Layer Driver Architecture

The CDD DMA software is organized into three layers. Each layer has a single, well-defined responsibility and communicates only with adjacent layers.

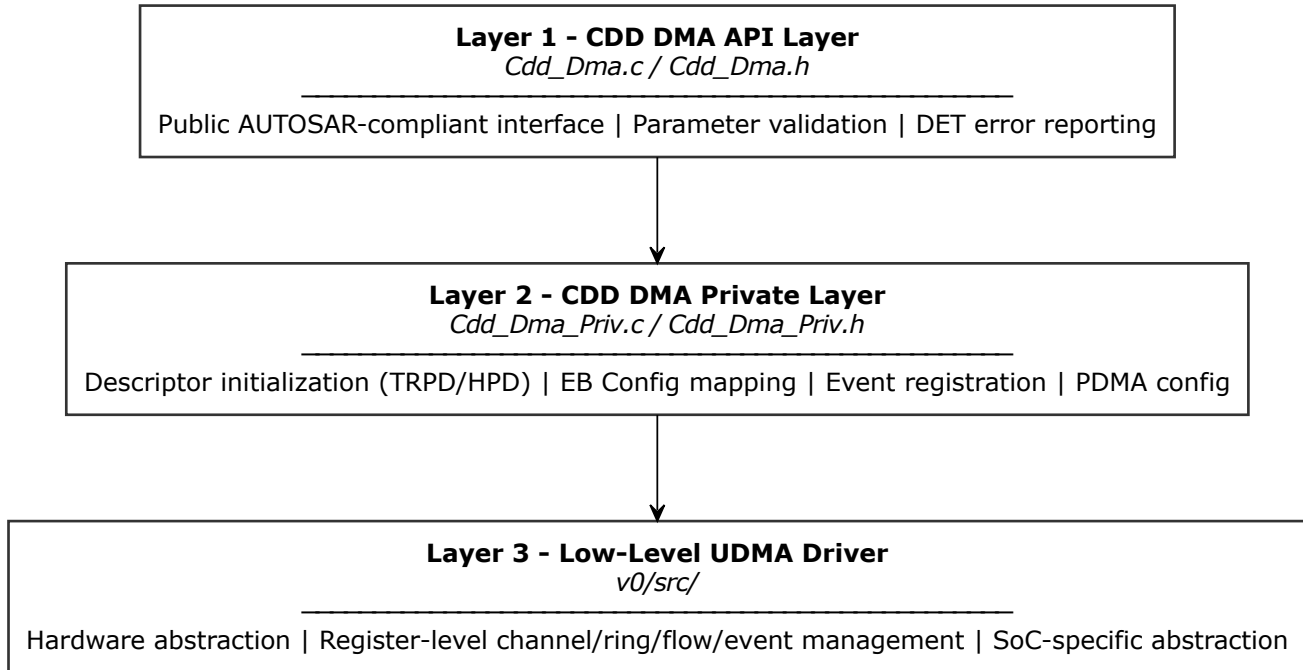


Figure 4-1. CDD DMA Three-Layer Architecture

4.1.1 API Layer

This is the interface through which applications interact with the DMA service.

4.1.1.1 Key Responsibilities

- **Request Validation:** Makes sure all parameters are sensible (such as checking that an address is not in restricted memory)
- **Standardized Interface:** Consistent function names and parameter structures across use cases
- **Error Translation:** Converts low-level hardware issues into understandable error codes
- **Resource Tracking:** Keeps track of what has been allocated and what is available

4.1.2 Private Layer

This layer handles the preparation and coordination work.

4.1.2.1 Key Responsibilities

- **Descriptor Preparation:** Converts high-level transfer requests into TRPD/HPD formats
- **Configuration Mapping:** Translates EB Tresos settings into hardware register values
- **Event Management:** Sets up notification preferences (interrupts, polling, and so forth)
- **Resource Coordination:** Makes sure all needed pieces (channels, rings, events) are available and compatible

4.1.3 Low-Level Driver

This is where the actual work happens.

4.1.3.1 Key Responsibilities:

- **Hardware Abstraction:** Presents a consistent interface regardless of underlying silicon variations
- **Register-Level Control:** Directly manages DMA hardware registers for best performance
- **Resource Management:** Actually, allocates and configures channels, rings, flows, and events
- **SoC-Specific Optimizations:** Takes advantage of hardware features and capabilities

4.2 Memory Requirements

All memory allocated for DMA operations must be in DDR (not R5F TCM) and must be 128-byte cache-line aligned. Misalignment leads to data corruption.

Table 4-1. DMA Memory Requirements

Memory Type	Who Allocates	Alignment	Notes
Ring memory (ringMem)	Application (Block Copy) / Peripheral driver (Peripheral mode)	128 bytes	Size = ringElemCount × 8 bytes
TRPD descriptor memory	Application	128 bytes	Block Copy only; minimum 256 bytes recommended
HPD descriptor memory	Peripheral driver	128 bytes	Peripheral mode only; driver manages content
TX data buffer	Application	128 bytes	Source data for TX and Block Copy
RX data buffer	Application	128 bytes	Destination buffer; must be in DDR (not TCM)

Note

Use `__attribute__((aligned(128)))` in C to verify 128-byte alignment for all DMA buffers. Buffers in R5F ATCM or Tightly Coupled Memory (BTCM) cannot be accessed by the DMA hardware. The FLS driver automatically falls back to CPU copy if the destination buffer is in TCM.

4.3 AUTOSAR Integration

This section covers AUTOSAR-specific integration requirements for the CDD DMA module, including Schedule Manager exclusive area usage, error reporting strategy, boot/shutdown sequencing, ISR category configuration, memory protection considerations, and inter-module initialization dependencies.

4.3.1 Schedule Manager (SchM) Integration

Per the AUTOSAR BSW specification, *"a CDD shall protect its critical resources by defining critical sections that are handled by the Schedule Manager (SchM) or OS mechanisms."*

In conformance with this requirement, CDD DMA uses the Schedule Manager to protect critical resources accessed concurrently from task and ISR contexts; for example, the ring queue and dequeue operations within the UDMA driver, which are shared state that must remain consistent across preemptive execution contexts.

The module defines the exclusive area for this purpose, making sure that all access to these critical resources are guarded by AUTOSAR-compliant critical section entry and exit calls.

Internally, two SchM wrapper functions are defined in `Cdd_Dma_Priv.c` and registered as OSAL callbacks during initialization:

```
SchM_Enter_Cdd_Dma_DMA_EXCLUSIVE_AREA_0(); /* disable interrupts */
SchM_Exit_Cdd_Dma_DMA_EXCLUSIVE_AREA_0(); /* restore interrupts */
```

These callbacks are passed to the underlying UDMA driver at DMA initialization time, so all internal UDMA resource allocation and ring operations use AUTOSAR-compliant critical section protection.

Integrators must make sure that `SchM_Cdd_Dma.h` is generated by the AUTOSAR toolchain and that the exclusive area `DMA_EXCLUSIVE_AREA_0` is defined in the OS configuration. For systems using an

AUTOSAR OS, this exclusive area must be mapped to a SUSPEND_ALL_INTERRUPTS or interrupt-level-based implementation depending on the timing requirements of the application.

4.3.2 Development Error Tracer (DET)

The current CDD DMA implementation uses the Development Error Tracer (DET) for error reporting during development. The following DET error codes are defined:

DET reporting is conditionally compiled through CDD_DMA_DEV_ERROR_DETECT and is intended for development builds only.

CDD DMA — DET Error Codes (Development Error Tracer)

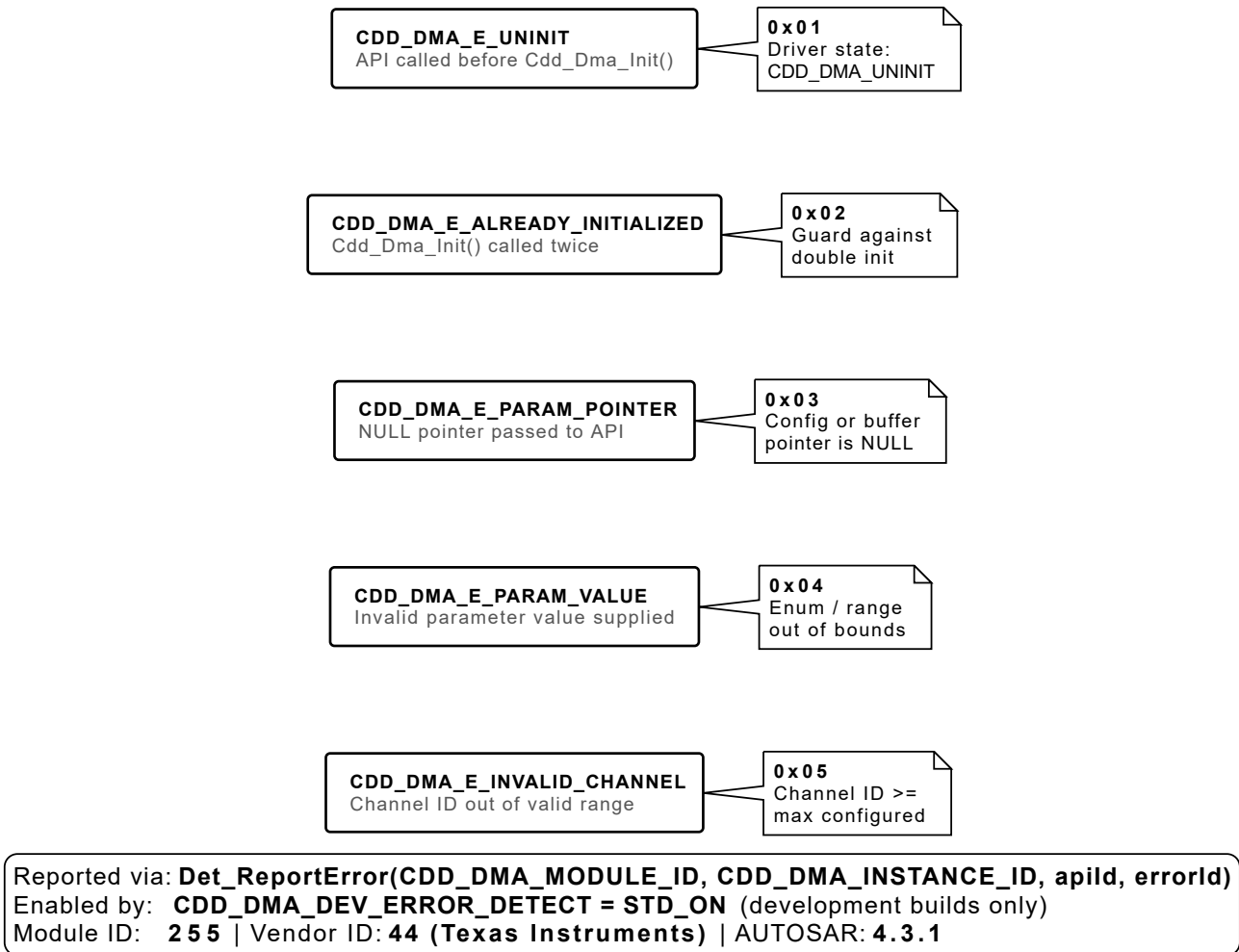


Figure 4-2. CDD DMA DET Errors

4.3.3 ECU Manager (EcuM) Integration and Boot Sequence

CDD DMA does not internally hook into the EcuM module. The integrator is responsible for calling Cdd_Dma_Init () and Cdd_Dma_DeInit () at the correct points in the AUTOSAR boot and shutdown sequence.

Startup: Cdd_Dma_Init () must be called during the EcuM_StartupTwo phase, after the following prerequisites are satisfied:

1. Mcu_Init () has completed - device clocks, power domains, and base addresses are configured.
2. The System Controller Interface (TISCI/ScClient) is operational - CDD DMA relies on TISCI to allocate Ring Accelerator resources and configure IRQ routing.

Shutdown: Cdd_Dma_DeInit () must be called during the EcuM_GoDownPhase, before MCU clock gating or power domain shutdown. Any in-progress DMA transfers must be completed or aborted before calling Cdd_Dma_DeInit (). Channels must be individually closed through Cdd_Dma_ChClose () before global de-initialization.

Failing to respect this sequence can result in incomplete DMA transfers, or unserved interrupt events at next start-up.

4.3.4 AUTOSAR OS vs FreeRTOS - ISR Category Configuration

CDD DMA supports three ISR compilation modes, selected at configuration time through CDD_DMA_ISR_TYPE in Cdd_Dma_Cfg.h:

CDD DMA — ISR Compilation Modes (CDD_DMA_ISR_TYPE)

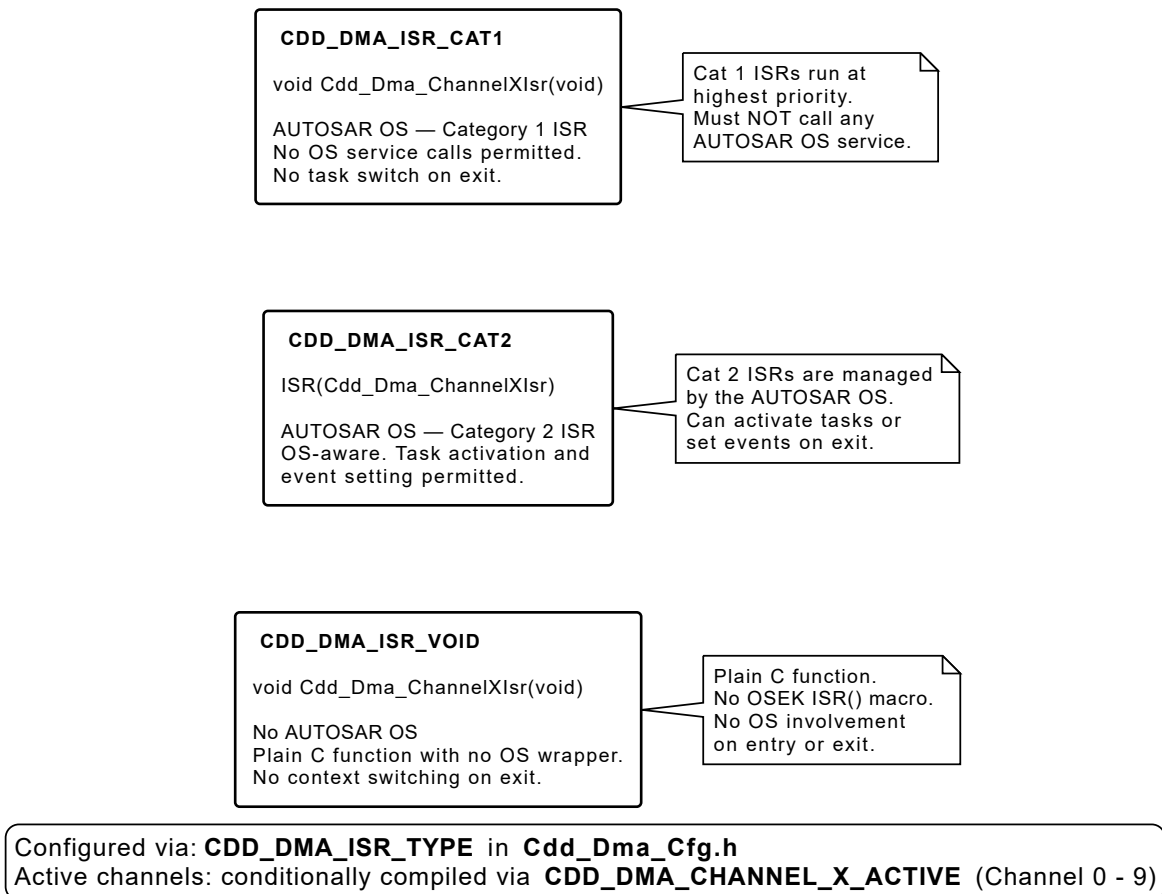


Figure 4-3. CDD DMA DET Errors

For deployments using an AUTOSAR OS:

- Set CDD_DMA_ISR_TYPE to CDD_DMA_ISR_CAT2 if the DMA ISR activates tasks or sets events.
- Set CDD_DMA_ISR_TYPE to CDD_DMA_ISR_CAT1 if the ISR only sets flags or manipulates shared variables within the interrupt context, with no OS service calls.
- Category 1 ISRs must not call any OS services. Violating this causes undefined OS behavior.

Each channel ISR (Cdd_Dma_Channel0Isr through Cdd_Dma_Channel9Isr) is conditionally compiled based on CDD_DMA_CHANNEL_X_ACTIVE. This CDD_DMA_CHANNEL_X_ACTIVE is generated only when CDD_DMA_INTERRUPT_MODE is configured for the respective peripheral. Only active channels generate ISR symbols, reducing code size.

4.3.5 Memory Protection Unit (MPU) Considerations for DMA Buffers

DMA operations on AM62x, AM62Px, AM275x, AM62Dx and AM62Ax devices operate on physical memory addresses. DMA hardware accesses memory directly, by passing the CPU cache. This imposes the following requirements on all DMA transfer buffers:

1. Cache line alignment

All Transfer Request Packet Descriptor (TRPD) and Host Packet Descriptor (HPD) buffers must be aligned to 128 bytes, matching the L3 cache line size on these devices (CDD_DMA_CACHELINE_ALIGNMENT = 128U). Buffers that are not 128-byte aligned cause descriptor fetch errors or silent data corruption.

2. Cache coherency

Application buffers used as DMA source or destination must be cache-coherent with the DMA engine. Before initiating a DMA write (CPU to peripheral), the application must flush the cache lines covering the source buffer. After a DMA read (peripheral to CPU), the application must invalidate the cache lines covering the destination buffer before reading the data. Failure to do so results in stale data.

3. MPU configuration

For systems with an active Memory Protection Unit, DMA descriptor memory regions (TRPD, HPD, ring memory) must be mapped as: accessible by the DMA hardware (not restricted to CPU-only access); non-cacheable or managed with explicit cache maintenance operations; and assigned appropriate privilege levels if the system enforces memory partitioning for ASIL isolation.

CDD DMA does not perform MPU configuration internally. The integrator must verify the linker places DMA descriptor buffers in a memory region with the correct MPU attributes prior to calling Cdd_Dma_Init ().

4.3.6 Inter-Module AUTOSAR Dependencies and Initialization Order

CDD DMA has implicit dependencies on several other BSW modules that must be satisfied before Cdd_Dma_Init () is called. These dependencies are not enforced by the module; the integrator must verify the following BSW initialization order:

1. **Step 1 - Mcu_Init ():**

Configures device clocks, power domains, and peripheral base addresses required by the DMA subsystem.

2. **Step 2 - ScClient / TISCI:**

Allocates Ring Accelerator resources, configures BCDMA/PKTDMA channel ownership, and programs the Interrupt Aggregator (INTAGGR) routing. CDD DMA initialization fails silently if TISCI is unavailable.

3. **Step 3 - Cdd_Dma_Init ():**

Initializes BCDMA and PKTDMA instances, registers SchM OSAL callbacks, and prepares the driver for channel open requests.

4. **Step 4 - Spi_Init() / Uart_Init() (or other DMA clients):**

Must be called after CDD DMA is initialized, as these modules call Cdd_Dma_ChOpen () during their own initialization.

Port_Init () is not required before Cdd_Dma_Init () because DMA channels are internal to the device subsystem and do not involve pad/pin configuration.

The ScClient callback structure (Cdd_Dma_ScClientType) must be populated in the generated configuration file (Cdd_Dma_Cfg.c) and point to valid TISCI proxy functions before Cdd_Dma_Init () is called. If TISCI callbacks are NULL, the UDMA initialization fails and the driver state remains CDD_DMA_UNINIT, causing all subsequent API calls to report CDD_DMA_E_UNINIT through DET.

5 Use Cases

5.1 Flash Copy with DMA

This section demonstrates how DMA is used in practice, using the most common CDD DMA use case: DMA-accelerated Flash (FLS) read operations through the OSPI peripheral.

5.1.1 Flash DMA Overview

The FLS (Flash) MCAL driver supports DMA-accelerated reads in DAC (Direct Access / memory-mapped) mode. When Flash DMA is enabled, the FLS driver uses CDD DMA to transfer data from the memory-mapped flash address to a DDR buffer - without the CPU reading and writing every byte.

Table 5-1. Flash Operations and DMA Support

Operation	DMA Supported?	Mode Required
Flash Read (DAC mode)	Yes	FlsDacEnable = true AND FlsDmaEnable = true
Flash Read (INDAC mode)	No	CPU-based transfer only
Flash Write	No	CPU-based transfer always
Flash Erase	No	CPU-based transfer always

5.1.2 How DMA Enhances Flash Performance

With CPU-based flash reads, the processor must fetch each word individually. For large blocks (for example, reading a 1MB firmware image), this ties up the CPU entirely. DMA removes this bottleneck:

Table 5-2. Flash Read Performance - CPU Versus DMA

Metric	CPU-Based DAC Read	DMA-Based DAC Read
CPU involvement	CPU reads every byte	CPU submits descriptor; DMA does the work
CPU available for other tasks	No	Yes - CPU is free during transfer
Read throughput for 2MB of data in interrupt mode	Approximately 3.471MB/s	Approximately 25.34MB/s

5.1.3 Initialization Sequence

The initialization order is critical. Fls_Init () calls Fls_OspiDmaOpen () internally to acquire a CDD DMA channel. If Cdd_Dma_Init () has not been called first, Fls_OspiDmaOpen () fails silently, and Flash DMA is permanently disabled at runtime.

CDD DMA FLS Driver Init Flow

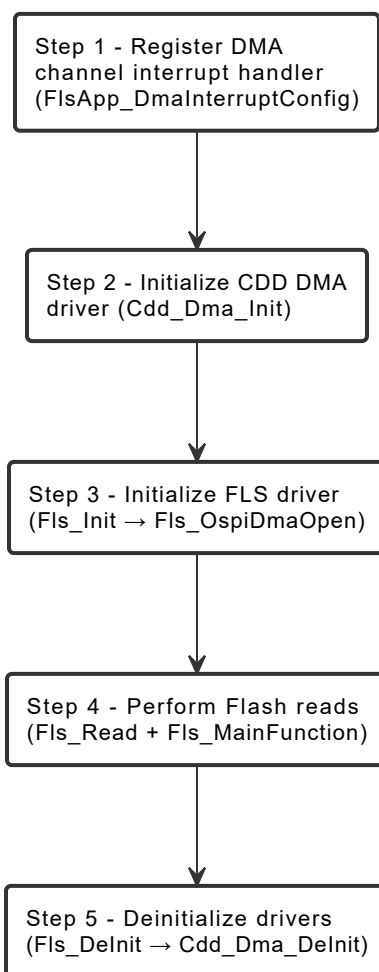


Figure 5-1. Correct Initialization Sequence for Flash DMA

Note

WRONG order:

Fls_Init() → Cdd_Dma_Init () ← DMA is be DISABLED

CORRECT order:

Cdd_Dma_Init () → Fls_Init() ← DMA is be ACTIVE

5.1.4 Constraints and Fallback Behavior

Table 5-3. Flash DMA Constraints and Resolutions

Constraint	Behavior	Resolution
FlsDacEnable = false	DMA only works in DAC mode; without DAC, DMA is ignored	Set FlsDacEnable = true in EB Tresos
FlsDmaEnable = false	DMA feature is compiled out	Set FlsDmaEnable = true in EB Tresos
Cdd_Dma_Init() not called before Fls_Init()	Fls_OspiDmaOpen () fails; DMA permanently disabled for that session	Always call Cdd_Dma_Init () first

5.2 Use Case: Amplifier Data Transmission Using CDD UART With DMA

This use case demonstrates data (for example, audio amplifier output, vibration amplifier readings) using CDD UART with DMA enabled. Similar to how the FLS driver uses CDD DMA to offload flash reads from the CPU, the CDD UART driver uses CDD DMA PKTDMA channels to offload serial data transmission and reception, freeing the CPU for real-time signal processing while data flows through the UART peripheral autonomously.

5.2.1 UART DMA Overview

The CDD UART driver supports DMA-accelerated data transmission (TX) and reception (RX). When DMA is enabled, the UART driver acquires CDD DMA PKTDMA-TX and PKTDMA-RX channels internally during `Cdd_Uart_Init()`. The application interacts only with UART APIs - the DMA handshake is transparent.

Table 5-4. UART Operations and DMA Support

Operation	DMA Supported?	Mode Required
UART Transmit (TX)	Yes	PKTDMA-TX channel; CDD_DMA_PERIPH_TX_MODE
UART Receive (RX)	Yes	PKTDMA-RX channel; CDD_DMA_PERIPH_RX_MODE

5.2.2 How DMA Enhances Amplifier Data Throughput

With CPU-based UART transmission, the processor writes each amplifier sample byte-by-byte to the UART FIFO, blocking the CPU during the entire burst. For continuous high-speed amplifier data streams (for example, audio or vibration data at kHz sample rates), this leaves no CPU headroom for processing. With DMA-enabled UART, the CPU prepares one write call and is freed immediately:

Table 5-5. Amplifier Data Transmission Performance - CPU Versus DMA

Metric	CPU-Based UART TX	DMA-Based UART TX
CPU involvement per transfer	CPU writes every byte to FIFO	CPU submits one <code>Cdd_Uart_Write()</code> ; DMA manages the rest
CPU available for other tasks	No - CPU is blocked	Yes - CPU is free after submitting the transfer
Data jitter	Variable; depends on CPU preemption	Deterministic; hardware-timed transfer
Practical for amplifier streaming	Limited - high CPU load	Yes - sustained streams at UART baud rate

5.2.3 Initialization Sequence

The initialization order is identical in principle to the FLS+DMA flow. `Cdd_Uart_Init()` calls `Uart_DmaOpen()` internally, which in turn calls `Cdd_Dma_ChOpen()`. If `Cdd_Dma_Init()` has not been called first, `Uart_DmaOpen()` fails silently and UART DMA is permanently disabled for that session.

CDD DMA UART Driver Init Flow

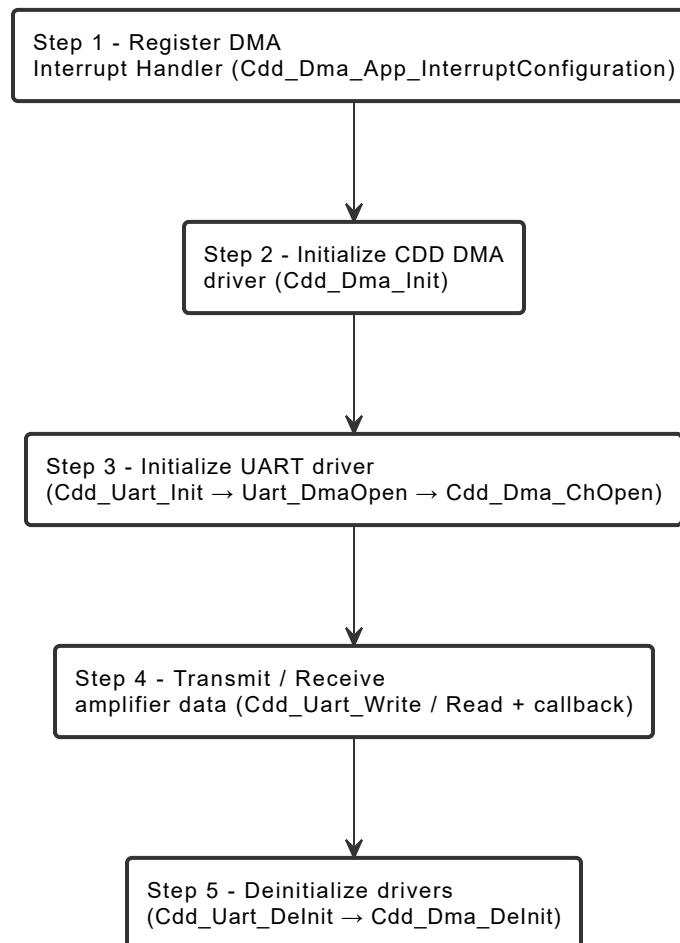


Figure 5-2. Correct Initialization Sequence for Amplifier Data UART and DMA

Note

WRONG order:

Cdd_Uart_Init() → Cdd_Dma_Init() ← UART DMA is DISABLED

CORRECT order:

Cdd_Dma_Init() → Cdd_Uart_Init() ← UART DMA is ACTIVE

6 Step-by-Step Guide to Write a CDD DMA Application

This section provides a conceptual step-by-step guide for writing your own CDD DMA application from scratch. Follow these steps in order.

6.1 Step 1 - Configure EB Tresos

- In `CddDmaChannelConfig`, create a channel entry
- Set `Cdd_Dma_TransType` → `CDD_DMA_BC_MODE` (for memory copy) or `CDD_DMA_PERIPH_TX_MODE/`
`CDD_DMA_PERIPH_RX_MODE` (for peripheral)
- Set `Cdd_Dma_InstID` → `CDD_DMA_ID_0` (BCDMA) for block copy OR `CDD_DMA_ID_1` (PKTDMA)
- Set `Cdd_Dma_Mode` → `CDD_DMA_MODE_POLLING` or `CDD_DMA_MODE_INTERRUPT`
- Set `Cdd_Dma_Trlcnt0/1/2/3` → as per the transfer length (In case of BCDMA)
- Set `Cdd_Dma_RingElemCount` → 1 for single-descriptor use (increase for pipelining)
- Implement and Provide Sciclient callback function pointer names in `Cdd_Dma_SciclientCallbacks`

6.2 Step 2 - Allocate Memory (128-Byte Aligned, in DDR)

- Ring memory: size = `RingElemCount` × 8 bytes, aligned to 128 bytes (Block Copy only)
- TRPD descriptor: aligned to 128 bytes (Block Copy only)
- Source buffer: aligned to 128 bytes
- Destination buffer: aligned to 128 bytes, must be in DDR (not TCM)

6.3 Step 3 - Initialize the Driver

- Register DMA interrupt handler (if using interrupt mode)
- Call `Cdd_Dma_Init()` to initialize the DMA hardware and allocate resources

6.4 Step 4 - Open and Enable the Channel

- Populate `Cdd_Dma_ParamsType`: pass `ringMem`, `ringElemCnt`, `channelType`, `eventCb` (or NULL for polling)
- Call `Cdd_Dma_ChOpen()` - hardware ring is configured, channel is allocated
- Call `Cdd_Dma_ChEnable()` - PSIL thread paired (peripheral), channel registers enabled

6.5 Step 5 - Submit the Transfer

- Set `dmaParams.srcBuf`, `destBuf`, `length` in `Cdd_Dma_ParamsType`
- For Block Copy: also pass `trpdMem` (your 128-byte aligned TRPD buffer)
- Call `Cdd_Dma_StartTransfer()` - descriptor is initialized and pushed to FQ ring
- DMA hardware begins autonomously

6.6 Step 6 - Wait for Completion

- Polling mode: call `Cdd_Dma_GetTransferStatus()` with a defined timeout until completion
- Interrupt mode: the registered callback is called from the ISR; wait for a flag to be set

6.7 Step 7 - Clean Up

- Call `Cdd_Dma_ChDisable()` - disables channel, unpairs PSIL threads
- Call `Cdd_Dma_ChClose()` - releases channel ID and ring resources
- Call `Cdd_Dma_DeInit()` when no more DMA is needed in the application life cycle

6.8 Common Mistakes to Avoid

Table 6-1. Common DMA Programming Mistakes

Mistake	Symptom	Fix
Buffer not 128-byte aligned	Data corruption; transfer appears to succeed but data is wrong	Provide properly aligned buffers
<code>Cdd_Dma_ChEnable()</code> skipped	Transfer timeout; <code>StartTransfer</code> returns <code>E_NOT_OK</code>	Always call <code>ChEnable()</code> after <code>ChOpen()</code>
Sciclient callbacks not implemented	Init fails with DET error	Implement all <code>Sciclient_rm*</code> wrapper functions
Using TCM buffer with FLS DMA	FLS silently falls back to CPU copy; DMA not used	Use DDR buffer for <code>Fls_Read()</code> destination

7 Summary

This application note introduced the CDD DMA driver for Texas Instruments' MCAL embedded software. The following key points summarize everything a developer needs to get started:

Table 7-1. Application Note Summary

Topic	Key Takeaway
What is DMA?	Hardware-managed data transfer; frees the CPU to run other tasks in parallel
Block Copy DMA	Memory-to-memory; uses TRPD descriptors; application allocates descriptor memory
Peripheral DMA	Memory↔Peripheral via PSIL threads; driver manages HPD descriptors; peripheral driver calls CDD DMA internally
Polling versus Interrupt	Polling: simple but CPU-intensive. Interrupt: efficient and real-time-friendly
Channel Chaining	Trigger channel auto-starts chained channel on completion;
RINGACC	Hardware queue manager; FQ ring submits work, CQ ring signals completion; all memory 128-byte aligned in DDR
PSIL	Peripheral interconnect fabric; thread pairing done automatically at ChEnable/ChDisable
Descriptors	TRPD for BCDMA (app allocates); HPD for PKTDMA (driver manages)
Interrupt Aggregator	Consolidates DMA events into CPU interrupts; supports shared VINTR for multiple channels
Flash DMA	DMA accelerates DAC-mode reads; Cdd_Dma_Init() must precede Fls_Init(); TCM buffers fall back to CPU
Memory Rules	128-byte alignment; DDR only (not TCM); applies to ring mem, descriptors, and data buffers
Init Order	Register interrupt → Cdd_Dma_Init() → peripheral Init() → transfers → peripheral DeInit() → Cdd_Dma_DeInit()

8 References

- Texas Instruments, [AM62x Processors Texas Instruments Families of Products](#), technical reference manual
- Texas Instruments, [MCUSW Documentation](#)
- Texas Instruments, [MCUSW Documentation](#)

8.1 Example Applications (in MCAL):

Table 8-1. Reference Example Applications

Application	Path	Demonstrates
cdd_dma_memcpy_polling_app	mcal/examples/Cdd_Dma/ cdd_dma_memcpy_polling_app/	Block Copy DMA in polling mode
cdd_dma_memcpy_interrupt_app	mcal/examples/Cdd_Dma/ cdd_dma_memcpy_interrupt_app/	Block Copy DMA with interrupt completion
cdd_dma_chaining_app	mcal/examples/Cdd_Dma/ cdd_dma_chaining_app/	DMA channel chaining
cdd_uart_dma_echo_app	mcal/examples/Uart/ cdd_uart_dma_echo_app/	UART peripheral DMA (TX and RX)
fls_app_dma	mcal/examples/Fls/fls_app_dma/	Flash DMA read with interrupt

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to [TI's Terms of Sale](#), [TI's General Quality Guidelines](#), or other applicable terms available either on [ti.com](#) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025