Texas Instruments

EDMA3 Driver

User Guide

October 2010

Document Version 02.10.04.XX

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI

Mailing Address:
Texas Instruments
Post Office Box 655303, Dallas, Texas 75265

LICENSE

Read This First

About This Manual

This User's Manual serves as a software programmer's handbook for working with the **EDMA3 Driver Version 02.10.04.XX**. This manual provides necessary information regarding how to effectively install, build and use **EDMA3 Driver** in user systems and applications.

This manual provides details regarding how the **EDMA3 Driver** is Architected, its composition, its functionality, the requirements it places on the hardware and software environment where it can be deployed, how to customize/configure it to specific requirements, how to leverage the supported run-time interfaces in user's own application etc.,

This manual also provides supplementary information regarding steps to be followed for proper installation/ un-installation of the EDMA3 Driver. Also included are appendix sections on related Glossary, Web sites and Pointers for gathering further information on the EDMA3 Driver.

Terms and Abbreviations

Add any longer explanations for terms before the table.

Add any abbreviations and short explanations to the table.

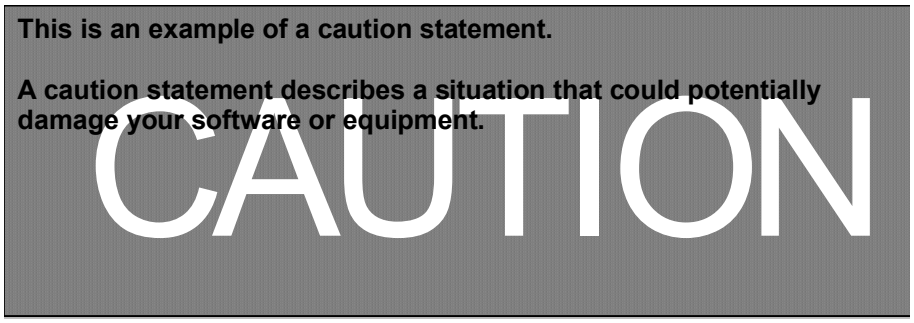
Term/Abbreviation	Description
EDMA	Enhanced Direct Memory Access
EDMA3 Controller	Consists of the EDMA3 channel controller (EDMA3CC) and EDMA3 transfer memory access controller(s) (EDMA3TC). Is referred to as EDMA3 in this document.
DMA	Direct Memory Access
QDMA	Quick DMA
TCC	Transfer Completion Code (basically Interrupt Channel)
ISR	Interrupt Service Routine
CC	Channel Controller
TC	Transfer Controller
RM	Resource Manager
TR	Transfer Request. A command for data movement that is issued from the EDMA3CC to the EDMA3TC. A TR includes source and destination addresses, counts, indexes, options, etc.

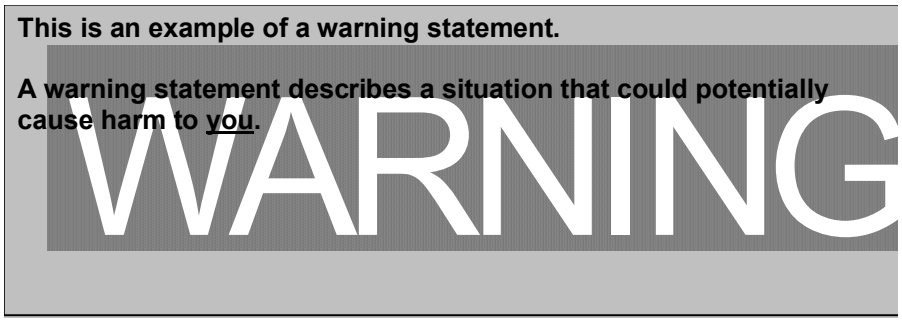
Notations

Explain any special notations or typefaces used (such as for API guides, special typefaces for functions, variables, etc.)

Information about Cautions and Warnings

This book may contain cautions and warnings.





The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation

Internal

- EDMA3 Channel Controller (TPCC), version 3.0.2 (Available at PDS)
- EDMA3 Transfer Controller (TPTC), version 3.0.1 (Available at PDS)

Trademarks

Revision History

Date	Author	Revision History	Version
October 16, 2008	Anuj Aggarwal	First release supporting platform DA830 on BIOS 6.	02.00.00.XX
June 3, 2009	Anuj Aggarwal	Patch release for DA830 platform on BIOS 6.	02.00.01.XX
December 7, 2009	Anuj Aggarwal	a) Migration to new BSD license b) Added support for TCI6498 platform. See release notes for more details.	02.10.00.XX
April 9,2009	Imtiaz SMA	Added the support for the c6748 and OMAPL138 platforms. See release notes for more details.	02.10.01.XX
May 12, 2010	Vinay K Nooji	Added the support for the OMAPL138 ARM platform. See release notes for more details.	02.10.02.XX
Sep 6, 2010	Sundaram Raju	Support for the TI816X Simulator & platform, C6472 & TCI6486 platform and TI814X platform have been added.	02.10.03.XX
Oct 12, 2010	Sundaram Raju	Support for C66x(ELF) in Generic library of Resource Manager and bug fixes	02.10.04.XX

Contents

Tables

Chapter 1

EDMA3 Driver Introduction

This chapter introduces the **EDMA3 Driver** to the user by providing a brief overview of the purpose and construction of the **EDMA3 Driver** along with hardware and software environment specifics in the context of **EDMA3 Driver** Deployment.

1.1 Overview

This section describes the functional scope of the **EDMA3 Driver** and its feature set.

A brief definition of the component is provided at this point – its main characteristics and purpose.

1.1.1 System Partitioning

EDMA3 peripheral supports data transfers between two memory mapped devices. It supports EDMA as well as QDMA channels for data transfer. This peripheral IP is being re-used in different SoCs with only a few configuration changes like number of DMA and QDMA channels supported, number of PARAM sets available, number of event queues and transfer controllers etc.

The EDMA3 peripheral is used by other peripherals for their DMA needs thus the EDMA3 Driver needs to cater to the requirements of device drivers of these peripherals as well as other application software that may need to use the 3rd party DMA services.

The **EDMA3 Driver** provides functionality that allows device drivers and applications for submitting and synchronizing with EDMA3 based DMA transfers. In order to simplify the usage, this component internally uses the services of the **EDMA3 Resource Manager** and provides one consistent interface for applications or device drivers.

The **EDMA3 Resource Manager** comprises of the following two parts:

- **Physical Driver**: This component is responsible for the management of several resources within the EDMA3 peripheral like DMA and QDMA channels, TCC codes, PARAM entry, all global EDMA3 registers, queues etc.
- **Interrupt Manager**: This module provides the different interrupt handlers (ISRs) for various EDMA3 interrupts like transfer completion interrupt, CC error interrupt and TC error interrupt. Since interrupts could be associated with TCC codes in EDMA3, this module also provides the functionality of accepting application registration callbacks for TCC codes and calls the callback functions upon receipt of the given interrupt (TCC).

  Moreover, these ISRs are NOT registered with the underlying OS, since Resource Manager is an OS-agnostic module. The user application has to do the registration / un-registration of ISRs by itself.

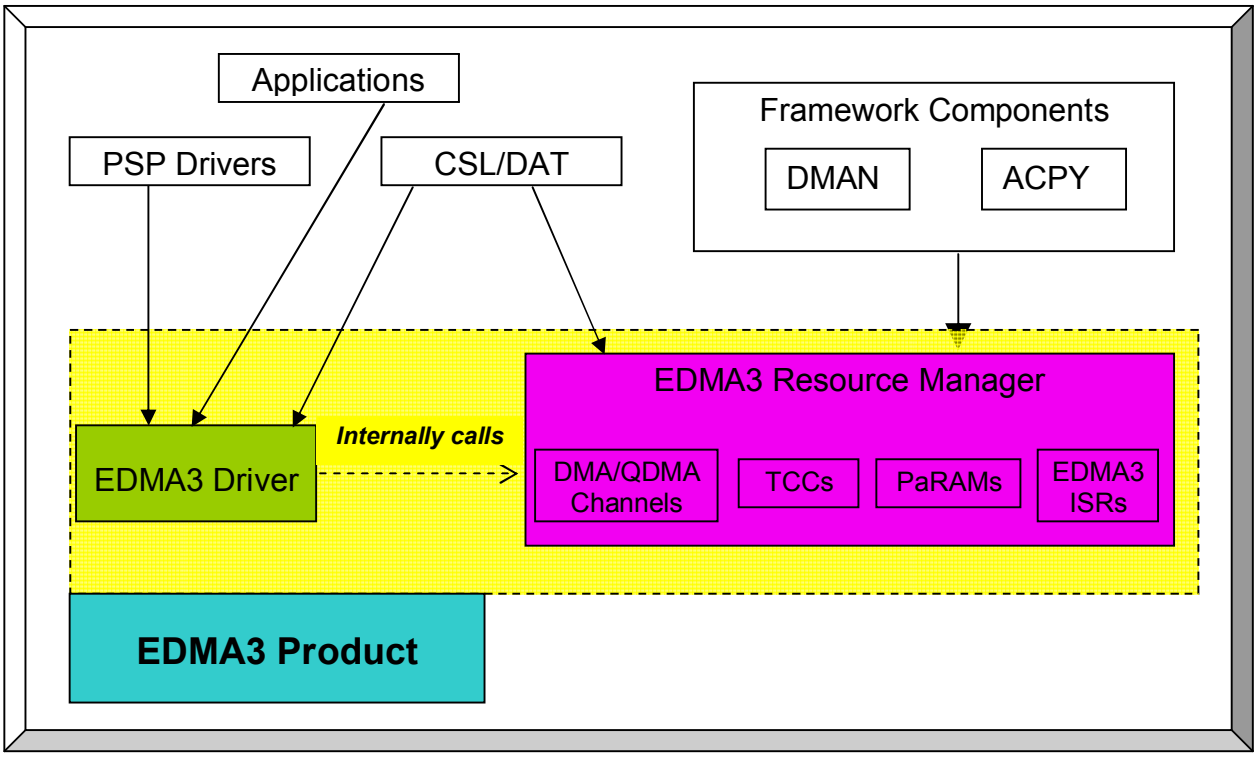


Figure 1: EDMA3 Related Software Product and Packages Structure

Typically, each master (ARM, DSP etc.) within the SoC shall open an instance of EDMA3 Driver, which internally will open a Resource Manager Instance. Resources could be allocated statically or dynamically to the EDMA3 Driver Instance. This

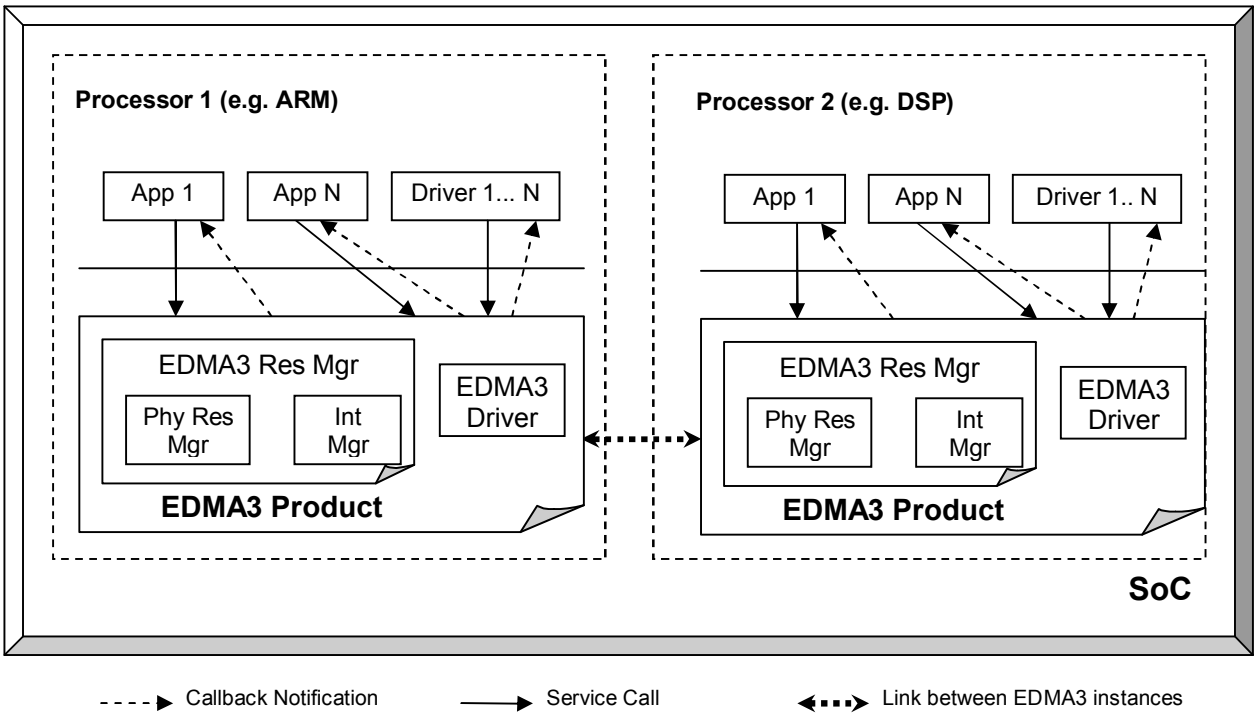


Figure 2: EDMA3 Related Software Product and Packages Structure

EDMA3 Driver Instance should be used by the users (device drivers or applications) to call all other EDMA3 Driver APIs. This instance will use the appropriate shadow region registers (specific to its master) to program EDMA3 hardware. Please note that the shadow region registers are master specific and there is only and only one set of shadow region registers for each master. If a master tries to program EDMA3 using other sets of shadow region registers (tied to other masters in the system), it could result in unexpected behavior with the possible loss of EDMA3 interrupts and EDMA3 resources' conflict. So it should be avoided in normal circumstances.

EDMA3 Driver doesn't allow multiple instances for a single master on the respective shadow region. It permits only one instance for each master which will be tied to its specific shadow region. This is done to prevent any potential problem which could arise due to EDMA3 resources' conflict among these different instances.

However, it is possible to have multiple EDMA3 Driver Instances, running on the same processor. These different EDMA3 Driver instances would be tied to different masters (and hence different shadow regions) to cater their specific requests. The EDMA3 resources should be carefully allocated among all those instances to avoid any possible conflict.

All software entities intending to use the services of the EDMA3 peripheral on the given processor shall use the services of the EDMA3 Product (Resource manager OR EDMA3 Driver) as desired.

1.1.2 Supported Services

Following are the services provided by the **EDMA3 Driver:**

1.1.2.1 ***Request and Free DMA channel:*** It provides an interface that applications or device drivers can use to request and free DMA channels. Channels in EDMA3 module are categorized as:

- DMA Channel (mapped to a hardware sync event),
- DMA Channel (NOT mapped to a hardware sync event),
- QDMA Channel, and
- Link Channel (a PARAM Set in EDMA3).

1.1.2.2 ***Programs DMA channel:*** It provides an interface that applications or device drivers can use to program a DMA transaction. This typically involves setting the DMA source and destination parameters.

Following types of transactions are supported:

- Event triggered (peripheral driven transfers),
- Chain triggered (issuing a chain of transfers initiated by single event),
- Manual triggered (CPU generated sync-event), and
- QDMA transfer (triggered on a write to the QDMA Trigger word).

An API is also provided to get the current status of the DMA/QDMA channel.

1.1.2.3 ***Start and Synchronize with DMA transfers***: It provides an interface that applications or device drivers can use to start and synchronize with a DMA transaction.

1.1.2.4 ***Provides DMA transaction completion callback to applications:*** It provides an interface that applications or device drivers can use to register a transaction completion (final or intermediate) callback or error interrupt callback. EDMA3 driver calls this application or device driver specifc callback routine, with the appropriate status message.

1.1.2.5 ***Supports Linking and chaining feature:*** EDMA3 peripheral provides linking and chaining capabilities. EDMA driver provides an interface that applications or device drivers can use to use this functionality.

1.1.2.6 ***Supports multiple instances of EDMA driver on a single processor:*** It supports multiple instances of itself, running on the same processor, but tied to different masters (and hence different shadow regions). These different instances will run on the same processor but manage same/different set of EDMA3 resources and are tied to different shadow regions. Please note that EDMA3 Driver doesn't allow multiple instances for a single master on the respective shadow region.

1.1.2.7 ***Read/Write a specific CC register:*** It also provides an interface which enables users to read/write any EDMA3 Channel Controller register. These APIs are for advanced users and could be used for debugging purposes.

1.1.2.8 ***Support for Polled Mode DMA Transfers:*** It provides an interface which enables the application or device driver to use it in an interrupt-less (and further in an OS-less) environment. In this scenario, the application does not register the callback function with the resource manager and itself polls the EDMA3 hardware for the completion interrupt, using the specific APIs.

1.1.2.9 ***Non-RTSC Environment Support:*** EDMA3 Driver module should gets built in non-RTSC environment also. All the CCS PJT files should come for non-RTSC environment too.

1.1.2.10 ***IOCTL interface support:*** EDMA3 Driver shall provide an IOCTL interface for toggling the option whether PaRAM Sets should be cleared during allocation or not. This interface could also be extended in future for other misc requirements.

1.1.2.11 ***Registration and Un-registration of TCC callbacks:*** It provides an interface that can be called by applications to register/un-register for TCC callbacks. It handles EDMA3 interrupts and calls the respective TCC callback function with appropriate status.

1.1.2.12 ***Big Endian platforms support:*** EDMA3 driver can also be used for big endian platforms.

1.1.2.13 ***Enable/diable transfer controller error interrupts:*** It provides an interface that can be used to enable or disable specific transfer controller error interrupts.

1.1.2.14 ***Map Cross bar events to the DMA channels:*** It provides and interface than can be used to map the cross bar mapped events to the specific DMA channel.

Chapter 2

Installation Guide

This chapter discusses the **EDMA3 Driver** installation, how and what software and hardware components to be availed in order to complete a successful installation of **EDMA3 Driver**.

2.1 Component Folder

Upon installing the **EDMA3 LLD package,** the following directory structure is found in the main directory. A viewgraph of the actual directory tree (as seen in the final deployed environment) is inserted here for clarity.

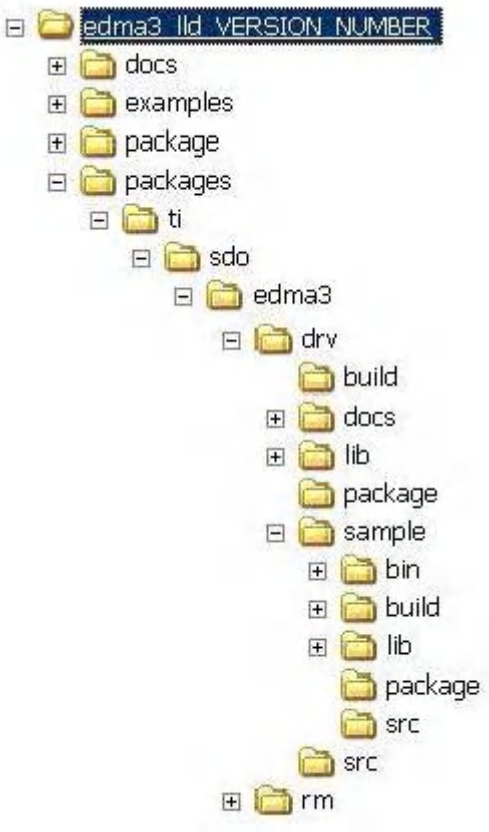


Figure 3: EDMA3 Driver Directory Structure

The sections below describe the folder contents:

edma3_lld_<<*version_number*>>

Top level installation directory. Contains the source code, examples and the documents.

docs

Contains release notes for EDMA3 Driver and Resource Manager.

examples

Contains the stand-alone applications for EDMA3 Driver (for all the supported platforms) and the DAT example.

packages

All components (Driver, Resource Manager, sample OS-abstraction layers etc) fall under packages/ti/sdo/edma3 directory, under their individual directories. For e.g., EDMA3 driver lies under packages/ti/sdo/edma3/drv folder, sample initialization library for EDMA3 Driver lies under packages/ti/sdo/edma3/drv/sample folder etc.

a) **drv** -> Top level folder for the EDMA3 Driver.

b) **drv\docs** -> User guide, datasheet etc.

c) **drv\lib** -> EDMA3 Driver libraries for all the supported platforms.

d) **drv\sample** -> Sample code for how to use the EDMA3 Driver, along-with the pre-built libraries for the same.

e) **drv\src** -> Source files for EDMA3 Driver.

Just to clarify, the *sample* folder inside the edma3/drv folder DOESN'T contain the sample applications. It provides the:

a) Sample initialization code to properly configure the EDMA3 hardware, and,

b) Sample OS abstraction layer to provide the OS-specific hooks to the EDMA3 package.

This sample code is provided for reference purpose only. To start with, the user is advised to use the sample code/library as it is, and later modify/create his own initialization code, as per the requirements.

The stand-alone applications are provided in the top level *examples* folder as mentioned above. Please note that these examples use the above mentioned sample initialization/OS abstraction libraries and the EDMA3 Driver libraries.

2.2 Development Tools Environment(s)

This section describes the development tools environment(s) for software development with **EDMA3 Driver**. It describes the tools used and their setup, for each supported environment.

2.2.1 Development Tools

Describe here the tools that need to be installed, the installation order and specific configuration. Including: 3rd party components/ libraries, Operating System and auxiliary Tools.

Table 1: Development Tools/components

Development tool/ component	Version	Comments
Code Composer Studio (CCS)	4.2.0.09000	IDE
Code Gen Tools	7.0.x	Code generation utilities
DSP BIOS	6.30.02.42	Operating System
XDC tool chain	3.20.02.59	XDC tools
TCI6498 Simulator	0.7.1	Simulator
TI816x Simulator	0.7.1	Simulator

2.3 Installation Guide

This section describes the EDMA3 LLD installation and un-installation.

2.3.1 Installation and Usage Procedure

1) Install the products mentioned in the development tools requirements section, as per instructions provided along with the products.

2) Install the EDMA3 package using the self-extracting installer into preferred drive/folder. It is recommended to install the EDMA3 LLD into the default drive/folder as indicated by the self-extracting installer.

3) As a part of installation process, an environment variable "EDMA3LLD_BIOS6_INSTALLDIR" is created with its value as the current EDMA3 installation directory. Moreover, in case the variable exists prior to this installation, the same will be updated with the current (latest) EDMA3 installation directory. This environment variable can be used by other users of EDMA3 package for e.g. BIOS PSP drivers package.

4) For building the downloadable images, refer to section 2.4 – Integration Guide.

5) Download the image (.out) onto the platform using CCS.

6) Run the program.

2.3.2 Un-installation

1) Uninstall the EDMA3 package by using the uninstall.exe in the install directory.

2) Un-install the products mentioned in the development tools requirements section as per the instructions provided with the product.

2.4 Integration Guide

This section describes the EDMA3 LLD package usage. The package provides pre-built libraries for all the different components: EDMA3 Driver, Resource Manager along with their sample initialization libraries. Moreover, demo applications are also provided to check the basic functionality for the supported components.

The EDMA3LLD package for BIOS-6 is an XDC based package, and hence requires a "config.bld" file in the *<EDMA3LLD_BIOS6_INSTALLDIR>\packages* location. A "_config.bld" file is shipped in that folder. Rename that file to "config.bld" before building any EDMA3 LLD libraries or standalone applications.

2.4.1 Building EDMA3 Libraries

The EDMA3 package contains pre-built libraries for all EDMA3 components. But user can also build them by following the below mentioned steps in case of source code modification or some other specific use cases described below.

1) Install the products mentioned in the development tools requirements section (section 2.2), as per instructions provided along with the products. See note 1 below for more details.

2) Build the required libraries using the *xdc* command at the command prompt:

   a. Example:

```
Z:\edma3_lld_<<version_number>>\packages\ti\sdo\edma3\drv> xdc
```

3) All EDMA3 public APIs provide a mechanism to disable input parameter checking. This is intended to reduce the number of CPU cycles spent in the parameter checking and hence provide more efficient libraries. To do that, user has to modify the "package.bld" file, found in the component base folder itself, and re-build the libraries. By default, the parameter checking is enabled for all the public APIs.

   For e.g., following code snippet in the edma3\drv\package.bld file is used to create the EDMA3 Driver libraries:

```
Pkg.addLibrary("lib/Debug/" + Pkg.name, targ,
      { defs:"", profile: "debug"}
      ).addObjects(objList);
Pkg.addLibrary("lib/Release/" + Pkg.name, targ,
      { defs:"", profile: "release"}
      ).addObjects(objList);
```

   By default, parameter checking is enabled in both Debug and Release modes for all the public APIs. If user wants to disable the same in Release mode (for example), he has to modify the above code as:

```
Pkg.addLibrary("lib/Debug/" + Pkg.name, targ,
      { defs:"", profile: "debug"}
      ).addObjects(objList);
```

```
Pkg.addLibrary("lib/Release/" + Pkg.name, targ,

      {defs:"-DEDMA3_DRV_PARAM_CHECK_DISABLE",profile:
      "release"}

      ).addObjects(objList);
```

The Release mode library generated now will have input parameter check disabled for all the public APIs. User is advised to use this configuration option with caution.

4) All EDMA3 private functions use the standard C **assert** mechanism to enable/disable input parameter checking. This is intended to reduce the number of CPU cycles spent in the parameter checking and hence provide more efficient libraries. To do that, user has to modify the "package.bld" file, found in the component base folder itself, and re-build the libraries. By default, the parameter checking is enabled for all the private functions.

   For e.g., following code snippet in the edma3\drv\package.bld file is used to create the EDMA3 Driver libraries:

```
Pkg.addLibrary("lib/Debug/" + Pkg.name, targ,

      { defs:"", profile: "debug"}

      ).addObjects(objList);

Pkg.addLibrary("lib/Release/" + Pkg.name, targ,

      { defs:"", profile: "release"}

      ).addObjects(objList);
```

   By default, parameter checking is enabled in both Debug and Release modes for all the private functions. If user wants to disable the same in Release mode (for example), he has to modify the above code as:

```
Pkg.addLibrary("lib/Debug/" + Pkg.name, targ,

      { defs:"", profile: "debug"}

      ).addObjects(objList);

Pkg.addLibrary("lib/Release/" + Pkg.name, targ,

      { defs:"-DNDEBUG", profile: "release"}

      ).addObjects(objList);
```

   The Release mode library generated now will have input parameter check disabled for all the private functions. User is advised to use this configuration option with caution.

5) The event queue number registers for DMA/QDMA channels are programmed during run-time, depending on the application requirements. User has to specify the desired queue number for the specific channel while calling the EDMA3_DRV_requestChannel () API.

This behavior can be changed by re-compiling the EDMA3 Driver libraries and passing "EDMA3_PROGRAM_QUEUE_NUM_REGISTER_INIT_TIME" to the compiler. Now the EDMA3 driver will pre-allocate the event queues for the DMA/QDMA channels present in the system and program the appropriate registers during the EDMA3 initialization; it will not program the same registers at run-time anymore.

The mapping between DMA/QDMA channels and different queue numbers should be provided by the system integrator using the structure `edma3DmaQdmaQueueNumConfig` in file "packages\ti\sdo\edma3\drv\src\edma3_drv_init.c".

For e.g., following code snippet in the edma3\drv\package.bld file is used to create the EDMA3 Driver libraries:

```
Pkg.addLibrary("lib/Debug/" + Pkg.name, targ,

      { defs:"", profile: "debug"}

      ).addObjects(objList);

Pkg.addLibrary("lib/Release/" + Pkg.name, targ,

      { defs:"", profile: "release"}

      ).addObjects(objList);
```

By default, event queue registers will be programmed at run-time. If user wants to disable the same and instead program the registers at init-time itself, he has to modify the above code as:

```
Pkg.addLibrary("lib/Debug/" + Pkg.name, targ,
{defs:"-DEDMA3_PROGRAM_QUEUE_NUM_REGISTER_INIT_TIME",
profile: "debug"}).addObjects(objList);


Pkg.addLibrary("lib/Release/" + Pkg.name, targ,
{defs:"-DEDMA3_PROGRAM_QUEUE_NUM_REGISTER_INIT_TIME",
profile: "release"}).addObjects(objList);
```

2.4.2 Building the EDMA3 Driver Stand-alone Applications

The EDMA3 package contains separate sample applications for EDMA3 Driver for each of the supported platforms. Following steps are required to build the same:

1) Install the products mentioned in the development tools requirements section (section 2.2), as per instructions provided along with the products. See note 1 below for more details.

2) Build the required libraries using the *xdc* command at the command prompt:

   a. Example:

```
Z:\edma3_lld_<<version_number>>\packages\ti\sdo\edma3\drv> xdc
```

This step is required only if the source code is modified and new libraries need to be generated.

3) Setup the CCS4 to set the underlying platform and use the appropriate DSP gel file, if required.

4) Load CCS project.
   a. Open C/C++ perspective: Window -> Open Perspective -> C/C++.
   b. CCS v4 uses "Linked Resources" to create portable paths. This is required to link different source files into the CCS project without making copies of the source files. To enable proper linking of the source files, user must modify the path variable "EDMA3LLD_BIOS6_INSTALLDIR" to point it to the EDMA3 installation directory. The value of this variable can also be fetched from the environment variable "EDMA3LLD_BIOS6_INSTALLDIR".
      i. Go to Window -> Preferences.
      ii. Modify the path variable "EDMA3LLD_BIOS6_INSTALLDIR" under General -> Workspace -> Linked Resources section, as suggested above.
   c. Go to "Project -> Import Existing CCS Eclipse Project" menu item.
   d. Point to the directory of the sample application needed to run.
      - Example: `Z:\edma3_lld_<<version_number>>\examples\edma3_driver\<<your_platform>>\`
   e. Set required Debug/Release configuration.
   f. "Project -> Rebuild Active Project" will build the .out executable. E.g. edma3_drv_bios6_*<<your_platform>>*_st_sample.out.

5) Target -> Launch TI Debugger for your platform.

6) Use "Target -> Advanced -> Connect target" to connect to DSP target. The GEL would configure and setup the DSP to be used by the DSP window.

7) Use "Target -> Load Program" to download the .out executable.

NOTES:

1. edma3_lld_<<version_number>>\packages\config.bld

a. It uses an environment variable CGTOOLS to locate the right codegen tool chain version. Make sure that this variable is set in your environment and points to the desired toolchain version.

Example: If the code generation tools are installed in "C:\Program Files\Texas Instruments\ccsv4\tools\compiler\c6000", then CGTOOLS should point to:

```
CGTOOLS= C:\Program Files\Texas
Instruments\ccsv4\tools\compiler\c6000
```

b. It uses an environment variable TMS470_CGTOOLS to locate the right ARM codegen tool chain version for compilitation. Make sure that this variable is set in your environment and points to the desired toolchain version.

Example: If the code generation tools are installed in "C:\Program Files\Texas Instruments\ccsv4\tools\compiler\tms470", then TMS470_CGTOOLS should point to:

```
TMS470_CGTOOLS= C:\Program Files\Texas
Instruments\ccsv4\tools\compiler\tms470
```

2. The following environmental variables must be set

a. XDCPATH – Should include BIOS v6 package installation directory.

Example:

```
XDCPATH= D:/Program Files/Texas Instruments/bios_6_21_00_06_eng/packages
```

2.4.3 Building the DAT Example

The EDMA3 package contains CSL 2.0 DAT Adapter Reference Implementation using EDMA3 Low Level Driver. The same can be built using the steps shown in the previous section. The application can be located at "edma3_lld_<<version_number>>\examples\CSL2_DAT_DEMO\demo\" in the platform specific folder.

2.4.4 Build Options

This section enumerates and describes alongside each of the allowed build options. It also tells the default configurations available.

Build option Reference	Default Configuration	Description
EDMA3_INSTRUMENTATION_ENABLED	Instrumentation disabled	To enable/disable Real Time Instrumentation support.
EDMA3_DRV_PARAM_CHECK_DISABLE	Parameter checking enabled (public APIs)	Disable parameter checking for public APIs, if required. See note 1 below.
NDEBUG	Parameter checking enabled (private functions)	Disable parameter checking for private functions, if required. See note 2 below.
_BIG_ENDIAN	NA	Used while building libraries for Big Endian platforms.

Table 2: Build Options

Note 1: All EDMA3 public APIs provide a mechanism to disable input parameter checking. This is intended to reduce the number of CPU cycles spent in the parameter checking and hence provide more efficient libraries. To do that, user has to modify the build environment (for e.g. the package.bld file), and re-build the libraries. By default, the parameter checking is enabled for all the public APIs.

Note 2: All EDMA3 private functions use the standard C **assert** mechanism to enable/disable input parameter checking. This is intended to reduce the number of CPU cycles spent in the parameter checking and hence provide more efficient libraries. To do that, user has to modify the build environment (for e.g. the package.bld file), and re-build the libraries. By default, the parameter checking is enabled for all the private functions.

Chapter 3

Run-Time Interfaces/Integration Guide

This chapter discusses the **EDMA3 Driver** run-time interfaces that comprise the API specification & usage scenarios, in association with its data types and structure definitions.

3.1 Symbolic Constants and Enumerated Data types

This section summarizes all the symbolic constants specified as either #define macros and/or enumerated C data types. Described alongside the macro or enumeration is the semantics or interpretation of the same in terms of what value it stands for and what it means.

Table 3: Symbolic Constants and Enumerated Data types Table for common header file edma3_common.h

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
Driver Global Defines	EDMA3_DRV_DEBUG	This define is used to enable/disable EDMA3 Driver debug messages
	EDMA3_DRV_PRINTF	If EDMA3_DRV_DEBUG is defined, EDMA3_DRV_PRINTF will be used to print the debug messages on the user specified output.
	EDMA3_DRV_SOK	EDMA3 Driver Result OK
	EDMA3_OSSEM_NO_TIMEOUT	This define is used to specify a blocking call without timeout while requesting a semaphore.
Defines used to support the maximum resources supported by the EDMA3 controller. These are used to allocate the maximum memory for different data structures of the EDMA3 Driver and Resource Manager.	EDMA3_MAX_ EDMA3_INSTANCES	Maximum EDMA3 Controllers on the SoC
	EDMA3_MAX_DMA_CH	Maximum DMA channels supported by the EDMA3 Controller
	EDMA3_MAX_QDMA_CH	Maximum QDMA channels supported by the EDMA3 Controller
	EDMA3_MAX_PARAM_SETS	Maximum PaRAM Sets supported by the EDMA3 Controller
	EDMA3_MAX_LOGICAL_CH	Maximum Logical channels supported by the EDMA3 Package
	EDMA3_MAX_TCC	Maximum TCCs (Interrupt Channels) supported by the EDMA3 Controller
	EDMA3_MAX_EVT_QUE	Maximum Event Queues supported by the EDMA3 Controller
	EDMA3_MAX_TC	Maximum Transfer Controllers supported by the EDMA3 Controller
	EDMA3_MAX_REGIONS	Maximum Shadow Regions supported by the EDMA3 Controller
	EDMA3_MAX_DMA_CHAN_DWRDS	Maximum Words (4-bytes region)

		required for the book-keeping information specific to the maximum possible DMA channels.
	EDMA3_MAX_QDMA_CHAN_DWRDS	Maximum Words (4-bytes region) required for the book-keeping information specific to the maximum possible QDMA channels.
	EDMA3_MAX_PARAM_DWRDS	Maximum Words (4-bytes region) required for the book-keeping information specific to the maximum possible PaRAM Sets.
	EDMA3_MAX_TCC_DWRDS	Maximum Words (4-bytes region) required for the book-keeping information specific to the maximum possible TCCs.
Defines for the level of OS protection needed when calling edma3OsProtectXXX()	EDMA3_OS_PROTECT_INTERRUPT	Protection from All Interrupts required
	EDMA3_OS_PROTECT_SCHEDULER	Protection from scheduling required
	EDMA3_OS_PROTECT_INTERRUPT_XFER_COMPLETION	Protection from EDMA3 Transfer Completion Interrupt required
	EDMA3_OS_PROTECT_INTERRUPT_CC_ERROR	Protection from EDMA3 CC Error Interrupt required
	EDMA3_OS_PROTECT_INTERRUPT_TC_ERROR	Protection from EDMA3 TC Error Interrupt required

Table 4: Symbolic Constants and Enumerated Data types Table for EDMA3 Driver header file edma3_drv.h

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
Driver Error Codes	EDMA3_DRV_E_OBJ_NOT_DELETED	Before a Driver Object could be created, it must be in the 'Deleted' state. Since it is not yet 'Deleted', it cannot be created.
	EDMA3_DRV_E_OBJ_NOT_CLOSED	Before a Driver Object could be deleted, it must be in the 'Closed' state. Since it is not yet 'Closed', it cannot be deleted.
	EDMA3_DRV_E_OBJ_NOT_OPENED	Before a Driver Object could be closed, it must be in the 'Opened' state. Since it is not yet 'Opened', it cannot be closed.
	EDMA3_DRV_E_RM_CLOSE_FAIL	While closing EDMA3 Driver Object, Resource Manager Object has to be closed. If the 'Close' fails, this error is returned.
	EDMA3_DRV_E_DMA_CHANNEL_UNAVAIL	DMA channel requested for allocation is not available.
	EDMA3_DRV_E_QDMA_CHANNEL_UNAVAIL	QDMA channel requested for allocation is not available.
	EDMA3_DRV_E_PARAM_SET_UNAVAIL	PARAM Set requested for allocation is not available.
	EDMA3_DRV_E_TCC_UNAVAIL	TCC requested for allocation is not available.
	EDMA3_DRV_E_TCC_REGISTER_FAIL	Registration of the callback function against a specific TCC failed.
	EDMA3_DRV_E_CH_PARAM_BIND_FAIL	The binding of Channel and PaRAM Set failed.
	EDMA3_DRV_E_ADDRESS_NOT_ALIGNED	While in FIFO mode, the address of the memory location passed as argument is not properly aligned. It should be 32 bytes aligned.
	EDMA3_DRV_E_INVALID_PARAM	Invalid Parameter passed to API.
	EDMA3_DRV_E_INVALID_STATE	Invalid State of EDMA3 Driver Object.
	EDMA3_DRV_E_INST_ALREADY_EXISTS	EDMA3 Driver instance already exists for the specified region. Multiple EDMA3 Driver instances on the same shadow region are NOT allowed.
	EDMA3_DRV_E_FIFO_WIDTH_NOT_SUPPORTED	FIFO width not supported by the requested Transfer Controller.
	EDMA3_DRV_E_SEMAPHORE	Semaphore handling related error.

	EDMA3_DRV_E_INST_NOT_OPENED	EDMA3 Driver Instance does not exist, it is not opened yet.
Driver Global Defines	EDMA3_DRV_CH_NO_PARAM_MAP	This define is used to say that the DMA channel is not tied to any PaRAM Set and hence any available PaRAM Set could be used for that DMA channel. It could be used in *dmaChannelPaRAMMap [EDMA3_MAX_DMA_CH]*, in global configuration structure *EDMA3_DRV_GblConfigParams*. **This value should mandatorily be used to mark DMA channels with no initial mapping to a specific PaRAM Set.**
	EDMA3_DRV_CH_NO_TCC_MAP	This define is used to say that the DMA/QDMA channel is not tied to any TCC and hence any available TCC could be used for that DMA/QDMA channel. It could be used in *dmaChannelTccMap [EDMA3_RM_NUM_DMA_CH]*, in global configuration structure *EDMA3_DRV_GblConfigParams*. **This value should mandatorily be used to mark DMA channels with no initial mapping to a specific TCC.**
	EDMA3_DRV_DMA_CHANNEL_ANY	Used to specify any available DMA Channel while requesting one. It is used in the API EDMA3_DRV_requestChannel (). DMA channel from the pool of (owned && non_reserved && available_right_now) DMA channels will be chosen and returned.
	EDMA3_DRV_QDMA_CHANNEL_ANY	Used to specify any available QDMA Channel while requesting one. It is used in the API EDMA3_DRV_requestChannel (). QDMA channel from the pool of (owned && non_reserved && available_right_now) QDMA channels will be chosen and returned.
	EDMA3_DRV_TCC_ANY	Used to specify any available TCC while requesting one. Used in the API EDMA3_DRV_requestChannel (), for both DMA and QDMA channels. Interrupt channel (TCC) from the pool of (owned && non_reserved && available_right_now) TCCs will be chosen and returned.

	EDMA3_DRV_LINK_CHANNEL	Used to specify any PaRAM Set. It is used as the *channelId* when requesting ANY available PaRAM set for linking. It is used in the API EDMA3_DRV_requestChannel (). PaRAM Set from the pool of (owned && non_reserved && available_right_now) PaRAM Sets will be chosen and returned.
	EDMA3_DRV_LINK_CHANNEL_WITH_TCC	Used to specify any available PaRAM Set while requesting one. Used in the API EDMA3_DRV_requestChannel(), for Link channels. TCC code should also be specified and it will be used to populate the LINK field of the PaRAM Set. Without TCC code, the call will fail. PaRAM Set from the pool of (owned && non_reserved && available_right_now) PaRAM Sets will be chosen and returned.
	EDMA3_DRV_QDMA_CHANNEL_0	QDMA Channel 0 define. It used while requesting the specific QDMA channel.
	EDMA3_DRV_QDMA_CHANNEL_1	QDMA Channel 1 define. It used while requesting the specific QDMA channel.
	EDMA3_DRV_QDMA_CHANNEL_2	QDMA Channel 2 define. It used while requesting the specific QDMA channel.
	EDMA3_DRV_QDMA_CHANNEL_3	QDMA Channel 3 define. It used while requesting the specific QDMA channel.
	EDMA3_DRV_QDMA_CHANNEL_4	QDMA Channel 4 define. It used while requesting the specific QDMA channel.
	EDMA3_DRV_QDMA_CHANNEL_5	QDMA Channel 5 define. It used while requesting the specific QDMA channel.
	EDMA3_DRV_QDMA_CHANNEL_6	QDMA Channel 6 define. It used while requesting the specific QDMA channel.
	EDMA3_DRV_QDMA_CHANNEL_7	QDMA Channel 7 define. It used while requesting the specific QDMA channel.
	EDMA3_DRV_CHANNEL_CLEAN	Channel status define: Channel is clean; no pending event, completion interrupt and event miss interrupt.
	EDMA3_DRV_CHANNEL_EVENT_PENDING	Channel status define: Pending event is detected on the DMA channel.

	EDMA3_DRV_CHANNEL_XFER_COMPLETE	Channel status define: Transfer completion interrupt is detected on the DMA/QDMA channel.
	EDMA3_DRV_CHANNEL_ERR	Channel status define: Event miss error interrupt is detected on the DMA/QDMA channel.
Enum EDMA3_DRV_HW_CHANNEL_EVENT	EDMA3_DRV_HW_CHANNEL_EVENT_0 = 0, EDMA3_DRV_HW_CHANNEL_EVENT_1, EDMA3_DRV_HW_CHANNEL_EVENT_2, ...	DMA Channels assigned to different Hardware Events. They should be used while requesting a specific DMA channel. One possible usage is to maintain a SoC specific file, which will contain the mapping of these hardware events to the respective peripherals for better understanding and lesser probability of errors. Also, if any event associated with a particular peripheral gets changed, only that SoC specific file needs to be changed.
Enum EDMA3_DRV_OptField	EDMA3_DRV_OPT_FIELD_SAM	Source addressing mode (INCR / FIFO)
	EDMA3_DRV_OPT_FIELD_DAM	Destination addressing mode (INCR / FIFO)
	EDMA3_DRV_OPT_FIELD_SYNCDIM	Transfer synchronization dimension (A-synchronized / AB-synchronized)
	EDMA3_DRV_OPT_FIELD_STATIC	Static/non-static PaRAM set
	EDMA3_DRV_OPT_FIELD_FWID	FIFO Width. Applies if either SAM or DAM is set to FIFO mode.
	EDMA3_DRV_OPT_FIELD_TCCMODE	Transfer complete code mode. Indicates the point at which a transfer is considered completed for chaining and interrupt generation.
	EDMA3_DRV_OPT_FIELD_TCC	Transfer complete code. This 6-bit code is used to set the relevant bit in chaining enable register (CER[TCC]/CERH[TCC]) for chaining or in interrupt pending register (IPR[TCC]/IPRH[TCC]) for interrupts.
	EDMA3_DRV_OPT_FIELD_TCINTEN	Transfer complete interrupt enable/disable.
	EDMA3_DRV_OPT_FIELD_ITCINTEN	Intermediate transfer complete interrupt enable/disable.
	EDMA3_DRV_OPT_FIELD_TCCHEN	Transfer complete chaining enable/disable.
	EDMA3_DRV_OPT_FIELD_ITCCHEN	Intermediate transfer completion chaining enable/disable.
Enum	EDMA3_DRV_ADDR_MODE_INCR	Increment (INCR) mode. Source

EDMA3_DRV_AddrMode		addressing within an array increments. Source is not a FIFO.
	EDMA3_DRV_ADDR_MODE_FIFO	FIFO mode. Source addressing within an array wraps around upon reaching FIFO width.
Enum EDMA3_DRV_SyncType	EDMA3_DRV_SYNC_A	A-synchronized. Each array is submitted as one TR. (BCNT*CCNT) number of sync events are needed to completely service a PaRAM set (where BCNT = Num of Arrays in a Frame; CCNT = Num of Frames in a Block). (S/D)CIDX = (Address of First array in next frame) - (Address of Last array in present frame) (where CIDX is the Inter-Frame index).
	EDMA3_DRV_SYNC_AB	AB-synchronized. Each frame is submitted as one TR. Only CCNT number of sync events are needed to completely service a PaRAM set (where CCNT = Num of Frames in a Block). (S/D)CIDX = (Address of First array in next frame) - (Address of first array of present frame) (where CIDX is the Inter-Frame index).
Enum EDMA3_DRV_StaticMode	EDMA3_DRV_STATIC_DIS	PaRAM set is not Static. PaRAM set is updated or linked after TR is submitted. A value of 0 should be used for DMA channels and for non-final transfers in a linked list of QDMA transfers.
	EDMA3_DRV_STATIC_EN	PaRAM set is Static. PaRAM set is not updated or linked after TR is submitted. A value of 1 should be used for isolated QDMA transfers or for the final transfer in a linked list of QDMA transfers.
Enum EDMA3_DRV_FifoWidth	EDMA3_DRV_W8BIT	The user can set the width of the FIFO as 8 bits using it. This is done via the OPT register. This is valid only if the EDMA3_DRV_ADDR_MODE_FIFO value is used for the enum EDMA3_DRV_AddrMode.
	EDMA3_DRV_16WBIT	FIFO width is 16-bit.
	EDMA3_DRV_32WBIT	FIFO width is 32-bit.
	EDMA3_DRV_64WBIT	FIFO width is 64-bit.
	EDMA3_DRV_128WBIT	FIFO width is 128-bit.
	EDMA3_DRV_256WBIT	FIFO width is 256-bit.
Enum EDMA3_DRV_TccMode	EDMA3_DRV_TCCMODE_NORMAL	Normal completion: A transfer is considered completed after the data has been transferred.

	EDMA3_DRV_TCCMODE_EARLY	Early completion: A transfer is considered completed after the EDMA3CC submits a TR to the EDMA3TC. TC may still be transferring data when interrupt/chain is triggered.
Enum EDMA3_DRV_TcintEn	EDMA3_DRV_TCINTEN_DIS	Transfer complete interrupt is disabled.
	EDMA3_DRV_TCINTEN_EN	Transfer complete interrupt is enabled. When enabled, the interrupt pending register (IPR/IPRH) bit is set on transfer completion (upon completion of the final TR in the PaRAM set). The bit (position) set in IPR or IPRH is the TCC value specified. In order to generate a completion interrupt to the CPU, the corresponding IER [TCC] / IERH [TCC] bit must be set to 1.
Enum EDMA3_DRV_ItcintEn	EDMA3_DRV_ITCINTEN_DIS	Intermediate transfer complete interrupt is disabled.
	EDMA3_DRV_ITCINTEN_EN	Intermediate transfer complete interrupt is enabled. When enabled, the interrupt pending register (IPR/IPRH) bit is set on every intermediate transfer completion (upon completion of every intermediate TR in the PaRAM set, except the final TR in the PaRAM set). The bit (position) set in IPR or IPRH is the TCC value specified. In order to generate a completion interrupt to the CPU, the corresponding IER [TCC] / IERH [TCC] bit must be set to 1.
Enum EDMA3_DRV_TcchEn	EDMA3_DRV_TCCHEN_DIS	Transfer complete chaining is disabled.
	EDMA3_DRV_TCCHEN_EN	Transfer complete chaining is enabled. When enabled, the chained event register (CER/CERH) bit is set on final chained transfer completion (upon completion of the final / last TR in the PaRAM set). The bit (position) set in CER or CERH is the TCC value specified.
Enum EDMA3_DRV_ItcchEn	EDMA3_DRV_ITCCHEN_DIS	Intermediate transfer complete chaining is disabled.
	EDMA3_DRV_ITCCHEN_EN	Intermediate transfer complete chaining is enabled. When enabled, the chained event register (CER/CERH) bit is set on every intermediate chained transfer completion (upon completion of every intermediate TR in the PaRAM set, except the

		final TR in the PaRAM set). The bit (position) set in CER or CERH is the TCC value specified.
Enum EDMA3_DRV_TrigMode	EDMA3_DRV_TRIG_MODE_MANUAL	EDMA Trigger Mode Selection: Set the Trigger mode to Manual. The CPU manually triggers a transfer by writing a 1 to the corresponding bit in the event set register (ESR/ESRH).
	EDMA3_DRV_TRIG_MODE_QDMA	EDMA Trigger Mode Selection: Set the Trigger mode to QDMA. A QDMA transfer is triggered when a CPU (or other EDMA3 programmer) writes to the trigger word of the QDMA channel parameter set (auto-triggered) or when the EDMA3CC performs a link update on a PaRAM set that has been mapped to a QDMA channel (link triggered).
	EDMA3_DRV_TRIG_MODE_EVENT	EDMA Trigger Mode Selection: Set the Trigger mode to Event. Allows for a peripheral, system, or externally-generated event to trigger a transfer request.
Enum EDMA3_DRV_PaRAMEntry	EDMA3_DRV_PARAM_ENTRY_OPT	PaRAM Set Entry type: The OPT field (Offset Address 0h Bytes)
	EDMA3_DRV_PARAM_ENTRY_SRC	PaRAM Set Entry type: The SRC field (Offset Address 4h Bytes)
	EDMA3_DRV_PARAM_ENTRY_ACNT_BCNT	PaRAM Set Entry type: The (ACNT+BCNT) field (Offset Address 8h Bytes)
	EDMA3_DRV_PARAM_ENTRY_DST	PaRAM Set Entry type: The DST field (Offset Address Ch Bytes)
	EDMA3_DRV_PARAM_ENTRY_SRC_DST_BIDX	PaRAM Set Entry type: The (SRCBIDX+DSTBIDX) field (Offset Address 10h Bytes)
	EDMA3_DRV_PARAM_ENTRY_LINK_BCNTRLD	PaRAM Set Entry type: The (LINK+BCNTRLD) field (Offset Address 14h Bytes)
	EDMA3_DRV_PARAM_ENTRY_SRC_DST_CIDX	PaRAM Set Entry type: The (SRCCIDX+DSTCIDX) field (Offset Address 18h Bytes)
	EDMA3_DRV_PARAM_ENTRY_CCNT	PaRAM Set Entry type: The (CCNT+RSVD) field (Offset Address 1Ch Bytes)
Enum EDMA3_DRV_PaRAMField	EDMA3_DRV_PARAM_FIELD_OPT	PaRAM Set Field type: OPT field of PaRAM Set
	EDMA3_DRV_PARAM_FIELD_SRCADDR	PaRAM Set Field type: Starting byte address of Source. For FIFO

		mode, srcAddr must be a 256-bit aligned address.
	EDMA3_DRV_PARAM_FIELD_ACNT	PaRAM Set Field type: Number of bytes in each Array (ACNT)
	EDMA3_DRV_PARAM_FIELD_BCNT	PaRAM Set Field type: Number of Arrays in each Frame (BCNT)
	EDMA3_DRV_PARAM_FIELD_DESTADDR	PaRAM Set Field type: Starting byte address of destination. For FIFO mode, destAddr must be a 256-bit aligned address.
	EDMA3_DRV_PARAM_FIELD_SRCBIDX	PaRAM Set Field type: Index between consecutive arrays of a Source Frame (SRCBIDX). If SAM is set to 1 (via channelOptions), then srcInterArrIndex should be an even multiple of 32 bytes.
	EDMA3_DRV_PARAM_FIELD_DESTBIDX	PaRAM Set Field type: Index between consecutive arrays of a Destination Frame (DESTBIDX). If DAM is set to 1 (via channelOptions), then destInterArrIndex should be an even multiple of 32 bytes.
	EDMA3_DRV_PARAM_FIELD_LINKADDR	PaRAM Set Field type: Address for linking (Auto-Reloading of a PaRAM Set). This must point to a valid aligned 32-byte PaRAM set. A value of 0xFFFF means no linking. Linking is especially useful for use with ping-pong buffers and circular buffers.
	EDMA3_DRV_PARAM_FIELD_BCNTRELOAD	PaRAM Set Field type: Reload value of the numArrInFrame (BCNT). Relevant only for A-sync transfers.
	EDMA3_DRV_PARAM_FIELD_SRCCIDX	PaRAM Set Field type: Index between consecutive frames of a Source Block (SRCCIDX).
	EDMA3_DRV_PARAM_FIELD_DESTCIDX	PaRAM Set Field type: Index between consecutive frames of a Dest Block (DSTCIDX).
	EDMA3_DRV_PARAM_FIELD_CCNT	PaRAM Set Field type: Number of Frames in a block (CCNT).
Enum EDMA3_DRV_IoctlCmd	EDMA3_DRV_IOCTL_MIN_IOCTL	EDMA3 Driver IOCTL commands. Min IOCTL.
	EDMA3_DRV_IOCTL_SET_PARAM_CLEAR_OPTION	PaRAM Sets will be cleared OR will not be cleared during allocation, depending upon this option. For e.g., To clear the PaRAM Sets during allocation, cmdArg = (void *)1; To NOT clear the PaRAM Sets

		during allocation, cmdArg = (void *)0; For all other values, it will return error. By default, PaRAM Sets will be cleared during allocation. Note: Since this enum can change the behavior how the resources are initialized during their allocation, user is adviced to not use this command while allocating the resources. User should first change the behavior of resources' initialization and then should use start allocating resources.
	EDMA3_DRV_IOCTL_GET_PARAM_CLEAR_OPTION	To check whether PaRAM Sets will be cleared or not during allocation. If the value read is '1', it means that PaRAM Sets are getting cleared during allocation. If the value read is '0', it means that PaRAM Sets are NOT getting cleared during allocation. For e.g., unsigned short isParamClearingDone; cmdArg = &paramClearingRequired;
	EDMA3_DRV_IOCTL_MAX_IOCTL	Max IOCTL.
Enum EDMA3_DRV_Tc_Err	EDMA3_DRV_TC_ERR_BUSERR_DIS	Interrupt disable for bus error
	EDMA3_DRV_TC_ERR_BUSERR_EN	Interrupt enable for bus error
	EDMA3_DRV_TC_ERR_TRERR_DIS	Interrupt disable for transfer request error
	EDMA3_DRV_TC_ERR_TRERR_EN	Interrupt enable for transfer request error
	EDMA3_DRV_TC_ERR_MMRAERR_DIS	Interrupt disable for MMR address error
	EDMA3_DRV_TC_ERR_MMRAERR_EN	Interrupt enable for MMR address error
	EDMA3_DRV_TC_ERR_DIS	Disable all TC error interrupts
	EDMA3_DRV_TC_ERR_EN	Enable all TC error interrupts

3.2 Data Structures

This section summarizes the entire user visible data structure elements pertaining to the **EDMA3 Driver** run-time interfaces.

3.2.1 EDMA3_DRV_GblConfigParams

This configuration structure is used to specify the EDMA3 Resource Manager global settings, specific to the SoC. For e.g. number of DMA/QDMA channels, number of PaRAM sets, TCCs, event queues, transfer controllers, base addresses of CC global registers and TC registers, interrupt number for EDMA3 transfer completion, CC error, event queues' priority, watermark threshold level etc.

This configuration information is SoC specific and could be provided by the user at run-time while creating the EDMA3 Driver Object. In case user doesn't provide it, this information could be taken from the SoC specific configuration file edma3_<SOC_NAME>_cfg.c, in case it is available.

Member	**Description**
numDmaChannels	Number of DMA Channels supported by the underlying EDMA3 Controller
numQdmaChannels	Number of QDMA Channels supported by the underlying EDMA3 Controller
numTccs	Number of Interrupt Channels supported by the underlying EDMA3 Controller
numPaRAMSets	Number of PaRAM Sets supported by the underlying EDMA3 Controller
numEvtQueue	Number of Event Queues in the underlying EDMA3 Controller
numTcs	Number of Transfer Controllers (TCs) in the underlying EDMA3 Controller
numRegions	Number of Regions in the underlying EDMA3 controller
dmaChPaRAMMapExists	Channel mapping existence: A value of 0 (No channel mapping) implies that there is fixed association between a DMA channel and a PaRAM Set or, in other words, DMA channel n can ONLY use PaRAM Set n (No availability of DCHMAP registers) for transfers to happen. A value of 1 implies the presence of DCHMAP registers for the DMA channels and hence the flexibility of associating any DMA channel to any PaRAM Set. In other words, ANY PaRAM Set can be used for ANY DMA channel (like QDMA Channels).

memProtectionExists	Existence of memory protection feature
globalRegs	Base address of EDMA3 CC memory mapped registers.
tcRegs[EDMA3_MAX_TC]	Base address of EDMA3 TCs memory mapped registers.
xferCompleteInt	EDMA3 transfer completion interrupt line (could be different for ARM and DSP)
ccError	EDMA3 CC error interrupt line (could be different for ARM and DSP)
tcError[EDMA3_MAX_TC]	EDMA3 TCs error interrupt line (could be different for ARM and DSP)
evtQPri [EDMA3_MAX_EVT_QUE]	User can program the priority of the Event Queues at a system-wide level. This means that the user can set the priority of an IO initiated by either of the TCs (Transfer Controllers) relative to IO initiated by the other bus masters on the device (ARM, DSP, USB, etc).
evtQueueWaterMarkLvl [EDMA3_MAX_EVT_QUE]	To Configure the Threshold level of number of events that can be queued up in the Event queues. EDMA3CC error register (CCERR) will indicate whether or not at any instant of time the number of events queued up in any of the event queues exceeds or equals the threshold/watermark value that is set in the queue watermark threshold register (QWMTHRA).
tcDefaultBurstSize[EDMA3_MAX_TC]	To Configure the Default Burst Size (DBS) of TCs. An optimally-sized command is defined by the transfer controller default burst size (DBS). Different TCs can have different DBS values. It is defined in Bytes.
dmaChannelPaRAMMap [EDMA3_MAX_DMA_CH]	If channel mapping exists (DCHMAP registers are present), this array stores the respective PaRAM Set for each DMA channel. User can initialize each array member with a specific PaRAM Set or with EDMA3_DRV_CH_NO_PARAM_MAP. If channel mapping doesn't exist, it is of no use as the EDMA3 driver automatically uses the right PaRAM Set for that DMA channel.
dmaChannelTccMap [EDMA3_MAX_DMA_CH]	This array stores the respective TCC (interrupt channel) for each DMA channel. User can initialize each array member with a specific TCC or with EDMA3_DRV_CH_NO_TCC_MAP. This specific TCC code will be returned when the transfer is completed on the mapped DMA channel.
dmaChannelHwEvtMap [EDMA3_MAX_DMA_CHAN_DWRDS]	Each bit in this array corresponds to one DMA channel and tells whether this DMA channel is tied to any peripheral. That is whether any peripheral can send the synch event on this DMA channel or not. 1 means the channel is tied to some peripheral; 0 means it is not.

	DMA channels which are tied to some peripheral are RESERVED for that peripheral only. They are not allocated when user asks for 'ANY' DMA channel. All channels need not be mapped, some can be free also.

3.2.2 EDMA3_DRV_InstanceInitConfig

This configuration structure is used to specify which EDMA3 resources are owned and reserved by the EDMA3 driver instance. This configuration structure is shadow region specific and will be provided by the user at run-time while calling EDMA3_RM_open ().

Owned resources:

EDMA3 Driver Instances are tied to different shadow regions and hence different masters. Regions could be:

a) ARM,

b) DSP,

c) IMCOP (Imaging Co-processor) etc.

User can assign each EDMA3 resource to a shadow region using this structure. In this way, user specifies which resources are owned by the specific EDMA3 Driver Instance.

This assignment should also ensure that the same resource is not assigned to more than one shadow regions (unless desired in that way). Any assignment not following the above mentioned approach may have catastrophic consequences.

Reserved resources:

During EDMA3 driver initialization, user can reserve some of the EDMA3 resources for future use, by specifying which resources to reserve in the configuration data structure. These (critical) resources are reserved in advance so that they should not be allocated to someone else and thus could be used in future for some specific purpose.

User can request different EDMA3 resources using two methods:

a) by passing the resource type and the actual resource id,

b) by passing the resource type and ANY as resource id

For e.g. to request DMA channel 31, user will pass 31 as the resource id. But to request ANY available DMA channel (mainly used for memory-to-memory data transfer operations), user will pass EDMA3_DRV_DMA_CHANNEL_ANY as the resource id.

During initialization, user may have reserved some of the DMA channels for some specific purpose (mainly for peripherals using EDMA). These reserved DMA channels then will not be returned when user requests ANY as the resource id.

Same logic applies for QDMA channels and TCCs.

For PaRAM Set, there is one difference. If the DMA channels are one-to-one tied to their respective PaRAM Sets (i.e. user cannot 'choose' the PaRAM Set for a particular DMA channel), EDMA3 Driver automatically reserves all those PaRAM Sets which are tied to the DMA channels. Then those PaRAM Sets would not be returned when user requests for ANY PaRAM Set (specifically for linking purpose). This is done in order to avoid allocating the PaRAM Set, tied to a particular DMA channel, for linking purpose. If this constraint is not there, that DMA channel thus could not be used at all, because of the unavailability of the desired PaRAM Set.

Member	**Description**
ownPaRAMSets [EDMA3_MAX_PARAM_DWRDS]	PaRAM Sets owned by the EDMA3 Driver Instance.
ownDmaChannels [EDMA3_MAX_DMA_CHAN_DWRDS]	DMA channels owned by the EDMA3 Driver Instance.
ownQdmaChannels [EDMA3_MAX_QDMA_CHAN_DWRDS]	QDMA channels owned by the EDMA3 Driver Instance.
ownTccs [EDMA3_MAX_TCC_DWRDS]	TCCs owned by the EDMA3 Driver Instance.
resvdPaRAMSets [EDMA3_MAX_PARAM_DWRDS]	PaRAM Sets reserved during initialization for future use. These will not be given when user requests for ANY available PaRAM Set using 'EDMA3_DRV_LINK_CHANNEL' as resource/channel id.
resvdDmaChannels [EDMA3_MAX_DMA_CHAN_DWRDS]	DMA channels reserved during initialization for future use. These will not be given when user requests for ANY available DMA channel using 'EDMA3_DRV_DMA_CHANNEL_ANY' as resource/channel id.
resvdQdmaChannels [EDMA3_MAX_QDMA_CHAN_DWRDS]	QDMA channels reserved during initialization for future use. These will not be given when user requests for ANY available QDMA channel using 'EDMA3_DRV_QDMA_CHANNEL_ANY' as resource/channel id.
resvdTccs [EDMA3_MAX_TCC_DWRDS]	TCCs reserved during initialization for future use. These will not be given when user requests for ANY available TCC using 'EDMA3_DRV_TCC_ANY' as resource/TCC id.

3.2.3 EDMA3_DRV_InitConfig

This configuration structure is used to initialize the EDMA3 Driver Instance. This configuration information is passed while opening the driver instance.

Member	**Description**
regionId	Shadow region identifier. Note that only one EDMA3 driver instance can be opened for each shadow region.
isMaster	It tells whether the EDMA3 driver instance is Master or not. Only the shadow region associated with this master instance will receive the EDMA3 interrupts (if enabled).
drvInstInitConfig	EDMA3 resources related shadow region specific information. Which all EDMA3 resources are owned and reserved by this particular instance are told in this configuration structure. User can also pass this structure as NULL. In that case, default static configuration would be taken from the platform specific configuration files (part of the Resource Manager), if available.
drvSemHandle	Driver Instance specific semaphore handle. It is used to share EDMA3 resources (DMA/QDMA channels, PaRAM Sets, TCCs etc) among different users.
gblerrCb	Driver Instance wide global callback function to catch non-channel specific errors from the Channel Controller. for e.g., TCC error, queue threshold exceed error etc.
gblerrData	Application data to be passed back to the global error callback function

3.2.4 EDMA3_DRV_MiscParam

This configuration structure is used to specify some misc options while creating the Driver object. New options may also be added into this structure in future.

Member	**Description**
isSlave	In a multi-master system (for e.g. ARM + DSP), this option is used to distinguish between Master and Slave. Only the Master is allowed to program the global EDMA3 registers (like Queue priority, Queue water-mark level, error registers etc).
param	For future use

3.2.5 EDMA3_DRV_ChainOptions

This configuration structure is used to configure the interrupt (final and intermediate) generation and chaining (final and intermediate) options.

Member	Description
tcchEn	Transfer complete chaining enable. When enabled, the chained event register (CER/CERH) bit is set on final chained transfer completion (upon completion of the final/last TR in the PaRAM set). The bit (position) set in CER or CERH is the TCC value specified.
itcchEn	Intermediate transfer completion chaining enable. When enabled, the chained event register (CER/CERH) bit is set on every intermediate chained transfer completion (upon completion of every intermediate TR in the PaRAM set, except the final TR in the PaRAM set). The bit (position) set in CER or CERH is the TCC value specified.
tcintEn	Transfer complete interrupt enable. When enabled, the interrupt pending register (IPR/IPRH) bit is set on transfer completion (upon completion of the final TR in the PaRAM set). The bit (position) set in IPR or IPRH is the TCC value specified. In order to generate a completion interrupt to the CPU, the corresponding Interrupt Enable Register: TCC (IER [TCC]/IERH [TCC]) bit must be set to 1.
itcintEn	Intermediate transfer completion interrupt enable. When enabled, the interrupt pending register (IPR/IPRH) bit is set on every intermediate transfer completion (upon completion of every intermediate TR in the PaRAM set, except the final TR in the PaRAM set). The bit (position) set in IPR or IPRH is the TCC value specified. In order to generate a completion interrupt to the CPU, the corresponding Interrupt Enable Register: TCC (IER[TCC]/IERH[TCC]) bit must be set to 1.

3.2.6 EDMA3_DRV_PaRAMRegs

This configuration structure is EDMA3 PaRAM Set in user configurable format. This is a mapping of the EDMA3 PaRAM set provided to the user for ease of modification of the individual fields.

Member	Description
opt	OPT field of PaRAM Set. It consists of various transfer related configuration options. Like interrupt generation options, chaining options, FIFO related options etc.
srcAddr	The 32-bit source address parameter specifies the starting byte address of the source. For FIFO mode transfers, user must program the source address to be aligned to a 256-bit aligned address (5 LSBs of address must be 0). The EDMA3TC will signal an error if this rule is violated.
aCnt	ACNT represents the number of bytes within the 1st dimension of a transfer. ACNT is a 16-bit unsigned value with valid values between 0 and 65535. Therefore, the maximum number of bytes in an array is 65535 bytes. ACNT must be greater than or equal to 1 for a TR to be submitted to EDMA3TC. An ACNT equal to 0 is considered either a null or dummy transfer. A dummy or null transfer generates a completion code depending on the settings of the completion bit fields in OPT.
bCnt	BCNT is a 16-bit unsigned value that specifies the number of arrays of length ACNT. For normal operation, valid values for BCNT are between 1 and 65535. Therefore, the maximum number of arrays in a frame is 65535. A BCNT equal to 0 is considered either a null or dummy transfer. A dummy or null transfer generates a completion code depending on the settings of the completion bit fields in OPT.
destAddr	The 32-bit destination address parameter specifies the starting byte address of the destination. For FIFO mode, user must program the destination address to be aligned to a 256-bit aligned address (5 LSBs of address must be 0). The EDMA3TC will signal an error if this rule is violated.
srcBIdx	SRCBIDX is a 16-bit signed value (2s complement) used for source address modification between each array in the 2nd dimension. Valid values for SRCBIDX are between –32768 and 32767. It provides a byte address offset from the beginning of the source array to the beginning of the next source array. It applies to both A-synchronized and AB-synchronized transfers.
destBIdx	DSTBIDX is a 16-bit signed value (2s complement) used for destination address modification between each array in the 2nd dimension. Valid values for DSTBIDX are between –32768 and 32767. It provides a byte address offset from the beginning of the destination array to the beginning of the next destination array within the current frame. It applies to both A-synchronized and AB-synchronized transfers.
linkAddr	The EDMA3CC provides a mechanism, called linking, to reload the current PaRAM set upon its natural termination (that is, after the count fields are

	decremented to 0) with a new PaRAM set. The 16-bit parameter LINK specifies the byte address offset in the PaRAM from which the EDMA3CC loads/reloads the next PaRAM set during linking. User should make sure to program the LINK field correctly, so that link update is requested from a PaRAM address that falls in the range of the available PaRAM addresses on the device. A LINK value of FFFFh is referred to as a NULL link that should cause the EDMA3CC to perform an internal write of 0 to all entries of the current PaRAM set, except for the LINK field that is set to FFFFh.
bCntReload	BCNTRLD is a 16-bit unsigned value used to reload the BCNT field once the last array in the 2nd dimension is transferred. This field is only used for A-synchronized transfers. In this case, the EDMA3CC decrements the BCNT value by 1 on each TR submission. When BCNT (conceptually) reaches 0, the EDMA3CC decrements CCNT and uses the BCNTRLD value to reinitialize the BCNT value. For AB-synchronized transfers, the EDMA3CC submits the BCNT in the TR and the EDMA3TC decrements BCNT appropriately. For AB-synchronized transfers, BCNTRLD is not used.
srcCIdx	SRCCIDX is a 16-bit signed value (2s complement) used for source address modification in the 3rd dimension. Valid values for SRCCIDX are between –32768 and 32767. It provides a byte address offset from the beginning of the current array (pointed to by SRC address) to the beginning of the first source array in the next frame. It applies to both A-synchronized and AB-synchronized transfers.
destCIdx	DSTCIDX is a 16-bit signed value (2s complement) used for destination address modification in the 3rd dimension. Valid values are between –32768 and 32767. It provides a byte address offset from the beginning of the current array (pointed to by DST address) to the beginning of the first destination array TR in the next frame. It applies to both A-synchronized and AB-synchronized transfers.
cCnt	CCNT is a 16-bit unsigned value that specifies the number of frames in a block. Valid values for CCNT are between 1 and 65 535. Therefore, the maximum number of frames in a block is 65 535 (64K – 1 frames). A CCNT equal to 0 is considered either a null or dummy transfer. A dummy or null transfer generates a completion code depending on the settings of the completion bit fields in OPT. A CCNT value of 0 is considered either a null or dummy transfer.

3.2.7 EDMA3_DRV_EvtQuePriority

This configuration structure is used to set the event queues' priorities. It allows to change the priority of the individual queues and the priority of the transfer request (TR) associated with the events queued in the queue.

3.2.8 EDMA3_DRV_GblXbarToChanConfigParams

This configuration structure is used to map the cross bar events to DMA channels. This setting is done at initialization time. For the cross bar event if the DMA channel is to be mapped then DMA channel number is stored in the event array location, otherwise -1 is written.

3.3 API Specification

The application programming interface (API) for the **EDMA3 Driver** can be found at:

EDMA3_Driver.chm

3.4 EDMA3 Driver Initialization

EDMA3 Driver should be initialized first before it can be used by the peripheral drivers or application. During initialization, EDMA3 driver object is created first and then a region specific EDMA3 driver instance is opened. Following are the APIs which are used for the initialization:

```
/* EDMA3 Driver Object Creation */

EDMA3_DRV_Result EDMA3_DRV_create (unsigned int
phyCtrllerInstId, const EDMA3_DRV_GblConfigParams
*gblCfgParams, const void *param)

/* EDMA3 Driver Instance Opening */

EDMA3_DRV_Result EDMA3_DRV_open (unsigned int
phyCtrllerInstId, const EDMA3_DRV_InitConfig *initCfg,
EDMA3_DRV_Result *errorCode)
```

These APIs should be mandatorily called once by the global initialization routine or by the user itself, for EDMA3 driver functioning. Also, they can be called further for other usage.

Note 1: During the initialization sequence, EDMA3 Driver, being an OS independent module, doesn't register various interrupt handlers with the underlying OS. The application which is using the EDMA3 Driver should register the various Interrupt Handlers (ISRs in Resource Manager) with the underlying OS on which it is running. Similarly, the application should un-register the previously registered Interrupt Handlers when the Driver instance is no more required.

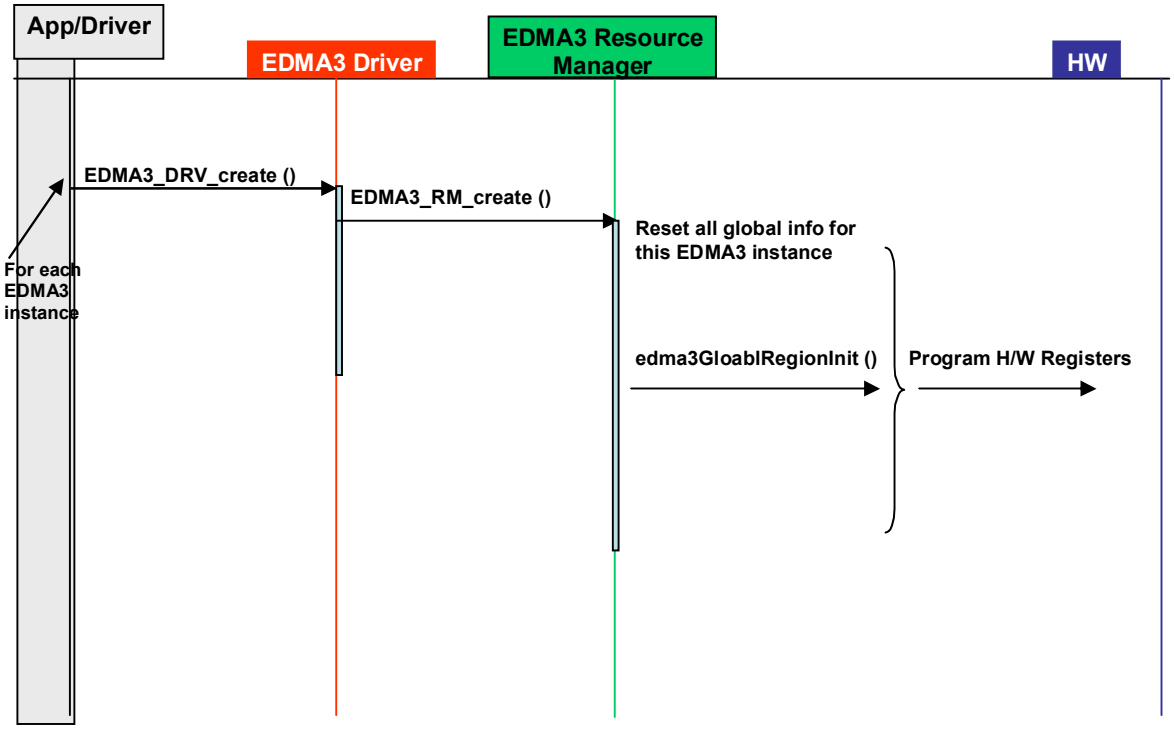
Note 2: While un-registering the interrupt handlers, it should be taken care by the application that no other applications, using the interrupt functionality, are functioning. Otherwise, the un-registration done by one application may stop other applications. The un-registration should be done only when no more applications, using the interrupt functionality, are functioning.

Note 3: While initialization of the driver if the platform supports cross bar events then EDMA3_DRV_initXbarEventMap function have to be called before allocating any channel to cross bar mapped event.

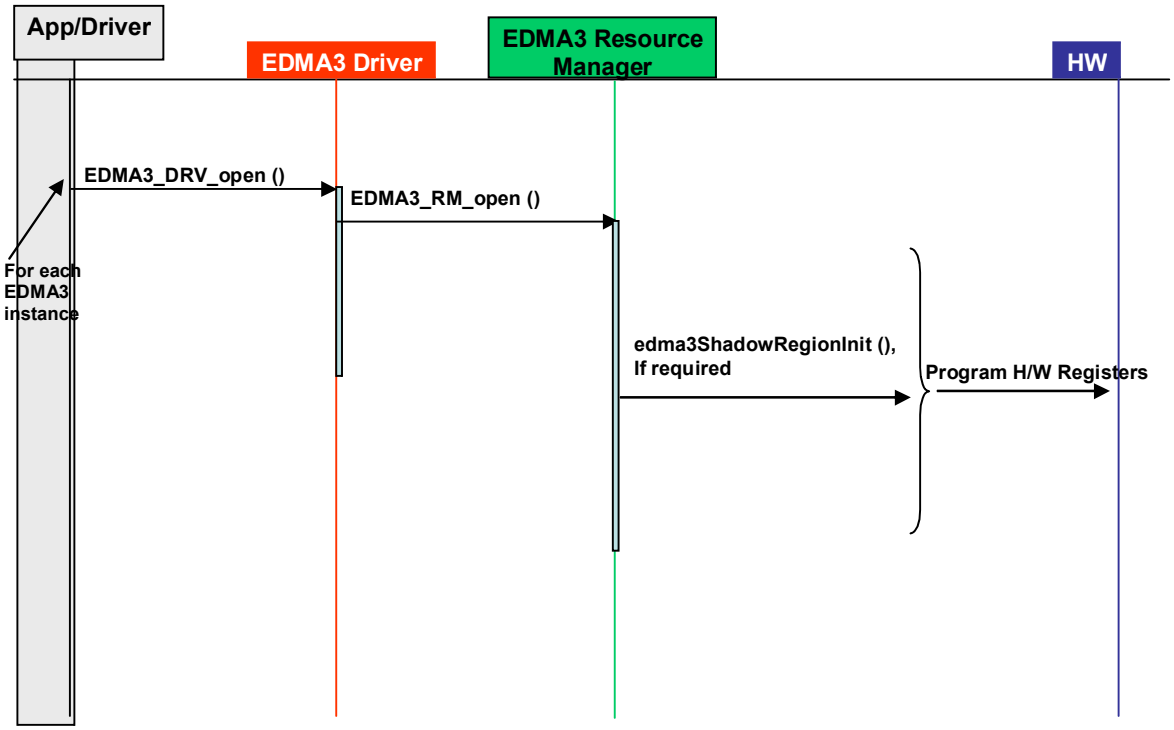
3.5 API Flow Diagram

Below are the flow diagrams for some **EDMA3 Driver** APIs which interact with the **EDMA3 Resource Manager** for their functioning.

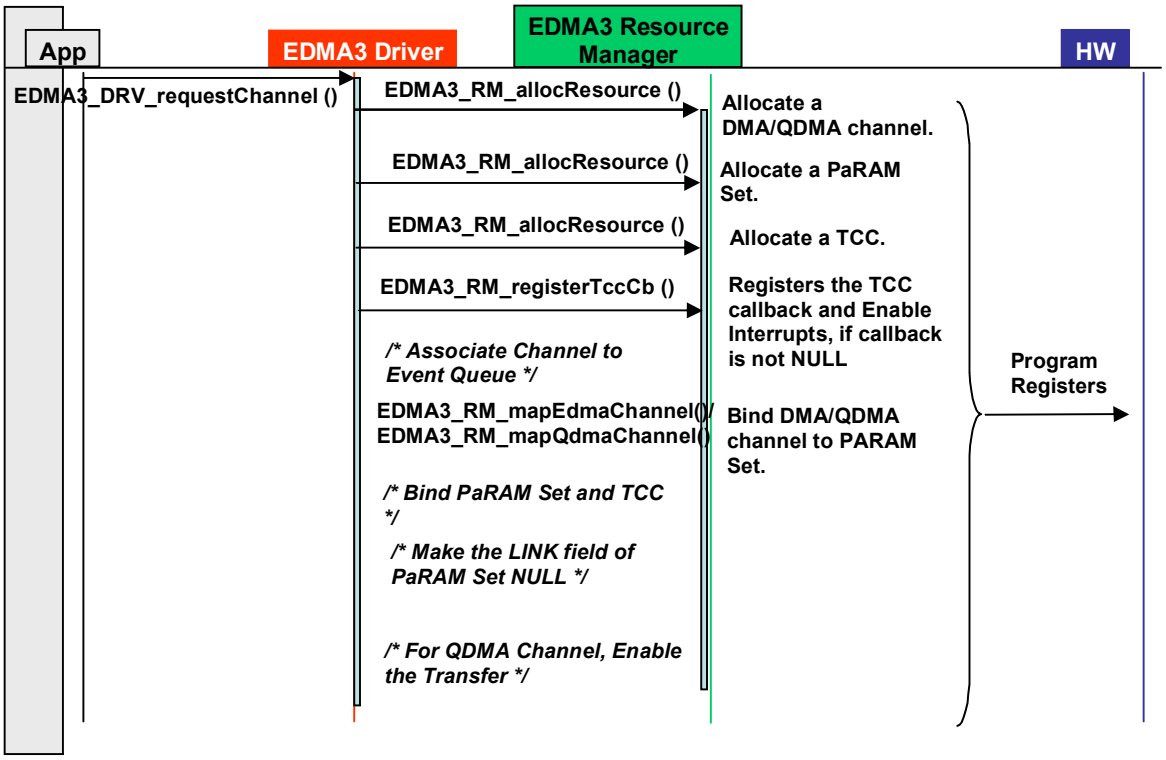
3.5.1 EDMA3 Driver Creation



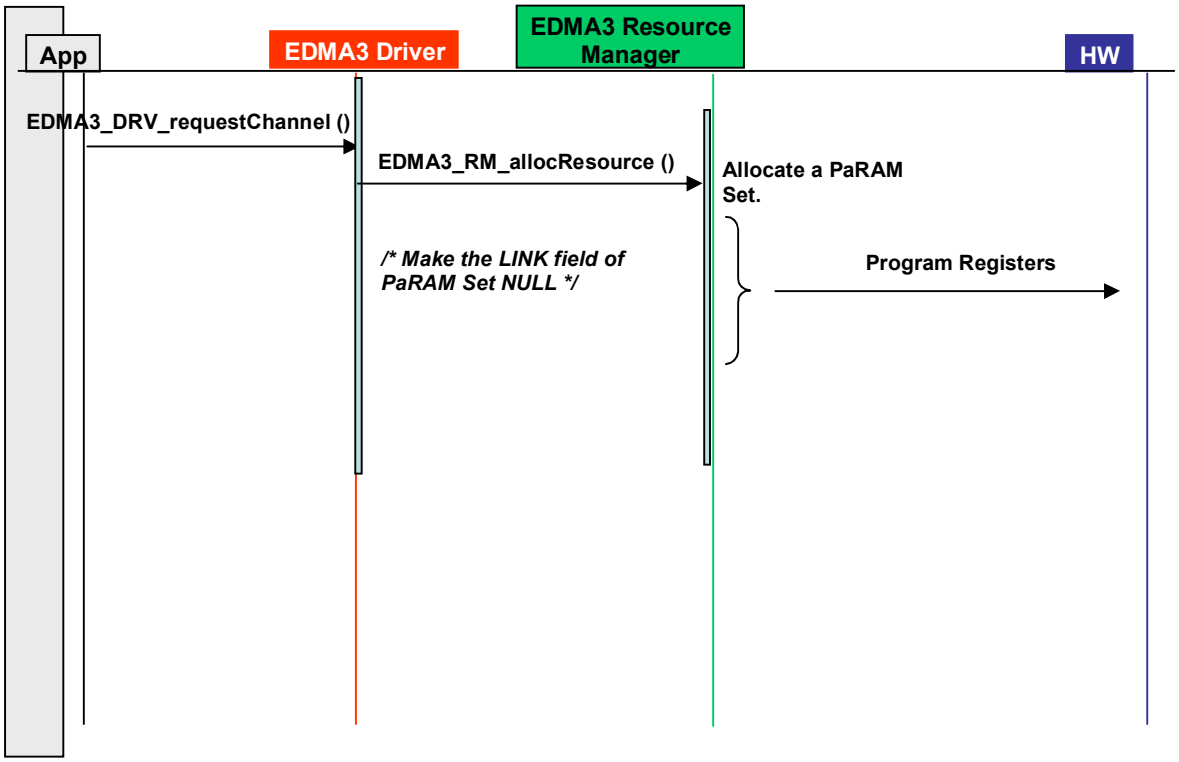
3.5.2 EDMA3 Open



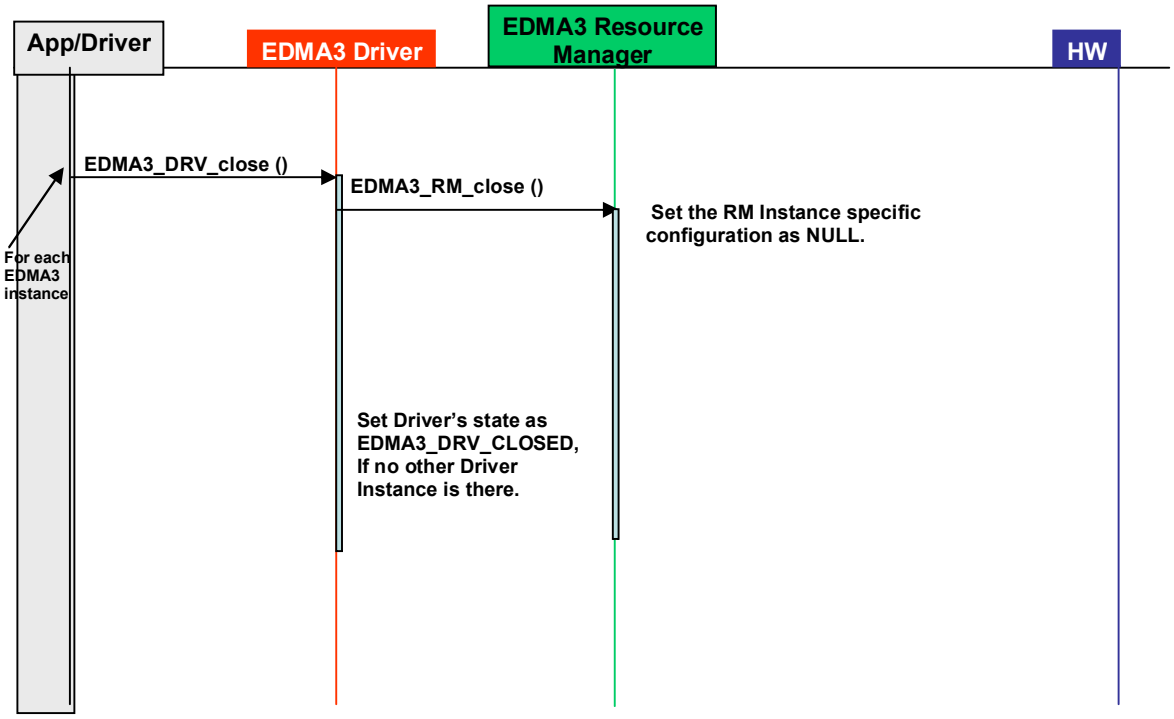
3.5.3 EDMA3 Request Channel (DMA / QDMA Channel)



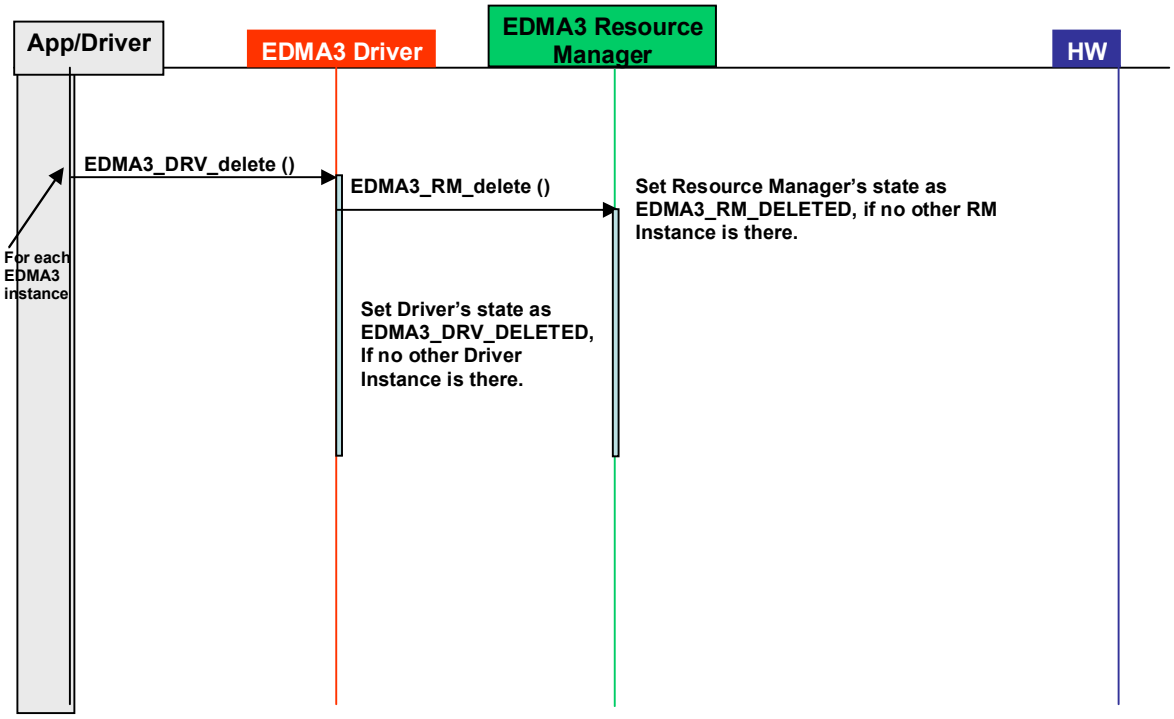
3.5.4 EDMA3 Request Channel (LINK Channel)



3.5.5 EDMA3 Close



3.5.6 EDMA3 Delete



3.6 API Usage Example

Below are the steps required to create the Driver Object and then initialize a region specific Driver Instance. Afterwards, if required, the application has to register the various interrupt handlers with the underlying OS.

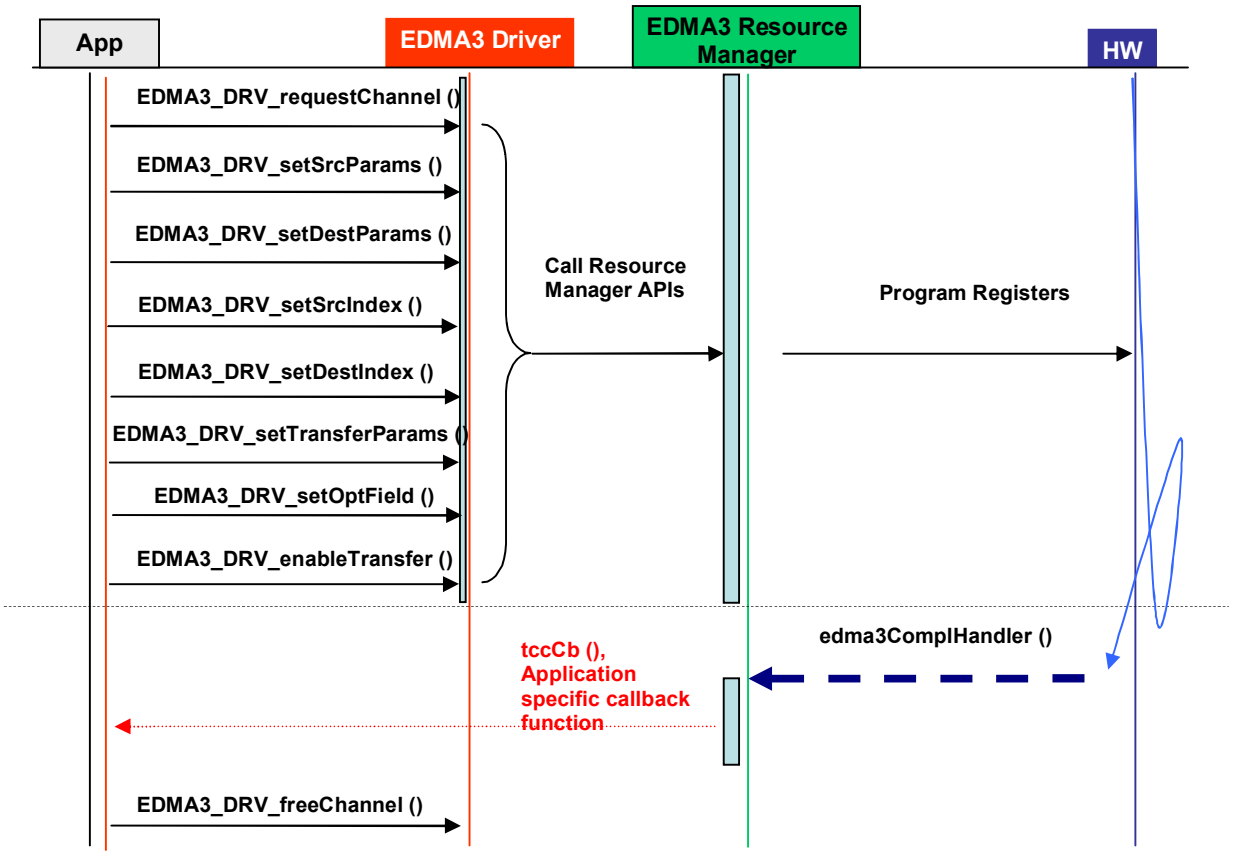
After the successful opening, the Driver instance can be used to call other EDMA3 Driver APIs.

/* Below are the steps required to create the Driver Object and then initialize a region specific Driver Instance. Afterwards, if required, the application has to register the various interrupt handlers with the underlying OS. */

```
/** EDMA3 Driver Instance specific Semaphore handle */
extern EDMA3_OS_Sem_Handle semHandle[];

EDMA3_DRV_Handle edma3init (unsigned int edma3Id, EDMA3_DRV_Result *errorCode)
   {
   EDMA3_DRV_InitConfig initCfg;
   EDMA3_DRV_Result edma3Result = EDMA3_DRV_SOK;
   Semaphore_Params semParams;
   EDMA3_RM_MiscParam miscParam;
   EDMA3_DRV_Handle hEdma = NULL;

      /* configuration structure for the Driver */
      initCfg.isMaster = TRUE;
      initCfg.regionId = (EDMA3_RM_RegionId)1u;
      initCfg.drvSemHandle = NULL;
      /* Driver instance specific config NULL */
      initCfg.drvInstInitConfig = NULL;
      initCfg.gblerrCb = NULL;
      initCfg.gblerrData = NULL;

      miscParam.isSlave = FALSE;

      /* Create EDMA3 Driver Object first. */
      edma3Result = EDMA3_DRV_create (edma3InstanceId, NULL, (void *)&miscParam);
      if (edma3Result != EDMA3_DRV_SOK)
         {
         /* Report error */
         return hEdma;
         }
      else
         {
         /**
           * Driver Object created successfully.
           * Create a semaphore now for driver instance.
           */
         Semaphore_Params_init(&semParams);
         edma3Result = edma3OsSemCreate(1, & semParams, &initCfg.drvSemHandle);
         if (edma3Result != EDMA3_DRV_SOK)
            {
            /* Report error */
            return hEdma;
            }
         else
            {
            /* Save the semaphore handle for future use */
            semHandle[edma3Id] = initCfg.drvSemHandle;

            /* Open the Driver Instance */
            hEdma = EDMA3_DRV_open (edma3InstanceId, (void *) &initCfg, &edma3Result);
            if(NULL == hEdma)
               {
               /* Report error */
               return hEdma;
               }
```

```
            else
              {
#if CROSS_BAR_EVENTS_
               /* Initialize the Cross bar map function
               */
               {
               if(hEdma && (edma3Result == EDMA3_DRV_SOK))
                    {
                    edma3Result = sampleInitXbarEvt(hEdma, edma3Id);
                    }
               }
#endif
               /**
                * Register Interrupt Handlers for various interrupts
                * like transfer completion interrupt, CC error
                * interrupt, TC error interrupts etc, if required.
                */
               /*  registerEdma3Interrupts(edma3Id); */
               }
            }
         }
      }

   *errorCode = edma3Result;
    return hEdma;
   }
```

Below is the flow diagram for an application requesting a DMA channel to transfer data. After the transfer completion, EDMA3 Resource Manager calls the application specific call-back function, along with the status code.



Below is the sample code describing the steps required to close the already opened EDMA3 Driver Instance and then delete the EDMA3 Driver Object. It should be done when EDMA3 driver functionality is no more required.

```
EDMA3_DRV_Result edma3deinit (unsigned int edma3Id, EDMA3_DRV_Handle hEdma)
   {
   EDMA3_DRV_Result edma3Result = EDMA3_DRV_E_INVALID_PARAM;

   /* Unregister Interrupt Handlers first, if required */
   /* unregisterEdma3Interrupts(edma3Id); */

   /* Delete the semaphore */
   edma3Result = edma3OsSemDelete(semHandle[edma3Id]);

   if (EDMA3_DRV_SOK == edma3Result )
      {
      /* Make the semaphore handle as NULL. */
      semHandle[edma3Id] = NULL;

      /* Now, close the EDMA3 Driver Instance */
      edma3Result = EDMA3_DRV_close (hEdma, NULL);
      }

   if (EDMA3_DRV_SOK == edma3Result )
      {
      /* Now, delete the EDMA3 Driver Object */
      edma3Result = EDMA3_DRV_delete (edma3Id, NULL);
      }

   return edma3Result;
   }
```

Chapter 4

EDMA3 Driver Porting

This chapter discusses how to port **EDMA3 Driver (and EDMA3 Resource Manager)** to other supported target platforms and operating systems.

3.7 Getting Started

The **EDMA3 Driver** is based upon PSP Framework architecture making portability and re-usability as prime requirements. Based upon the architecture, the EDMA3 Driver is made like it can be ported to another platform very easily. EDMA3 Driver itself is completely platform independent. So for its proper functioning, user has to provide the platform specific configuration, which will be used by the Resource Manager internally for managing all the resources.

The platform specific configuration can be provided in two ways:

a) Provide the configuration during init time only while calling the APIs: EDMA3_DRV_create () (for providing the global hardware specific configuration) and EDMA3_DRV_open () (for providing the shadow regions specific configuration), OR,

b) Create the platform specific configuration file "edma3_<PLATFORM_NAME>_cfg.c" in "edma3_lld_<VERSION_NUMBER>\packages\ti\sdo\edma3\rm\src\configs" folder, if it is not already there. Use this configuration file as input and generate the required platform specific library.

Support is already provided for many platforms; please see the release notes for more details. To port to a new platform, user is advised to look the existing files.

Also, the EDMA3 Driver module is completely OS-agnostic, for make it's porting to a different OS completely hassle-free. It is designed in such a way that the OS dependent part has to be provided by the user for its proper functioning. This is done in order to make the EDMA3 Driver OS independent.

The following OS dependent part of the EDMA3 Package has to be provided by the user:

a) **Critical section entry and exit functions**: They should be implemented by the application for proper linking with the EDMA3 Driver. The Driver uses these functions for proper sharing of resources (among various users) and for other purposes and assumes the implementation of these functions to be provided by the application. Without the definitions being provided, the image won't get linked properly.

/ Entry to critical section */**

extern void edma3OsProtectEntry (unsigned int edma3InstanceId, int level, unsigned int *intState);

/ Exit from critical section */**

extern void edma3OsProtectExit (unsigned int edma3InstanceId, int level, unsigned int intState);

These APIs should be mandatorily implemented once by the global initialization routine or by the user itself, for proper linking.

b) **Semaphore related functions**: They should be implemented by the application for proper linking with the EDMA3 Driver and Resource Manager. The EDMA3 Resource Manager uses these functions for proper sharing of resources (among various users) and assumes the implementation of these functions to be provided by the application. Without the definitions being provided, the image won't get linked properly.

```
/** EDMA3 OS Semaphore Take */

extern EDMA3_DRV_Result edma3OsSemTake
(EDMA3_OS_Sem_Handle hSem, int mSecTimeout);

/** EDMA3 OS Semaphore Give */

extern EDMA3_DRV_Result edma3OsSemGive
(EDMA3_OS_Sem_Handle hSem);
```

c) **Interrupts registration and un-registration:** It is not done by the EDMA3 Driver or the Resource Manager. The application which is using the EDMA3 Driver should register the various Interrupt Handlers (ISRs in Resource Manager) with the underlying OS on which it is running. Similarly, the application should un-register the previously registered Interrupt Handlers when the Driver instance is no more required.

Public header file "edma3_lld_<VERSION_NUMBER>\packages\ti\sdo\edma3\rm\edma3_common.h" contains all the OS dependent part which needs to be provided by the user application.

Sample initialization libraries are already provided for multiple platforms which provide the DSP/BIOS 6 side OS adaptation layer implementation and platform specific configuration for proper functioning of the EDMA3 Driver. User is encouraged to look at them and use them in the porting activity.

3.8 Step-by-Step procedure for porting

This section provides illustrative description on how to port the EDMA3 Driver to the selected platform and the OS.

3.8.1 edma3_<PLATFORM_NAME>_cfg.c:

EDMA3_DRV_GblConfigParams is the initialization structure which is used to specify the EDMA3 Hardware specific global settings, specific to the SoC. For e.g. number of DMA/QDMA channels, number of PaRAM sets, TCCs, event queues, transfer controllers, base addresses of CC global registers and TC registers, interrupt number for EDMA3 transfer completion, CC error, event queues' priority, watermark threshold level etc. This configuration information is SoC specific and could be provided by the user at run-time also while creating the EDMA3 Driver object. In case user doesn't provide it, this information will be taken from the configuration file, in case it is available for the specific SoC.

Similarly, *EDMA3_DRV_InstanceInitConfig* is the initialization structure which is used to specify the EDMA3 Resource Manager Region specific settings. For e.g. resources (DMA/QDMA channels, PaRAM sets, TCCs) owned and reserved by this EDMA3 driver instance. This configuration information is shadow region (or master) specific and could be provided by the user at run-time while creating the EDMA3 Driver instance. In case user doesn't provide it, this information will be taken from the configuration file, in case it is available for the specific SoC for the specific shadow region.

To summarize, this file contains the global and region specific configuration information for EDMA3 for the specific platform. User can create this file by adding the desired information for the new SoC, or he/she can provide this info at init-time.

User can find the sample configuration files for different platforms at:
"edma3_lld_<VERSION_NUMBER>\packages\ti\sdo\edma3\rm\src\configs". On the same lines, user can create different configuration file for another platform.

3.8.2 Package.bld file for the Resource Manager

Platform specific EDMA3 configuration file will be included as a source file in the package.bld file. The bld file has variable arrays which will be used to generate the platform specific Resource Manager libraries.

User can find the package.bld file at "edma3_lld_<VERSION_NUMBER>\packages\ti\sdo\edma3\rm\" and modify it appropriately to add support for the desired platform.

3.8.3 OS-dependent (sample) Implementation

Following is the sample implementation of OS dependent functions.

DSP/BIOS version 6.10.00.28 is the reference OS chosen here for the DA830 platform.

/* Below is the sample configuration file which specifies EDMA3 hardware related information like number of transfer controllers, various interrupt ids etc. It is used while interrupts enabling / disabling, in the sample application. */

```
/* DA830 Specific EDMA3 Information */

#include <ti/sdo/edma3/drv/edma3_drv.h>

/** Number of PaRAM Sets available */
#define EDMA3_NUM_PARAMSET                  128u
/** Number of TCCS available */
#define EDMA3_NUM_TCC                       32u
/** Number of Event Queues available */
#define EDMA3_NUM_EVTQUE                    2u
/** Number of Transfer Controllers available */
#define EDMA3_NUM_TC                        2u

/** Interrupt no. for Transfer Completion */
#define EDMA3_CC_XFER_COMPLETION_INT        8u
/** Interrupt no. for CC Error */
#define EDMA3_CC_ERROR_INT                  56u
/** Interrupt no. for TCs Error */
#define EDMA3_TC0_ERROR_INT                 57u
#define EDMA3_TC1_ERROR_INT                 58u
#define EDMA3_TC2_ERROR_INT                 0u
#define EDMA3_TC3_ERROR_INT                 0u
#define EDMA3_TC4_ERROR_INT                 0u
#define EDMA3_TC5_ERROR_INT                 0u
#define EDMA3_TC6_ERROR_INT                 0u
#define EDMA3_TC7_ERROR_INT                 0u

/**
* EDMA3 interrupts (transfer completion, CC error etc.) correspond to different
* ECM events (SoC specific). These ECM events come
* under ECM block XXX (handling those specific ECM events). Normally, block
* 0 handles events 4-31 (events 0-3 are reserved), block 1 handles events
* 32-63 and so on. This ECM block XXX (or interrupt selection number XXX)
* is mapped to a specific HWI_INT YYY in the tcf file.
* Define EDMA3_HWI_INT_XFER_COMP to specific HWI_INT, corresponding
* to transfer completion interrupt.
* Define EDMA3_HWI_INT_CC_ERR to specific HWI_INT, corresponding
* to CC error interrupts.
* Define EDMA3_HWI_INT_TC_ERR to specific HWI_INT, corresponding
* to TC error interrupts.
*/
#define EDMA3_HWI_INT_XFER_COMP             (6u)
#define EDMA3_HWI_INT_CC_ERR                (7u)
#define EDMA3_HWI_INT_TC_ERR                (7u)
```

```
/**
 * \brief Mapping of DMA channels 0-31 to Hardware Events from
 * various peripherals, which use EDMA for data transfer.
 * All channels need not be mapped, some can be free also.
 * 1: Mapped
 * 0: Not mapped
 *
 * This mapping will be used to allocate DMA channels when user passes
 * EDMA3_DRV_DMA_CHANNEL_ANY as dma channel id (for eg to do memory-to-memory
 * copy). The same mapping is used to allocate the TCC when user passes
 * EDMA3_DRV_TCC_ANY as tcc id (for eg to do memory-to-memory copy).
 *
 * To allocate more DMA channels or TCCs, one has to modify the event mapping.
 */
                                                        /* 31    0 */
#define EDMA3_DMA_CHANNEL_TO_EVENT_MAPPING_0        (0xCFFFFFFFu)

/**
 * \brief Mapping of DMA channels 32-63 to Hardware Events from
 * various peripherals, which use EDMA for data transfer.
 * All channels need not be mapped, some can be free also.
 * 1: Mapped
 * 0: Not mapped
 *
 * This mapping will be used to allocate DMA channels when user passes
 * EDMA3_DRV_DMA_CHANNEL_ANY as dma channel id (for eg to do memory-to-memory
 * copy). The same mapping is used to allocate the TCC when user passes
 * EDMA3_DRV_TCC_ANY as tcc id (for eg to do memory-to-memory copy).
 *
 * To allocate more DMA channels or TCCs, one has to modify the event mapping.
 */
/* DMA channels 32-63 DOES NOT exist in DA830. */
#define EDMA3_DMA_CHANNEL_TO_EVENT_MAPPING_1        (0x0u)

/* Variable which will be used internally for referring number of Event Queues. */
unsigned int numEdma3EvtQue = EDMA3_NUM_EVTQUE;

/* Variable which will be used internally for referring number of TCs. */
unsigned int numEdma3Tc = EDMA3_NUM_TC;

/**
 * Variable which will be used internally for referring transfer completion
 * interrupt.
 */
unsigned int ccXferCompInt = EDMA3_CC_XFER_COMPLETION_INT;

/**
 * Variable which will be used internally for referring channel controller's
 * error interrupt.
 */
unsigned int ccErrorInt = EDMA3_CC_ERROR_INT;

/**
 * Variable which will be used internally for referring transfer controllers'
 * error interrupts.
 */
unsigned int tcErrorInt[8] =    {
                        EDMA3_TC0_ERROR_INT, EDMA3_TC1_ERROR_INT,
                        EDMA3_TC2_ERROR_INT, EDMA3_TC3_ERROR_INT,
                        EDMA3_TC4_ERROR_INT, EDMA3_TC5_ERROR_INT,
                        EDMA3_TC6_ERROR_INT, EDMA3_TC7_ERROR_INT
                        };
```

```
/**
 * Variables which will be used internally for referring the hardware interrupt
 * for various EDMA3 interrupts.
 */
unsigned int hwIntXferComp = EDMA3_HWI_INT_XFER_COMP;
unsigned int hwIntCcErr = EDMA3_HWI_INT_CC_ERR;
unsigned int hwIntTcErr = EDMA3_HWI_INT_TC_ERR;


/* Driver Object Initialization Configuration */
EDMA3_DRV_GblConfigParams sampleEdma3GblCfgParams =
   {
   /** Total number of DMA Channels supported by the EDMA3 Controller */
   32u,
   /** Total number of QDMA Channels supported by the EDMA3 Controller */
   8u,
   /** Total number of TCCs supported by the EDMA3 Controller */
   32u,
   /** Total number of PaRAM Sets supported by the EDMA3 Controller */
   128u,
   /** Total number of Event Queues in the EDMA3 Controller */
   2u,
   /** Total number of Transfer Controllers (TCs) in the EDMA3 Controller */
   2u,
   /** Number of Regions on this EDMA3 controller */
   4u,
   /**
    * \brief Channel mapping existence
    * A value of 0 (No channel mapping) implies that there is fixed association
    * for a channel number to a parameter entry number or, in other words,
    * PaRAM entry n corresponds to channel n.
    */
   0u,
   /** Existence of memory protection feature */
   0u,
   /** Global Register Region of CC Registers */
   (void *)0x01C00000u,
   /** Transfer Controller (TC) Registers */
      {
      (void *)0x01C10000u,
      (void *)0x01C10400u,
      (void *)NULL,
      (void *)NULL,
      (void *)NULL,
      (void *)NULL,
      (void *)NULL,
      (void *)NULL
      },
   /** Interrupt no. for Transfer Completion */
   EDMA3_CC_XFER_COMPLETION_INT,
   /** Interrupt no. for CC Error */
   EDMA3_CC_ERROR_INT,
   /** Interrupt no. for TCs Error */
      {
      EDMA3_TC0_ERROR_INT,
      EDMA3_TC1_ERROR_INT,
      EDMA3_TC2_ERROR_INT,
      EDMA3_TC3_ERROR_INT,
      EDMA3_TC4_ERROR_INT,
      EDMA3_TC5_ERROR_INT,
      EDMA3_TC6_ERROR_INT,
      EDMA3_TC7_ERROR_INT
      },
```

```
    /**
     * \brief EDMA3 TC priority setting
     *
     * User can program the priority of the Event Queues
     * at a system-wide level.  This means that the user can set the
     * priority of an IO initiated by either of the TCs (Transfer Controllers)
     * relative to IO initiated by the other bus masters on the
     * device (ARM, DSP, USB, etc)
     */
      {
      0u,
      1u,
      0u,
      0u,
      0u,
      0u,
      0u,
      0u
      },
    /**
     * \brief To Configure the Threshold level of number of events that can be queued up in the Event queues.
EDMA3CC error register (CCERR) will indicate whether or not at any instant of time the number of events queued
up in any of the event queues exceeds or equals the threshold/watermark value that is set in the queue
watermark threshold register (QWMTHRA).
     */
      {
      16u,
      16u,
      0u,
      0u,
      0u,
      0u,
      0u,
      0u
      },
    /**
     * \brief To Configure the Default Burst Size (DBS) of TCs.
     * An optimally-sized command is defined by the transfer controller
     * default burst size (DBS). Different TCs can have different
     * DBS values. It is defined in Bytes.
     */
      {
      16u,
      16u,
      0u,
      0u,
      0u,
      0u,
      0u,
      0u
      },
    /**
     * \brief Mapping from each DMA channel to a Parameter RAM set,
     * if it exists, otherwise of no use.
     */
      {
      0u, 1u, 2u, 3u,
      4u, 5u, 6u, 7u,
      8u, 9u, 10u, 11u,
      12u, 13u, 14u, 15u,
      16u, 17u, 18u, 19u,
      20u, 21u, 22u, 23u,
      24u, 25u, 26u, 27u,
      28u, 29u, 30u, 31u,
```

```
    /* DMA channels 32-63 DOES NOT exist in DA830. */
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS,
    EDMA3_MAX_PARAM_SETS, EDMA3_MAX_PARAM_SETS
    },

  /**
   * \brief Mapping from each DMA channel to a TCC. This specific
   * TCC code will be returned when the transfer is completed
   * on the mapped channel.
   */
    {
    0u, 1u, 2u, 3u,
    4u, 5u, 6u, 7u,
    8u, 9u, 10u, 11u,
    12u, 13u, 14u, 15u,
    16u, 17u, 18u, 19u,
    20u, 21u, EDMA3_RM_CH_NO_TCC_MAP, EDMA3_RM_CH_NO_TCC_MAP,
    24u, 25u, 26u, 27u,
    EDMA3_RM_CH_NO_TCC_MAP, EDMA3_RM_CH_NO_TCC_MAP, 30, 31,
    /* DMA channels 32-63 DOES NOT exist in DA830. */
    EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC,
    EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC,
    EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC,
    EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC,
    EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC,
    EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC,
    EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC,
    EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC, EDMA3_MAX_TCC
    },

 /**
  * \brief Mapping of DMA channels to Hardware Events from
  * various peripherals, which use EDMA for data transfer.
  * All channels need not be mapped, some can be free also.
  */
    {
    EDMA3_DMA_CHANNEL_TO_EVENT_MAPPING_0,
    EDMA3_DMA_CHANNEL_TO_EVENT_MAPPING_1
    }
 };
```

```
/* Driver Instance Initialization Configuration */
EDMA3_DRV_InstanceInitConfig sampleInstInitConfig =
   {
      /* Resources owned by Region 1 */
      /* ownPaRAMSets */
      /* 31    0        63     32     95    64       127    96 */
      {0xFFFFFFFFu, 0xFFFFFFFFu, 0xFFFFFFFFu, 0xFFFFFFFFu,
      /* 159    128     191    160     223    192     255    224 */
       0x00000000u, 0x00000000u, 0x00000000u, 0x00000000u,
      /* 287    256     319    288     351    320     383    352 */
       0x00000000u, 0x00000000u, 0x00000000u, 0x00000000u,
      /* 415    384     447    416     479    448     511    480 */
       0x00000000u, 0x00000000u, 0x00000000u, 0x00000000u,},

      /* ownDmaChannels */
      /* 31       0      63      32 */
     {0xFFFFFFFFu, 0x00000000u},

      /* ownQdmaChannels */
      /* 31        0 */
      {0x000000FFu},

      /* ownTccs */
      /* 31       0      63      32 */
      {0xFFFFFFFFu, 0x00000000u},

      /* Resources reserved by Region 1 */
      /* resvdPaRAMSets */
      /* 31    0        63     32     95    64       127    96 */
      {0xFFFFFFFFu, 0x00000000u, 0x00000000u, 0x00000000u,
      /* 159    128     191    160     223    192     255    224 */
       0x00000000u, 0x00000000u, 0x00000000u, 0x00000000u,
      /* 287    256     319    288     351    320     383    352 */
       0x00000000u, 0x00000000u, 0x00000000u, 0x00000000u,
      /* 415    384     447    416     479    448     511    480 */
       0x00000000u, 0x00000000u, 0x00000000u, 0x00000000u,},

      /* resvdDmaChannels */
      /* 31                                      0 */
      {EDMA3_DMA_CHANNEL_TO_EVENT_MAPPING_0,
      /* 63                                     32 */
      EDMA3_DMA_CHANNEL_TO_EVENT_MAPPING_0},

      /* resvdQdmaChannels */
      /* 31        0 */
      {0x00000000u},

      /* resvdTccs */
      /* 31                                      0 */
      {EDMA3_DMA_CHANNEL_TO_EVENT_MAPPING_0,
      /* 63                                     32 */
      EDMA3_DMA_CHANNEL_TO_EVENT_MAPPING_0},
   };

EDMA3_DRV_GblXbarToChanConfigParams sampleXbarChanInitConfig=
/* Event to channel map for region 0 */
      {
        -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1, -1,
        -1, -1, -1, -1, -1, -1, -1
      }
/* End of File */
```

```
/** File:   bios6_edma3_drv_sample.h
 *
 *                  Header file for the sample application of the EDMA3 Driver.
 */

#include <stdio.h>
#include <ti/sysbios/ipc/Semaphore.h>

/* Include EDMA3 Driver */
#include <ti/sdo/edma3/drv/edma3_drv.h>

/**
 * Cache line size on the underlying SoC. It needs to be modified
 * for different cache line sizes, if the Cache is Enabled.
 */
#define EDMA3_CACHE_LINE_SIZE_IN_BYTES      (128u)

/* Error returned in case of buffers are not aligned on the cache boundary */
#define EDMA3_NON_ALIGNED_BUFFERS_ERROR     (-1)

/* Error returned in case of data mismatch */
#define EDMA3_DATA_MISMATCH_ERROR           (-2)

/**
 * \brief   EDMA3 Initialization
 *
 * This function initializes the EDMA3 Driver for the given EDMA3 controller
 * and opens a EDMA3 driver instance. It internally calls EDMA3_DRV_create() and
 * EDMA3_DRV_open(), in that order.
 *
 * It also registers interrupt handlers for various EDMA3 interrupts like
 * transfer completion or error interrupts.
 *
 *  \param  edma3Id       [IN]              EDMA3 Controller Instance Id (Hardware
 *                                                                    instance id, starting from 0)
 *  \param  errorCode     [IN/OUT]          Error code while opening DRV instance
 *  \return EDMA3_DRV_Handle: If successfully opened, the API will return the
 *                      associated driver's instance handle.
 */
EDMA3_DRV_Handle edma3init (unsigned int edma3Id, EDMA3_DRV_Result *errorCode);

/**
 * \brief   EDMA3 De-initialization
 *
 * This function de-initializes the EDMA3 Driver for the given EDMA3 controller
 * and closes the previously opened EDMA3 driver instance. It internally calls
 * EDMA3_DRV_close and EDMA3_DRV_delete(), in that order.
 *
 * It also un-registers the previously registered interrupt handlers for various
 * EDMA3 interrupts.
 *
 *  \param  edma3Id       [IN]              EDMA3 Controller Instance Id (Hardware
 *                                                                    instance id, starting from 0)
 *  \param  hEdma         [IN]              EDMA3 Driver handle, returned while using
 *                                                                    edma3init().
 *  \return  EDMA3_DRV_SOK if success, else error code
 */
EDMA3_DRV_Result edma3deinit (unsigned int edma3Id, EDMA3_DRV_Handle hEdma);
```

```
/**
 *  \brief   EDMA3 Cache Invalidate
 *
 *  This function invalidates the D cache.
 *
 *  \param  mem_start_ptr [IN]      Starting address of memory.
 *                          Please note that this should be
 *                          aligned according to the cache line size.
 *  \param  num_bytes [IN]          length of buffer
 *  \return  EDMA3_DRV_SOK if success, else error code in case of error
 *         or non-alignment of buffers.
 *
 * Note: This function is required if the buffer is in DDR.
 * For other cases, where buffer is NOT in DDR, user
 * may or may not require the below implementation and
 * should modify it according to her need.
 */
EDMA3_DRV_Result Edma3_CacheInvalidate(unsigned int mem_start_ptr,
                    unsigned int num_bytes);


/**
 * \brief   EDMA3 Cache Flush
 *
 *  This function flushes (cleans) the Cache
 *
 *  \param  mem_start_ptr [IN]      Starting address of memory.
 *                          Please note that this should be
 *                          aligned according to the cache line size.
 *  \param  num_bytes [IN]          length of buffer
 *  \return  EDMA3_DRV_SOK if success, else error code in case of error
 *         or non-alignment of buffers.
 *
 * Note: This function is required if the buffer is in DDR.
 * For other cases, where buffer is NOT in DDR, user
 * may or may not require the below implementation and
 * should modify it according to her need.
 */
EDMA3_DRV_Result Edma3_CacheFlush(unsigned int mem_start_ptr,
                unsigned int num_bytes);
```

```
/**
 * Counting Semaphore related functions (OS dependent) should be
 * called/implemented by the application. A handle to the semaphore
 * is required while opening the driver/resource manager instance.
 */

/**
 * \brief   EDMA3 OS Semaphore Create
 *
 *      This function creates a counting semaphore with specified
 *      attributes and initial value. It should be used to create a semaphore
 *      with initial value as '1'. The semaphore is then passed by the user
 *      to the EDMA3 driver/RM for proper sharing of resources.
 * \param   initVal [IN] is initial value for semaphore
 * \param   semParams [IN] is the semaphore attributes.
 * \param   hSem [OUT] is location to receive the handle to just created
 *      semaphore
 * \return  EDMA3_DRV_SOK if successful, else a suitable error code.
 */
EDMA3_DRV_Result edma3OsSemCreate(int initVal,
                      const Semaphore_Params *semParams,
                      EDMA3_OS_Sem_Handle *hSem);

/**
 * \brief   EDMA3 OS Semaphore Delete
 *
 *      This function deletes or removes the specified semaphore
 *      from the system. Associated dynamically allocated memory
 *      if any is also freed up.
* \param   hSem [IN] handle to the semaphore to be deleted
 * \return  EDMA3_DRV_SOK if successful else a suitable error code
 */
EDMA3_DRV_Result edma3OsSemDelete(EDMA3_OS_Sem_Handle hSem);
```

/* Below is the sample code which show how to define the OS dependent critical section handling routines. These functions should be mandatorily defined by the user. */

```
#include <ti/sysbios/family/c64p/EventCombiner.h>
#include <ti/sysbios/hal/Cache.h>
#include <ti/sysbios/hal/Hwi.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/ipc/Semaphore.h>

#include <ti/sdo/edma3/drv/sample/bios6_edma3_drv_sample.h>

/** Entry to critical section */
void edma3OsProtectEntry (unsigned int edma3InstanceId,
                                                        int level, unsigned int *intState)
  {
  if (((level == EDMA3_OS_PROTECT_INTERRUPT)  || (level == EDMA3_OS_PROTECT_INTERRUPT_TC_ERROR))
     && (intState == NULL))
     {
     return;
     }
  else
     {
     switch (level)
        {
        /* Disable all (global) interrupts */
        case EDMA3_OS_PROTECT_INTERRUPT :
           *intState = Hwi_disable();
           break;

        /* Disable scheduler */
        case EDMA3_OS_PROTECT_SCHEDULER :
           Task_disable();
           break;

        /* Disable EDMA3 transfer completion interrupt only */
        case EDMA3_OS_PROTECT_INTERRUPT_XFER_COMPLETION :
           EventCombiner_disableEvent(ccXferCompInt[edma3InstanceId][dsp_num]);
           break;

        /* Disable EDMA3 CC error interrupt only */
        case EDMA3_OS_PROTECT_INTERRUPT_CC_ERROR :
           EventCombiner_disableEvent(ccErrorInt[edma3InstanceId]);
           break;

        /* Disable EDMA3 TC error interrupt only */
        case EDMA3_OS_PROTECT_INTERRUPT_TC_ERROR :
           switch (*intState)
              {
              case 0:
              case 1:
              case 2:
              case 3:
              case 4:
              case 5:
              case 6:
              case 7:
                 /* Fall through... */
                 /* Disable the corresponding interrupt */
                 EventCombiner_disableEvent(tcErrorInt[edma3InstanceId][*intState]);
                 break;

               default:
                 break;
              }

           break;

        default:
           break;
        }
     }
  }
```

```
/** Exit from critical section */

void edma3OsProtectExit (unsigned int edma3InstanceId,
                         int level, unsigned int intState)
  {
  switch (level)
    {
    /* Enable all (global) interrupts */
    case EDMA3_OS_PROTECT_INTERRUPT :
      Hwi_restore(intState);
      break;

    /* Enable scheduler */
    case EDMA3_OS_PROTECT_SCHEDULER :
      Task_enable();
      break;

    /* Enable EDMA3 transfer completion interrupt only */
    case EDMA3_OS_PROTECT_INTERRUPT_XFER_COMPLETION :
      EventCombiner_enableEvent(ccXferCompInt[edma3InstanceId][dsp_num]);
      break;

    /* Enable EDMA3 CC error interrupt only */
    case EDMA3_OS_PROTECT_INTERRUPT_CC_ERROR :
      EventCombiner_enableEvent(ccErrorInt[edma3InstanceId]);
      break;

    /* Enable EDMA3 TC error interrupt only */
    case EDMA3_OS_PROTECT_INTERRUPT_TC_ERROR :
      switch (intState)
        {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
        case 6:
        case 7:
          /* Fall through... */
          /* Enable the corresponding interrupt */
          EventCombiner_enableEvent(tcErrorInt[edma3InstanceId][intState]);
          break;

         default:
          break;
        }

      break;

    default:
      break;
    }
  }
```

```
/**
 *  \brief   EDMA3 Cache Invalidate
 *
 *  This function invalidates the D cache.
 *
 *  \param  mem_start_ptr [IN]      Starting address of memory. Please note that this should be aligned according
* to the cache line size.
 *  \param  num_bytes [IN]          length of buffer
 *  \return  EDMA3_DRV_SOK if success, else error code in case of error or non-alignment of buffers.
* Note: This function is required if the buffer is in DDR. For other cases, where buffer is NOT in DDR, user may   *
or may not require the below implementation and should modify it according to her need.
 */
EDMA3_DRV_Result Edma3_CacheInvalidate(unsigned int mem_start_ptr,
                    unsigned int    num_bytes)
   {
   EDMA3_DRV_Result cacheInvResult = EDMA3_DRV_SOK;

   /* Verify whether the start address is cache aligned or not */
   if((mem_start_ptr & (EDMA3_CACHE_LINE_SIZE_IN_BYTES-1u))    !=    0)
      {
#ifdef EDMA3_DRV_DEBUG
      EDMA3_DRV_PRINTF("\r\n Cache : Memory is not %d bytes alinged\r\n",
EDMA3_CACHE_LINE_SIZE_IN_BYTES);
#endif
      cacheInvResult = EDMA3_NON_ALIGNED_BUFFERS_ERROR;
      }
   else
      {
      Cache_inv((Ptr)mem_start_ptr, num_bytes, Cache_Type_ALL, TRUE);
      }
   return cacheInvResult;
}


/**
 * \brief   EDMA3 Cache Flush
 *
 *  This function flushes (cleans) the Cache
 *
 *  \param  mem_start_ptr [IN]      Starting address of memory. Please note that this should be aligned according
* to the cache line size.
 *  \param  num_bytes [IN]          length of buffer
 *  \return  EDMA3_DRV_SOK if success, else error code in case of error or non-alignment of buffers.
* Note: This function is required if the buffer is in DDR. For other cases, where buffer is NOT in DDR, user may
*or may not require the below implementation and should modify it according to her need.
 */
EDMA3_DRV_Result Edma3_CacheFlush(unsigned int mem_start_ptr,
                  unsigned int num_bytes)
   {
   EDMA3_DRV_Result cacheFlushResult = EDMA3_DRV_SOK;

   /* Verify whether the start address is cache aligned or not */
   if((mem_start_ptr & (EDMA3_CACHE_LINE_SIZE_IN_BYTES-1u))    !=    0)
      {
#ifdef EDMA3_DRV_DEBUG
      EDMA3_DRV_PRINTF("\r\n Cache : Memory is not %d bytes alinged\r\n",
                     EDMA3_CACHE_LINE_SIZE_IN_BYTES);
#endif
      cacheFlushResult = EDMA3_NON_ALIGNED_BUFFERS_ERROR;
      }
   else
      {
      Cache_wb((Ptr)mem_start_ptr, num_bytes, Cache_Type_ALL, TRUE);
      }
   return cacheFlushResult;
}
```

/* Below is the sample code demonstrating how to create and delete a semaphore with a specific initial value. It also shows how to acquire and later release a semaphore. */

```
/* Function to create OS Semaphore */

EDMA3_DRV_Result edma3OsSemCreate(int initVal,
                    const Semaphore_Params *semParams,
                    EDMA3_OS_Sem_Handle *hSem)
   {
   EDMA3_DRV_Result semCreateResult = EDMA3_DRV_SOK;

   if(NULL == hSem)
      {
      semCreateResult = EDMA3_DRV_E_INVALID_PARAM;
      }
   else
      {
      *hSem = (EDMA3_OS_Sem_Handle)Semaphore_create(initVal, semParams, NULL);
      if ( (*hSem) == NULL )
         {
         semCreateResult = EDMA3_DRV_E_SEMAPHORE;
         }
      }

   return semCreateResult;
   }



/* Function to delete OS Semaphore */

EDMA3_DRV_Result edma3OsSemDelete (EDMA3_OS_Sem_Handle hSem)
   {
   EDMA3_DRV_Result semDeleteResult = EDMA3_DRV_SOK;

   if(NULL == hSem)
      {
      semDeleteResult = EDMA3_DRV_E_INVALID_PARAM;
      }
   else
      {
      SEM_delete(hSem);
      }

   return semDeleteResult;
   }
```

```
/* Function to take OS Semaphore */


EDMA3_DRV_Result edma3OsSemTake(EDMA3_OS_Sem_Handle hSem, int mSecTimeout)
   {
   EDMA3_DRV_Result semTakeResult = EDMA3_DRV_SOK;
   unsigned short semPendResult;

   if(NULL == hSem)
      {
      semTakeResult = EDMA3_DRV_E_INVALID_PARAM;
      }
   else
      {
         semPendResult = Semaphore_pend(hSem, mSecTimeout);
         if (semPendResult == FALSE)
            {
            semTakeResult = EDMA3_DRV_E_SEMAPHORE;
            }
      }

   return semTakeResult;
   }



/* Function to give OS Semaphore */

EDMA3_DRV_Result edma3OsSemGive(EDMA3_OS_Sem_Handle hSem)
   {
   EDMA3_DRV_Result semGiveResult = EDMA3_DRV_SOK;

   if(NULL == hSem)
      {
      semGiveResult = EDMA3_DRV_E_INVALID_PARAM;
      }
   else
      {
         Semaphore_post (hSem);
      }

   return semGiveResult;
   }
```

/* Below is the sample code demonstrating how to register/un-register the various interrupt handlers with the underlying OS. Here, application is registering interrupt handlers with the DSP/BIOS OS. */

```
#include <ti/sysbios/hal/Hwi.h>
#include <ti/sysbios/ipc/Semaphore.h>
#include <ti/sysbios/family/c64p/EventCombiner.h> #include
<ti/sdo/edma3/drv/sample/bios6_edma3_drv_sample.h>

/**
  * EDMA3 TC ISRs which need to be registered with the underlying OS by the user
  * (Not all TC error ISRs need to be registered, register only for the
  * available Transfer Controllers).
  */
void (*ptrEdma3TcIsrHandler[EDMA3_MAX_TC])(unsigned int arg) =
                                    {
                                    &lisrEdma3TC0ErrHandler0,
                                    &lisrEdma3TC1ErrHandler0,
                                    &lisrEdma3TC2ErrHandler0,
                                    &lisrEdma3TC3ErrHandler0,
                                    &lisrEdma3TC4ErrHandler0,
                                    &lisrEdma3TC5ErrHandler0,
                                    &lisrEdma3TC6ErrHandler0,
                                    &lisrEdma3TC7ErrHandler0,
                                    };


/**  To Register the ISRs with the underlying OS, if required. */
static void registerEdma3Interrupts(void)
   {
   static UInt32 cookie = 0;
   unsigned int numTc = 0;

   /* Disabling the global interrupts */
   cookie = Hwi_disable();

   /* Enable the Xfer Completion Event Interrupt */
   EventCombiner_dispatchPlug(ccXferCompInt, (EventCombiner_FuncPtr)(&lisrEdma3ComplHandler0),
                                    NULL, 0);
   EventCombiner_enableEvent(ccXferCompInt);

   /* Enable the CC Error Event Interrupt */
   EventCombiner_dispatchPlug(ccErrorInt, (EventCombiner_FuncPtr)(&lisrEdma3CCErrHandler0),
                                                                    NULL, 0);
   EventCombiner_enableEvent(ccErrorInt);
```

```
    /* Enable the TC Error Event Interrupt, according to the number of TCs. */
    while (numTc < numEdma3Tc)
              {
       EventCombiner_dispatchPlug(tcErrorInt[numTc],
                    (EventCombiner_FuncPtr)(ptrEdma3TcIsrHandler[numTc]),
                    NULL, 0);
       EventCombiner_enableEvent(tcErrorInt[numTc]);
       numTc++;
           }

   /**
    * Enabling the HWI_ID.
    * EDMA3 interrupts (transfer completion, CC error etc.)
    * correspond to different ECM events (SoC specific). These ECM events come
    * under ECM block XXX (handling those specific ECM events). Normally, block
    * 0 handles events 4-31 (events 0-3 are reserved), block 1 handles events
    * 32-63 and so on. This ECM block XXX (or interrupt selection number XXX)
    * is mapped to a specific HWI_INT YYY in the tcf file. So to enable this
    * mapped HWI_INT YYY, one should use the corresponding bitmask in the
    * API C64_enableIER(), in which the YYY bit is SET.
    */
          Hwi_enableInterrupt(hwIntXferComp);
          Hwi_enableInterrupt(hwIntCcErr);
          Hwi_enableInterrupt(hwIntTcErr);

    /* Restore interrupts */
    Hwi_restore(cookie);
    }


/**  To Unregister the ISRs with the underlying OS, if previously registered. */
static void unregisterEdma3Interrupts(void)
    {
    static UInt32 cookie = 0;
    unsigned int numTc = 0;

    /* Disabling the global interrupts */
    cookie = Hwi_disable();

    /* Disable the Xfer Completion Event Interrupt */
          EventCombiner_disableEvent(ccXferCompInt);

    /* Disable the CC Error Event Interrupt */
          EventCombiner_disableEvent(ccErrorInt);

    /* Enable the TC Error Event Interrupt, according to the number of TCs. */
    while (numTc < numEdma3Tc)
          {
       EventCombiner_disableEvent(tcErrorInt[numTc]);
       numTc++;
           }

    /* Restore interrupts */
    Hwi_restore(cookie);
    }
```

```
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_3_MASK (0x3F000000u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_3_SHIFT (0x00000018u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_3_RESETVAL (0x00000000u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_2_MASK (0x003F0000u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_2_SHIFT (0x00000010u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_2_RESETVAL (0x00000000u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_1_MASK (0x00003F00u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_1_SHIFT (0x00000008u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_1_RESETVAL (0x00000000u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_0_MASK (0x0000003Fu)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_0_SHIFT (0x00000000u)
#define CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_0_RESETVAL (0x00000000u)

/*This function reads from the sample configuration structure which specifies
 * cross bar events mapped to DMA channel.*/
EDMA3_DRV_Result sampleMapXbarEvtToChan (unsigned int eventNum,
            unsigned int *chanNum,
            const EDMA3_DRV_GblXbarToChanConfigParams * edmaGblXbarConfig)
        {
   EDMA3_DRV_Result edma3Result = EDMA3_DRV_E_INVALID_PARAM;
   unsigned int xbarEvtNum = 0;
   int         edmaChanNum = 0;
        if ((eventNum < EDMA3_MAX_CROSS_BAR_EVENTS_TI814X) &&
                (chanNum != NULL) &&
                (edmaGblXbarConfig != NULL))
                {
                xbarEvtNum = eventNum - EDMA3_NUM_TCC;
                edmaChanNum = edmaGblXbarConfig->dmaMapXbarToChan[xbarEvtNum];
                if (edmaChanNum != -1)
                        {
                        *chanNum = edmaChanNum;
                        edma3Result = EDMA3_DRV_SOK;
                        }
                }
        return (edma3Result);
        }

/* Sample initialization function*/
EDMA3_DRV_Result sampleInitXbarEvt(EDMA3_DRV_Handle hEdma,
                           unsigned int edma3Id)
   {
   EDMA3_DRV_Result retVal = EDMA3_DRV_SOK;
   const EDMA3_DRV_GblXbarToChanConfigParams *sampleXbarToChanConfig =
                        &(sampleXbarChanInitConfig[edma3Id][dsp_num]);
   if (hEdma != NULL)
      {
      retVal = EDMA3_DRV_initXbarEventMap(hEdma, sampleXbarToChanConfig, &sampleMapXbarEvtToChan,
                                                                        &sampleConfigScr);
      }

   return retVal;
   }
```

```
typedef struct  {
    volatile Uint32 DSP_INTMUX[21];
    volatile Uint32 DUCATI_INTMUX[15];
    volatile Uint32 TPCC_EVTMUX[16];
    volatile Uint32 TIMER_EVTCAPT;
    volatile Uint32 GPIO_MUX;
} CSL_IntmuxRegs;
typedef volatile CSL_IntmuxRegs     *CSL_IntmuxRegsOvly;

/* This function configures control config registers for the cross bar events mapped to the EDMA channel. */
EDMA3_DRV_Result sampleConfigScr (unsigned int eventNum,
                          unsigned int chanNum)
        {
    EDMA3_DRV_Result edma3Result = EDMA3_DRV_SOK;
    unsigned int scrChanOffset = 0;
    unsigned int scrRegOffset  = 0;
    unsigned int xBarEvtNum    = 0;
    CSL_IntmuxRegsOvly scrEvtMux = (unsigned int)(0x0);

        if ((eventNum < EDMA3_MAX_CROSS_BAR_EVENTS_TI814X) &&
                (chanNum < EDMA3_NUM_TCC))
                {
                scrRegOffset = chanNum / 4;
                scrChanOffset = chanNum - (scrRegOffset * 4);
                xBarEvtNum = (eventNum - EDMA3_NUM_TCC) + 1;

                switch(scrChanOffset)
                        {
                        case 0:
                        scrEvtMux->TPCC_EVTMUX[scrRegOffset] |=
                                (xBarEvtNum & CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_0_MASK);
                        break;
                        case 1:
                        scrEvtMux->TPCC_EVTMUX[scrRegOffset] |=
                                ((xBarEvtNum < CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_1_SHIFT) &
                                (~(CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_1_MASK)));
                        break;
                        case 2:
                        scrEvtMux->TPCC_EVTMUX[scrRegOffset] |=
                                ((xBarEvtNum < CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_2_SHIFT) &
                                (~(CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_2_MASK)));
                        break;
                        case 3:
                        scrEvtMux->TPCC_EVTMUX[scrRegOffset] |=
                                ((xBarEvtNum < CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_3_SHIFT) &
                                (~(CSL_INTMUX_TPCC_EVTMUX_TPCCEVT_MUX_3_MASK)));
                        break;
                        default:
                        edma3Result = EDMA3_DRV_E_INVALID_PARAM;
                        break;
                        }
                }
        else
                {
                edma3Result = EDMA3_DRV_E_INVALID_PARAM;
                }
        return edma3Result;
        }
```