Application Report

SPRAAN4A–June 2008

How to Use the EDMA3 Driver on a TMS320DM643x Device

Zhengting He

ABSTRACT

This application report describes how to use the existing enhanced direct memory access (EDMA3) driver on the TMS320DM643x devices. It also explains several complicated transfers required for certain real-life applications using the EDMA3.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/SPRAAN4.

Contents

1 Introduction

Four sets of examples are included with this document. The first set demonstrates how to convert an image in NTSC D1 resolution (720×480) from YUV422 interleaved format to YUV420 planar format. The second set demonstrates how to copy a two-dimensional array from external to internal memory, using the QDMA channel. The third set demonstrates how to convert two-dimensional data to one-dimensional data and copy them from external to internal memory, using the link feature. The fourth set demonstrates how to achieve a PING-PONG buffering scheme and optimize a transfer by taking advantage of the link and chain feature.

It also reports the transfer performance for the provided examples. However, these performance numbers should not be interpreted as EDMA3 bench marking numbers.

This document does not detail the EDMA3 hardware or application programming interfaces (APIs).

1.1 Introduction to the EDMA3

This section details important concepts for the EDMA3 hardware module in the TMS320DM643x devices. For more details on the EDMA3, see the *TMS320DM643x DMP Enhanced Direct Memory Access (EDMA3) Controller User's Guide* (SPRU987) [1].

Figure 1 illustrates a block diagram of the EDMA3.

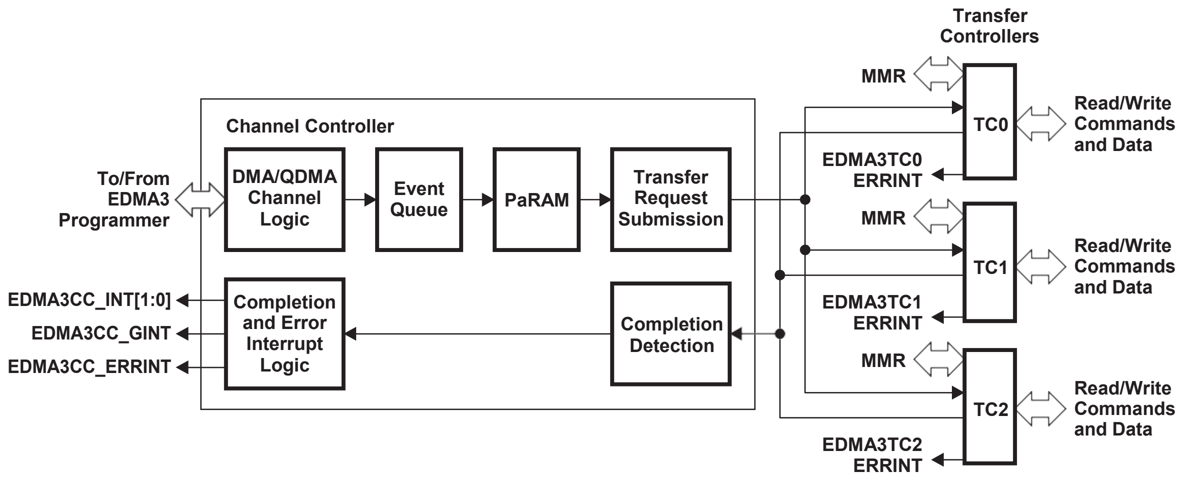

Figure 1. EDMA3 Block Diagram

The EDMA3's primary purpose is to service user-programmed data transfers between two memory-mapped slave endpoints on the device. It consists of two principal blocks:

- EDMA3 channel controller (EDMA3CC)
- EDMA3 transfer controller(s) (EDMA3TC)

The EDMA3CC is the user interface. It includes parameter RAM (PaRAM), channel control registers, and interrupt control registers. It prioritizes incoming software requests or events from peripherals, and submits transfer requests to an EDMA3TC based on the transfer parameter.

The EDMA3TCs are slaves to the EDMA3CC, realizing data movement by issuing read/write commands to the source and destination addresses.

The EDMA3CC has the following features:

- Fully orthogonal transfer description
  - Supports one-dimensional (1-D), two-dimensional (2-D) and three-dimensional (3-D) transfers
  - Transfer can either be synchronized once each 1-D transfer is completed (A-sync), or once each 2-D transfer is completed (AB-sync)
  - Independent indexes (offsets) on source and destination
  - Increment or constant addressing modes on source and destination
- Flexible transfer definition
  - Linking mechanism allows automatic PaRAM set update after a transfer is completed by reloading the current PaRAM set from the linked PaRAM set
  - Chaining feature allows multiple transfers to be triggered automatically after the current transfer is completed
- Interrupt generation for:
  - Intermediate or final transfer completion. A transfer can be split into multiple smaller transfers. For instance, a 3-D transfer can be split into multiple 2-D transfers. The final transfer is the last small transfer and an intermediate transfer is a small transfer that is not performed last.
  - Error conditions
- 64 regular direct memory access (DMA) channels
- Eight QDMA channels
- 128 PaRAM sets. Each PaRAM set can be used for a DMA channel, QDMA channel, or as a Link set.

The EDMA3TC has following features:

- Three transfer controllers with 64-bit wide read and write ports per transfer controller
- Up to four in-flight transfer requests
- Programmable priority level
- Supports 2-D transfers with independent indexes on source and destination (EDMA3CC manages the third dimension)
- Support for increment or constant addressing mode transfers
- Interrupt and error support
- Little-endian or big-endian operations

1.2 Introduction to EDMA3 Driver

This section introduces the EDMA3 driver. For more details, see the *EDMA3 0.3.0 User's Manual* provided with the EDMA3 driver code [2].

Figure 2 shows the software structure of the EDMA3 driver. It allows other device drivers and applications to submit and synchronize EDMA3-based DMA transfers. To simplify the usage, this component internally uses the services of the EDMA3 Resource Manager and provides one consistent interface for applications or other device drivers.

The EDMA3 Resource Manager includes the following parts:

- Physical driver: Manages several resources within the EDMA3 peripheral, including TCC codes, PARAM entry, DMA and QDMA channels, all global EDMA3 registers, etc.
- Interrupt manager: Handles EDMA3 interrupts by registering with the ported OS interrupt handling mechanism. Because interrupts are associated with TCC codes in the EDMA3 module, this module provides an option to accept application registration callbacks for TCC codes and calls the callback functions upon receipt of the given interrupt (TCC). Note that the application/driver using the EDMA3 resource manager has to register/unregister the interrupt handlers with the underlying OS. The resource manager does not do this by itself.

The EDMA3 hardware module can be used with different TI devices by changing a few resource configurations, such as the number of DMA and QDMA channels supported, the number of PARAM sets available, etc. Using the EDMA3 resource manager allows you to maintain one version of the EDMA3 driver source code across different devices by providing the driver device-specific configuration information. Typically, each device runs an instance of the resource (and interrupt) manager and EDMA driver. It is also possible to have multiple instances of the resource manager or the EDMA driver running on the same device if the device has multiple EDMA3 hardware modules.

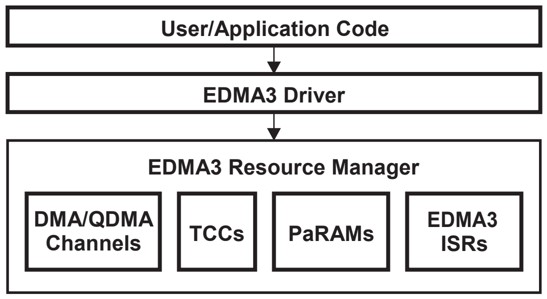

Figure 2. EDMA3 Driver Structure

The following are services provided by the EDMA3 driver:

- Request and free DMA channel: This provides an interface that applications or other device drivers can use to request and free DMA channels. Channels in the EDMA3 module are categorized as:
  - Regular DMA channel (mapped to a hardware sync event)
  - Regular DMA channel (not mapped to a hardware sync event)
  - QDMA channel
  - Link channel (a PARAM set in EDMA3)

- Program DMA channel: This provides an interface that applications or other device drivers can use to program a DMA transaction. This typically involves setting the DMA source and destination parameters. The following types of transactions are supported:
  - Event-triggered (peripheral driven transfers)
  - Chain-triggered (issuing a chain of transfers initiated by a single event)
  - Manual-triggered (CPU generated)
- Start and synchronize with DMA transfers: This provides an interface that applications or other device drivers can use to start and synchronize with a DMA transaction.
- Provide DMA transaction completion callback to applications: This provides an interface that applications or other device drivers can use to register a transaction completion (final or intermediate) callback. The EDMA3 driver calls this application or device driver callback with appropriate completion status.
- Support link and chain feature: The EDMA3 peripheral provides link and chain services in hardware. The EDMA driver provides an interface so applications or other device drivers can use the service.

2 Example A: Convert Image From YUV422 Format to YUV420 Format

This section explains how to use the EDMA3 driver to convert an image in NTSC D1 resolution (720×480 pixels) from the YUV422 interleaved format to YUV420 planner format. The input and output are assumed to be in the external memory. YUV422 to YUV420 conversion is often a necessary preprocessing step for video compression-based applications in which video signals captured from a charge-coupled device (CCD) camera are in a YUV422 format and they must be converted to a YUV420 format before being processed by the video encoder. This example shows how to offload the DSP core by performing the conversion task on the EDMA3.

Figure 3 shows the data organization of an NTSC D1 resolution image in YUV422 format. In the provided example, YUV components are assumed in UYVY order although they can also be in YUYV order, depending on the video capture port.

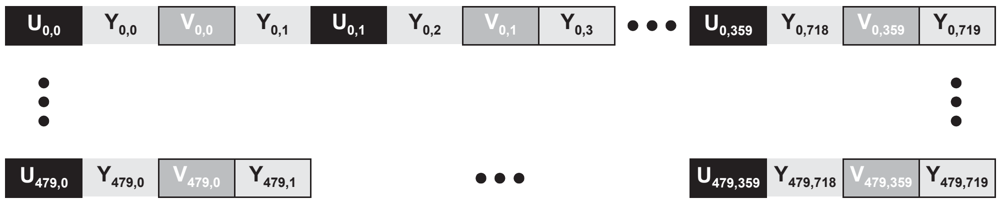
$U_{0,0}$ $Y_{0,0}$ $V_{0,0}$ $Y_{0,1}$ $U_{0,1}$ $Y_{0,2}$ $V_{0,1}$ $Y_{0,3}$ $U_{0,359}$ $Y_{0,718}$ $V_{0,359}$ $Y_{0,719}$
$U_{479,0}$ $Y_{479,0}$ $V_{479,0}$ $Y_{479,1}$ $U_{479,359}$ $Y_{479,718}$ $V_{479,359}$ $Y_{479,719}$

Figure 3. NTSC D1 YUV422 Interleaved Format (UYVY)

Figure 4 shows the data organization of a NTSC D1 resolution image in YUV420 format. Compared to the YUV422 interleaved format, the main differences are:

- The Y, U and V components are separated
- U and V are down-sampled vertically by half

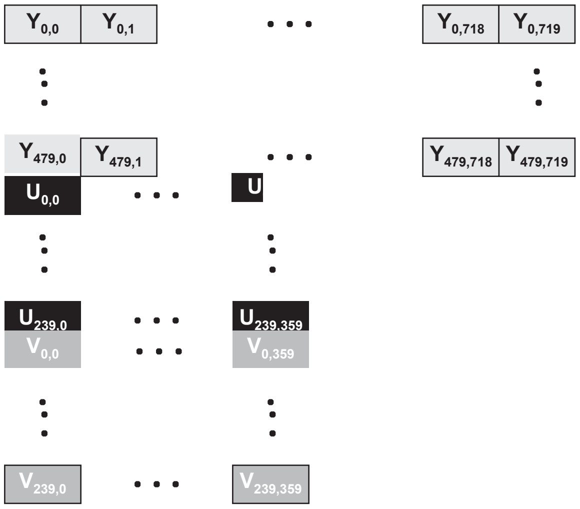

Figure 4. NTSC D1 YUV420 Planar Format

The following steps show how to use the EDMA3 driver to copy the Y plane.

1. Allocate a regular channel by calling EDMA3_DRV_requestChannel(). This API allocates a TCC code and a PaRAM set along with the regular channel to the caller. You need to provide a callback. The purpose of the callback is explained in step 3.
2. Configure the PaRAM set associated with the allocated channel by calling the EDMA3_DRV_setSrcParams(). The values of the fields in the PaRAM set are shown below:
   - *SRC* should be the address of the first Y component of the input image.
   - *DST* should be the address of the output Y plane.
   - *SRCBIDX = 2, SRCCIDX = 1440, DSTBIDX = 1 and DSTCIDX = 720.* This indicates that the EDMA3 copies every other byte (the Y component) from the input image to the output Y plane buffer. The data in the Y plane buffer is contiguous.
   - *ACNT = 1, BCNT = 720 and CCNT = 480.* This indicates that the transfer is configured as a 3-D transfer. The 1-D transfer copies every Y component (1 byte). The 2-D transfer consists of 720 1-D transfers to copy a row. The 3-D transfer consists of 480 2-D transfers to copy the whole Y plane.
   - *OPT.SYNCDIM = SYNC_AB, OPT.TCINTEN = ENABLE, OPT.ITCINTEN = ENABLE, and OPT.TCCMODE = NORMAL.* This indicates that after each 2-D transfer is completed, you will be notified by an interrupt.
3. Manually enable the first 2-D transfer by calling EDMA3_DRV_enableTransfer(). Once it is completed, an interrupt is generated. Inside the callback, you are expected to enable the next 2-D transfer until all 480 rows have been copied.

The following steps show how to use the EDMA3 driver to copy the U plane.

1. Allocate a regular channel by calling EDMA3_DRV_requestChannel() or reuse the same channel which is used to copy the Y plane.
2. Configure the PaRAM set associated with the allocated channel by calling the EDMA3_DRV_setSrcParams(). The values of the fields in the PaRAM set are shown below:
   - *SRC* should be the address of the first U component of the input image.
   - *DST* should be the address of the output U plane.
   - *SRCBIDX = 4, SRCCIDX = 2880, DSTBIDX = 1 and DSTCIDX = 360.* This indicates that the EDMA3 copies one of every four bytes (the U component) in every odd row of the input image to the output U plane buffer. The data in the U plane buffer is contiguous.
   - *ACNT = 1, BCNT = 360 and CCNT = 240.* This indicates that the transfer is configured as a 3-D transfer. The 1-D transfer copies every U component (1 byte) in the odd row of the input image. The 2-D transfer consists of 360 1-D transfers to copy every odd row. The 3-D transfer consists of 240 2-D transfers to copy the whole U plane.
   - *OPT.SYNCDIM = SYNC_AB, OPT.TCINTEN = ENABLE, OPT.ITCINTEN = ENABLE, and OPT.TCCMODE = NORMAL.* This indicates that after each 2-D transfer is completed, you will be notified by an interrupt.
3. Manually enable the first 2-D transfer by calling EDMA3_DRV_enableTransfer(). Once it is completed, an interrupt is generated. Inside the callback, you are expected to enable the next 2-D transfer until all 240 rows have been copied.

Steps and parameters to copy the V plane are similar to the steps required to copy the U plane, except that *SRC* should be the address of the first V component of the input image and *DST* should be the address of the output V plane.

After the output YUV420 image is generated, the regular EDMA channel is freed by calling EDMA3_DRV_freeChannel().

3 Example B: Copy a 2-D Array From External Memory to Internal Memory

This section describes using the EDMA3 driver to extract a subset of a 2-D array from a Y plane image in NTSC D1 resolution (720×480). The input image is assumed to reside in the external memory and the output is in the internal memory, which can either be L1 data memory or L2 memory. The array size is 128 by 128 bytes, although you can change it to any desirable value as long as the array fits in the internal memory (see Figure 5).

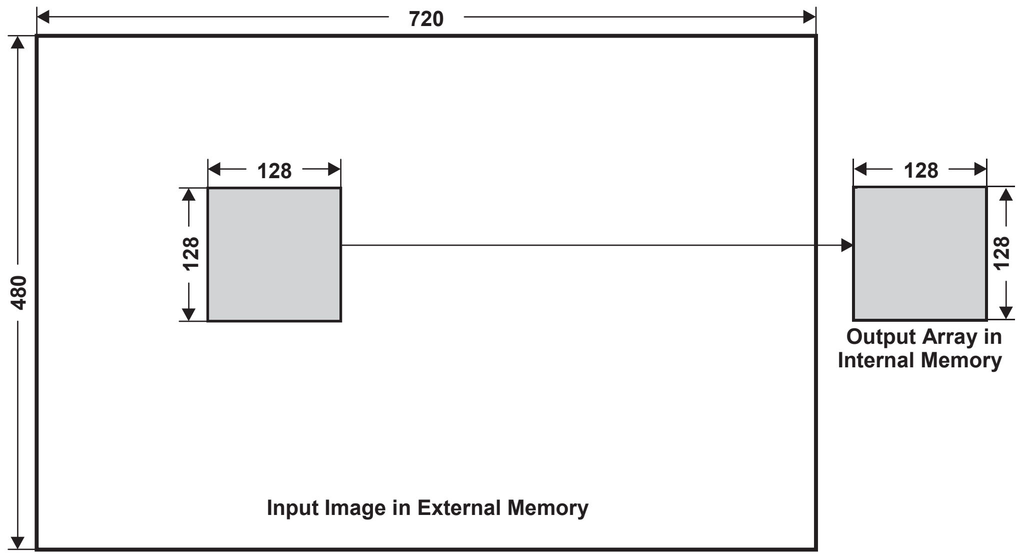
720
128
128
128
128
480
Output Array in Internal Memory
Input Image in External Memory

Figure 5. Extract a 2-D Subset Array From an Input Image

Two examples are provided. The first example uses a QDMA channel to perform the data transfer and an interrupt notifies you that the transfer is completed. Currently, the EDMA3 driver can only use interrupts to notify you that the transfer is complete.

The second example shows how to program the EDMA3 IECR(H) register to disable the interrupt and poll the TCC bit in the IPR(H) register to check that the transfer is complete. Due to the interrupt handling overhead, the polling mode may be more efficient and easier to use for some applications. This example demonstrates a workaround solution that achieves better performance for the EDMA3 driver. TI strongly recommends that you use the updated version of the driver whenever possible.

3.1 QDMA 2-D Copy – Interrupt Notification

The following steps show how to use a QDMA channel to copy the 2-D array (128×128 bytes) from external memory to internal memory.

1. Allocate a QDMA channel by calling EDMA3_DRV_requestChannel(). This API allocates a TCC code and a PaRAM set along with the QDMA channel to the caller. You need to provide a callback that is called when the transfer is completed.
2. Set the QDMA trigger word to be the *DST* field in the PaRAM set. The trigger word can be any field in the PaRAM set. This example uses the *DST* field to demonstrate the QDMA usage.
3. Configure the PaRAM set associated with the allocated channel by calling EDMA3_DRV_setSrcParams(). The values of the fields in the PaRAM set are as follows:
   - *SRC* should be the start address of the 128×128 array in the input image to be extracted.
   - *SRCBIDX = 720 and DSTBIDX = 128.* This indicates that the offset between two neighboring rows of the 128×128 array in the input image and output buffer is 720 and 128, respectively. The values of *SRCCIDX* and *DSTCIDX* do not matter in this example because it is a 2-D copy.
   - *ACNT = 128, BCNT = 128 and CCNT = 1.* This indicates that the transfer is configured as a 2-D transfer. The 1-D transfer copies every row (128 bytes). The 2-D transfer consists of 128 1-D transfers to copy the 128×128 array.
   - *OPT.SYNCDIM = SYNC_AB, OPT.TCINTEN = ENABLE, OPT.ITCINTEN = DISABLE, and OPT.TCCMODE = NORMAL.* This indicates that after the 2-D transfer is completed, you will be notified by an interrupt.
4. Manually enable the transfer by calling EDMA3_DRV_setPaRAMEntry(). The *DST* field, which is the trigger word, is set to the address of the output 2-D array in the internal memory. Once the transfer is completed, an interrupt is generated.
5. After the example is finished, the QDMA channel is freed by calling EDMA3_DRV_freeChannel().

3.2 QDMA 2-D Copy – Polling Transfer Completeness

The following steps show how to use a QDMA channel to copy the 2-D array (128×128 bytes) from external memory to internal memory, using the polling method to check that the transfer is complete.

1. Allocate a QDMA channel by calling EDMA3_DRV_requestChannel(). This API allocates a TCC code and a PaRAM set along with the QDMA channel to the caller. You do not need to provide a callback.
2. Set the TCC bit in the EDMA3 IECR(H) register, which disables the interrupt to be generated when a transfer is completed.
3. Set the QDMA trigger word to be the *DST* field in the PaRAM set. The trigger word can be any field in the PaRAM set. This example uses the *DST* field to demonstrate the QDMA usage.
4. Configure the PaRAM set associated with the allocated channel by calling EDMA3_DRV_setSrcParams(). The values of the fields in the PaRAM set are as follows:
   - *SRC* should be the start address of the 128×128 array of the input image to be extracted.
   - *SRCBIDX = 720 and DSTBIDX = 128.* This indicates that the offset between two neighboring rows of the 128×128 array in the input image and output image is 720 and 128, respectively. The values of *SRCCIDX* and *DSTCIDX* do not matter in this example because it is a 2-D copy.
   - *ACNT = 128, BCNT = 128 and CCNT = 1.* This indicates that the transfer is configured as a 2-D transfer. The 1-D transfer copies every row (128 bytes). The 2-D transfer consists of 128 1-D transfers to copy the 128×128 array.
   - *OPT.SYNCDIM = SYNC_AB, OPT.TCINTEN = ENABLE, OPT.ITCINTEN = DISABLE, and OPT.TCCMODE = NORMAL.* This indicates that after the 2-D transfer is completed, the TCC bit in the IPR(H) register is set.
5. Manually enable the transfer by calling EDMA3_DRV_setPaRAMEntry(). The *DST* field, which is the trigger word, is set to the address of the output 2-D array in the internal memory.
6. Poll the transfer status by reading the TCC bit in the IPR(H) register. Once the transfer is completed, the bit is raised to 1. The bit needs to be cleared by writing to the TCC bit in the ICR(H) register.
7. After the example is finished, the QDMA channel is freed by calling EDMA3_DRV_freeChannel().

4 Example C: Convert a 2-D Image to 1-D Data

This section describes how to copy each of the 8×8 2-D blocks in the original Y plane image in NTSC D1 resolution (720×480) from the external memory to the internal memory, while converting them to 1-D data. The Y plane image consists of macroblocks (MBs) in size 16×16. Each MB is divided into four 8×8 blocks. When transferring a MB, the provided example also converts each 8×8 block to a 1-D vector, as shown in Figure 6.

Many video encoders compress the image MB by MB, each of which is divided into four 8×8 blocks and processed separately. When implementing these video encoders, you should copy the blocks from the external memory to the internal memory and convert them to 1-D data before processing starts. This example shows you how to use EDMA3 to perform these tasks.

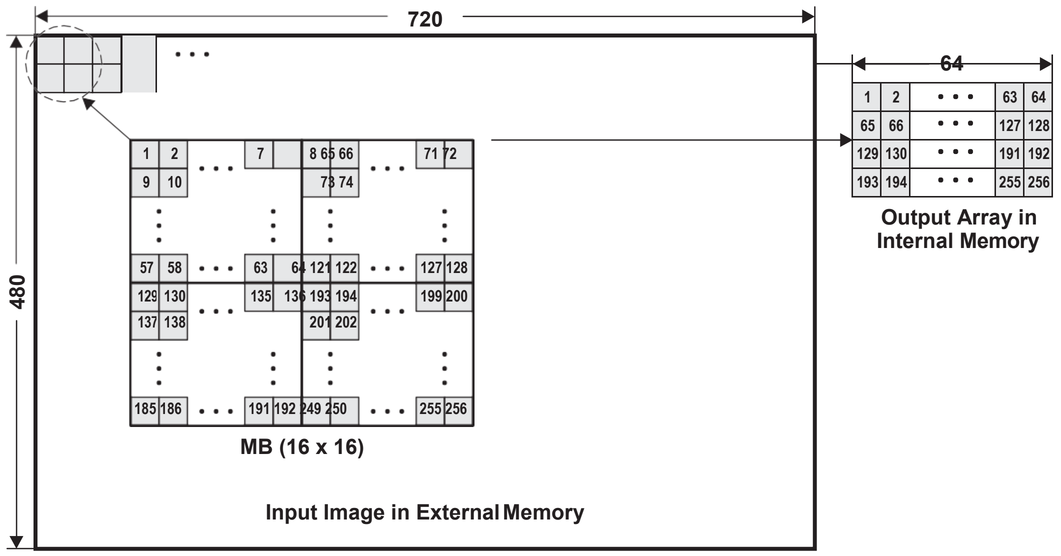
720
480
64
MB (16 x 16)
Input Image in External Memory
Output Array in Internal Memory

Figure 6. Convert 2-D Image to 1-D Data

The Y plane image in NTSC D1 resolution consists of 45 MBs per row and 30 MBs per column. The provided example copies one MB each time. Thus, 1350 transfers are needed. To copy and convert each MB, two regular channels are used. The first channel copies the upper two 8×8 blocks in each MB and the second channel copies the lower two blocks.

Using the DSP core to reprogram the PaRAM set each time after a transfer is completed is not efficient. In fact, all the field values of the PaRAM set should remain the same across consecutive transfers, except for the *SRC* and *DST* fields, which should be updated to point to the correct input and output buffer addresses. To reduce the overhead of reprogramming the PaRAM set, a link channel is allocated for each regular channel. The field values of its PaRAM set are the same as those of the PaRAM set corresponding to the regular channel. After each transfer is completed, the PaRAM set of the regular channel is reloaded from that of the link channel. By reprogramming only the *SRC* and *DST* fields, the next transfer can be started.

The following steps demonstrate how to copy 2-D blocks in the original Y plane image from the external memory to the internal memory, while converting them to 1-D data.

1. Allocate two regular channels by calling EDMA3_DRV_requestChannel() twice. This API allocates a TCC code and a PaRAM set along with the channel to the caller. You need to provide a callback for each allocation. The purpose of the callback is explained in step 3.
2. Allocate two link channels by calling EDMA3_DRV_requestChannel() twice. You do not need to provide a callback here.

3. Configure the PaRAM set associated with each regular channel by calling EDMA3_DRV_setSrcParams().
   - The field values of the PaRAM set corresponding to the first regular channel are as follows:
     – *SRC* should be the start address of the input Y plane image.
     – *DST* should be the start address of the output buffer.
     – *SRCBIDX* = *720, SRCCIDX* = *8, DSTBIDX* = *8 and DSTCIDX* = *64.* This indicates that two upper 8×8 blocks are copied and converted to two vectors and the output buffer, with each vector length being 64.
     – *ACNT* = *8, BCNT* = *8 and CCNT* = *2.* This indicates that the transfer is configured as a 3-D transfer. The 1-D transfer copies each row of the 8×8 block. The 2-D transfer consists of eight 1-D transfers to copy the whole block. The 3-D transfer consists of two 2-D transfers to copy two upper blocks.
     – *OPT.SYNCDIM* = *SYNC_AB, OPT.TCINTEN* = *ENABLE, OPT.ITCINTEN* = *ENABLE, and OPT.TCCMODE* = *NORMAL.* This indicates that after one 8×8 block is transferred, you will be notified by an interrupt. Inside the callback, you should trigger a new transfer to copy the next block.
   - The field values of the PaRAM set corresponding to the second channel are as follows:
     – *SRC* should be the start address of the input Y plane image, plus 720×8 bytes.
     – *DST* should be the start address of the output buffer, plus 128 bytes.
     – Values of *SRCBIDX*, *SRCCIDX*, *DSTBIDX*, and *DSTCIDX* are the same as for the first channel.
     – Values of *ACNT*, *BCNT,* and *CCNT* are the same as for the first channel.
     – Values of *OPT.SYNCDIM*, *OPT.TCINTEN*, *OPT.ITCINTEN,* and *OPT.TCCMODE* are the same as for the first channel.
4. For each regular channel, link it with the corresponding link channel by calling EDMA3_DRV_linkChannel(), then copy its PaRAM set content to the link channel's PaRAM set.
5. Manually enable the first transfer for each regular channel by calling EDMA3_DRV_enableTransfer(). As mentioned previously, once the first intermediate (2-D) transfer is completed, an interrupt is generated. Inside the callback, you are expected to enable the next 2-D transfer one more time to copy the next 8×8 block. The channel considers that the transfer is complete after the two 8×8 blocks are copied.
6. After both channels have finished copying and converting the two 8×8 blocks, reprogram the *SRC* and *DST* fields in their PaRAM sets to the appropriate values. For instance, the *SRC* field of the first channel should point to the start address of the next MB in the input image.
7. The transfer continues until all 1350 MBs have been copied. After that, all the channels need to be freed by calling EDMA3_DRV_freeChannel().

5 Example D: PING-PONG Buffering Scheme

This section explains how to use the PING-PONG buffering scheme that is required for many DSP applications. Two buffers reside in the internal memory. When the PING (or PONG) buffer is processed by the DSP, the PONG (or PING) buffer is filled by the EDMA3. By the time the PING (or PONG) buffer has been processed, the DSP can start processing the PONG (or PING) buffer without waiting for it to be filled.

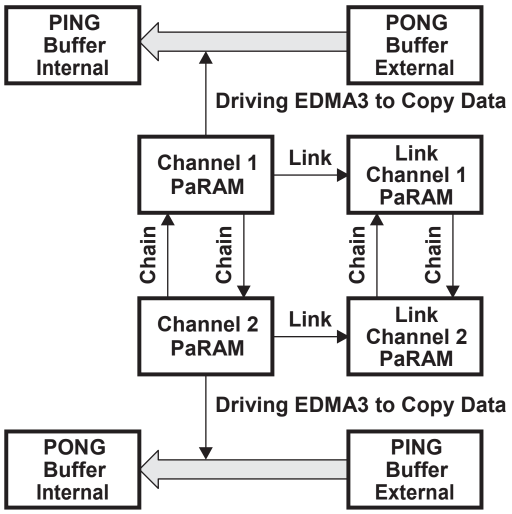

Figure 7. PING-PONG Buffer Copy

This example uses the link and chain feature to automatically trigger the next transfer once the previous one is finished without using the DSP to reprogram any PaRAM set or reenable the transfer. The idea is illustrated in Figure 7. Two regular channels are allocated and fill the PING and PONG buffer, respectively. In addition, two link channels are allocated and linked to the corresponding regular channel. When the current transfer is finished, the regular channel PaRAM set can be reloaded from the link channel's PaRAM set without DSP intervention. Two regular (link) channels are also chained to each other by programming the LINK field in their PaRAM sets. As a result, when the PING (or PONG) buffer is filled by channel 1, filling the PONG (or PING) buffer can be automatically started. Such an automatic transfer-trigger feature imposes a critical real-time requirement to the application. It implies that you have to make sure the processing time in each round is less than the data transfer time.

In the provided example, both the internal and external PING (or PONG) buffer size are set to 4096 bytes. The element size is set to 32 bits. Each transfer copies 1K elements from the external PING (or PONG) buffer to the internal PING (or PONG) buffer.

The following steps demonstrate the PING-PONG buffering scheme using the EDMA3 driver:

1. Allocate two regular channels by calling EDMA3_DRV_requestChannel() twice. This API allocates a TCC code and a PaRAM set along with the channel to the caller. You need to provide a callback for each allocation. In this example, the only purpose of the callback is to notify you that the transfer is completed.
2. Allocate two link channels by calling EDMA3_DRV_requestChannel() twice. You do not need to provide a callback here.

3. Configure the PaRAM set associated with each regular channel by calling EDMA3_DRV_setSrcParams().
   - The field values of the PaRAM set corresponding to the first channel are as follows:
     – *SRC* should be the start address of the PING buffer in the external memory.
     – *DST* should be the start address of the PING buffer in the internal memory.
     – *SRCBIDX = 4096, DSTBIDX = 4096, ACNT = 4096, BCNT = 1 and CCNT = 1.* This indicates that each transfer is configured as a 1-D transfer copying 4096 bytes (1K elements). The values of *SRCCIDX* and *DSTCIDX* do not matter in this example.
     – *OPT.SYNCDIM = SYNC_AB, OPT.TCINTEN = ENABLE, and OPT.TCCMODE = NORMAL.* This indicates that after the PING buffer is copied, you will be notified by an interrupt.
   - The field values of the PaRAM set corresponding to the second channel are as follows:
     – *SRC* should be the start address of the input PONG buffer in the external memory.
     – *DST* should be the start address of the PONG buffer in the internal memory.
     – Values of *SRCBIDX*, *SRCCIDX*, *ACNT*, *BCNT*, and *CCNT* are the same as for the first channel.
     – Values of *OPT.SYNCDIM*, *OPT.TCINTEN*, and *OPT.TCCMODE* are the same as for the first channel.
4. Chain the regular channels to each other by calling the EDMA3_DRV_chainChannel() twice.
5. For each regular channel, link it with the corresponding link channel by calling EDMA3_DRV_linkChannel(), then copy its PaRAM set content to the linked channel's PaRAM set.
6. Manually enable the first transfer of the first channel to fill the PING buffer by calling EDMA3_DRV_enableTransfer(). Once it is completed, an interrupt is generated and the next transfer of the second channel to fill the PONG buffer is triggered automatically. When the PONG buffer is filled, the data in the PING buffer can be processed.
7. The transfer continues for a specified number of times. After the example is finished, all the channels need to be freed by calling EDMA3_DRV_freeChannel().

6 Package Usage Guide

6.1 Hardware and Software Requirement

The following equipment and software are needed to compile and execute the provided example code.

- A PC with Microsoft® Windows XP® operating system pre-installed
- Texas Instruments Code Composer Studio™ 3.3 (CCStudio) software already installed on PC
- A TMS320DM6437 evaluation module (EVM) board

6.2 Package Contents

The package provided is compressed as a file called spraan4.zip. Unzipping the package to the PC creates a folder called edma3test. There are six sub-folders in the edma3test folder: CSL_inc, driver, Debug, Release, test, and edma3test.CS_.

The CSL_inc folder contains the chip support library (CSL) header files of all the peripherals on TMS320DM643x. These files implement the polling mode that checks that the transfer is complete, which works around the limitation of the current EDMA3 driver. These header files will not be needed if you use a future driver without the limitation.

The driver folder contains various libraries of the EDMA3 driver in object format. Table 1 lists the files in the driver folder.

Table 1. Contents of the edma3test/driver Folder

File(s)	Description
edma3_drv_bios_dbg.lib	Debug version of the driver library.
edma3_drv_bios_rel.lib	Release version of the driver library.
edma3_rm_bios_dbg.lib	Debug version of the resource manager. The edma3_drv_bios_dbg.lib file depends on it.
edma3_rm_bios_rel.lib	Release version of the resource manager. The edma3_drv_bios_rel.lib file depends on it.
bios_edma3_sample_cfg.c	Defines the resource of the EDMA3 hardware module on the TMS320DM673x. It comes with the EDMA3 driver. If the driver needs to be reused for a different device, this file is the only place that you need to modify to correctly define the resource on the device.
bios_edma3_sample_cs.c	Implements the functions related to OS service required by the driver. It comes with the EDMA3 driver.
bios_edma3_sample_init.c	Implements the functions initializing the EDMA3 hardware module and the driver. It comes with the EDMA3 driver.
bios_edma3_sample.h, edma3_common.h, edma3_drv.h, edma3_log.h, edma3_rl_cc.h, edma3_rl_tc.h, edma3_rm.h	Driver head files. To use the driver, you only need to include file bios_edma3_sample.h.

The Debug folder contains the binary code generated from compiling the debug version of the example package. The executable is edma3test.out. The memory map file is edma3test.map.

The Release folder contains the binary code generated from compiling the release version of the example package. The executable is edma3test.out. The memory map file is edma3test.map.

The test folder contains the source code that demonstrates how to program the EDMA3 driver to implement the examples that were previously described. Table 2 lists the files in the test folder.

Table 2. Contents of the edma3test/test Folder

File(s)	Description
callback.c	Contains the callback routine that is called when an intermediate or a final transfer is completed.
main.c	Contains the main entry function of the examples.
test.h	Defines the data structure, function prototypes, and global variable prototypes used in the examples.
test_2D1D.c	Implements the example described in Section 4.
test_2D2D.c	Implements the example described in Section 3.
test_422to420_conversion.c	Implements the example described in Section 2.
test_PingPong.c	Implements the example described in Section 5.
timer.lib	Implements the function _C64P_getltime() to help measure the performance.

The folder edma3 includes the following files, among others.

- The edma3test.pjt project file is the Code Composer Studio (CCStudio) software project file.
- The edma3test.tcf configuration file configures the DSP/BIOS™ software kernel foundation that is the operating system running on the TMS320DM643x.
- The edma3test.cmd command file defines some memory buffer sections in the L1, L2, and external memory.

6.3 How to Run the Example

The following steps show how to run the example using Code Composer Studio:

1. Start Code Composer Studio and open the project file edma3test.pjt.
2. To run the debug (release) version of the example, you must load the .out file to the board. To load a file to the board, select File → Load Program, then choose the edma3test.out file from either the Debug or Release folder of the project.
3. Reset the CPU by selecting *Debug→Reset CPU*.
4. Restart the program by selecting *Debug→Restart*.
5. Open the RTDX log window by selecting *DSP/BIOS→Message Log*.
6. Hit the F5 key to run the program, or select *Debug→Run*.
7. When the messages in the RTDX log window have been printed, stop the program using the shift+F5 keys. The correct messages are shown below.

Note: Your performance numbers may differ from the numbers shown below.

```
0    Starting EDMA3 examples!

1    test_422to420_conversion() started:
2       cache operation cycles = 319982
3       Tran(Y) cycles = 4751786
4       Tran(U) cycles = 1300288
5       Tran(V) cycles = 1308284
6    test_422to420_conversion() passed!

7    test_2D2D() started:
8       cache operation cycles = 23514
9       Tran cycles = 15316
10   test_2D2D() to L2 mem passed!

11   test_2D2D_nointr() started:
12       cache operation cycles = 35276
13       Tran cycles = 7462
14   test_2D2D_nointr() to L2 mem passed!

15   test_PINGPONG_LinkChain() started:
16       cache operation cycles = 3204
17       Tran cycles = 567987
18   test_PINGPONG_LinkChain() passed!

19   test_2D1D() started:
20       cache operation cycles = 500167
21       Tran cycles = 13053419
22   test_2D1D() passed!

23   All EDMA3 examples are done!
```

6.4 How to Compile the Example

To compile the example using Code Composer Studio:

1. Start Code Composer Studio and open the project file edma3test.pjt.
2. To compile the debug (or release) version of the example, select *Debug* (*or Release*) in the configuration window. Select *Project → Build* to build the code

7 References

1. *DMP Enhanced Direct Memory Access (EDMA3) Controller User's Guide* (SPRU987)
2. *EDMA3 0 3.0 User's Manual*, included with the digital video software development kit (DVSDK) software package of the TMS320DM6437 EVM board