CMX DMA for MA2x5x

User Guide

v1.0

Movidius Ltd.
1730 South El Camino Real, Suite 200
San Mateo, CA 94402
http://www.movidius.com/

Table of Contents

1 Overview

This document briefly describes the features of the new CMX DMA driver for MA2x5x and is intended to serve as a basic user guide. The current guide is meant for the new CMX DMA driver. This driver is intended to replace the existing old driver, but in order to ease the migration both drivers are present in the current MDK release.

2 Hardware Overview

The CMX DMA resides between the 128-bit MXI bus and CMX memory. It provides for scheduling high-bandwidth data transfers between CMX and DRAM in either direction. It also supports data transfers from DRAM back to DRAM or from CMX to CMX, allowing data to be relocated within the same physical location.

The CMX DMA engine processes a linked-list of DMA descriptors, which are created by the driver.

For more details about the hardware functionality please refer to the hardware specification document for MA2x5x.

3 CMX DMA Driver Design

The driver is designed to hide the hardware's complexity from the user. The main purpose of the driver is to expose functionality for the following 4 simple operations that the user needs:

- Initialize the driver (and the hardware);
  - Using an `Initialize` function
- Create individual (multiple if needed) CMX DMA descriptors and link them in a linked-list;
  - Using `CreateTransaction` and `AddTransaction` functions
- Send a linked-list of CMX DMA descriptors to the hardware;
  - Using `StartTransfer`
- Determine the completion of a linked-list that was transmitted to the hardware.
  - Blocking `StartTransfer` when possible, or polling explicitly by using `WaitTransaction`

Using these design elements, a simple test could be written as a sequence of the above functions. A fully functional example for both LEON and SHAVE code follows the description of the driver's API.

4 Implementation notes

- The new driver performance is similar with the performance of the old driver
- The CMX DMA descriptors must always be placed into CMX. CMX DMA controller cannot read the descriptors placed in DDR.
- The CMX DMA descriptors have to be aligned to 64-bits, based on HW specification. This alignment is already part of the driver's definitions, but if for any reason the user overrides them, he has to ensure for the alignment explicitly.
- A difference from the old driver is the way in which the wait functionality was implemented.

  The new driver allows out of order waiting for multiple lists of task.

Example:

```
ScCmxDmaStartTransfer(hnd1);
ScCmxDmaStartTransfer(hnd2);
ScCmxDmaStartTransfer(hnd3);
 // wait for all transactions to finish
 ScCmxDmaWaitTransaction(hnd2);
 ScCmxDmaWaitTransaction(hnd3);
 ScCmxDmaWaitTransaction(hnd1);
```

5 CMX DMA Driver Interface

LEON RTEMS functions start with `OsDrvCmxDma`, while SHAVE functions with `ScCmxDma`. The drivers for LEON and SHAVE have an identical interface for all functions (except for one), thus only the LEON driver will be described below. When the interface is different, both drivers will be described in parallel.

- To initialize the driver on the local processor, the following function can be used

  ```
  OsDrvCmxDmaInitialize(config)
  ```

  This function initializes the driver's global variables for the local processor and hardware, if not already initialized by another processor. The config parameter is optional (and unused by the SHAVE driver) and for the LEONs it defines the interrupt priority used for the driver. If left empty, the default interrupt priority is 7.

- To create a DMA transaction, the following function can be used

  ```
  OsDrvCmxDmaCreateTransaction(handle, transaction, src, dst, size)
  ```

  This function initializes both handle and transaction fields. First, it creates a new transaction descriptor, and then updates our local records in handle. The transaction descriptor contains the data transfer related fields, letting all other fields in the descriptor with their default values. These default values can be changes by the `CmxDmaTransactionConfig` function.

- To create a stride DMA transaction, with similar functionality as the above function, the following function can be used

  ```
  OsDrvCmxDmaCreateStrideTransaction(handle, transaction, src, dst,
                  src_width, dst_width, src_stride, dst_stride, size)
  ```

- To modify the configuration of an already created transaction, the following function can be called

  ```
  OsDrvCmxDmaTransactionConfig(transaction, params)
  ```

  This function allows the user to modify any pre-configured parameters of a transaction. Currently the only parameter that can be configured is the burst size. When a new Transaction is created, the maximum burst size (16 bytes) is used as the default value.

- To add a transaction to an existing handle, the following function can be called

  ```
  OsDrvCmxDmaAddTransaction(handle, transaction, src, dst, size)
  ```

- Similarly, to add a stride transaction to an existing handle

  ```
  OsDrvCmxDmaAddStrideTransaction(handle, transaction, src, dst,
              src_width, dst_width, src_stride, dst_stride, size)
  ```

- To link together a variable number of transactions in a single linked list

  ```
  OsDrvCmxDmaLinkTransactions(handle, cnt, ...)
  ```

  This function creates a va_list from the given arguments, and the listed transactions are linked together. Handle becomes their linked-list handler.

- To transfer an already created linked list of transactions

```
LEON: OsDrvCmxDmaStartTransfer(handle, wait)
SHAVE: ScCmxDmaStartTransfer(handle)
```

In the case of LEON, the user is allowed to set an additional flag if he wants to wait for the transaction to be completed before returning to the user.

- To check the state of an existing transaction

  ```
  OsDrvCmxDmaGetTransactionStatus(handle, status)
  ```

  This function checks for the status of a pending transfer. It has slightly different functionality for SHAVEs and LEONs. In LEONs, if the user did not want to created a blocking Transfer, the function can be used to selectively poll for the transaction statuses on demand.

6 Using the driver

The current documentation describes the latest implementation of the CMX DMA driver. This is a complete reimplementation of the existing driver, with a new API and slightly different functionality. This new API is not backward compatible with the old one.

In order to ease the migration and keep the existing project using the old driver working, both the old and the new driver are included in MDK. Only one driver is active at any time. By default the active driver is the old one. In order to activate the new driver the USE_CMX_DMA_NEW_DRIVER variable needs to be set. The easiest way to set it is by adding it as new define in the compiler option. This can be done by adding the following line in the project Makefile:

```
CCOPT += -D'USE_CMX_DMA_NEW_DRIVER=1' # if driver is used on LOS
CCOPT_LRT += -D'USE_CMX_DMA_NEW_DRIVER=1' # if driver is used on LRT
MVCCOPT += -D'USE_CMX_DMA_NEW_DRIVER=1' # if driver is used on SHAVES
```

Since two versions of the same driver are present they need to use different names for the includes. The table below shows the headers that need to be included for each driver version:

	Old Driver	New Driver
Include for Leon BM	`#include “DrvCmxDma.h”`	`#include “DrvCdma.h”`
Include for Leon OS	`#include “OsDrvCmxDma.h”`	`#include “OsDrvCdma.h”`
Include for SHAVEs	`#include “swcCdma.h”`	`#include “scCmxDma.h”`

7 Limitations

Reusing a list of one or more descriptors without reinitialization (by using `CmxDmaCreateTransaction`) can lead to a wrong behavior. This happens because the tail of the list may be modified when a list from another processor is linked to it.

8 Examples

The examples below describe one the simpler way to use the driver from both LEON and SHAVE perspectives.

LEON RTEMS example

```
#include "OsDrvCdma.h"
#define WAIT    1
#define NO_WAIT 0

static OsDrvCmxDmaTransaction list[LIST_SIZE];
static OsDrvCmxDmaTransactionHnd dma_handle;

static OsDrvCmxDmaSetupStruct setup = {
    .irq_priority = IRQ_PRIO,
    .agent = DRV_CMX_DMA_AGENT1
};
// use NULL as parameter instead of config for default configuration
int32_t status = OsDrvCmxDmaInitialize(&config);

if (status == OS_MYR_DRV_SUCCESS) {
  OsDrvCmxDmaCreateTransaction(&dma_handle, &list[0], src_buff, dst_buff,
                               transfer_size);
  for (i = 1; i < LIST_SIZE; i++)
    OsDrvCmxDmaAddTransaction(&dma_handle, &list[i], src_buff, dst_buff,
                             transfer_size);
  // Blocking wait
  OsDrvCmxDmaStartTransfer(&dma_handle, WAIT);
  ...
  // Non blocking
  OsDrvCmxDmaStartTransfer(&dma_handle, NO_WAIT);
  ...
  // check transaction status
  status = OsDrvCmxDmaGetTransactionStatus(&dma_handle, &tr_status);
  if ((status == OS_MYR_DRV_SUCCESS) &&
      (tr_status == OS_DRV_CMX_DMA_FINISHED)) {
    //transaction finished
  } else {
    // transaction not finished yet
  }
}
```

LEON BM example

```
#include "DrvCdma.h"

static DrvCmxDmaTransaction list[LIST_SIZE]
    __attribute__((section(".cmx_direct.bss")));
static DrvCmxDmaTransactionHnd dma_handle;

static DrvCmxDmaSetupStruct setup = {
    .irq_priority = IRQ_PRIO,
    .irq_enable = 1,  // 0 to disable CMX DMA interrupts
    .agent = DRV_CMX_DMA_AGENT1
};

// use NULL as parameter instead of config for default configuration
int32_t status = DrvCmxDmaInitialize(&config);

if (status == MYR_DRV_SUCCESS) {
  DrvCdmaCreateTransaction(&dma_handle, &list[0], src_buff, dst_buff,
                           transfer_size);
  for (i = 1; i < LIST_SIZE; i++)
    DrvCmxDmaAddTransaction(&dma_handle, &list[i], src_buff, dst_buff,
                            transfer_size);
  // callback and context are not mandatory; replace their
  // corresponding parameters with NULL if not needed
  DrvCmxDmaStartTransfer(&dma_handle, callback, context);
  DrvCmxDmaWaitTransaction(&dma_handle);
}
```

SHAVE example

```
#include "ScCmxDma.h"

 static ScCmxDmaTransaction list[LIST_SIZE]
   __attribute__((section(".cmx_direct.bss")));
 static ScCmxDmaTransactionHnd dma_handle;

 static ScCmxDmaSetupStruct setup = {
 .agent = DRV_CMX_DMA_AGENT1
 };

 // use NULL as parameter instead of config for default configuration
 int32_t status = ScCmxDmaInitialize(&config);

 if (status == MYR_DRV_SUCCESS) {
 ScCdmaCreateTransaction(&dma_handle, &list[0], src_buff, dst_buff,
             transfer_size);
 for (i = 1; i < LIST_SIZE; i++)
   ScCmxDmaAddTransaction(&dma_handle, &list[i], src_buff, dst_buff,
```

SHAVE example

```
                 transfer_size);
    ScCmxDmaStartTransfer(&dma_handle);
    ScCmxDmaWaitTransaction(&dma_handle);
}
```