

Chapter 4: Direct Memory Access (DMA)

After completing this chapter, you will understand the capabilities and some use cases of the DMA Controller in XMC7000 devices.

Table of contents

4.1	Overview	2
4.1.1	DMA Architecture.....	2
4.1.2	Priorities and preemption.....	3
4.1.3	Data transfer widths.....	3
4.1.4	Chaining descriptors/channels.....	4
4.2	DMA Configuration	6
4.3	XMC7000 DMA	7
4.3.1	Introduction	7
4.3.2	Features.....	9
4.3.3	Configuration	9
4.3.4	Transfer modes	11
4.3.5	Descriptor chaining.....	12
4.4	Exercises	13
	Exercise 1: 1-1 Transfer.....	13
	Exercise 2: 1-N Transfer	15
	Exercise 3: N-1 Transfer	17
	Exercise 4: N-N Transfer.....	19
	Exercise 5: N-NxM Transfer	22
	Exercise 6: Descriptor chaining	24
	Exercise 7: Channel chaining.....	26

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	<code>CY_ISR_PROTO(MyISR);</code> <code>make build</code>
<i>Italics</i>	Displays file names and paths	<i>sourcefile.hex</i>
[bracketed, bold]	Displays keyboard commands in procedures	[Enter] or [Ctrl] [C]
Menu > Selection	Represents menu paths	File > New Project > Clone
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the Debugger icon, and then click Next .

4.1 Overview

DMA is a feature that allows you to access memory independently of the CPU. With DMA, you can read/write from any memory location accessible to the XMC7000 devices. This includes the registers of peripherals, RAM, internal flash, and even external memories.

DMA is enabled by HW blocks that are specifically designed for data movement. These HW blocks are so effective at transferring data that they can actually transfer large blocks of data more quickly than the CPU could. There are two types of the DMA HW block that we will discuss in this chapter. The architecture and feature set of each type is very similar but they each differ in their performance, use cases, and implementations. First, we will discuss, in broad terms, the features that all of the blocks have in common, then we will look more closely at the specific implementations of each block.

4.1.1 DMA Architecture

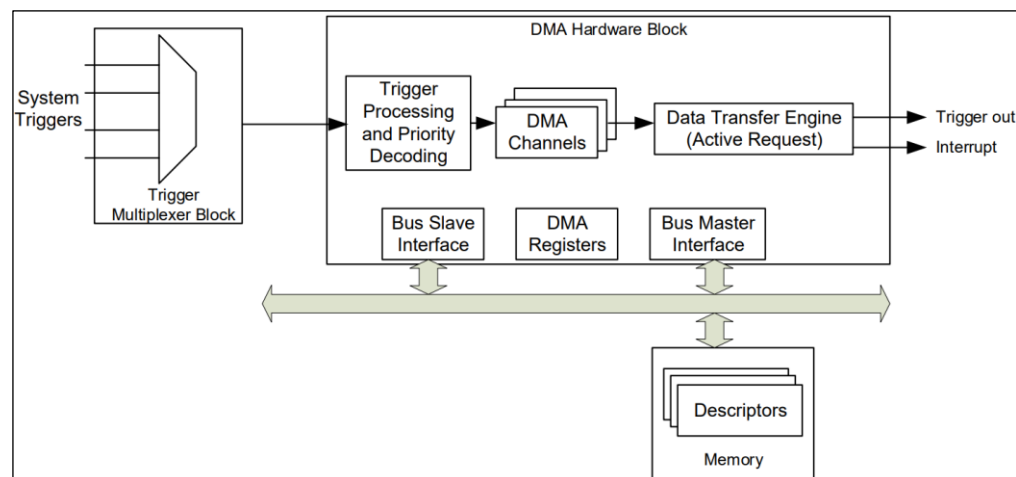
The XMC7000 devices have two types of DMA blocks – DMA Controller (DMAC) and DMA DataWire (DMA DW). This section talks about DMA in general. Later sections discuss the specific XMC7000 blocks.

Each DMA HW block implements multiple DMA channels that can be independently configured for different data transfers. Each of these channels, when enabled, waits for a trigger signal to begin running its data transfer protocol. Only one channel in a DMA HW block can be active at a time: if other channels are triggered while one is already active, they are placed into a pending state. When the DMA HW block completes the active channel's data transfer, pending channels are evaluated and run according to their priorities.

Each DMA channel has a trigger input, trigger output, and interrupt output line. The trigger signals are routed through a trigger multiplexer block, which enables the routing of trigger signals from different peripherals to the DMA block and vice versa. The trigger multiplexer block's architecture is device specific and, on most devices, only certain DMA channels can connect their trigger lines to certain peripherals. For details on which DMA channels can connect their trigger lines to which peripherals, you will need to refer to your device's documentation.

The data transfer protocol associated with a particular DMA channel is defined by a "descriptor", a structure you create that specifies different aspects of the transfer such as the size of each datum, the number of datums to transfer, and the source/destination addresses.

DMA Architecture

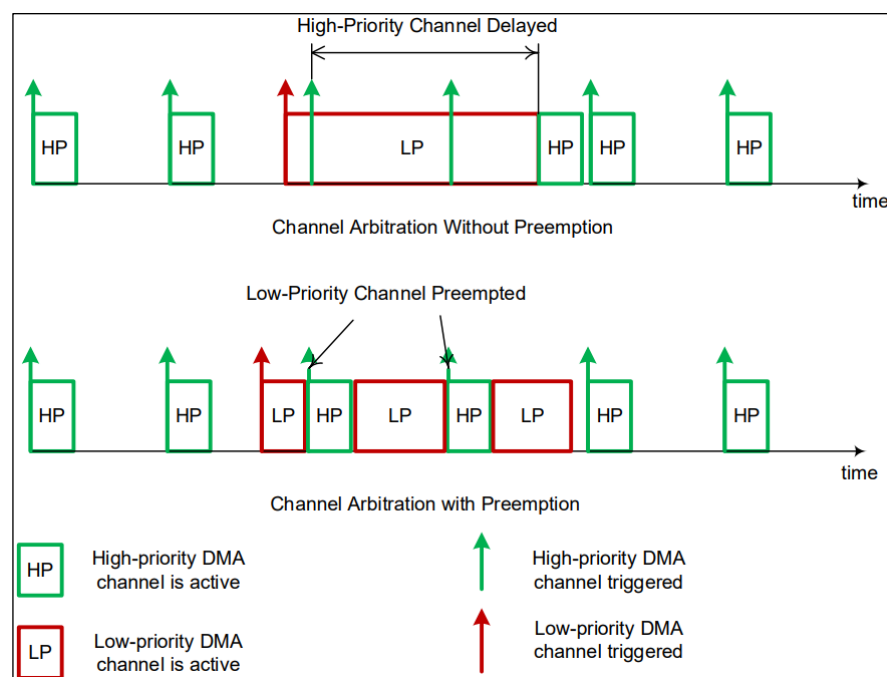


4.1.2 Priorities and preemption

There are four priority levels available to a DMA channel (0-3). Every DMA channel has an associated priority value that the DMA HW block uses to determine which channel to run when multiple channels are pending. In such a situation the channel with the lowest priority number becomes active. In the case where there are multiple channels with the same priority that are pending, a round robin scheme of arbitration is employed.

By default, DMA channels complete their data transfer before yielding to another channel. That is, if there is a low-priority channel in the middle of a large data transfer when a high-priority channel is triggered, the high-priority channel will not be able to run until the low-priority channel has completed its transfer. This could be a problem if the high-priority channel's data transfer is time sensitive. To address this, each channel can be marked as "preemptible". This parameter allows a higher-priority channel to preempt the marked channel when it is active. If a low-priority preemptible channel is active when a channel with a higher priority is triggered, the DMA HW block will pause the low-priority channel and allow the high-priority channel to run to completion. The low-priority channel will then be allowed to complete its data transfer, as long as another high-priority channel is not triggered. A low-priority preemptible channel can be preempted multiple times during a single transfer.

DMA Channel Arbitration and Preemption Example



4.1.3 Data transfer widths

Data transfer width is a descriptor parameter that determines the width of the data being accessed at the source and destination. This setting also determines the value of each source/destination address increment.

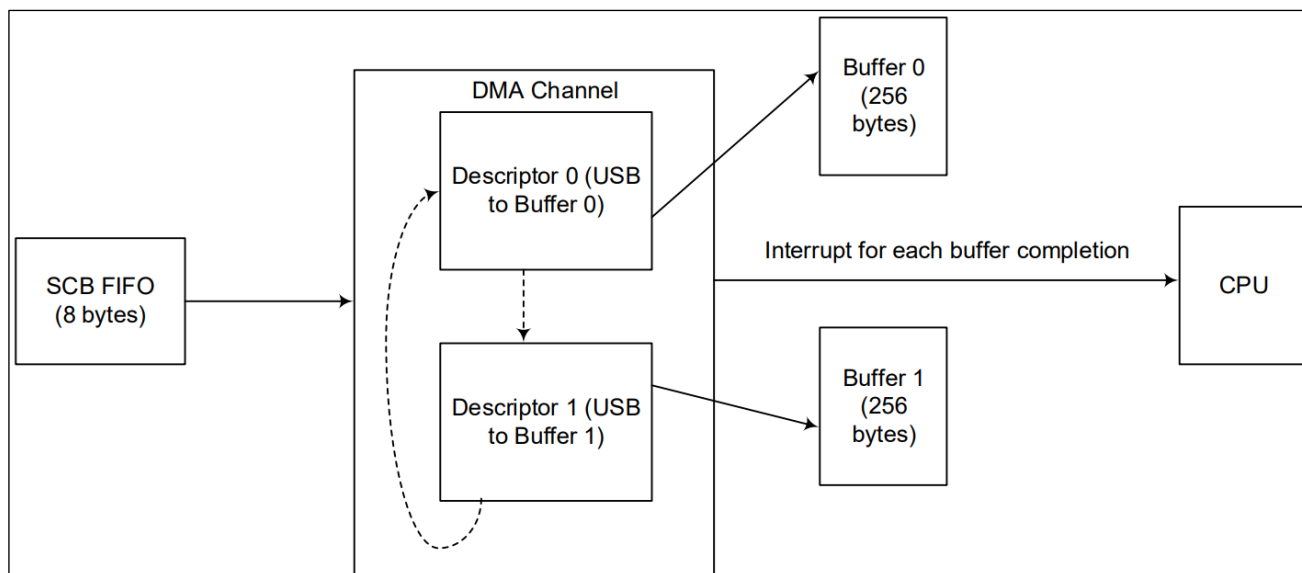
The XMC7000 DMA HW blocks support 32-bit (word), 16-bit (half word), and 8-bit (byte) reads and writes. A DMA channel's source width does not have to be equal to its destination width. When a channel's destination width is smaller than its source width, the higher bits will be truncated (i.e. not transferred). When a channel's destination width is larger than its source width, the higher bits will be padded with zeros. The data transfer widths you select must be a width supported by the peripheral or memory being accessed.

4.1.4 Chaining descriptors/channels

All XMC7000 DMA HW blocks support both descriptor and channel chaining in one way or another. For specifics on how to chain descriptors or channels, refer to the device specific sections of this chapter.

Descriptor Chaining

Descriptor chaining is used to link multiple descriptors together within one channel so that they run one after another. This can be useful when you need multiple different types of transfers to occur in succession. Each of the chained descriptors can have a completely different configuration including different source/destination addresses, trigger settings, interrupt settings, transfer modes, and data widths. It is possible to have a circular descriptor chain; in a circular chain, DMA execution will continue indefinitely until there is an error or the channel is disabled by user code.

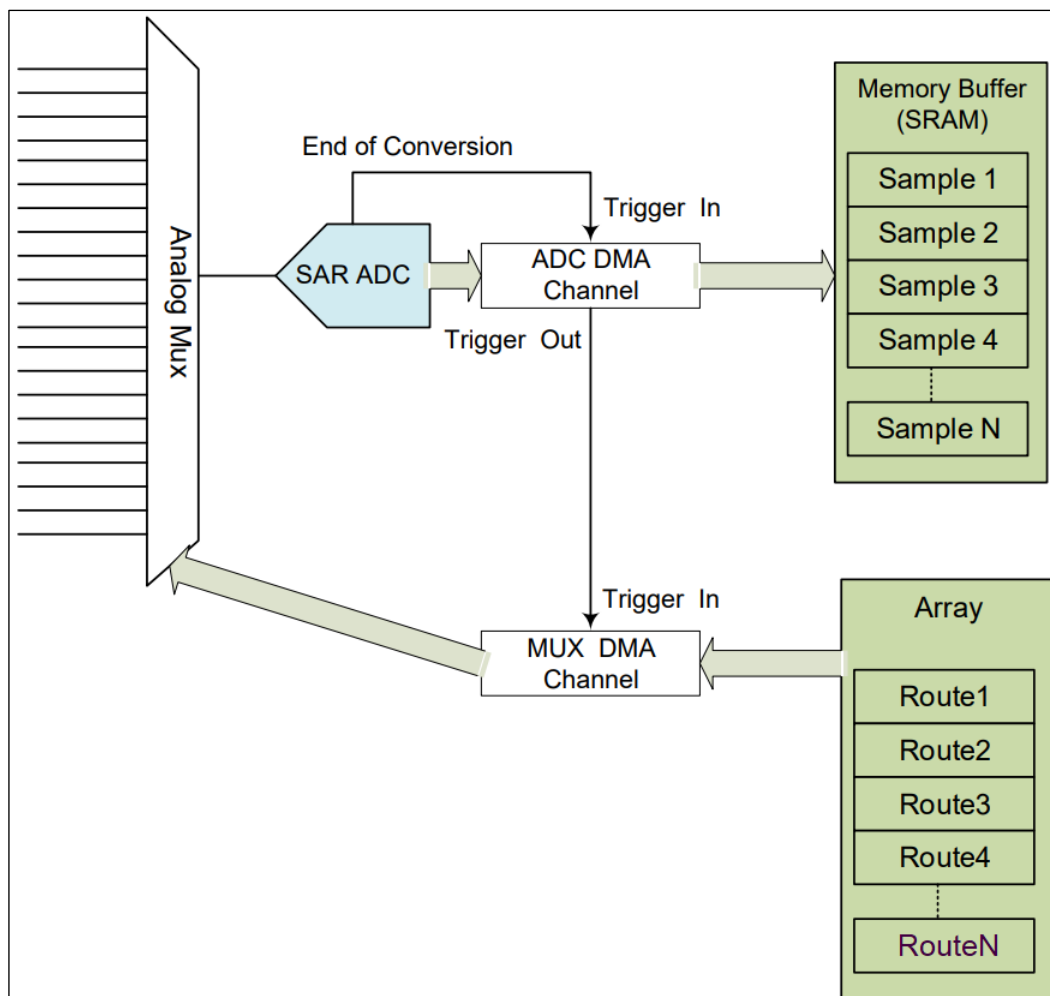


In this example the input data comes in 8-byte blocks in the SCB FIFO, which need to be processed by the CPU. Two buffers are used so that one buffer can be filled by the SCB FIFO while the CPU processes data from the other buffer. The buffers are 256 bytes each and thus can accommodate 32 FIFO's worth of data before overflowing. A single DMA channel is used for the transfer, with two descriptors that are each chained to the other. Both descriptors are set up for a 2D transfer (Read about this transfer mode in the Transfer Modes section of this chapter) with the FIFO as the source. Descriptor 0 has buffer 0 set as its destination while descriptor 1 has buffer 1 set as its destination. First descriptor 0 runs and fills buffer 0. When it completes, it triggers descriptor 1 and interrupts the CPU so that it can begin processing the data. As soon as descriptor 1 is triggered it begins shuttling data from the FIFO to buffer 1. When it is finished, it triggers descriptor 0 and interrupts the CPU so that the cycle can continue.

Channel Chaining

Channel chaining is used to initiate a second channel immediately after a first channel completes its transfer. This is done by connecting the trigger output signal of the first channel to the trigger input signal of another channel. Depending on the specific trigger multiplexer routing in a given MCU, only certain DMA channels will have the ability to chain. Refer to your device's documentation to find out which channels support this.

The following example shows a use case for channel chaining. In this example, DMA is used to move ADC results to memory and then re-configure the ADC's input for the next conversion.

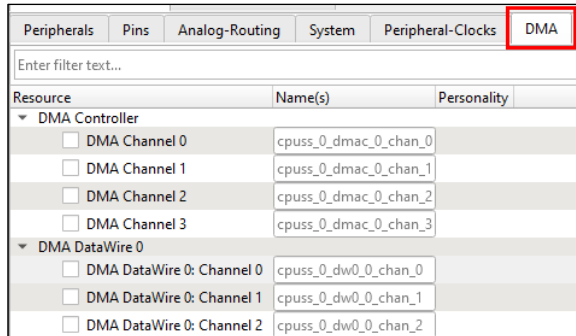


Here the input to the ADC is a large analog multiplexer. Because the number of inputs to the mux is so large, indexing the mux is not supported by the inbuilt multiplexer in the SAR ADC HW. Instead this analog mux has a routing register where different values can be written into to select which input is connected to the ADC.

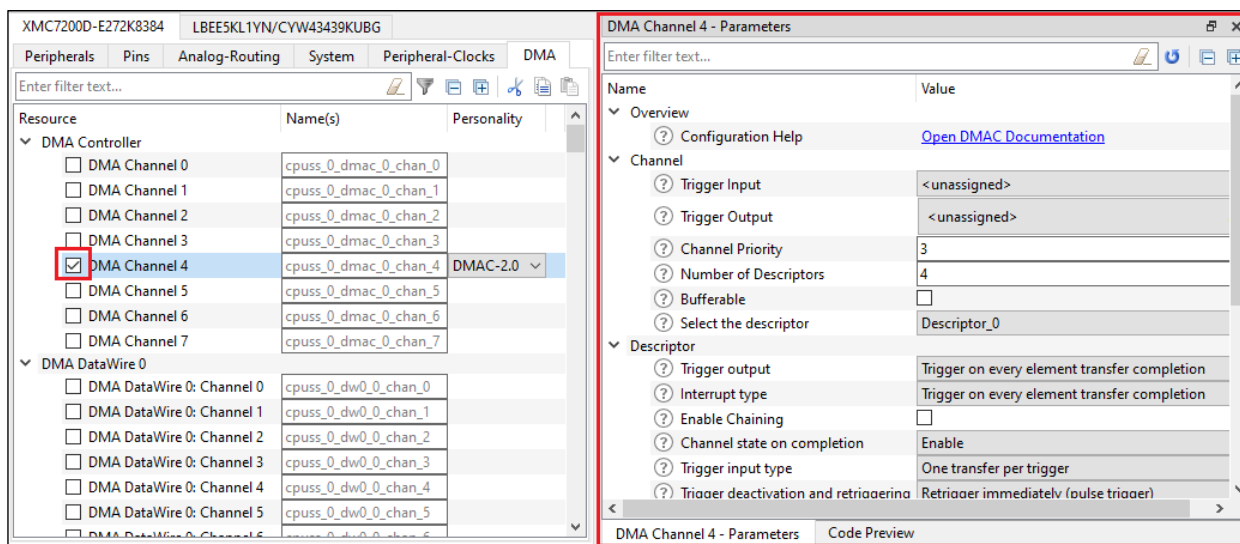
When an ADC conversion is completed, the ADC DMA channel is triggered. This moves data from the ADC result register to a memory buffer. After the transfer is completed, the ADC DMA channel triggers the MUX DMA channel. The source for the MUX DMA channel is a set of memory locations with preset index values for the mux's routing register. Whenever the MUX DMA channel is triggered, it transfers the new index value to the mux's routing register, thereby switching the ADC's input.

4.2 DMA Configuration

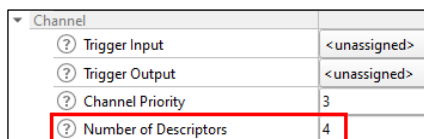
To set up DMA on your device, open the Device Configurator and go the **DMA** tab. Here you will see a list of all the DMA blocks that your device has. Under each block is a drop-down menu of all of that block's channels:



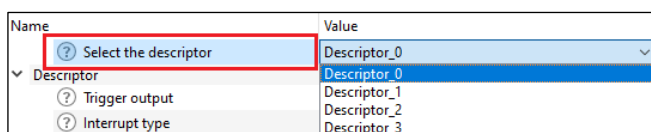
To enable and configure a DMA channel, simply check the box next to the channel you want to use and enter a name for it. Then, on the right side of the screen, you will be presented with a list of DMA parameters to configure for the selected channel:



The DMA channels may have up to 256 descriptors at once. You set the number of descriptors by configuring the **Number of Descriptors** parameter:



Then to configure each descriptor, select the descriptor you want to configure in the **Select the descriptor** drop-down menu:



Then the parameters for the selected descriptor will be available for configuration under the "Descriptor..." headers:

Name	Value
? Select the descriptor	Descriptor_2
▼ Descriptor	
? Trigger output	Trigger on every element transfer completion
? Interrupt type	Trigger on every element transfer completion
? Enable Chaining	<input type="checkbox"/>
? Channel state on completion	Enable
? Trigger input type	One transfer per trigger
? Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
? Data prefetch	<input type="checkbox"/>
? Data transfer width	Word to Word (full 32 bit)
▼ Descriptor X loop settings	
? Number of data elements to transfer	1
? Source increment every cycle by	1
? Destination increment every cycle by	1
? Scatter transfer	<input type="checkbox"/>
▼ Descriptor Y loop settings	
? Number of X-loops to execute	1
? Source increment every cycle by	1
? Destination increment every cycle by	1
▼ Advanced	
? Store Config in Flash	<input checked="" type="checkbox"/>

Note: The screenshots here are from the DMAC block channel settings, but the DMA DW block channel settings work the same way.

4.3 XMC7000 DMA

4.3.1 Introduction

The XMC7000 DMA controllers can seamlessly transfer data between memory and on-chip peripherals, or between memories without CPU intervention. This allows the CPU to handle other tasks while the DMA controller transfers data. As previously mentioned, there are two types of DMA HW block – DMAC and DMA DW. Both DW and DMAC have multiple independent DMA channels. Each DMA channel has a separate DMA request input that initiates the transaction and can independently transfer data. See the device datasheet for the number of DMA channels available for each device.

The DMA DW blocks are optimized for peripheral-to-memory and memory-to-peripheral low-latency data transfers for many channels. These are also sometimes called Peripheral DMA or P-DMA.

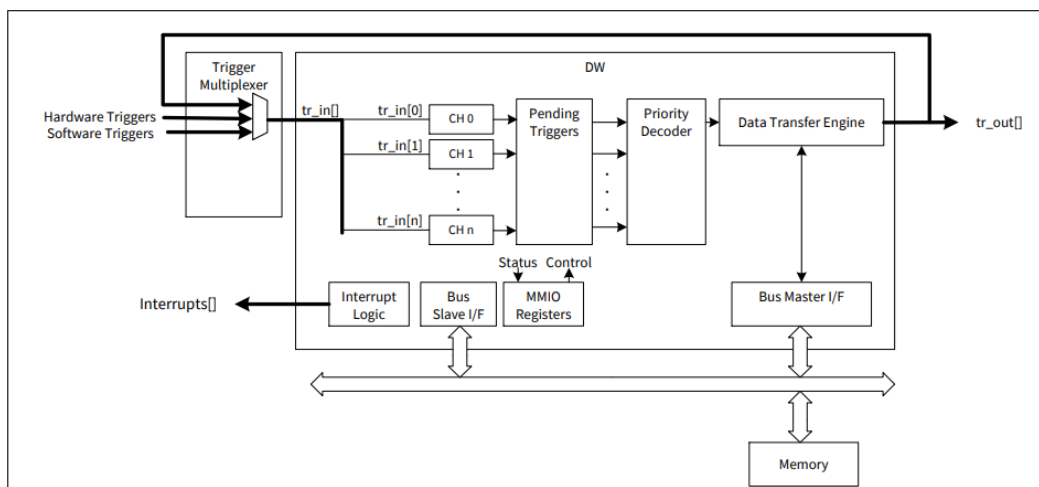
The DMAC blocks are optimized for memory-to-memory high-memory-bandwidth data transfer for a small number of channels. These are also sometimes called Memory DMA or M-DMA.

These DMA controllers have descriptors that specify the transfer operation, and it corresponds flexibly to various applications. Descriptors can be chained and it is possible to have circular lists. To understand more about XMC7000 DMA, see the “Direct Memory Access” chapter of the architecture technical reference manual (TRM).

4.3.1.1 P-DMA (DMA DW)

A DMA DW block consists of channels (CH0 – CHn), a pending trigger block, priority decoder, data transfer engine, and interrupt logic. The DW transfer engine is shared by all channels within a given DMA DW HW block. See the architecture TRM for details of each block.

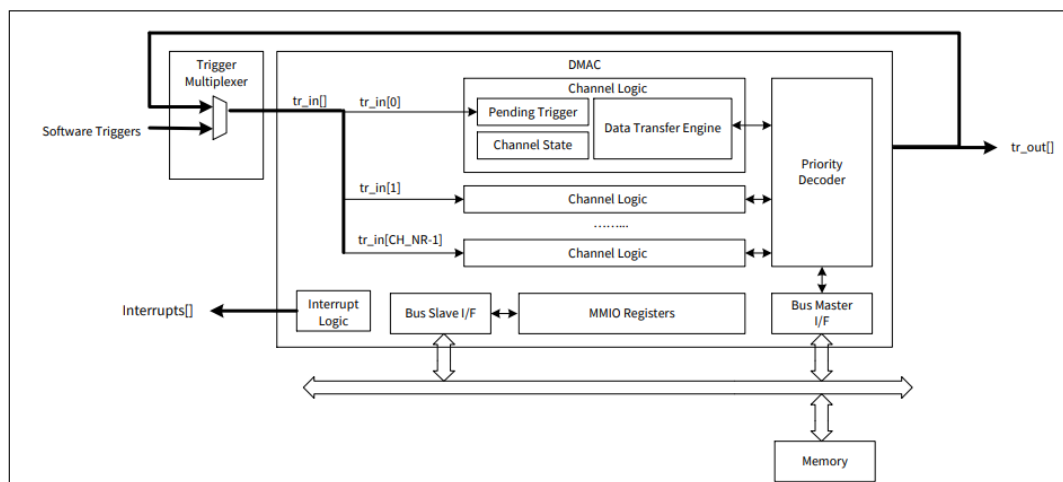
As mentioned earlier, DW trigger inputs can be a hardware trigger, software trigger, or trigger output. These triggers are provided via the trigger multiplexer. The trigger output can be used as its own trigger input, or it can be used as the trigger input to trigger different transfers of other channels. The memory that is used to store descriptors is outside the DMA block. When the transfer engine activates the next pending channel, the transfer engine reads the descriptor corresponding to the channel from the memory and starts the transfer.



4.3.1.2 M-DMA (DMAC)

The DMAC block consists of the channel logic, priority decoder, and registers. The channel logic itself stores the pending trigger and hosts the current channel state and data transfer engine. DMAC has transfer engines dedicated for each channel. See the architecture TRM for details of each block.

As trigger inputs, DMAC supports software trigger and its own trigger output. These trigger inputs are provided via the trigger multiplexer. Note that unlike DW, DMAC does not support hardware triggers.



4.3.2 Features

Feature	DW	DMAC
Focuses on	Low latency	High memory bandwidth
Useful for	Transfer between peripheral and memory	Transfer between memories
Transfer engine	Shared all channels	Dedicated for each channel
Transfer size	8-bit, 16-bit, 32-bit	8-bit, 16-bit, 32-bit
Channel priority	<ul style="list-style-type: none"> Four levels Preemptable 	Four levels
Descriptor type	<ul style="list-style-type: none"> Single transfer 1D/2D transfer CRC transfer 	<ul style="list-style-type: none"> Single transfer 1D/2D transfer Memory copy Scatter

Descriptor	<ul style="list-style-type: none"> Source and destination address Transfer size Descriptor type Trigger-in type (four types) Trigger-out type (four types) Interrupt type (four types) Descriptor chaining 	<ul style="list-style-type: none"> Source and destination address Transfer size Descriptor type Trigger-in type (four types) Trigger-out type (four types) Interrupt type (four types) Descriptor chaining
Trigger input	<ul style="list-style-type: none"> Hardware trigger 	<ul style="list-style-type: none"> Software trigger
	<ul style="list-style-type: none"> Software trigger Trigger output (tr_out) 	<ul style="list-style-type: none"> Trigger output (tr_out)

4.3.3 Configuration

The descriptor settings are almost identical for DMA DW and DMAC. To utilize DMA on XMC7000, you will need to configure the following parameters:

Channel Level Parameters

- Trigger Input – The signal to connect to the trigger input of the DMA channel. A rising edge on this input will cause the channel to begin a transfer.
- Trigger Output – What to connect the DMA channel's trigger output signal to. This signal will be sent every time a descriptor's transfer is completed.
- Channel Priority – The priority level of the transfer
- Number of Descriptors – The number of descriptors to associate with the channel. This can be up to 256.
- Bufferable – Whether or not to make the channel's data transactions bufferable

- Preemptable – Whether or not to allow the channel to be preempted by a higher priority channel

Descriptor Level Parameters

- Trigger Output – What events should trigger the DMA output trigger. The options for this are:
 - Every Element – After the transfer of each data element
 - Every X Loop – After the completion of each X loop
 - Entire Descriptor – After the descriptor's entire transfer completes
 - Descriptor Chain – After the entire descriptor chain completes. (After a descriptor whose "Trigger Input Type" parameter is not set to "Entire Descriptor Chain" is run)
- Interrupt Type – What events should trigger an interrupt. The options for this are:
 - Every Element – After the transfer of each data element
 - Every X Loop – After the completion of each X loop
 - Entire Descriptor – After the descriptor's entire transfer completes
 - Descriptor Chain – After the entire descriptor chain completes. (After a descriptor whose "Trigger Input Type" parameter is not set to "Entire Descriptor Chain" is run)
- Enable Chaining – Whether or not to enable chaining for this descriptor. If this is not enabled, the descriptor's "Next Descriptor" parameter will be set to `NULL`.
- Next Descriptor – The descriptor to chain to the current descriptor. If chaining is enabled, this will be the descriptor that runs immediately after the current descriptor.
- Channel State on Completion – Whether or not to disable the channel after the descriptor completes its transfer.
- Trigger Type – The number of data elements to transfer per trigger. The options for this are:
 - Single Element – Only a single data element is transferred per trigger
 - One X Loop – The X loop of the descriptor will be run once
 - Entire Descriptor – The entire transfer specified by the descriptor will be run
 - Entire Descriptor Chain – The current descriptor and all descriptors chained to it are run
- Trigger Deactivation and Retriggering – How long to wait before reactivating the DMA channel's input trigger.
- Data Prefetch (DMAC Only) – Whether or not to prefetch data. The prefetch occurs as soon as the channel is enabled.
- Data Transfer Width – The number of bytes to read from the source address and write to the destination address. The source and destination transfer widths are independently configurable.
- X Loop Data Count – The number of data elements to transfer per X loop.
- X Loop Source Address Increment – The number of data elements to increment the source address after every data element read. (The size of a data element is determined by source transfer size) The increments applied during the X loop are reset after the completion of the X loop.
- X Loop Destination Address Increment – The number of data elements to increment the destination address by after every data element write. (The size of a data element is determined by destination transfer size) The increments applied during the X loop are reset after the completion of the X loop.
- Scatter Transfer (DMAC Only) – Whether or not this descriptor is a scatter transfer.

- Y Loop Count – The number of X loops to run per Y loop.
- Y Loop Source Address Increment – The number of data elements to increment the source address after every completion of an X loop.
- Y Loop Destination Address Increment – The number of data elements to increment the destination address after every completion of an X loop.

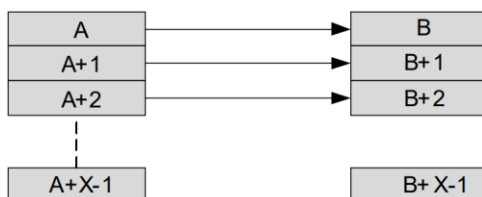
4.3.4 Transfer modes

The DMA blocks support the following transfer modes. Unless otherwise stated, the transfer modes apply to both DMA DW and DMAC:

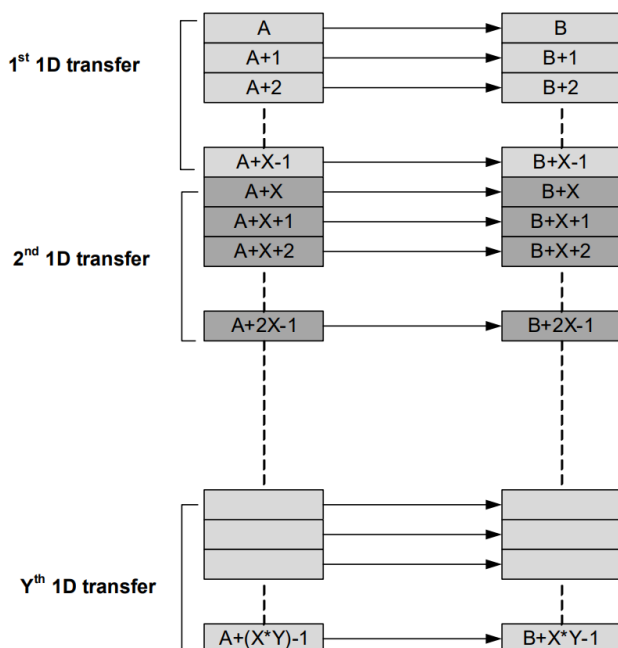
- Single Transfer – Every time the descriptor is triggered it will transfer only one data element. Each single transfer needs to be initiated by a trigger signal.



- 1D Transfer (X loop) – This allows for multiple data elements to be transferred as defined in a single descriptor. You can trigger the transfers one data element at a time or do the entire X loop at once.



- 2D Transfer (Y loop) – This allows for multiple 1D transfers to be defined in a single descriptor. You can choose to trigger the transfers one data element at a time, one X loop at a time, or you can trigger the entire 2D transfer at once.



- Scatter Transfer (DMAC only) – A special case of single transfer intended to transfer a set of data elements whose addresses are "scattered" around the address space. For more information on this mode refer to the above TRM.

Loops

The DMA descriptors allow you to use nested loops to define data transfers. Think of these as nested "for" loops. The inner loop, the X loop, allows you to perform 1D data transfers. These are useful for buffer-to-buffer transfers or peripheral-to-memory transfers. To configure the X loop, you need to populate the following descriptor parameters:

- X Loop Data Count
- X Loop Source Address Increment
- X Loop Destination Address Increment

The outer loop, the Y loop, enables what are referred to as 2D transfers, where a single descriptor can perform multiple 1D transfers in succession. The Y loop specifies how many times the X loop will be run. This allows for the transfer of significantly larger amounts of data and more complex data entities i.e. arrays of structs. To configure the Y loop, you need to populate the following descriptor parameters:

- Y Loop Count
- Y Loop Source Address Increment
- Y Loop Destination Address Increment

4.3.5 Descriptor chaining

Descriptor chaining is the same for DMA DW and DMAC. XMC7000 DMA enables descriptor chaining via the "Next Descriptor" descriptor parameter. This parameter is simply a pointer to another descriptor configuration structure. You can easily chain together as many descriptors as you want simply by setting this parameter to the address of the next descriptor you want to add to the chain. Think of this like a linked list. If the "Next Descriptor" parameter of a running descriptor is not `NULL`, upon completion of the running descriptor the DMA channel will automatically make the chained descriptor the active descriptor of the channel. By default, the next descriptor in the chain will not automatically be triggered. To automatically trigger chained descriptors, you must set the "Trigger Type" descriptor parameter in each descriptor in the chain to **Entire Descriptor Chain**. If the last descriptor in the chain is chained to first descriptor in the chain, the last descriptor's trigger mode should be set to something besides **Entire Descriptor Chain**, otherwise the DMA channel will run forever. If you have configured a chained descriptor's "Interrupt Type" parameter to be **Trigger on completion of entire descriptor chain**, the interrupt will be generated when a DMA descriptor is run that is not chained to anything (its "Next Descriptor" parameter is `NULL`).

4.4 Exercises

In the following exercises, we will focus on using the PDL to set up each DMA transfer. The HAL also supports DMA, but because each DMA block is different, we suggest that you use the PDL when you require fine-grained control of your hardware.

All exercises use the KIT_XMC72_EVK BSP.

Exercise 1: 1-1 Transfer

A 1-1 transfer allows for one data element to be transferred from a source to a destination. In this exercise we will transfer data from a UART Rx buffer to a UART Tx buffer so that anything a user types in a serial communication terminal will be echoed back to the terminal.



1. Create a new application called **ch04_ex01_XMC7000_echo** using the template **ch04_ex01_XMC7000_echo**. This template application can be found in *modustoolbox-level2-XMC7x/Templates*.

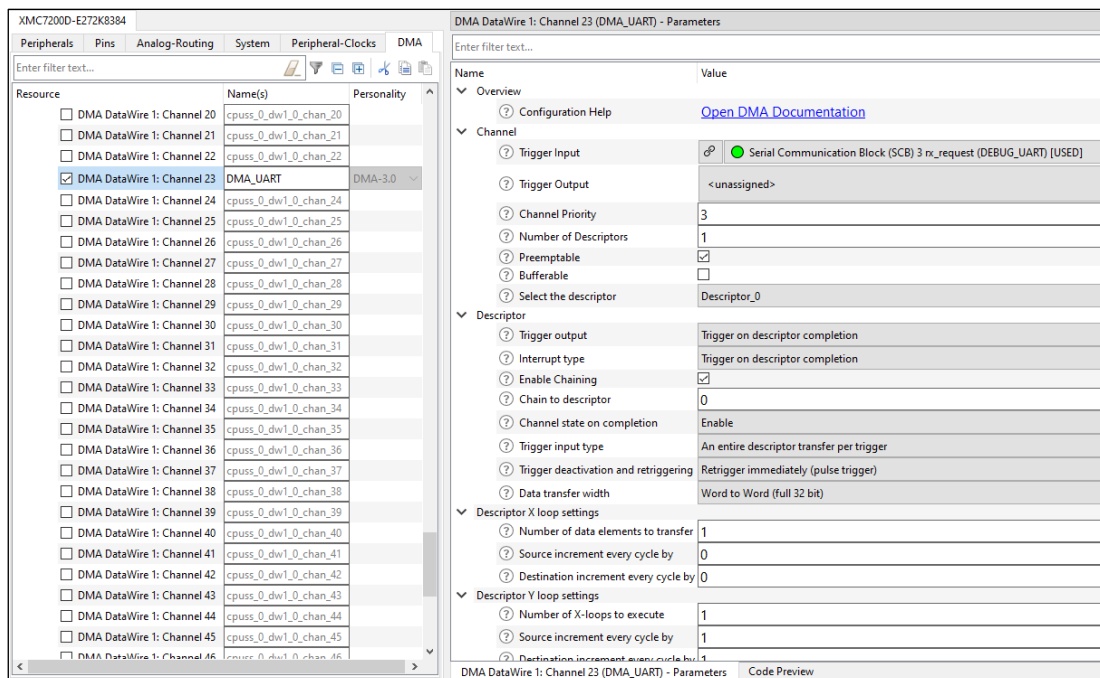
Note: This template comes with SCB3 preconfigured as a debug UART named `DEBUG_UART`. The file `main.c` includes the code to initialize and enable this peripheral so that you can focus on learning DMA.



2. Open the Device Configurator, enable DMA DW 1 Channel 23, name it "DMA_UART", and configure it as follows:

Note: DMA DW 1 Channel 23 must be used because it can connect its trigger input to the debug UART (SCB3) rx request signal on this kit's hardware.

If you know the peripheral you want to connect to but not the DMA block that connects to it, the easiest thing to do is to go to the peripheral configuration and select a DMA block in the trigger input or output list. Since that list only contains possible connections, it will only list DMA blocks that allow the connection that you need.



Name	Value
Overview	
Configuration Help	Open DMA Documentation
Channel	
Trigger Input	Serial Communication Block (SCB) 3 rx_request (DEBUG_UART) [USED]
Trigger Output	<unassigned>
Channel Priority	3
Number of Descriptors	1
Preemptable	<input checked="" type="checkbox"/>
Bufferable	<input type="checkbox"/>
Select the descriptor	Descriptor_0
Descriptor	
Trigger output	Trigger on descriptor completion
Interrupt type	Trigger on descriptor completion
Enable Chaining	<input checked="" type="checkbox"/>
Chain to descriptor	0
Channel state on completion	Enable
Trigger input type	An entire descriptor transfer per trigger
Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
Data transfer width	Word to Word (full 32 bit)
Descriptor X loop settings	
Number of data elements to transfer	1
Source increment every cycle by	0
Destination increment every cycle by	0
Descriptor Y loop settings	
Number of X-loops to execute	1
Source increment every cycle by	1
Destination increment every cycle by	1

Note: *If you do not enable chaining, the `nextDescriptor` descriptor parameter will be set to `NULL` in the code generated by the Device Configurator.*

- ☐ 3. Save this configuration and close the Device Configurator.
- ☐ 4. In your application code initialize and enable the DMA channel.

Note: *To initialize and enable your DMA channel, you will need to call these functions:*

```
Cy_DMA_Descriptor_Init  
Cy_DMA_Descriptor_SetSrcAddress  
Cy_DMA_Descriptor_SetDstAddress  
Cy_DMA_Channel_Init  
Cy_DMA_Channel_SetDescriptor  
Cy_DMA_Enable  
Cy_DMA_Channel_Enable
```

Note: *For examples of how to initialize a DMA channel you can refer to the CAT1 PDL API reference, or to one of the DMA code examples.*

Note: *For the source address and destination address, we want to use the UART RX and TX FIFOs respectively:*

```
Cy_DMA_Descriptor_SetSrcAddress(&UART_DMA_Descriptor_0,  
(void *) &(DEBUG_UART_HW->RX_FIFO_RD));  
  
Cy_DMA_Descriptor_SetDstAddress(&UART_DMA_Descriptor_0,  
(void *) &(DEBUG_UART_HW->TX_FIFO_WR));
```

- ☐ 5. Print something to the UART at startup. For example:

```
/* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */  
Cy_SCB_UART_PutString(DEBUG_UART_HW, "\x1b[2J\x1b[;H");  
Cy_SCB_UART_PutString(DEBUG_UART_HW, "*****\r\n");  
Cy_SCB_UART_PutString(DEBUG_UART_HW, "UART echo using DMA\r\n");  
Cy_SCB_UART_PutString(DEBUG_UART_HW, "*****\r\n\r\n");  
Cy_SCB_UART_PutString(DEBUG_UART_HW, "Start typing to see the echo on the screen \r\n\r\n");
```

- ☐ 6. Program the project to your kit and verify that what you type in the serial terminal is echoed.

Exercise 2: 1-N Transfer

A 1-N data transfer allows for the data from a single source address to be transferred to multiple destination addresses. In this exercise we will transfer characters sent over UART to a memory buffer where they can be reversed by the CPU before printing them.

- ☐ 1. Create a new application called **ch04_ex02_XMC7000_buffer** using the previous completed exercise **ch04_ex01_XMC7000_echo** as the template.
- ☐ 2. Open the Device Configurator and make the following changes to your DMA descriptor's parameters:
 - a. Set the X loop **Number of data elements to transfer** to 5
 - b. Set **Data Transfer Width** to **Word to Byte**
 - c. Set the X loop **Destination increment every cycle** to 1
 - d. Set **Triggering Input Type** to **One transfer per trigger**

Descriptor	
Trigger output	Trigger on descriptor completion
Interrupt type	Trigger on descriptor completion
Enable Chaining	<input checked="" type="checkbox"/>
Chain to descriptor	0
Channel state on completion	Enable
Trigger input type	One transfer per trigger
Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
Data transfer width	Word to Byte
Descriptor X loop settings	
Number of data elements to transfer	5
Source increment every cycle by	0
Destination increment every cycle by	1
CRC	<input type="checkbox"/>

- ☐ 3. Save the configuration and close the Device Configurator. Modify the code in *main.c* so that the DMA channel writes to a 5-byte buffer that you declare instead of writing to the UART TX buffer.

```
char buffer[5];
...
Cy_DMA_Descriptor_SetDstAddress (&UART_DMA_Descriptor_0, (void *) buffer);
```

Note: The buffer should be a global variable so that it can be accessed by the DMA ISR which we will setup next.

- ☐ 4. Add an ISR that will run every time the DMA descriptor finishes. To do this, at the top of *main.c* declare the following macro and function prototype:

```
// Macros
#define DMA_INT_PRIORITY      (3u)

// Function Prototypes
void Isr_DMA(void);
```



5. Then at the bottom of *main.c*, define the function `Isr_DMA`, this will be the ISR that will run every time the DMA descriptor finishes:

```
void Isr_DMA(void)
{
    /* Check if the Dma channel response is successful for current transfer */
    if(!(CY_DMA_INTR_CAUSE_COMPLETION == Cy_DMA_Channel_GetStatus(UART_DMA_HW, UART_DMA_CHANNEL))){
        Cy_SCB_UART_PutString(DEBUG_UART_HW, "DMA Error Occurred. Halting Execution.\r\n");
        CY_ASSERT(0);
    }
    /* Clear Dma channel interrupt */
    Cy_DMA_Channel_ClearInterrupt(UART_DMA_HW, UART_DMA_CHANNEL);
}
```

Note: The ISR provided here checks to make sure no DMA errors occurred during the transfer.



6. In in the DMA ISR, add code so that the contents of the buffer prints in reverse every time the ISR runs. You can use the function `Cy_SCB_UART_PutArray` for this.

Note: The DMA ISR will only run when the entire descriptor has been completed (when the buffer is full).

Note: When the descriptor completes, it's destination address will be reset to what it was originally (the beginning of the buffer).



7. Next, in the DMA initialization section of *main.c*, add code to initialize and enable the DMA Interrupt:

```
/* DMA interrupt initialization structure */
cy_stc_sysint_t DMA_INT_cfg =
{
    .intrSrc      = ((NvicMux0_IRQn << 16) | DMA_UART_IRQ),
    .intrPriority = DMA_INT_PRIORITY,
};
/* Initialize and enable the DMA interrupt */
Cy_SysInt_Init(&DMA_INT_cfg, &Isr_DMA);
NVIC_EnableIRQ((IRQn_Type) NvicMux0_IRQn);

/* Enable interrupt for DMA channel */
```

Note: `Cy_DMA_Channel_SetInterruptMask(UART_DMA_HW, UART_DMA_CHANNEL, CY_DMA_INTR_MASK);`



8. Program the project to your kit and verify that every time you type 5 characters, the characters are echoed in reverse order.

Exercise 3: N-1 Transfer

An N-1 transfer increments the source address while keeping the destination address constant. In this exercise we will use a counter to trigger the transfer of text from two memory buffers into the UART Tx buffer so that the text is printed on a serial communication terminal. We will use descriptor chaining to perform the two distinct transfers.



1. Create a new application called **ch04_ex03_XMC7000_printBuffer** using the template **ch04_ex03_XMC7000_printBuffer**. This template application can be found in *modustoolbox-level2-XMC7x/Templates*.

Note: This template comes with a custom configuration that configures SCB3 as a debug UART named DEBUG_UART. TCPWM 0 Group2 is configured as a timer that will produce a compare signal once every second. The timer is named MY_TIMER. The file main.c includes the code to initialize and enable both these peripherals.



2. Open the Device Configurator. The DMA channel we used in previous exercises is unable to connect its trigger input to any timers, so we will have to choose a different DMA channel. DMA DW 0 Channel 16 can connect to the preconfigured timer, so we will use it.



3. Enable DMA DW 0 Channel 16, name it DMA_UART, and configure it as follows:
 - a. Set the **Number of Descriptors** to 2
 - b. Configure both Descriptors as follows:
 - i. Check the box for **Enable Chaining**
 - ii. Chain each descriptor to the other
 - iii. Set **Trigger input type** to **An entire descriptor transfer per trigger**
 - iv. Set **Data transfer width** to **Byte to Word**
 - v. Set the X loop **Number of data elements to transfer** to 14
 - vi. Set the X loop **Source increment every cycle** to 1
 - vii. Set the X loop **Destination increment every cycle** to 0


▼ Channel	
Trigger Input	<unassigned>
Trigger Output	<unassigned>
Channel Priority	3
Number of Descriptors	2
Preemptable	<input type="checkbox"/>
Bufferable	<input type="checkbox"/>
Select the descriptor	Descriptor_0
▼ Descriptor	
Trigger output	Trigger on every element transfer completion
Interrupt type	Trigger on every element transfer completion
Enable Chaining	<input checked="" type="checkbox"/>
Chain to descriptor	1
Channel state on completion	Enable
Trigger input type	An entire descriptor transfer per trigger
Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
Data transfer width	Byte to Word
▼ Descriptor X loop settings	
Number of data elements to transfer	14
Source increment every cycle by	1
Destination increment every cycle by	0
CRC	<input type="checkbox"/>

② Select the descriptor	Descriptor_1
▼ Descriptor	
② Trigger output	Trigger on every element transfer completion
② Interrupt type	Trigger on every element transfer completion
② Enable Chaining	<input checked="" type="checkbox"/>
② Chain to descriptor	0
② Channel state on completion	Enable
② Trigger input type	An entire descriptor transfer per trigger
② Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
② Data transfer width	Byte to Word
▼ Descriptor X loop settings	
② Number of data elements to transfer	14
② Source increment every cycle by	1
② Destination increment every cycle by	0
② CRC	<input type="checkbox"/>



4. Set the DMA channel's **Trigger Input** to be the compare signal from the timer:

▼ Channel

② Trigger Input	 TCPWM[0] Group[2] 32-bit Counter 0 out 0 (MY_TIMER) [USED]
-----------------	--



5. Save this configuration and close the Device Configurator.



6. In your application code, add the code required to initialize and enable the DMA.

Note: Don't forget to initialize both of your descriptors.



7. In your application code, declare two 14 byte buffers and fill them with two unique messages to print:

```
char buffer0[14] = "Hello World!\r\n";
char buffer1[14] = "DMA is cool!\r\n";
```



8. Configure your DMA descriptors' source addresses to point to the beginning of these buffers:

```
Cy_DMA_Descriptor_SetSrcAddress(&TIMER_DMA_Descriptor_0, (void *) buffer0);
Cy_DMA_Descriptor_SetSrcAddress(&TIMER_DMA_Descriptor_1, (void *) buffer1);
```



9. Configure your DMA descriptors' destination addresses to point to the UART TX FIFO.



10. Program the project to your kit and verify that the two messages you wrote, print once every other second, alternating back and forth.

```
Hello World!
DMA is cool!
Hello World!
DMA is cool!
Hello World!
DMA is cool!
Hello World!
DMA is cool!
Hello World!
DMA is cool!
Hello World!
DMA is cool!
```

Exercise 4: N-N Transfer

An N-N transfer increments both the source and destination addresses. In this exercise we will use DMA to transfer a block of data from external flash memory into RAM, where it will be sent over the UART by the CPU.

- ☐ 1. Create a new application called **ch04_ex04_XMC7000_external** using the previous completed exercise **ch04_ex03_XMC7000_printBuffer** as the template.
- ☐ 2. Open the Device Configurator and make the following changes to your DMA parameters for channel 0:
 - a. Set the **Number of Descriptors** to 1
 - b. Set the **Interrupt Type** to **Trigger on Descriptor Completion**
 - c. Chain descriptor 0 to itself
 - d. Set the **Data transfer width** to **Byte to Byte**
 - e. Set the X loop **Destination increment every cycle by** to 1

Number of Descriptors	1
Preemptable	
Bufferable	
Select the descriptor	Descriptor_0
Descriptor	
Trigger output	Trigger on every element transfer completion
Interrupt type	Trigger on descriptor completion
Enable Chaining	✓
Chain to descriptor	0
Channel state on completion	Enable
Trigger input type	An entire descriptor transfer per trigger
Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
Data transfer width	Byte to Byte
Descriptor X loop settings	
Number of data elements to transfer	14
Source increment every cycle by	1
Destination increment every cycle by	1
CRC	
Descriptor Y loop settings	
Number of X-loops to execute	1
Source increment every cycle by	0
Destination increment every cycle by	0

- ☐ 3. Save this configuration and close the Device Configurator.
- ☐ 4. Remove the code for initializing descriptor 1 and for setting its source/destination addresses from *main.c*.
- ☐ 5. Open the Library Manager and add the serial-flash library to your project:

Name	Shared	Version
core-make	✓	1.8.0 release
Board Utils		
audio-codec-ak4954a		1.0.1 release
audio-codec-wm8960		1.0.0 release
bmi160		3.9.1 release
bmm150		2.0.0 release
CY8CKIT-028-EPD		2.1.0 release
CY8CKIT-028-SENSE		1.0.0 release
CY8CKIT-028-TFT		1.2.0 release
CY8CKIT-032		1.1.0 release
display-eink-e2271cs021		1.1.0 release
display-oled-ssd1306		1.0.1 release
display-tft-st7789v		1.0.1 release
retarget-io		1.2.0 release
rgb-led		1.2.1 release
sensor-light		1.0.1 release
sensor-motion-bmi160		1.1.0 release
sensor-orientation-bmx160		1.0.0 release
sensor-xensiv-dps3xx		1.0.0 release
sensor-xensiv-pasco2		0.6.0 release
serial-flash	✓	1.1.0 release
thermistor		2.0.0 release
whd-bsp-integration		1.2.0 release
BT Middleware libraries		

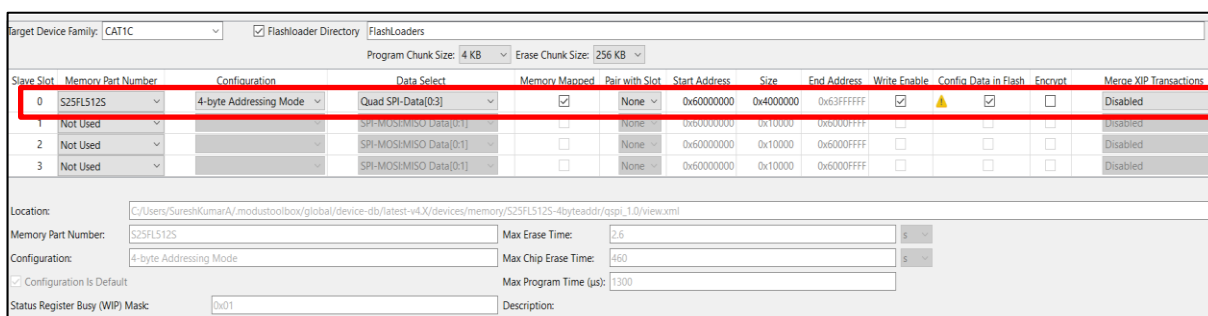


6. In your application code, #include the following header files to get access to the serial flash functions:

```
#include "cy_serial_flash_qspi.h"
#include "cycfg_qspi_memslot.h"
```

Note: The second header file gives you access to the memory configuration structures that are generated by the QSPI Configurator.

Note: Make sure you open qspi-configurator and do the below shown configuration.



Slave Slot	Memory Part Number	Configuration	Data Select	Memory Mapped	Pair with Slot	Start Address	Size	End Address	Write Enable	Config Data in Flash	Encrypt	Merge XIP Transactions
0	S25FL512S	4-byte Addressing Mode	Quad SPI-Data(0:3)	<input checked="" type="checkbox"/>	None	0x60000000	0x4000000	0x63FFFFFF	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Disabled
1	Not Used		SPI-MOSI-MISO Data(0:1)	<input type="checkbox"/>	None	0x60000000	0x10000	0x6000FFFF	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Disabled
2	Not Used		SPI-MOSI-MISO Data(0:1)	<input type="checkbox"/>	None	0x60000000	0x10000	0x6000FFFF	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Disabled
3	Not Used		SPI-MOSI-MISO Data(0:1)	<input type="checkbox"/>	None	0x60000000	0x10000	0x6000FFFF	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Disabled

Location: C:/Users/SureshKumarA/modustoolbox/global/device-db/latest-v4X/devices/memory/S25FL512S-4byteaddr/qspi_1.0/view.xml

Memory Part Number: S25FL512S Max Erase Time: 2.6 s

Configuration: 4-byte Addressing Mode Max Chip Erase Time: 460 s

☒ Configuration Is Default Max Program Time (µs): 1300

Status Register Busy (WIP) Mask: 0x01 Description:



7. In your application code, declare two 14-byte buffers. One to hold the message that will be written to the external flash and one for your DMA channel to transfer this message into:

```
char buffer[14] = "Hello World!\r\n";
char buffer_r[14] = "";
```



8. Remove the buffer0 and buffer1 from the previous exercise.



9. Copy the following code into the top of main.c, these macros and global vars will be needed to set up the external flash.

```
// Macros
#define MEM_SLOT_NUM          (0u) // QSPI configuration slot number
#define QSPI_BUS_FREQUENCY_HZ (50000000u)
#define MESSAGE_LENGTH        (14)

// Gloval Vars
uint32_t startOfFlashGlobal = 0x60000000; // Start of external flash memory in global
address space
uint32_t startOfFlash = 0x00; // Start of flash memory relative to beginning of flash
memory
```



10. Copy the following code into the initialization section of your application (after the UART is initialized). This code will initialize the external flash memory, and then write the contents of `buffer` to the beginning of its address space.

```
// Initialize qspi for external flash memory
result = cy_serial_flash_qspi_init(smifMemConfigs[MEM_SLOT_NUM], CYBSP_QSPI_D0,
CYBSP_QSPI_D1, CYBSP_QSPI_D2, CYBSP_QSPI_D3, NC, NC, NC, NC, CYBSP_QSPI_SCK, CYBSP_QSPI_SS,
QSPI_BUS_FREQUENCY_HZ);
if(result != CY_RSLT_SUCCESS){
    Cy_SCB_UART_PutString(DEBUG_UART_HW, "INIT FAILED\r\n");
    CY_ASSERT(0);
}

// Write buffer to beginning of external flash memory - need to erase first
result = cy_serial_flash_qspi_erase(startOfFlash,
cy_serial_flash_qspi_get_erase_size(startOfFlash));
if(result != CY_RSLT_SUCCESS){
    Cy_SCB_UART_PutString(DEBUG_UART_HW, "ERASE FAILED\r\n");
    CY_ASSERT(0);
}
result = cy_serial_flash_qspi_write(startOfFlash, MESSAGE_LENGTH, (const uint8_t *)buffer);
if(result != CY_RSLT_SUCCESS){
    Cy_SCB_UART_PutString(DEBUG_UART_HW, "WRITE FAILED\r\n");
    CY_ASSERT(0);
}

// Enable Execute in Place (memory mapped) mode - this allows external flash to be read by
DMA
result = cy_serial_flash_qspi_enable_xip(true);
if(result != CY_RSLT_SUCCESS){
    Cy_SCB_UART_PutString(DEBUG_UART_HW, "XIP MODE FAILED\r\n");
    CY_ASSERT(0);
}
```



11. Set your DMA descriptor's source address to the beginning of the external flash – where you wrote the message to. Set the descriptor's destination address to the read buffer you declared earlier:

```
Cy_DMA_Descriptor_SetSrcAddress(&TIMER_DMA_Descriptor_0, (void *)
startOfFlashGlobal);
Cy_DMA_Descriptor_SetDstAddress(&TIMER_DMA_Descriptor_0, (void *) buffer_r);
```

Note: When setting the DMA source address, make sure you use the global address for the beginning of the external flash (0x60000000), not the relative address (0x00).



12. Add a DMA ISR such that every time the ISR is run the contents of the read buffer is printed over the UART connection. You can use the function `Cy_SCB_UART_PutArray` for this.

Note: Refer to Exercise 7, steps 4-6 for the specifics on how to set up a DMA ISR.



13. Program the project to your kit and verify the message you wrote to the external flash prints every second.

Exercise 5: N-NxM Transfer

In this exercise we will use the DMAC X and Y loops to transfer an array of structures.



1. Create a new application called **ch04_ex05_XMC7000_struct** using the template **ch04_ex05_XMC7000_struct**. This template application can be found in *modustoolbox-level2-XMC7x/Templates*.

Take a look at the source files in this application to make sure you understand what is going on. In this exercise we will be transferring an array of structs from the internal flash to RAM using DMA. Once the transfer is complete, the data that was copied into RAM will be printed over the UART. The DMA transfer only takes place once and is triggered by a 1 second timer. To complete the application, you need to:

Note: This template comes with a custom configuration that configures SCB3 as a debug UART named DEBUG_UART. TCPWM 0 Group0 is configured as a timer that will produce a compare signal once every second. The timer is named MY_TIMER. DMAC Channel 0 is configured with its trigger input connected to the timer's compare output. Its triggering mode and interrupt type are set to entire descriptor. It is configured to perform a Byte to Byte transfer with one descriptor and to be disabled upon the completion of this transfer. This DMAC Channel is named MY_DMA. The file main.c includes the code to initialize and enable these peripherals.



2. Open the Device Configurator, navigate to the **DMA** tab and set the following for **DMA Controller > DMA Channel 0** based on the requirements of this application:
 - a. X loop – **Number of data elements to transfer**
 - b. X loop – **Source increment every cycle by**
 - c. X loop – **Destination increment every cycle by**
 - d. Y loop – **Number of X-loops to execute**
 - e. Y loop – **Source increment every cycle by**
 - f. Y loop – **Destination increment every cycle by**

Note: In this exercise we are using the DMAC HW block Channel 0, as the DMAC block is more efficient at transferring large blocks of data than the DMA DW block.

Each struct is 102 bytes long. You can see this in book.h – the title and author are 50 bytes each while the page count is 2 bytes.

After each X loop, the source address is reset, the increments applied during the X loop are undone. In the Y loop you will need to increment both the source and destination by the size of the X elements that were transferred so that each Y loop doesn't overwrite the previous data.

*The **Channel state on completion** is set to **Disable**, so the DMA will only execute one time.*



3. Save your configuration and close the Device Configurator.



4. Open *main.c* and search for the string "TODO". You need to fill in the source and destination addresses for the data transfer.

Note: The arrays of structs are defined at the top of main.c in the global variables section.



5. Program the project to your kit. If you've configured everything correctly you will see the contents of the structs print on your serial terminal. Verify that the values that print are the same as the original values that were copied (at the top of *main.c*)

```
ModusToolbox Level 2 - XMC7000 - Copying an array of structs
*****
Title: Moby Dick
Author: Herman Melville
Page Count: 378

Title: Around the World in 80 Days
Author: Jules Verne
Page Count: 188

Title: The Adventures of Tom Sawyer
Author: Mark Twain
Page Count: 274

Title: 1984
Author: George Orwell
Page Count: 328

Title: Frankenstein
Author: Mary Shelley
Page Count: 280
```

Exercise 6: Descriptor chaining

In this exercise we will modify the N-1 Transfer exercise so that rather than only printing two messages one at a time, it will print four messages all at once every second.

- ☐ 1. Create a new application called **ch04_ex06_XMC7000_descriptorChain** using completed exercise **ch04_ex03_XMC7000_printBuffer** as the template.
- ☐ 2. Open the Device Configurator and make the following changes to your DMA parameters for **DMA DataWire 0: Channel 0**:
 - a. Set the **Number of Descriptors** to 4
 - b. Chain each descriptor to the next, chain descriptor 3 to descriptor 0
 - c. Configure descriptors 2 and 3 in the same way that descriptors 0 and 1 are already configured (see images below)
 - d. Set each descriptor's **Trigger input type** to **Entire descriptor chain per trigger** except for the last descriptor which should be set to **An entire descriptor transfer per trigger**

Note: By setting the first three descriptors to "Entire descriptor chain per trigger", all four descriptors will run in sequence every time the timer expires. This is how we get it to print all four messages at once. The last descriptor is set to "An entire descriptor transfer per trigger" so that the chain stops after the last descriptor. The entire chain re-triggers the next time the timer expires.

Channel	
Trigger Input	TCPWM[0] Group[2] 32-bit Counter 0 out 0 (MY_TIMER) [USED]
Trigger Output	<unassigned>
Channel Priority	3
Number of Descriptors	4
Preemptable	<input type="checkbox"/>
Bufferable	<input type="checkbox"/>
Select the descriptor	Descriptor_0

Descriptor	
Trigger output	Trigger on every element transfer completion
Interrupt type	Trigger on descriptor completion
Enable Chaining	<input checked="" type="checkbox"/>
Chain to descriptor	1
Channel state on completion	Enable
Trigger input type	Entire descriptor chain per trigger
Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
Data transfer width	Byte to Word

Descriptor	
Select the descriptor	Descriptor_1
Trigger output	Trigger on every element transfer completion
Interrupt type	Trigger on descriptor completion
Enable Chaining	<input checked="" type="checkbox"/>
Chain to descriptor	2
Channel state on completion	Enable
Trigger input type	Entire descriptor chain per trigger
Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
Data transfer width	Byte to Word

?	Select the descriptor	Descriptor_2
▼	Descriptor	
?	Trigger output	Trigger on descriptor completion
?	Interrupt type	Trigger on descriptor completion
?	Enable Chaining	<input checked="" type="checkbox"/>
?	Chain to descriptor	3
?	Channel state on completion	Enable
?	Trigger input type	Entire descriptor chain per trigger
?	Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
?	Data transfer width	Byte to Word
▼	Descriptor X loop settings	
?	Number of data elements to transfer	14
?	Source increment every cycle by	1
?	Destination increment every cycle by	0
?	CRC	<input type="checkbox"/>

?	Select the descriptor	Descriptor_3
▼	Descriptor	
?	Trigger output	Trigger on descriptor completion
?	Interrupt type	Trigger on descriptor completion
?	Enable Chaining	<input checked="" type="checkbox"/>
?	Chain to descriptor	0
?	Channel state on completion	Enable
?	Trigger input type	An entire descriptor transfer per trigger
?	Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
?	Data transfer width	Byte to Word
▼	Descriptor X loop settings	
?	Number of data elements to transfer	14
?	Source increment every cycle by	1
?	Destination increment every cycle by	0

- ☐ 3. Save this configuration and close the Device Configurator.
- ☐ 4. In your application code define two new 14-byte buffers to print:


```
char buffer2[14] = "I love XMC!\r\n";
char buffer3[14] = "I'm so smrt!\r\n";
```
- ☐ 5. Initialize your two new descriptors and set their source addresses to the new buffers you declared. Set their destination addresses to the UART Tx buffer.
- ☐ 6. Program the project to your kit and verify that every second, all four messages are printed over the UART.

Exercise 7: Channel chaining

In this exercise we will rework a previous exercise so that it demonstrates the use of channel chaining.

- ☐ 1. Create a new application called **ch04_ex07_XMC7000_channelChain** using completed exercise **ch04_ex03_XMC7000_printBuffer** as the template.
- ☐ 2. Open the Device Configurator and make the following changes to your DMA parameters for **DMA DataWire 0: Channel 0**:
 - a. Set the **Number of Descriptors** to 1
 - b. Set the **Descriptor > Trigger output** to **Trigger on descriptor completion**
 - c. Chain descriptor 0 to itself

Number of Descriptors	1
Preemptable	<input type="checkbox"/>
Bufferable	<input type="checkbox"/>
Select the descriptor	Descriptor_0
▼ Descriptor	
Trigger output	Trigger on descriptor completion
Interrupt type	Trigger on descriptor completion
Enable Chaining	<input checked="" type="checkbox"/>
Chain to descriptor	0

- ☐ 3. Enable **DMA DataWire 0: Channel 1** and configure it as follows:
 - a. Name the block "UART_DMA".
 - b. Set the **Channel > Trigger Input** to connect to DMA DataWire 0 Channel 0's trigger output
 - c. Set the **Number of Descriptors** to 1
 - d. Chain the descriptor to itself
 - e. Set the **Trigger input type** to **An entire descriptor transfer per trigger**
 - f. Set the **Data transfer width** to **Byte to Word**
 - g. Set the X loop **Number of data elements to transfer** to 14
 - h. Set the X loop **Source increment every cycle by** to 1
 - i. Set the X loop **Destination increment every cycle by** to 0

Name	Value
Peripheral Documentation	
CRC	
Channel	
Trigger Input	DMA DataWire 0: Channel 0 tr_out (TIMER_DMA) [USED]
Trigger Output	<unassigned>
Channel Priority	3
Number of Descriptors	1
Preemptable	<input type="checkbox"/>
Bufferable	<input type="checkbox"/>
Select the descriptor	Descriptor_0
▼ Descriptor	
Trigger output	Trigger on every element transfer completion
Interrupt type	Trigger on every element transfer completion
Enable Chaining	<input checked="" type="checkbox"/>
Chain to descriptor	0
Channel state on completion	Enable
Trigger input type	An entire descriptor transfer per trigger
Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
Data transfer width	Byte to Word
▼ Descriptor X loop settings	
Number of data elements to transfer	14
Source increment every cycle by	1
Destination increment every cycle by	0
CRC	<input type="checkbox"/>

-
- ☐ 4. Save this configuration and close the Device Configurator.
 - ☐ 5. Remove the code for initializing and setting source/destinations addresses for descriptor 1 from *main.c*.
 - ☐ 6. Add code to initialize the new DMA channel and its descriptor. Set the new DMA channel's descriptor's source address to the second buffer `buffer1`. Set its destination address to the UART TX buffer.
 - ☐ 7. Program the project to your kit and verify that every second both messages are printed over the UART.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2022 Infineon Technologies AG.
All Rights Reserved.

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.