Chapter 5:   Direct Memory Access (DMA)

After completing this chapter, you will understand the capabilities and some use cases of the PSoC™ DMA hardware (HW).

Table of contents

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**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**.

5.1   Overview

DMA is a feature of PSoC™ devices that allows you to access memory independently of the CPU. With DMA you can read from any memory connected to the PSoC™. This includes the registers of peripherals, RAM, internal flash, and even external memories. However, you can only write to the registers of peripherals and RAM; you cannot write to flash (internal or external) using DMA.

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 three types of the DMA HW block that we will discuss in this chapter (one type in PSoC™ 4 and two types in PSoC™ 6). 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.

5.1.1   Architecture

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
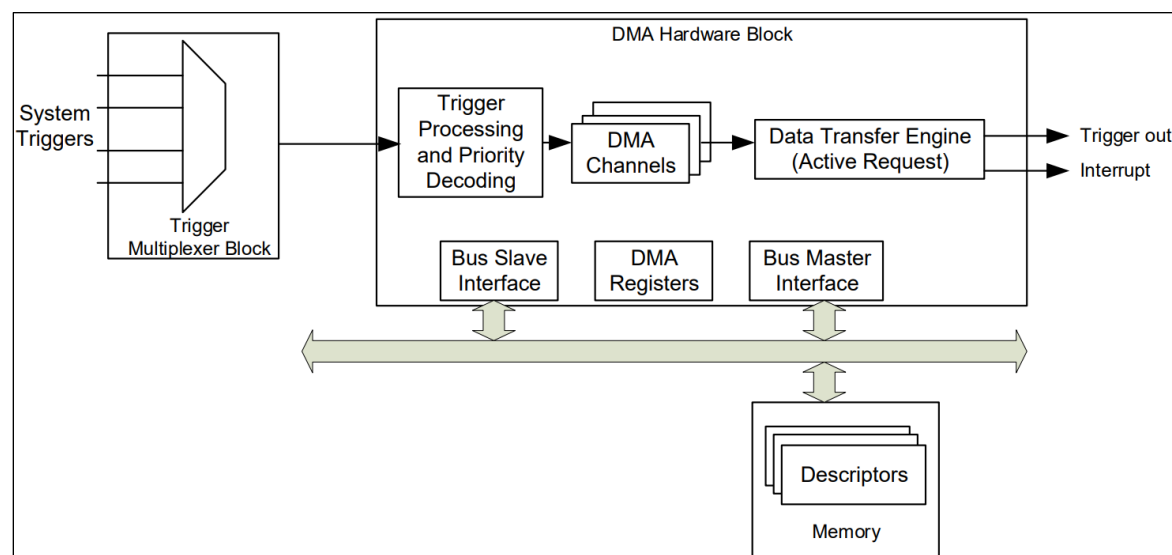
5.1.2   Priorities and preemption

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 "preemptable". This parameter allows a higher-priority channel to preempt the marked channel when it is active. If a low-priority preemptable 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 preemptable channel can be preempted multiple times during a single transfer.

DMA Channel Arbitration and Preemption Example
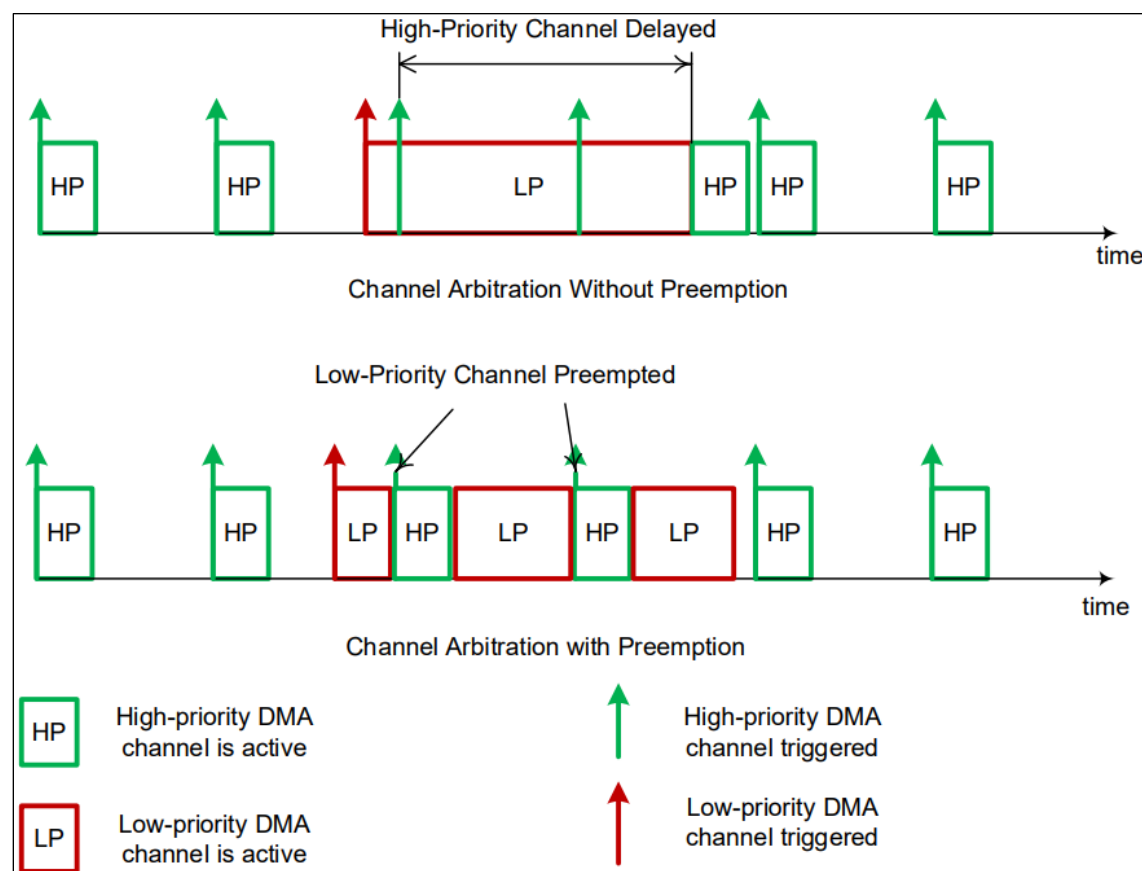
5.1.3 Data transfer widths

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 PSoC™ 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.

DATA_SIZE	SCR_TRANSFER_SIZE	DST_TRANSFER_SIZE	Typical Usage	Description
8-bit	8-bit	8-bit	Memory to Memory	No data manipulation
8-bit	32-bit	8-bit	Peripheral to Memory	Higher 24 bits from the source dropped
8-bit	8-bit	32-bit	Memory to Peripheral	Higher 24 bits zero padded at destination
8-bit	32-bit	32-bit	Peripheral to Peripheral	Higher 24 bits from the source dropped and higher 24 bits zero padded at destination
16-bit	16-bit	16-bit	Memory to Memory	No data manipulation
16-bit	32-bit	16-bit	Peripheral to Memory	Higher 16 bits from the source dropped
16-bit	16-bit	32-bit	Memory to Peripheral	Higher 16 bits zero padded at destination
16-bit	32-bit	32-bit	Peripheral to Peripheral	Higher 16 bits from the source dropped and higher 16-bit zero padded at destination
32-bit	32-bit	32-bit	Peripheral to Peripheral	No data manipulation

The data transfer widths you select must be a width supported by the peripheral or memory being accessed. For example: all PSoC™ peripherals support only a 32-bit data width, so whenever you are using DMA to read from or write to a peripheral you must select a 32-bit data width. PSoC™ RAM and flash memories support all of the data widths previously mentioned: 32-bit, 16-bit, and 8-bit.

5.1.4 Chaining descriptors/channels

All PSoC™ 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.

A good use case for descriptor chaining is double buffering (a.k.a. ping-pong):
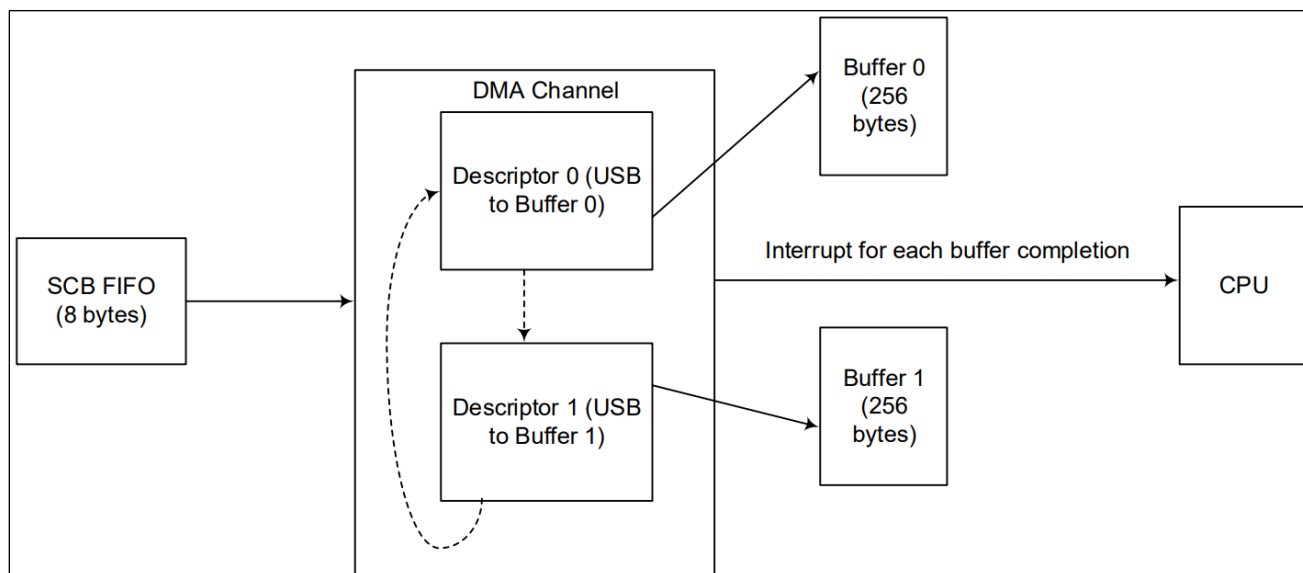
In this example the input data comes in 8-byte blocks in the SCB FIFO, which need to processed by the CPU. Two buffers are used so that one buffer can be filled by the SCB FIFO while the CPU processes data from the other buffer. The buffers are 256 bytes each and thus can accommodate 32 FIFO's worth 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 PSoC 6 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 2 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 devices 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.
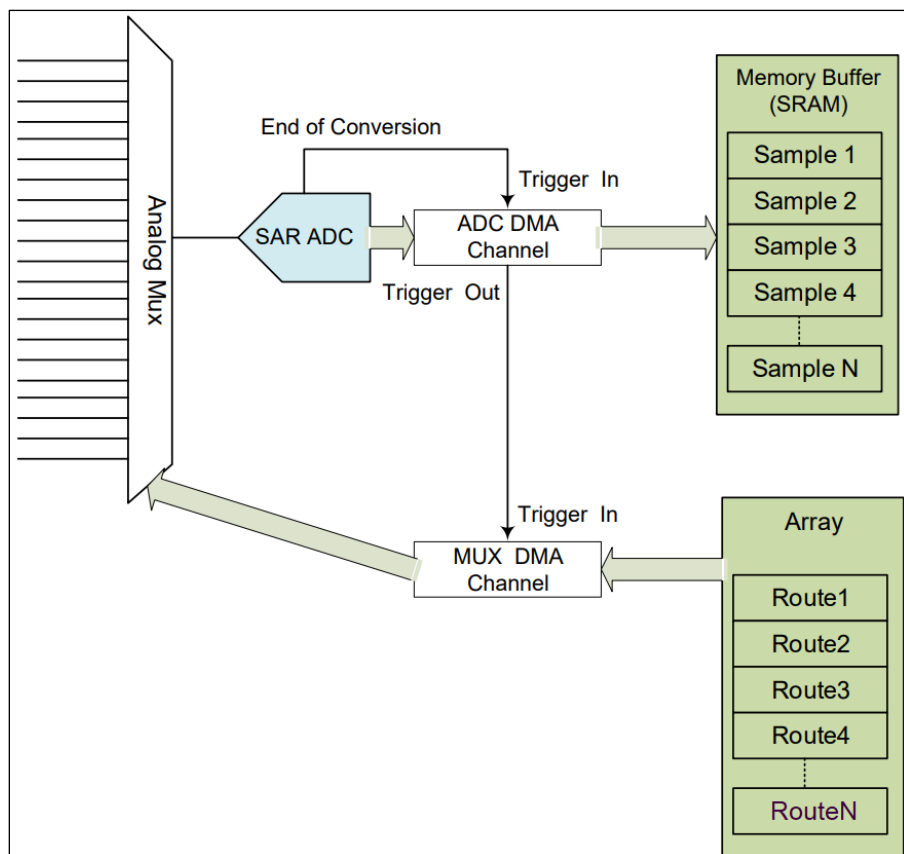
Here the input to the ADC is a large analog multiplexer. Because the number inputs to the mux is so large, indexing the mux it is not supported by the inbuilt multiplexer in the SAR ADC HW. Instead this analog mux has a routing register that different values can be written into to select which input 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.

5.2 DMA Configuration

To setup DMA on your device, you should 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 dropdown menu of all of that block's channels:

CY8CKIT-149	CY8CKIT-062S2-43012

Notice on the PSoC™ 6 kit, there are separate dropdowns for the DMAC block and the DMA DW blocks.

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:

CY8CKIT-149

CY8CKIT-062S2-43012

The PSoC™ 4 DMA channels can only have two descriptors at a time and all of the parameters for these two descriptors can be configured under the "Descriptor Ping" and "Descriptor Pong" headers:

In contrast, the PSoC™ 6 DMA channels are able to 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, you need to select the descriptor you want to configure in the **Select the descriptor** dropdown menu. Then the parameters for the selected descriptor will be available for configuration under the "Descriptor…" headers:

Note: The screenshots here are from the PSoC™ 6 DMAC HW block channel settings, but the DMA DW HW block channel settings work the same way.

5.3 PSoC™ 4 DMA

PSoC™ 4 devices have one type of DMA HW block that supports all of the features previously discussed. In this section we discuss the functionalities and capabilities of this block. For a more detailed look into the PSoC™ 4 DMA specs you can refer to the following resources:

- [PSoC™ 4 DMA Component Datasheet](#)
- The DMA sections of [PSoC™ 4 TRM](#)

PSoC™ 4 DMA Features

- The PSoC™ 4 DMA channels support four priority levels (0-3) and preemption. Lower priority numbers correspond to higher priorities.
- The PSoC™ 4 DMA block supports 32-bit (word), 16-bit (half word), and 8-bit (byte) reads and writes.
- The PSoC™ 4 DMA block supports descriptor chaining, but the way descriptors are implemented is slightly different than the other DMA HW blocks discussed in this chapter. We will discuss this more later.
- The PSoC™ 4 DMA block supports channel chaining by connecting the trigger output of one channel to the trigger input of another channel. Only certain channels have the ability to chain. To find out which channels support this on your device, refer to your device's documentation.

5.3.1 Configuration

The PSoC™ 4 DMA HW block implements descriptors and channels in a slightly different way than the other DMA HW blocks discussed in this chapter. Rather than being able to associate many descriptors with each channel, the PSoC™ 4 DMA channels can only have two associated descriptors at a time. Only one of the descriptors is active at any given time though; the other is inactive. In the ModusToolbox™ software, these two descriptors are referred to as "Descriptor Ping" and "Descriptor Pong". By default, "Ping" is initially the channel's active descriptor and will be the descriptor that runs when the channel is triggered. However, after the "Ping" descriptor has run, you can optionally "flip" descriptors so that the "Pong" descriptor will be the channel's active descriptor instead of "Ping". This is how the PSoC™ 4 DMA block implements descriptor chaining. We will discuss this more in the [PSoC™ 4 Descriptor Chaining](#) section of this chapter. In addition, the PSoC™ 4 DMA channels also have a slightly different set of configuration parameters when compared to the other descriptors discussed in this chapter.

To utilize DMA on PSoC™ 4, 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
- Active Descriptor – The channel's active descriptor upon initialization. (Ping or Pong)

Descriptor Level Parameters

- Data Count – The number of data elements that will be transferred by this descriptor. This can be a maximum of 65,536. (The size of a data element is determined by source transfer size)

- 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.

- Source Address Increment – Whether or not to increment the source address after every read. Unlike the PSoC™ 6 DMA blocks, the PSoC™ 4 DMA block can only increment by one data element after every read. (Here, the size of a data element is determined by source transfer width)

- Destination Address Increment – Whether or not to increment the destination address after every write. Unlike the PSoC™ 6 DMA blocks, the PSoC™ 4 DMA block can only increment by one data element after every write. (Here, the size of a data element is determined by destination transfer size)

- Trigger Deactivation and Retriggering – How long to wait before reactivating the DMA channel's input trigger.

- Invalidate on Completion – Whether or not to mark the descriptor invalid after it completes its transfer

- Interrupt on Completion – Whether or not to send an interrupt when the descriptor completes its transfer

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

- Flipping – Whether or not to make the channel's other descriptor active on completion of this descriptor (i.e. ping-pong)

- Trigger Type – The number of data elements to transfer per trigger. The options for this are:

  - Single Element – Only a single element is transferred

  - Single Descriptor – The entire descriptor will be run

  - Descriptor List (requires flipping to be enabled) – The current descriptor is run to completion, then the descriptor that is flipped to is automatically triggered and run to completion as well

- Source Address – The memory address to read data elements from. This parameter cannot be configured via the Device Configurator, you must configure it in your application code.

- Destination Address – The memory address to write data elements to. This parameter cannot be configured via the Device Configurator, you must configure it in your application code.

5.3.2   Transfer modes

The PSoC™ 4 DMA blocks support three transfer modes:

- Single data element – Every time the descriptor is triggered it will transfer only one data element. For example, if the descriptor's "Data Count" parameter is set to 10 data elements, it would need to be triggered 10 times before it completed.

- All data elements – Every time the descriptor is triggered it will transfer all of the data elements it is set up to transfer. For example, if the descriptor's "Data Count" parameter is set to 10 data elements, it will transfer all 10 elements and complete after being triggered once.

- All data elements and automatically trigger chained descriptor – Every time the descriptor is triggered it will run to completion, then the descriptor that is flipped to will automatically be triggered and run to completion as well. This will continue until a descriptor who's trigger type is not set to "Descriptor List" is run.

Note: *Each of these transfer modes corresponds to one of the options you can set the descriptor parameter "Trigger Mode" to.*

5.3.3　　　Descriptor chaining

As mentioned above, the PSoC™ 4 DMA HW block implements descriptor chaining via "flipping". When flipping is enabled, the DMA channel switches which of its two descriptors is active after the flipping descriptor is run. Enabling flipping only causes the active descriptor to be switched; it does not automatically trigger the descriptors that is flipped to. To automatically trigger descriptors that are flipped to, you must set the descriptor parameter "Trigger Type" to "Descriptor List". This will cause the next descriptor to be triggered automatically. If this option is not enabled, the channel will wait until it is triggered again to run the descriptor that was flipped to.

To chain more than two descriptors together, in addition to enabling flipping and setting the trigger type to "Descriptor List" on both of a channel's descriptors, you need to update one of the channel's descriptors, while it is not being run. To enable this, you can set the "Invalidate on Completion" descriptor parameter which causes the HW to reset the "valid" bit when the descriptor's transfer is completed. With this enabled, once the descriptor has run the HW will mark it invalid, at which point you can then overwrite it with the next descriptor you want to run. You can check whether a descriptor has been marked invalid from your application firmware by calling the PDL function `Cy_DMAC_Descriptor_GetState`. Then, to overwrite the descriptors parameters you use the provided PDL functions. There is a function to overwrite each of the descriptors parameters. Some examples are:

- `Cy_DMAC_Descriptor_SetSrcAddress`
- `Cy_DMAC_Descriptor_SetDataCount`
- `Cy_DMAC_Descriptor_SetDstIncrement`
- `Cy_DMAC_Descriptor_SetFlipping`
- `Cy_DMAC_Descriptor_SetTriggerType`
- `Cy_DMAC_Descriptor_SetPreemptable`

For a complete list of the DMA functions provided by the PDL, as well as a plethora of use examples, refer to the PDL documentation.

Note:　　　　A descriptor can only be overwritten when it is marked as "invalid".

After you have updated the descriptor's parameters, be sure to re-validate the descriptor. This is done using the `Cy_DMAC_Descriptor_SetState` function. If the descriptor is still marked invalid when it is triggered, it will not be run and the channel it is associated with will be disabled.

By updating descriptors like this, you can chain together as many descriptors as you need. The following is an example for chaining together 4 distinct descriptors referred to as "descriptor 1-4". In this example each of the descriptors has the flipping parameter enabled. Descriptors 1-3 have the trigger type parameter set to "Descriptor List" while descriptor 4 has the trigger type parameter set to "Single Descriptor".

1. Set "Descriptor Ping" to descriptor 1 and "Descriptor Pong" to descriptor 2

2. The channel is triggered, causing Ping-descriptor 1 to run

3. Ping-descriptor 1 completes, Pong-descriptor 2 is flipped to and automatically triggered

4. Before Pong-descriptor 2 completes, descriptor Ping is overwritten to descriptor 3

5. Pong-descriptor 2 completes, Ping-descriptor 3 is flipped to and automatically triggered

6. Before Ping-descriptor 3 completes, descriptor Pong is overwritten to descriptor 4

7. Ping-descriptor 3 completes, Pong-descriptor 4 is flipped to and automatically triggered

8. Before Pong-descriptor 4 completes, descriptor Ping is overwritten to descriptor 1

9. Pong-descriptor 4 completes, Ping-descriptor 1 is flipped to and waits to be triggered

10. Ping-descriptor 1 is triggered, but before it completes, descriptor Pong is overwritten to descriptor 2

11. Ping-descriptor 1 completes, Pong-descriptor 2 is flipped to and waits to be triggered

12. Repeat steps 4-11

5.4 PSoC™ 6 DMA

PSoC™ 6 devices have two types of DMA HW blocks: DMA DW (DMA Data Wire) and DMAC (DMA Controller). Both of these blocks support all of the features discussed in the [Overview](#) section of this chapter. In this section we will discuss the functionality and capabilities of both of these blocks. For a more detailed look into the PSoC™ 6 DMA specs you can refer to the following resources:

- [DMA on PSoC™ 6 MCU](#)

- The DMA sections of [PSoC™ 6 TRM](#)

PSoC™ 6 DMA Features

- The PSoC™ 6 DMA channels supports four priority levels (0-3) and preemption. Lower priority numbers correspond to higher priorities.

- The PSoC™ 6 DMA blocks support 32-bit (word), 16-bit (half word), and 8-bit (byte) reads and writes.

- The PSoC™ 6 DMA blocks support descriptor chaining.

- The PSoC™ 6 DMA blocks support channel chaining by connecting the trigger output of one channel to the trigger input of another channel. Only certain channels have the ability to chain. To find out which channels support this on your device, refer to your device's documentation.

- The PSoC™ 6 DMA DW block supports cyclic redundancy checks.

DMA DW

The DMA DW HW block offers a large number of channels meant for small data transfers, typically between peripherals, or between peripherals and memory. This block also has the ability to perform cyclic redundancy checks (CRCs).

DMAC

The DMAC HW Block offers a relatively small number of channels meant for large data transfers, typically from memory to memory. In general, the DMAC provides higher performance compared to DMA DW when transferring large blocks of data.

Which Block to Choose?

The exact performance of each HW block depends on several factors such as data count, transfer width, transfer mode, trigger scheme, preemption, and bus arbitration, just to name a few. The majority of these factors will be application specific, so in order to pick the most efficient block for your needs, you will have to analyze each of these factors in regard to your own situation. Sections 9, 10, and 11 of this document: [DMA on PSoC™ 6 MCU](#) contain a detailed discussion of all the factors relating to DMA performance that you should consider when deciding which block to use.

5.4.1 Configuration

The descriptor settings are almost identical for DMA DW and DMAC. Unlike the PSoC™ 4 DMA channels, each PSoC™ 6 DMA channel can have many descriptors associated with it at a time. This is enabled by a descriptor parameter that is a pointer to another descriptor. This is how descriptor chaining is implemented in PSoC™ 6 DMA. We will discuss this in more detail in the [PSoC™ 6 Descriptor Chaining](#) section of this chapter. The PSoC™ 6 DMA transfer modes differ vastly from the PSoC™ 4 DMA transfer modes. Rather than simply having a

number of data elements that will be transferred for each descriptor, PSoC™ 6 DMA descriptors allow you to perform 2D loop transfers. We will discuss this more in the PSoC™ 6 Transfer Modes section of this chapter. Because of these architectural differences between the PSoC™ 4 and PSoC™ 6 DMA HW blocks, the PSoC™ 6 DMA descriptors have a mostly different set of configuration parameters.

To utilize DMA on PSoC™ 4, 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

- <u>Trigger Deactivation and Retriggering</u> – How long to wait before reactivating the DMA channel's input trigger.
- <u>Data Prefetch (DMAC Only)</u> – Whether or not to prefetch data. The prefetch occurs as soon as the channel is enabled.
- <u>Data Transfer Width</u> – 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.
- <u>X Loop Data Count</u> – The number of data elements to transfer per X loop.
- <u>X Loop Source Address Increment</u> – 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.
- <u>X Loop Destination Address Increment</u> – 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.
- <u>CRC (DMA DW Only)</u> – Whether or not this descriptor is a CRC transfer. Other CRC configuration parameters are discussed in the <u>PSoC™ 6 Transfer Modes</u> section of this chapter.
- <u>Scatter Transfer (DMAC Only)</u> – Whether or not this descriptor is a scatter transfer.
- <u>Y Loop Count</u> – The number of X loops to run per Y loop.
- <u>Y Loop Source Address Increment</u> – The number of data elements to increment the source address after every completion of an X loop.
- <u>Y Loop Destination Address Increment</u> – The number of data elements to increment the destination address by after every completion of an X loop.

5.4.2   Transfer modes

The PSoC™ 6 DMA blocks support the following transfer modes. Unless otherwise stated, the transfer modes apply to both DMA DW and DMAC:

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

- <u>1D Transfer (X loop)</u> –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.
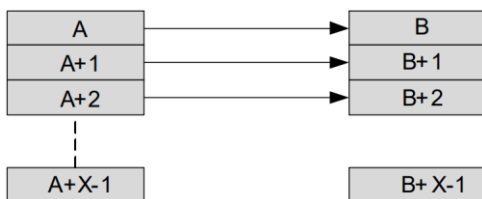
- 2D Transfer (Y loop) –This allows for multiple 1D transfers to be defined in a single descriptor. You can choose to trigger the transfers one data element at a time, one X loop at a time, or you can trigger the entire 2D transfer at once.
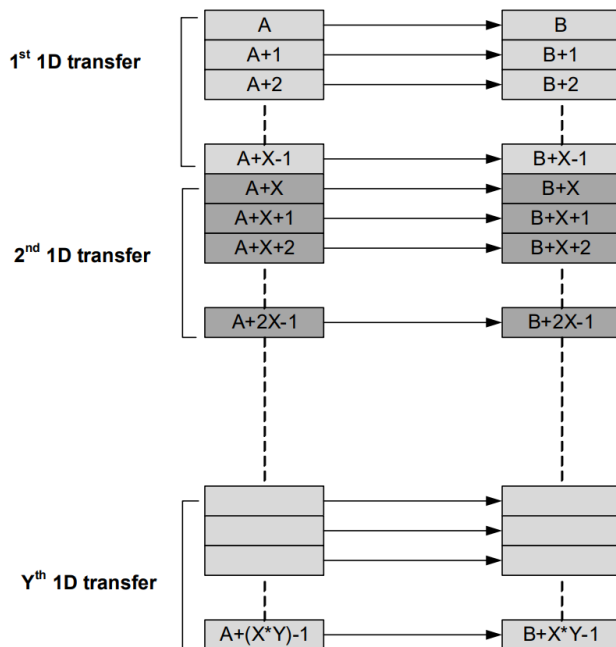
- CRC Transfer (DMA DW only) – This transfer mode does not actually transfer data. Instead a CRC is calculated over the source data, and is written to the destination address.

- Memory Copy (DMAC only) – A special case of the 1D transfer mode where the source and destination address increments are implicitly set to 1. For more information on this mode refer to this Technical Reference Manual (TRM): PSoC™ 6 TRM

- 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 PSoC™ 6 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

CRC (DMA DW only)

Cyclic redundancy check (CRC) is a type of error-detecting algorithm that the DMA DW HW block can perform. You can think of this like a checksum that is calculated based on the source data that the DMA block reads in a given transfer. CRC transfers do not actually transfer any data, instead they only calculate a CRC and write the calculated value to the destination address. CRC transfers can only be performed on 1D blocks of data. The DMA block reads each of the data elements specified by the X loop, calculates a CRC for them, and then writes the CRC to the destination address. The calculated CRC is for <u>all</u> of the data elements read in a given X loop, not for each individual data element. There is only one CRC calculated per CRC X loop. To perform a CRC transfer you need to specify the following CRC parameters:

- Data Reverse – This allows for bit reversal of the data bytes, providing support for serial interfaces that transfer bytes in both little-endian and big-endian order.
- Data XOR – Specifies a byte pattern with which each data byte is XOR'd. This allows for inversion of the data byte value.
- Reminder Reverse – Allows for bit reversal of the XOR'd data bytes
- Reminder XOR – Specifies a 32-bit pattern with which the CRC result will be XOR'd
- Polynomial – Specifies the polynomial to create the CRC with. The polynomial specification omits the high order bit and should be left aligned. For example, popular 32-bit and 16-bit CRC polynomials are specified as follows:
    - CRC32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
        - POLYNOMIAL[31:0] = 0x04c11db7
    - CRC16-CCITT: $x^{16} + x^{12} + x^5 + 1$
        - POLYNOMIAL[31:0] = (0x1021 << 16)
    - CRC16: $x^{16} + x^{15} + x^2 + 1$
        - POLYNOMIAL[31:0] = (0x8005 << 16)

For more information on CRC you can refer to the following resources:

- https://en.wikipedia.org/wiki/Cyclic_redundancy_check
- CRC section of PSoC™ 6 TRM

5.4.3   Descriptor chaining

Descriptor chaining is the same for DMA DW and DMAC. PSoC™ 6 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).

5.5   PSoC™ 4 Exercises

In the exercises below, 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 it is suggested that you use the PDL when you require fine grained control of your hardware.

All PSoC™ 4 exercises use the CY8CKIT-149.

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 **ch05_ex01_PSoC4_echo** using the template **ch05_ex01_PSoC4_echo**. This template application can be found in *modustoolbox-level2-PSoC/Templates*.

Note: *This template comes with a custom device configuration in COMPONENT_CUSTOM_DESIGN_MODUS 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, go to the **DMA** tab, enable DMA channel 0, name it "UART_DMA", and configure it as follows:
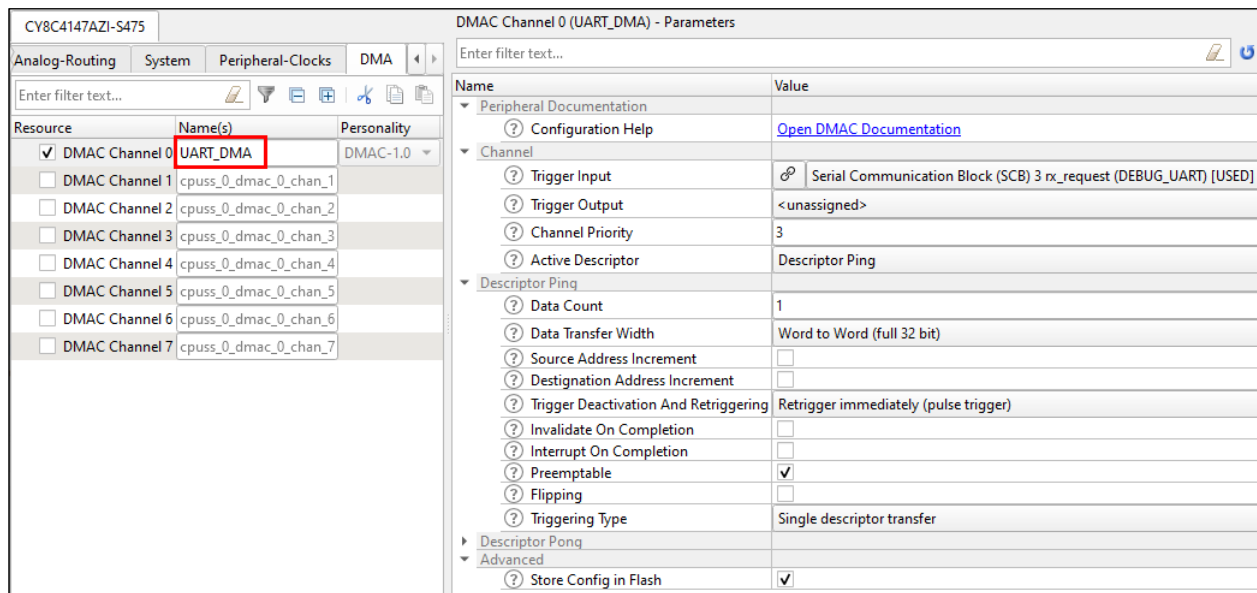
Note: *In this exercise we will only use the ping descriptor, so you can ignore the pong settings.*

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_DMAC_Descriptor_Init
Cy_DMAC_Descriptor_SetSrcAddress
Cy_DMAC_Descriptor_SetDstAddress
Cy_DMAC_Descriptor_SetState – To validate the descriptor
Cy_DMAC_Channel_Init
Cy_DMAC_Channel_SetCurrentDescriptor
Cy_DMAC_Enable
Cy_DMAC_Channel_Enable
```

Note: Refer to the CAT2 PDL API reference, or to one of the DMA code examples for additional details.

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

```
Cy_DMAC_Descriptor_SetSrcAddress(UART_DMA_HW, UART_DMA_CHANNEL,
CY_DMAC_DESCRIPTOR_PING, (void *) &(DEBUG_UART_HW->RX_FIFO_RD));

Cy_DMAC_Descriptor_SetDstAddress(UART_DMA_HW, UART_DMA_CHANNEL,
CY_DMAC_DESCRIPTOR_PING, (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\n");
Cy_SCB_UART_PutString(DEBUG_UART_HW, "Start typing to see the echo on the screen \r\n\n");
```

6. Leave the `for` loop empty. The CPU won't be doing anything in this exercise – it is completely free for other tasks.
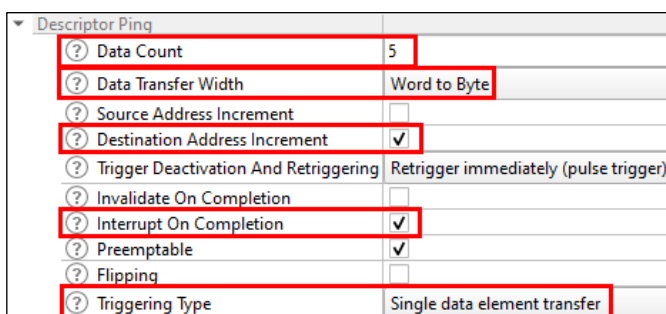
7. Program the project to your kit and verify that what you type in the serial terminal is echoed.

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 5 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 **ch05_ex02_PSoC4_buffer** using the previous completed exercise **ch05_ex01_PSoC4_echo** as the template.

2. Open the Device Configurator and make the following changes to your DMA descriptor's parameters:
    a. Set **Data Count** to 5
    b. Set **Data Transfer Width** to **Word to Byte**
    c. Enable **Destination Address Increment**
    d. Enable **Interrupt On Completion**
    e. Set **Triggering Type** to **Single data element transfer**

    Note: *Again, we will only be using descriptor ping, so you can ignore the pong settings.*

3. Save this 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_DMAC_Descriptor_SetDstAddress(UART_DMA_HW, UART_DMA_CHANNEL,
    CY_DMAC_DESCRIPTOR_PING, (void *) buffer);
    ```

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 macros and function prototype:

    ```
    // Macros
    #define DMA_IRQ             (cpuss_interrupt_dma_IRQn)
    #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_DMAC_DONE == Cy_DMAC_Descriptor_GetResponse(UART_DMA_HW, UART_DMA_CHANNEL,
CY_DMAC_DESCRIPTOR_PING)))
        {
                Cy_SCB_UART_PutString(DEBUG_UART_HW, "DMA Error Occurred. Halting Execution.\r\n");
                CY_ASSERT(0);
        }

        /* Clear Dma channel interrupt */
        Cy_DMAC_ClearInterrupt(UART_DMA_HW, CY_DMAC_INTR_CHAN_0);
}
```

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

☐ 6. 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      = (IRQn_Type)DMA_IRQ,
        .intrPriority = DMA_INT_PRIORITY,
};

/* Initialize and enable the DMA interrupt */
Cy_SysInt_Init(&DMA_INT_cfg, &Isr_DMA);
NVIC_EnableIRQ(DMA_INT_cfg.intrSrc);

/* Enable interrupt for DMA channel */
Cy_DMAC_SetInterruptMask(UART_DMA_HW,  CY_DMAC_INTR_CHAN_0);
```

☐ 7. Finally, 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).*

☐ 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 with flipping

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 flipping in order to print the text from both buffers.

1. Create a new application called **ch05_ex03_PSoC4_printBuffer** using the template **ch05_ex03_PSoC4_printBuffer**. This template application can be found in *modustoolbox-level2-PSoC/Templates*.

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

2. Open the Device Configurator, enable DMAC Channel 0, and make the following changes to your DMA descriptor Ping's parameters:

   a. Set **Data Count** to 14

   b. Set **Data Transfer Width** to **Byte to Word**

   c. Enable **Source Address Increment**

   d. Enable **Flipping**

   e. Set **Triggering Type** to **Single Descriptor Transfer**

3. Make the following changes to your DMA descriptor Pong's parameters:

   a. Set **Data Count** to 14

   b. Set **Data Transfer Width** to **Byte to Word**

   c. Enable **Source Address Increment**

   d. Enable **Flipping**

   e. Set **Triggering Type** to **Single Descriptor Transfer**
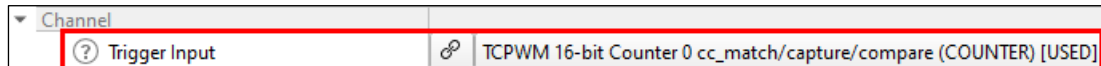
4. Set the DMA channel's **Trigger Input** to be the compare signal from the counter:

Channel		
⑦ Trigger Input	🔗	TCPWM 16-bit Counter 0 cc_match/capture/compare (COUNTER) [USED]

5. Save this configuration and close the Device Configurator. In your application code, add the code required to initialize and enable the DMA block and descriptors.

    Note: *Don't forget to initialize <u>both</u> the ping and pong DMA descriptors.*

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

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

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

```
Cy_DMAC_Descriptor_SetSrcAddress(UART_DMA_HW, UART_DMA_CHANNEL,
CY_DMAC_DESCRIPTOR_PING, (void *) pingBuffer);
Cy_DMAC_Descriptor_SetSrcAddress(UART_DMA_HW, UART_DMA_CHANNEL,
CY_DMAC_DESCRIPTOR_PONG, (void *) pongBuffer);
```

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

9. Program the project to your kit and verify that the two messages 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: Descriptor chaining

In this exercise we will modify the previous exercise so that rather than only printing two messages, it will print four messages.

1. Create a new application called **ch05_ex04_PSoC4_descriptorChain** using the previous completed exercise **ch05_ex03_PSoC4_printBuffer** as the template.

2. Declare two more 14-byte buffers with unique messages stored in them:

```
char pingBuffer1[14] = "I love PSoC!\r\n";
char pongBuffer1[14] = "I'm so smrt!\r\n";
```

3. Add a DMA ISR such that every time it is triggered, it determines which descriptor just finished and then updates that descriptor's source address to point to the appropriate buffer.

The ping descriptor's source address should alternate between `pingBuffer` and `pingBuffer1`, while the pong descriptor's source address should alternate between `pongBuffer` and `pongBuffer1`.

Note: *You can determine which descriptor finished by calling the function `Cy_DMAC_Descriptor_GetResponse`.*

Note: *Be sure to invalidate the descriptors before modifying them and to revalidate them before they run. This can be done using the function `Cy_DMAC_Descriptor_SetState`.*

Note: *Call the function `Cy_DMAC_Descriptor_GetSrcAddress` to determine what a descriptor's source address currently is. Call the function `Cy_DMAC_Descriptor_SetSrcAddress` to change the descriptor's source address.*

4. Program the project to your kit and verify that the four messages you wrote alternate printing one at a time.

Exercise 5: Channel chaining

In this exercise we will use DMA channel chaining to echo characters received over a UART. This does the same thing as exercise 1 but in this case, it demonstrates the use of channel chaining by copying the Rx FIFO data into an intermediate buffer in memory which is then copied to the Tx FIFO.

1. Create a new application called **ch05_ex05_PSoC4_channelChain** using the template **ch05_ex05_PSoC4_channelChain**. This template application can be found in *modustoolbox-level2-PSoC/Templates*.

Note: *This template comes with a custom configuration that configures SCB3 as a debug UART named DEBUG_UART. The file main.c includes the code to initialize and enable this peripheral.*

2. Enable two DMA channels. One of these channels will be responsible for transferring data from the UART RX buffer to a memory buffer you declare, while the other one is responsible for transferring data from the memory buffer to the UART TX buffer.

Note: *Both of these channels will only use the ping descriptor, so you can ignore the pong descriptor settings.*

3.   Configure the first DMA channel as follows:

Channel		
(?) Trigger Input	🔗	Serial Communication Block (SCB) 3 rx_request (DEBUG_UART) [USED]
(?) Trigger Output	🔗	DMAC Channel 1 tr_in (TX_DMA) [USED]
(?) Channel Priority	3	
(?) Active Descriptor	Descriptor Ping	
Descriptor Ping		
(?) Data Count	1	
(?) Data Transfer Width	Word to Byte	
(?) Source Address Increment	☐	
(?) Destination Address Increment	☐	
(?) Trigger Deactivation And Retriggering	Retrigger immediately (pulse trigger)	
(?) Invalidate On Completion	☐	
(?) Interrupt On Completion	✓	
(?) Preemptable	✓	
(?) Flipping	☐	
(?) Triggering Type	Single descriptor transfer	

Note: *Make sure the trigger output of the first DMA channel is connected to the trigger input of the second DMA channel.*

4.   Configure the second DMA channel as follows:

Channel		
(?) Trigger Input	🔗	DMAC Channel 0 tr_out (RX_DMA) [USED]
(?) Trigger Output	<unassigned>	
(?) Channel Priority	3	
(?) Active Descriptor	Descriptor Ping	
Descriptor Ping		
(?) Data Count	1	
(?) Data Transfer Width	Byte to Word	
(?) Source Address Increment	☐	
(?) Destination Address Increment	☐	
(?) Trigger Deactivation And Retriggering	Retrigger immediately (pulse trigger)	
(?) Invalidate On Completion	☐	
(?) Interrupt On Completion	✓	
(?) Preemptable	✓	
(?) Flipping	☐	
(?) Triggering Type	Single descriptor transfer	

5.   Save this configuration and close the Device Configurator. In your application code, initialize and enable the UART and the DMA channels.

6.   Declare a 1-byte buffer. Set this buffer to be the destination address of the first DMA channel and the source address of the second DMA channel.

Set the first DMA channel's source address to the UART RX buffer, and the second DMA channel's destination address to the UART TX buffer.

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

5.6 PSoC™ 6 Exercises

In the exercises below, 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 it is suggested that you use the PDL when you require fine grained control of your hardware.

 All PSoC™ 6 exercises use the CY8CKIT-062S2-43012.

Exercise 6: 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 **ch05_ex06_PSoC6_echo** using the template **ch05_ex06_PSoC6_echo.** This template application can be found in *modustoolbox-level2-PSoC/Templates*.

Note: *This template comes with a custom device configuration in COMPONENT_CUSTOM_DESIGN_MODUS with SCB5 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 0 Channel 27, name it "UART_DMA", and configure it as follows:

Note: *DMA DW 0 Channel 27 must be used because it can connect its trigger input to the debug UART (SCB5) 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.
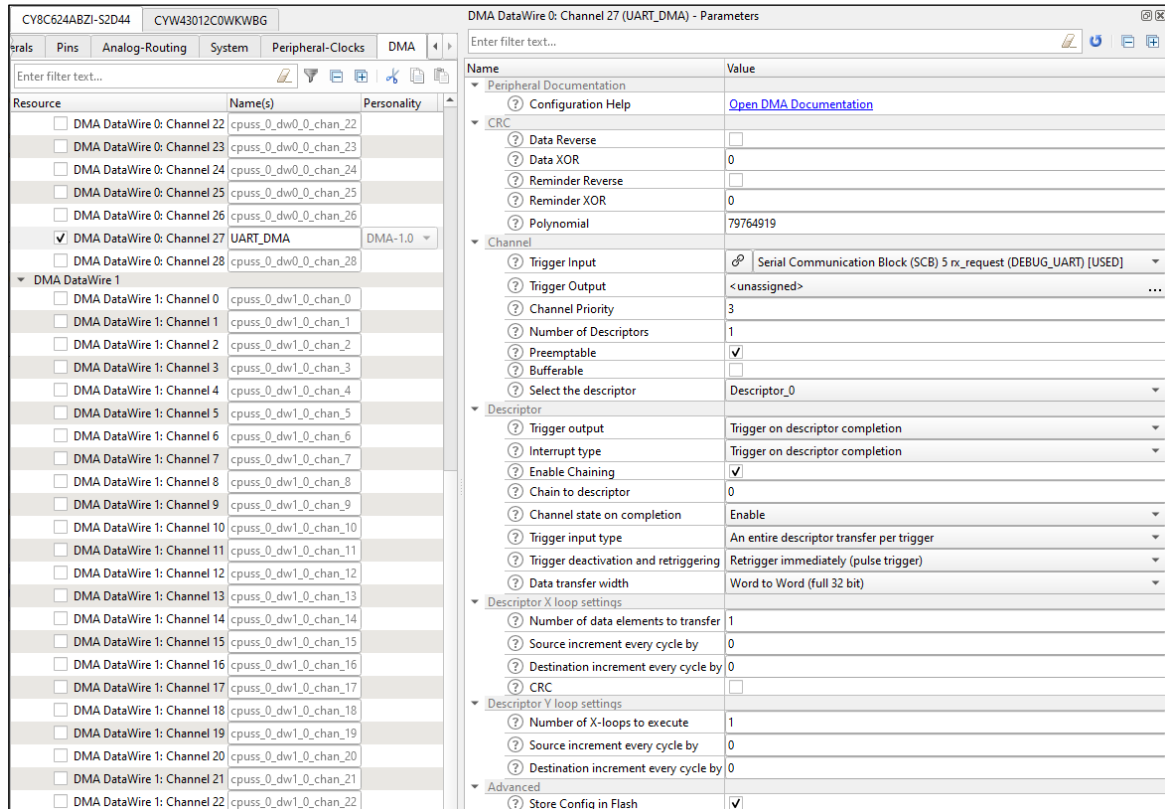
Note: *If you do not enable chaining, the `nextDescriptor` descriptor parameter will be set to `NULL` in the code generated by the Device Configurator. This is fine, but you will need to manually set this parameter in your application code using the `Cy_DMA_Descriptor_SetNextDescriptor` function to keep the descriptor channel from attempting to dereference the `NULL` pointer.*

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\n");
Cy_SCB_UART_PutString(DEBUG_UART_HW, "Start typing to see the echo on the screen \r\n\n");
```

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

Exercise 7: 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 **ch05_ex07_PSoC6_buffer** using the previous completed exercise **ch05_ex06_PSoC6_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	
⑦ Trigger output	Trigger on descriptor completion
⑦ Interrupt type	Trigger on descriptor completion
⑦ Enable Chaining	✔
⑦ Chain to descriptor	0
⑦ Channel state on completion	Enable
⑦ Trigger input type	One transfer per trigger
⑦ Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
⑦ Data transfer width	Word to Byte
Descriptor X loop settings	
⑦ Number of data elements to transfer	5
⑦ Source increment every cycle by	0
⑦ Destination increment every cycle by	1
⑦ CRC	☐

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. 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      = (IRQn_Type)UART_DMA_IRQ,
        .intrPriority = DMA_INT_PRIORITY,
};

/* Initialize and enable the DMA interrupt */
Cy_SysInt_Init(&DMA_INT_cfg, &Isr_DMA);
NVIC_EnableIRQ(DMA_INT_cfg.intrSrc);

/* Enable interrupt for DMA channel */
Cy_DMA_Channel_SetInterruptMask(UART_DMA_HW, UART_DMA_CHANNEL, CY_DMA_INTR_MASK);
```

7. Finally, 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).*

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

Exercise 8: 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 **ch05_ex08_PSoC6_printBuffer** using the template **ch05_ex08_PSoC6_printBuffer**. This template application can be found in *modustoolbox-level2-PSoC/Templates*.

Note: *This template comes with a custom configuration that configures SCB5 as a debug UART named DEBUG_UART. TCPWM 0 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 0 can connect to the preconfigured timer, so we will use it.

☐ 3. Enable DMA DW 0 Channel 0, name it UART_DMA, and configure it as follows:
   a. Set the **Number of Descriptors** to 2
   b. Configure <u>both</u> 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	
⑦ Trigger Input	\<unassigned\>
⑦ Trigger Output	\<unassigned\>
⑦ Channel Priority	3
⑦ Number of Descriptors	2
⑦ Preemptable	☐
⑦ Bufferable	☐
⑦ Select the descriptor	Descriptor_0
Descriptor	
⑦ Trigger output	Trigger on every element transfer completion
⑦ Interrupt type	Trigger on every element transfer completion
⑦ Enable Chaining	✓
⑦ Chain to descriptor	1
⑦ Channel state on completion	Enable
⑦ Trigger input type	An entire descriptor transfer per trigger
⑦ Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
⑦ Data transfer width	Byte to Word
Descriptor X loop settings	
⑦ Number of data elements to transfer	14
⑦ Source increment every cycle by	1
⑦ Destination increment every cycle by	0
⑦ CRC	☐

? Select the descriptor		Descriptor_1
▾ Descriptor		
? Trigger output		Trigger on every element transfer completion
? Interrupt type		Trigger on every element transfer 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 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	☐	

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

▾ Channel		
? Trigger Input	🔗	TCPWM[0] 32-bit Counter 0 cc_match/capture/compare (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 <u>both</u> 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 9: 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 **ch05_ex09_PSoC6_external** using the previous completed exercise **ch05_ex08_PSoC6_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:
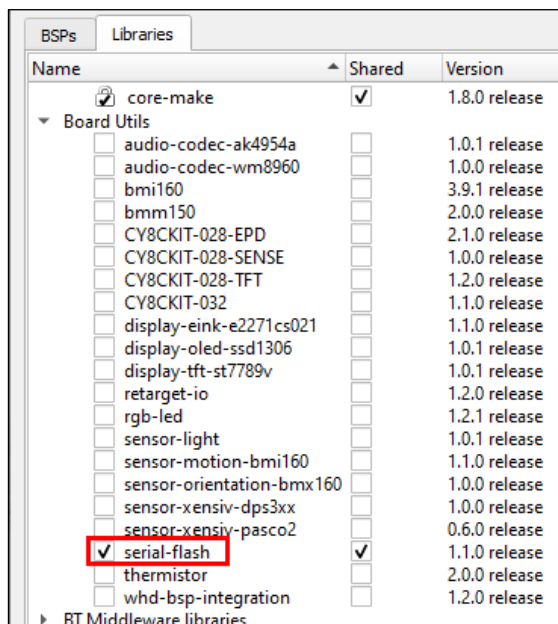
6. In your application code, `#include` the following header files to get access to the serial flash functions:

```
#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. The BSP for the CY8CKIT-062S2-43012 includes a memory configuration structure that is already configured for the external flash memory on the kit. You can open the QSPI Configurator to view this configuration if you are interested.*

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   (50000000lu)
#define MESSAGE_LENGTH       (14)

// Gloval Vars
uint32_t startOfFlashGlobal = 0x18000000; // 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 (0x18000000), 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 10: N-NxM Transfer

In this exercise we will use the PSoC™ 6 DMAC X and Y loops to transfer an array of structures.

1. Create a new application called **ch05_ex10_PSoC6_struct** using the template **ch05_ex10_PSoC6_struct**. This template application can be found in *modustoolbox-level2-PSoC/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 SCB5 as a debug UART named DEBUG_UART. TCPWM 0 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.*

   Note: *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.*

   Note: *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.*

   Note: *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 - PSoC - 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 11: 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 **ch05_ex11_PSoC6_descriptorChain** using completed exercise **ch05_ex08_PSoC6_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 desciptor 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	
⑦ Trigger Input	🔗 TCPWM[0] 32-bit Counter 0 cc_match/capture/compare (DMA_TIMER) [USED]
⑦ Trigger Output	\<unassigned\>
⑦ Channel Priority	3
⑦ Number of Descriptors	4
⑦ Preemptable	☐
⑦ Bufferable	☐
⑦ Select the descriptor	Descriptor_0
Descriptor	
⑦ Trigger output	Trigger on every element transfer completion
⑦ Interrupt type	Trigger on descriptor completion
⑦ Enable Chaining	☑
⑦ Chain to descriptor	1
⑦ Channel state on completion	Enable
⑦ Trigger input type	Entire descriptor chain per trigger
⑦ Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
⑦ Data transfer width	Byte to Word

⑦ Select the descriptor	Descriptor_1
Descriptor	
⑦ Trigger output	Trigger on every element transfer completion
⑦ Interrupt type	Trigger on descriptor completion
⑦ Enable Chaining	☑
⑦ Chain to descriptor	2
⑦ Channel state on completion	Enable
⑦ Trigger input type	Entire descriptor chain per trigger
⑦ Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
⑦ Data transfer width	Byte to Word

⑦ Select the descriptor	Descriptor_2
Descriptor	
⑦ Trigger output	Trigger on descriptor completion
⑦ Interrupt type	Trigger on descriptor completion
⑦ Enable Chaining	☑
⑦ Chain to descriptor	3
⑦ Channel state on completion	Enable
⑦ Trigger input type	Entire descriptor chain per trigger
⑦ Trigger deactivation and retriggering	Retrigger immediately (pulse trigger)
⑦ Data transfer width	Byte to Word
Descriptor X loop settings	
⑦ Number of data elements to transfer	14
⑦ Source increment every cycle by	1
⑦ Destination increment every cycle by	0
⑦ CRC	☐

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 PSoC!\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 12: 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 **ch05_ex12_PSoC6_channelChain** using completed exercise **ch05_ex06_PSoC6_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

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

4. Save this configuration and close the Device Configurator.

5. Remove the code for initializing and setting source/desitnations 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.