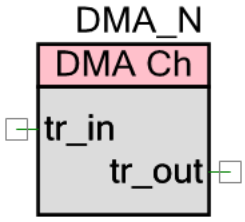
PSoC 4 Direct Memory Access (DMA) Channel

1.0



Features

- Support for up to 32 DMA channels; consult the device-specific datasheet to determine how many channels for a particular device
- Two independent descriptors per channel
- Four priority levels
- Byte, halfword (2 bytes), and word (4 bytes) transfers
- Transfer sizes up to 65536 data elements
- Configurable interrupt generation
- Output trigger on completion of transfer
- Three transfer modes
  - Single data element per trigger
  - All data elements per trigger
  - All data elements per trigger and automatically trigger chained descriptor

General Description

The DMA Channel component transfers data to and from memory, components, and registers. These transfers occur independent of the CPU. The DMA can transfer up to 65,536 data elements. These data elements can be a byte, halfword (2 bytes), or word (4 bytes) wide. The DMA starts each transaction through an external trigger that can come from a DMA channel (including itself), another DMA channel, a peripheral, or the CPU. The DMA is best used to offload data transfer tasks from the CPU.

When to Use a DMA Channel

The DMA Channel component can be used in any project that needs to transfer data without CPU intervention based on a hardware trigger signal from another component.

A common use is transferring data from memory to a peripheral, such as a UART. The DMA can be triggered by the UART FIFO not full signal. The DMA will load data in the UART until the FIFO fills.

The DMA can also be used to take data out of the UART and place it in memory. For example, the DMA can be triggered by the FIFO not empty signal, thus the DMA will transfer data as long as the FIFO is not empty.

Another common use is transferring data from the ADC to memory. The ADC's end of conversion (eoc) signal can be used to trigger the DMA to transfer the ADC result to memory.

The DMA can also be used to move blocks of memory from one memory location to another (RAM to RAM, FLASH to RAM); DMA cannot write to FLASH.

Each DMA channel can be triggered by a hardware signal as described above, or by a firmware register write, or both.

Each DMA channel has two descriptors. While one descriptor is running, the other can be updated by the CPU during run time. This allows for the creation of “extra” descriptors via firmware. The hardware makes this easier by providing a mechanism that optionally invalidates a descriptor when it is complete. While the descriptor is invalid, the CPU can update it.

Exercise care to ensure that the firmware updates the descriptor before it is triggered. The amount of time required for this will be application dependent; specifically, how quickly will the descriptor be re-triggered after it has finished. You must determine this time and the time it takes to update the descriptor. If the DMA engine triggers an invalid descriptor, it will not run it and will disable that channel. You can determine this has occurred by reading the descriptor status.

If updating the descriptor using API calls is too slow for your application, you can do it through direct register writes to the descriptor structure. Each descriptor structure has four 32-bit words:

```
CYDMA_DESCR_BASE.descriptor[channel][descriptor].src
CYDMA_DESCR_BASE.descriptor[channel][descriptor].dst
CYDMA_DESCR_BASE.descriptor[channel][descriptor].ctl
CYDMA_DESCR_BASE.descriptor[channel][descriptor].status
```

where `channel` is the associated DMA channel number and `descriptor` is the descriptor (0 or 1) being updated. For examples, refer to the API functions in *CyDMA.c* file, which you can open from the Workspace Explorer.

Another use is the firmware DMA. For this case, you do not need to place a DMA component on the schematic. You can allocate a DMA channel and configure it through API function calls the same as the component customizer. Then you can trigger the DMA from firmware.

Input/Output Connections

This section describes the various input and output connections for the DMA. An asterisk (*) in the list of I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

tr_in * – Input

The trigger in (tr_in) terminal is used to trigger a DMA transfer, as defined by the Transfer mode. The trigger can come from this DMA channel (tr_out), another DMA channel, another component, or the CPU. The terminal is visible if the Enable trigger input parameter is checked.

This is configured in the Trigger type setting; refer to that section for more details. If the tr_in terminal is not enabled the only way to trigger the DMA channel is through firmware.

The DMA channel is triggered by a logic HIGH/1 on the tr_in terminal. The minimum width of this logic HIGH/1 is 2 system clock (SYSCLK) cycles. All components that have DMA trigger outputs automatically comply with this convention. If you are creating your own trigger using UDB logic, ensure that the trigger complies with this requirement.

If tr_in goes high while the DMA engine is in the middle of another transfer, the DMA sets a flag indicating that the channel has a pending request. When the DMA finishes the current transfer, it evaluates all of the pending channels and starts a transfer for the pending channel with the highest priority. If this channel does not have the highest priority, its request will remain pending until its priority becomes the highest. Only one request can be pending per channel at any one time. Multiple pending requests are counted as one.

In some situations, the trigger may remain high to indicate a continuing need for DMA transactions. This type of trigger is called a level-sensitive trigger. These types of triggers often originate from sources such as Serial Communication Block (SCB) FIFOs. The FIFOs have status signals that indicate if they are not full, or empty. These signals can be used to trigger the DMA to transfer data to the FIFO until it is full, or transfer data from the FIFO until it is empty.

With these signals, it is undesirable for the DMA to transfer more data to the FIFO when it is full or to transfer data from the FIFO when it is empty. As stated previously, if the DMA is in the middle of a transfer and the trigger signal is high, it will set a flag indicating that the channel has a pending request. Thus, while the DMA is transferring the last byte to a FIFO, the trigger signal will remain high until the byte reaches the FIFO. Because of this, the DMA will be triggered again. In this situation, the FIFO is full but the DMA channel has a pending request. This is an undesirable situation.

To overcome this situation, the DMA can be configured to wait a certain amount of system clock (SYSCLK) cycles before re-triggering the channel, as set by the Trigger type parameter. It can be configured to wait four, eight, or an indefinite number of SYSCLK clock cycles after a transfer completes, before re-triggering the channel. If during this time the trigger goes low, the DMA channel will be re-triggered on the next logic HIGH/1 trigger. If the trigger doesn't go low, the DMA waits the specified number of clock cycles before re-triggering. Please note that setting this to indefinite has the potential of locking the DMA up if the trigger never goes low.

The following three figures demonstrate how this feature works. In each figure, the DMA is configured to wait four SYSCLK cycles before retriggering the DMA. In Figure 1, the DMA trigger remains high. In Figure 2, the trigger goes low and then back high. In Figure 3, the trigger goes low and remains low.

Figure 1. DMA trigger behavior; Trigger remains high

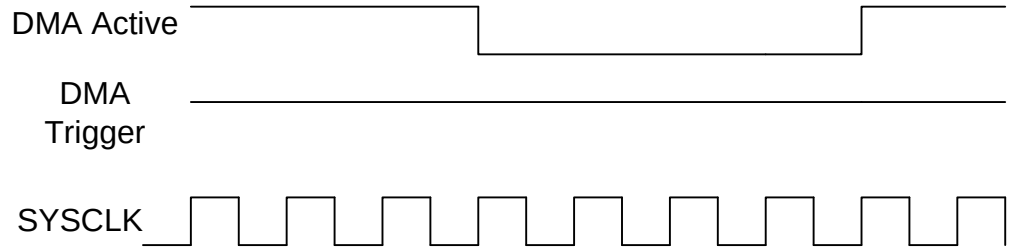


Figure 2. DMA trigger behavior; Trigger goes low then high

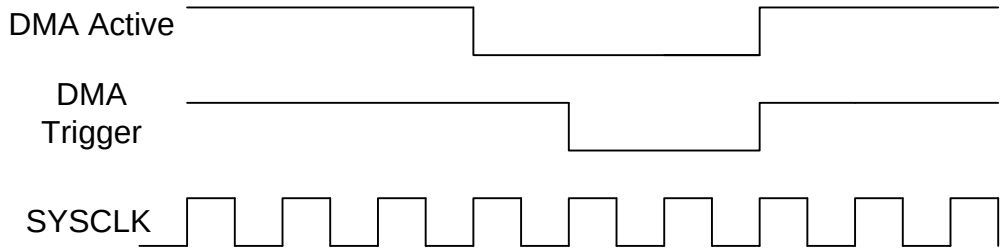
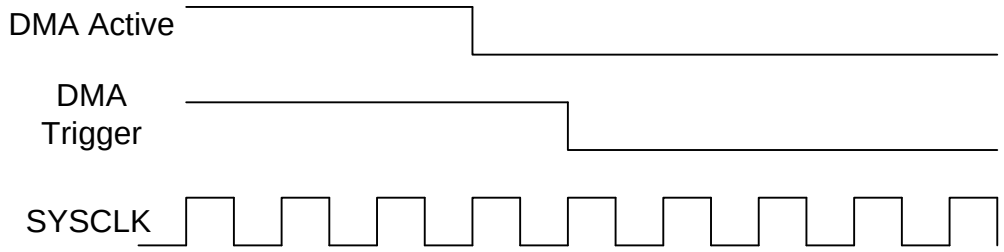


Figure 3. DMA trigger behavior; Trigger goes and stays low



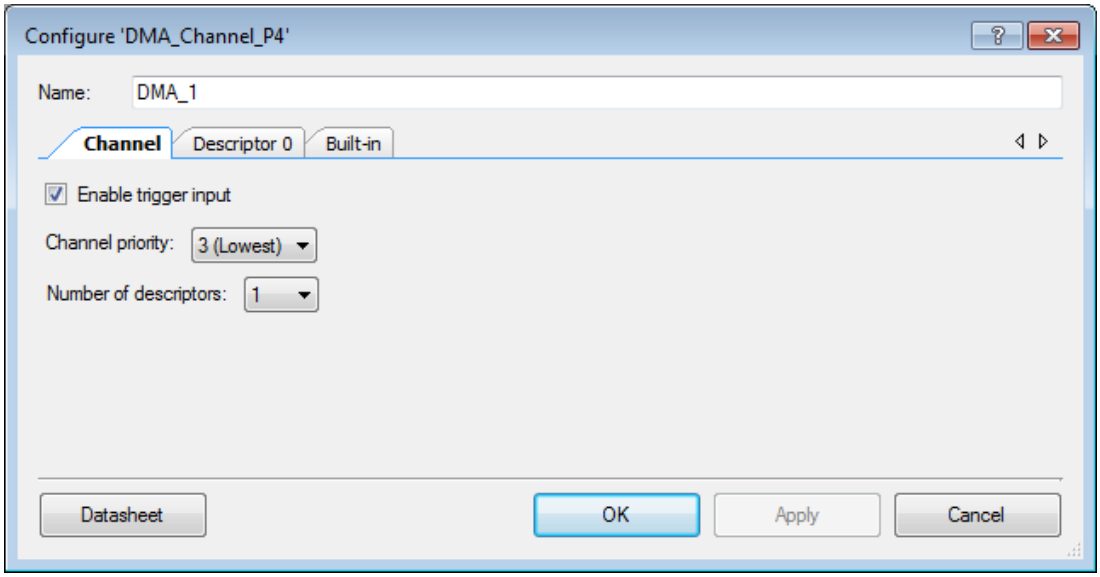
tr_out – Output

Each DMA channel has one output trigger (tr_out). This output trigger is a two cycle HIGH/1 pulse on SYSCLK. The trigger is generated on the completion of a transfer; a transfer is defined by the Transfer mode parameter.

Component Parameters

Drag a DMA Channel onto your design and double click it to open the Configure dialog. This dialog has the following tabs with different parameters.

Channel Tab



Enable trigger input

Enables or disables the trigger input terminal. If enabled, the DMA can be triggered by a hardware signal or by firmware. If disabled, the DMA can only be triggered by firmware. If enabled the terminal must be connected to a trigger source.

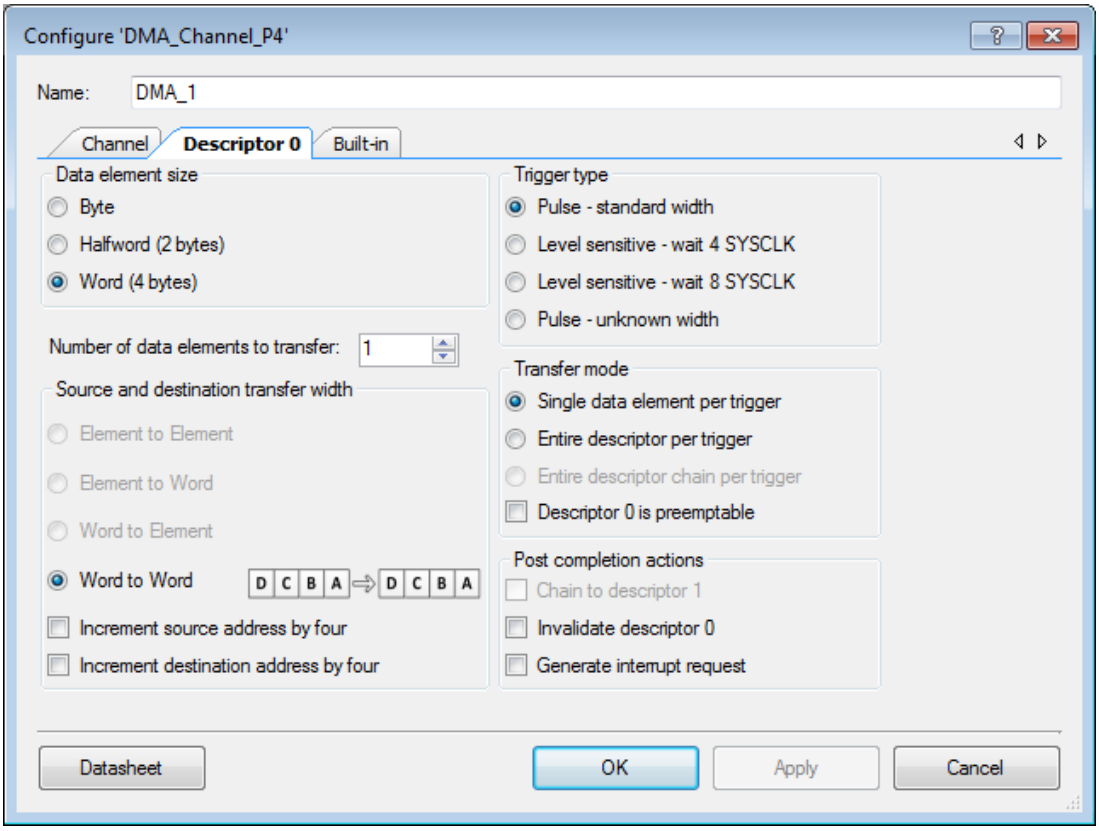
Channel priority

Specifies the channel priority. 0 is highest. Default setting is 3.

Number of descriptors

Specifies the number of descriptors used by the channel.

Descriptor 0/1 Tab



Data element size

Specifies the size of the data element. The DMA engine transfers one data element at a time. This parameter determines how wide the transfer is. This option can be set to **Byte**, **Halfword (2 bytes)**, or **Word (4 bytes)**. The default setting is **Word (4 bytes)**.

Number of data elements to transfer

Specifies the number of data elements to transfer. This value can be set between **1** and **65536**. The default setting is **1**.

Source and destination transfer width

Specifies the size of the source and destination locations. Source is listed first (left). This determines by how many bytes the source and destination address are incremented if increment is turned on. The default setting is **Word to Word**.

For memory locations, the source and destination width will typically equal the data element size. However, some components provide/require data elements that are smaller than their 32-bit bus interface width. In these cases, either the source or destination width needs to be a word, and the data element size needs to be smaller.

For example, an ADC has a 32-bit bus transfer size, but only provides a 16-bit data element. Thus, a transfer of a 16-bit sample from an ADC source to a memory destination will have the data element size set to **Halfword** and the source and destination width configured as **Word to halfword**.

The width of most PSoC 4 peripheral registers is 4 bytes (word) [1]. So typically the source or destination transfer width should be set to **Word** when DMA is using a peripheral as its source or destination. The source and destination transfer width for the DMA component must match the addressable width of the source and destination, regardless of the amount of data that needs to be moved.

For example, if a 16-bit PWM compare value register is used as a destination for DMA data, the destination transfer width must be set to word to match the width of the PWM register, because the peripheral register width for the TCPWM block (and most PSoC 4 peripherals) is always 4 bytes wide. However, in this example, the data element size for the destination may still be set to **Halfword** (2 bytes) because the 16-bit PWM only uses 2 bytes of data. SRAM and flash are 8-bit, 16-bit, or 32-bit addressable; they can use any transfer width to match the needs of the application.

Refer to the corresponding component datasheets for more details about their DMA access information.

Increment source address by *X*

After the transfer of each single data element increment the source address by the source transfer width. *X* will equal what is selected for the source location of the Source and destination transfer width parameter. Options = checked or unchecked (default).

Increment destination address by *X*

After the transfer of each single data element increment the source address by the source transfer width. *X* will equal what is selected for the source location of the Source and destination transfer width parameter. Options = checked or unchecked (default).

Trigger type

Specifies the type of DMA input trigger. See tr_in input section for more information on DMA trigger behavior. This value can be set to:

- **Pulse –** standard width (default). The pulse is a logic HIGH/1 with the minimum width of 2 system clock (SYSCLK) cycles.
- **Level sensitive –** wait 4 SYSCLK. DMA will wait 4 SYSCLK cycles after a transfer completes before re-triggering the channel.

---

1 Some UDB peripheral registers allow 8-bit or 16-bit addressing.

- **Level sensitive –** wait 8 SYSCLK. DMA will wait 8 SYSCLK cycles after a transfer completes before re-triggering the channel.
- **Pulse –** unknown width. DMA will wait an indefinite number of SYSCLK cycles after a transfer completes before re-triggering the channel. The trigger must go low for the DMA channel to be re-triggered on the next logic HIGH/1 trigger.

Level sensitive triggers have the ability to wait before re-triggering the DMA. This is best explained through the following pseudo code.

```
/* After transaction finishes */
if (trig == 1) /* If trigger is high */
{
    /* Wait for selected time or until trigger goes low */
    do
    {
       /* Increment on SYSCLK */
       num++;
    } while(num != triggerType && trig == 1)
}
/* Wait for new trigger */
while(trig == 0);
/* trigger new transaction */
trigger;
```

Transfer mode

Specifies how DMA reacts on a single trigger. The settings are:

- **Single data element per trigger** (default) – Each trigger causes the DMA engine to transfer one data element.
- **Entire descriptor per trigger** – Each trigger causes the DMA engine to transfer all data elements specified by Number of data elements to transfer parameter, one data element at a time.
- **Entire descriptor chain per trigger** – Each trigger causes the DMA engine to transfer all data elements specified by Number of data elements to transfer parameter, one data element at a time. After all data elements are transferred, automatically trigger chained descriptor.

  Note The option **Chain to descriptor 0/1** must be checked for Post completion actions to use this mode.

Descriptor 0/1 is preemptable

Specifies if the current transfer is allowed to complete undisturbed. If checked the current transfer can be preempted/interrupted by a DMA channel of higher priority. When this channel is preempted it is set as pending and will run next time its priority is the highest. Options = checked or unchecked (default).

Post completion actions

Specifies what occurs descriptor 0/1 completes. The settings are:

- **Chain to descriptor 1/0** - after the current descriptor completes chain to the next descriptor. If the Transfer mode parameter is set to **Entire descriptor chain per trigger**, the next descriptor is automatically triggered. Otherwise the next descriptor must be triggered.

  Note If the Transfer mode is set to **Entire descriptor chain per trigger**, then this option will be automatically checked.

- **Invalidate descriptor** - when the descriptor finishes transferring all data elements, clear the descriptors valid bit, making it invalid.

- **Generate interrupt request** - when the descriptor finishes transferring all data elements, generate an interrupt request. This interrupt request is shared among all DMA channels. Refer to the Interrupt Service Routine section for a detailed description on how to access this interrupt.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following sections list and describe the interface to each function. The subsequent sections cover each function in more detail.

API Functions per DMA Instance

By default, PSoC Creator assigns the instance name "DMA_1" to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is "DMA".

All the functions listed in the table below except DMA_Start() and DMA_Init() are inline functions. These functions call the corresponding global DMA API functions and have the channel number hardcoded. For example, DMA_Trigger() API calls CyDmaTrigger(DMA_CHANNEL). DMA_CHANNEL define is generated by PSoC Creator during design build process.

Function	Description
DMA_Start()	Configures DMA based on customizer settings, sets source and destination addresses and enables the channel.
DMA_Init()	Initializes DMA based on customizer settings.
DMA_ChEnable()	Enables the DMA channel.
DMA_ChDisable()	Disables the DMA channel.

Function	Description
DMA_Trigger()	Triggers the DMA channel.
DMA_SetConfiguration()	Sets configuration information for the specified channel and descriptor.
DMA_SetNextDescriptor()	Sets the next descriptor to run.
DMA_GetNextDescriptor()	Returns the next descriptor that will be run.
DMA_Validate_Descriptor()	Validates an invalid descriptor.
DMA_GetDescriptorStatus()	Returns current status of descriptor.
DMA_SetPriority()	Sets the channel priority.
DMA_GetPriority()	Returns the channel priority.
DMA_SetSrcAddress()	Sets the source address.
DMA_GetSrcAddress()	Returns the source address.
DMA_SetDstAddress()	Sets the destination address.
DMA_GetDstAddress()	Returns the destination address.
DMA_SetDataElementSize()	Sets the size of each data element.
DMA_GetDataElementSzie()	Returns the size of each data element.
DMA_SetNumDataElements()	Sets the number of data elements to transfer.
DMA_GetNumDataElements()	Returns the number of data elements to transfer.
DMA_SetSrcDstTransferWidth()	Sets the width of the source and destination.
DMA_GetSrcDstTransferWidth()	Returns the width of the source and destination.
DMA_SetAddressIncrement()	Sets if the source and destination addresses are incremented.
DMA_GetAddressIncrement()	Returns whether or not the source or destination addresses are incremented.
DMA_SetTriggerType()	Sets how long the DMA waits for its trigger to go low.
DMA_GetTriggerType()	Returns how long the DMA waits for its trigger to go low.
DMA_SetTransferMode()	Sets the DMA transfer mode.
DMA_GetTransferMode()	Returns the DMA transfer mode.
DMA_SetPreemptable()	Sets if the descriptor is preemptable.
DMA_GetPreemptable()	Checks if descriptor is preemptable.
DMA_SetPostCompletionActions()	Sets what occurs after the descriptor completes.
DMA_GetPostCompletionActions()	Returns what happens after the descriptor completes.
DMA_SetInterruptCallback()	Sets a user defined callback function to be called by the DMA interrupt.
DMA_GetInterruptCallback()	Returns pointer to a user defined interrupt callback function.

void DMA_Start(void * srcAddress, void * dstAddress)

Description: Calls Init() to configure the DMA channel based on customizer settings if the channel has not been initialized before.

Sets source and destination address, validates descriptor 0 and enables the channel. After calling this function the DMA channel is active and waiting for a trigger.

Parameters: srcAddress: Address of DMA transfer source.

dstAddress: Address of DMA transfer destination.

Return Value: None

Side Effects: None

void DMA_Init(void)

Description: Initializes the DMA channel based on the parameters set in the component customizer. It is not necessary to call DMA_Init() because the DMA_Start() API calls this function, which is the preferred method to begin component operation.

Parameters: None

Return Value: None

Side Effects: The first descriptor is set to descriptor 0. This function should not be called while the channel is enabled. All settings will be reset to their initial values.

Global DMA API Functions

DMA Controller API Functions

Function	Description
CyDmaEnable()	Enables the DMA transfer engine.
CyDmaDisable()	Disables the DMA transfer engine.
CyDmaGetActiveChannels()	Returns a bit-field with all of the currently active/pending channels.
CyDmaGetActiveSrcAddress()	Returns the source address currently used by DMA transfer engine.
CyDmaGetActiveDstAddress()	Returns the destination address currently used by the DMA transfer engine.
CyDmaGetStatus()	Returns status of DMA transfer engine.
CyDmaSetInterruptVector()	Sets the DMA interrupt vector.
CyDmaGetInterruptSource()	Returns which channels have active/pending interrupts.
CyDmaClearInterruptSource()	Clears specified active/pending interrupts.
CyDmaSetInterruptSourceMask()	Create an interrupt mask.
CyDmaGetInterruptSourceMask()	Returns an interrupt mask.

Function	Description
CyDmaGetInterruptSourceMasked()	Returns bit-wise AND of interrupt source and mask.
CyDmaSetInterruptCallback()	Sets a user defined callback function to be called by the DMA interrupt.
CyDmaGetInterruptCallback()	Returns pointer to a user defined interrupt callback function.
CyDmaInterrupt()	The default ISR for DMA interrupts.

void CyDmaEnable(void)

Description: Sets the default ISR to be called by the DMA interrupt and enables the DMA transfer engine.

Parameters: None

Return Value: None

Side Effects: Does not affect channel enable status. Overrides the interrupt vector set prior the first time CyDmaEnable() is called.

void CyDmaDisable(void)

Description: Disables the DMA transfer engine.

Parameters: None

Return Value: None

Side Effects: Does not affect channel enable status.

uint32 CyDmaGetActiveChannels(void)

Description: Returns a bit field of all the DMA channels that are either active or pending.

Parameters: None

Return Value: Bit field of active and pending channels.

Side Effects: None

*void * CyDmaGetActiveSrcAddress(void)*

Description: Returns the source address currently being used by the DMA transfer engine. This function can be used to debug the DMA, or observe where it is at in a transfer. It will not be used in normal DMA operation.

Parameters: None

Return Value: Source address currently being used by DMA transfer engine.

Side Effects: The series of function calls CyDmaGetStatus(), CyDmaGetActiveSrcAddress(), and CyDmaGetActiveDstAddress() are not atomic, the DMA engine may have advanced after one or more of these function calls. Meaning the returns from these three functions may not be related to each other.

*void * CyDmaGetActiveDstAddress(void)*

Description: Returns the destination address currently being used by the DMA transfer engine. This function can be used to debug the DMA, or observe where it is at in a transfer. It will not be used in normal DMA operation.

Parameters: None

Return Value: Destination address currently being used by DMA transfer engine.

Side Effects: The series of function calls CyDmaGetStatus(), CyDmaGetActiveSrcAddress(), and CyDmaGetActiveDstAddress() are not atomic, the DMA engine may have advanced after one or more of these function calls. Meaning the returns from these three functions may not be related to each other.

uint32 CyDmaGetStatus(void)

Description: Returns the status of the DMA transfer engine.

Parameters: None

Return Value: Returns the contents of the DMA status register. See the register section for more detail on bit meaning. Below is a table of defines (mask) for accessing the data in the register

Define (Mask)	Description	Bits
CYDMA_TRANSFER_INDEX	Transfer index of currently active channel. This value increase from zero to Number of data elements to transfer - 1.	[15:0]
CYDMA_CH_NUM	Channel number of currently active channel.	[20:16]
CYDMA_STATE	Current state of DMA transfer engine, see table below for descriptions of each state.	[26:24]
CYDMA_PRIO	Priority of currently active channel.	[29:28]
CYDMA_DESCRIPTOR	Active descriptor.	30
CYDMA_ACTIVE	Specifies if there is a currently active / pending transfer in the data transfer engine.	31

Defines for STATE: Only one define can be active at a time

Define	Description
CYDMA_IDLE	Idle state when the DMA is not active.
CYDMA_LOAD_DESCR	The DMA is loading the descriptor to the DMA transfer engine.
CYDMA_LOAD_SRC	DMA is getting the value from the source location.
CYDMA_STORE_DST	DMA is storing a value at the destination location.
CYDMA_STORE_DESCR	DMA is updating the descriptors after completion of transfer.
CYDMA_WAIT_TRIG_DEACT	The DMA is waiting for the level sensitive trigger to deactivate.
CYDMA_STORE_ERROR	There was an error during the transaction and the DMA is writing the error code to the channel status register.

Side Effects: The series of function calls CyDmaGetStatus(), CyDmaGetActiveSrcAddress(), CyDmaGetActiveDstAddress() are not atomic, the DMA engine may have advanced after one or more of these function calls. Meaning the returns from these three functions may not be related to each other.

void CyDmaSetInterruptVector(cyisraddress interruptVector)

Description: Sets the function that will be called by the DMA interrupt.

Note Calling CyDmaEnable() for the first time overrides any effect this API could have. Call CyDmaSetInterruptVector() after calling CyDmaEnable() to change the default ISR.

When defining ISR functions, the CY_ISR and CY_ISR_PROTO macros should be used to provide consistent definition across compilers:

Function definition example:

```
CY_ISR(MyISR)
{
    /* ISR Code here */
}
```

Function prototype example:

```
CY_ISR_PROTO(MyISR);
```

Parameters: Address of function that will be called by the DMA interrupt.

Return Value: None

Side Effects: Other components that use DMA may register their callback functions to be called from the default DMA ISR. Therefore, changing the DMA ISR to a user defined ISR using this API may prevent these other components from functioning correctly.

uint32 CyDmaGetInterruptSource(void)

Description: Returns the bit field of which channels generated an interrupt request.

Parameters: None

Return Value: Bit filed of which channels generated an interrupt request.

Side Effects: None

void CyDmaClearInterruptSource(uint32 interruptMask)

Description: Clears the pending interrupts.

Parameters: interruptMask: Bit field of interrupts to clear.

Return Value: None

Side Effects: None

void CyDmaSetInterruptSourceMask(uint32 interruptMask)

Description: Sets mask for interrupt source.

Parameters: interruptMask: Mask corresponding to interrupt bit field.

Return Value: None

Side Effects: None

uint32 CyDmaGetInterruptSourceMask(void)

Description: Returns mask for interrupt source.

Parameters: None

Return Value: Mask corresponding to interrupt bit field.

Side Effects: None

uint32 CyDmaGetInterruptSourceMasked (void)

Description: Returns the bitwise AND of the interrupt source and the interrupt mask.

Parameters: None

Return Value: Bitwise AND of the interrupt source and the interrupt mask.

Side Effects: None

cydma_callback_t CyDmaSetInterruptCallback(int32 channel, cydma_callback_t callback)

Description: Sets a user defined callback function to be called by the DMA interrupt. The function should contain code to process the interrupt request for the associated DMA channel.

Parameters: channel: Channel used by this function.

callback: Pointer to the user defined callback function.

Return Value: Pointer to the function previously set for the specified channel.

Side Effects: None

cydma_callback_t CyDmaGetInterruptCallback(int32 channel)

Description: Returns the pointer to the interrupt callback function for the specified DMA channel.

Parameters: channel: Channel used by this function.

Return Value: Callback function pointer for the specified channel.

Side Effects: None

void CyDmaInterrupt(void)

Description: The default ISR for DMA interrupts. The handler checks which DMA channel has triggered the interrupt and calls the user defined callback function. The callback function is set using CyDmaSetInteruptCallback() API.

Parameters: None

Return Value: None

Side Effects: This function clears the pending interrupts.

DMA Channel API Functions

Function	Description
CyDmaChAlloc()	Allocates a channel number for use with later DMA functions.
CyDmaChFree()	Frees a channel handle allocated by CyDmaChAlloc().
CyDmaChEnable()	Enables the DMA channel.
CyDmaChDisable()	Disables the DMA channel.
CyDmaTrigger()	Triggers the DMA channel.
CyDmaSetPriority()	Sets the channel priority.
CyDmaGetPriority()	Returns the channel priority.
CyDmaSetNextDescriptor()	Sets the next descriptor to run.
CyDmaGetNextDescriptor()	Returns the next descriptor that will be run.

int32 CyDmaChAlloc(void)

Description: Allocates a channel number for use with later DMA functions.

Parameters: None

Return Value: The allocated channel number. Zero is a valid channel number. CYDMA_INVALID_CHANNEL is returned if there are no channels available.

Side Effects: None

cystatus CyDmaChFree(int32 channel)

Description: Frees a channel number allocated by CyDmaChAlloc().

Parameters: channel: The channel previously returned by CyDmaChAlloc().

Return Value: CYRET_SUCCESS if successful.
CYRET_BAD_PARAM if channel is invalid.

Side Effects: None

void CyDmaChEnable(int32 channel)

Description: Enables the DMA channel.

Parameters: channel: Channel used by this function.

Return Value: None

Side Effects: If this function is called before CyDmaSetConfiguration(), and CyDmaSetSrcAddress() and CyDmaSetDstAddress() the operation of the DMA is undefined and could result in system data corruption.

void CyDmaChDisable(int32 channel)

Description: Disables the DMA channel.

Parameters: channel: Channel used by this function.

Return Value: None

Side Effects: If this function is called during a DMA transfer the transfer is aborted.

void CyDmaTrigger(int32 channel)

Description: Triggers the DMA channel.

Parameters: channel: Channel used by this function.

Return Value: None

Side Effects: None

void CyDmaSetPriority(int32 channel, int32 priority)

Description: Sets the priority for the channel.

Parameters: channel: Channel used by this function.

priority: Priority for channel. Priority can be 0,1,2, or 3. 0 is the highest priority.

Return Value: None

Side Effects: This function should not be called while the channel is enabled

int32 CyDmaGetPriority(int32 channel)

Description: Returns the priority for the channel.

Parameters: channel: Channel used by this function.

Return Value: Priority for channel. Priority can be 0, 1, 2, or 3. 0 is the highest priority.

Side Effects: None

void CyDmaSetNextDescriptor(int32 channel, int32 descriptor)

Description: Sets the descriptor that should be run the next time the channel is triggered.

Parameters: channel: Channel used by this function.

descriptor: Next descriptor to run (0 or 1).

Return Value: None

Side Effects: This function should not be called while the channel is enabled.

int32 CyDmaGetNextDescriptor(int32 channel)

Description: Returns the next descriptor that should be run, as set by CyDmaSetNextDescriptor().

Parameters: channel: Channel used by this function.

Return Value: descriptor: Next descriptor to run (0 or 1).

Side Effects: None

Descriptor API Functions

Function	Description
CyDmaSetConfiguration()	Sets configuration information for the specified channel and descriptor.
CyDmaValidateDescriptor()	Validates an invalid descriptor.
CyDmaGetDescriptorStatus()	Returns current status of descriptor.
CyDmaSetSrcAddress()	Sets the source address.
CyDmaGetSrcAddress()	Returns the source address.
CyDmaSetDstAddress()	Sets the destination address.
CyDmaGetDstAddress()	Returns the destination address.
CyDmaSetDataElementSize()	Sets the size of each data element.
CyDmaGetDataElementSize()	Returns the size of each data element.
CyDmaSetNumDataElements()	Sets the number of data elements to transfer.
CyDmaGetNumDataElements()	Returns the number of data elements to transfer.
CyDmaSetSrcDstTransferWidth()	Sets the width of the source and destination.
CyDmaGetSrcDstTransferWidth()	Returns the width of the source and destination.
CyDmaSetAddressIncrement()	Sets if the source and destination addresses are incremented.
CyDmaGetAddressIncrement()	Returns whether or not the source or destination addresses are incremented.
CyDmaSetTriggerType()	Sets how long the DMA waits for its trigger to go low.
CyDmaGetTriggerType()	Returns how long the DMA waits for its trigger to go low.
CyDmaSetTransferMode()	Sets the DMA transfer mode.
CyDmaGetTransferMode()	Returns the DMA transfer mode.
CyDmaSetPreemptable()	Sets if the descriptor is preemptable.
CyDmaGetPreemptable()	Checks if descriptor is preemptable.
CyDmaSetPostCompletionActions()	Sets what occurs after the descriptor completes.
CyDmaGetPostCompletionActions()	Returns what happens after the descriptor completes.

*void CyDmaSetConfiguration(int32 channel, int32 descriptor, cydma_init_struct * config)*

Description: Sets configuration information for the specified descriptor. This function is used to configure the DMA when the component is not placed in a schematic.

Parameters: channel: DMA channel modified by this function.

descriptor: Descriptor (0 or 1) modified by this function.

config: pointer to a structure that contains the following list of fields These fields match the selections available in the customizer.

Field	Description
uint32 dataElementSize	Specifies the size of the data element. Options: ▪ CYDMA_BYTE ▪ CYDMA_HALFWORD ▪ CYDMA_WORD See CyDmaSetDataElementSize() API for more information on configuration values.
int32 numDataElements	Total number of data elements this descriptor transfers. Valid ranges are 1 to 65536.
uint32 srcDstTransferWidth	Specifies the width of the source and destination. Options: ▪ CYDMA_ELEMENT_ELEMENT ▪ CYDMA_ELEMENT_WORD ▪ CYDMA_WORD_ELEMENT ▪ CYDMA_WORD_WORD See CyDmaSetSrcDstTransferWidth() API for more information on configuration values.
uint32 addressIncrement	Specifies whether the source or destination addresses are incremented after the transfer of each data element. Options (can be OR'd together): ▪ CYDMA_INC_SRC_ADDR ▪ CYDMA_INC_DST_ADDR ▪ CYDMA_INC_NONE See CyDmaSetAddressIncrement() API for more information on configuration values.
uint32 triggerType	Specifies the type of input trigger for the DMA. Level sensitive triggers can be configured to wait a number of system clocks (SYSCLK) for the trigger to go low (deactivate) before triggering the channel again. Options: ▪ CYDMA_PULSE ▪ CYDMA_LEVEL_FOUR ▪ CYDMA_ LEVEL_EIGHT ▪ CYDMA_PULSE_UNKNOWN See CyDmaSetTriggerType() API for more information on configuration values.

uint32 transferMode	Specifies how the DMA reacts to every trigger event. Options: ▪ CYDMA_SINGLE_DATA_ELEMENT ▪ CYDMA_ENTIRE_DESCRIPTOR ▪ CYDMA_ENTIRE_DESCRIPTOR_CHAIN See CyDmaSetTransferMode() API for more information on configuration values.
uint32 preemptable	Specifies if the descriptor is preemptable. Options: ▪ CYDMA_PREEMPTABLE ▪ CYDMA_NON_PREEMPTABLE See CyDmaSetPreemptable() API for more information on configuration values.
uint32 actions	Specifies what occurs after a descriptor completes. Options (can be OR'd together): ▪ CYDMA_CHAIN ▪ CYDMA_INVALIDATE ▪ CYDMA_GENERATE_IRQ See CyDmaSetPostCompletionActions() API for more information on configuration values.

Return Value: None

Side Effects: The status register associated with the specified descriptor is reset to zero after this function call. This function also validates the descriptor. This function should not be called while the descriptor is active. This can be checked by calling CyDmaGetStatus().

void CyDmaValidateDescriptor(int32 channel, int32 descriptor)

Description: Validates the specified descriptor after it has been invalidated.

Parameters: channel: Channel used by this function.

descriptor: Specifies descriptor (0 or 1) validated by this function.

Return Value: None

Side Effects: The status register associated with the specified descriptor is reset to zero after this function call.

This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

uint32 CyDmaGetDescriptorStatus(int32 channel, int32 descriptor)

Description: Returns the status of the specified descriptor.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) read by this function.

Return Value: Returns the contents of the specified descriptors status register. Below is a table of defines for accessing the data in the register.

Parameter Value (Mask)	Description	Bits
CYDMA_TRANSFER_INDEX	Number of data elements transferred by the descriptor. Counts up from 0 to Number of data elements. ▪ When a descriptor is done (Response is DONE), the field is set to '0' when the descriptor is valid and the field is set to Number of data elements when Invalidate descriptor is selected for this descriptor. ▪ When a descriptor is not done (Response is NO_ERROR), the field reflects the progress of a data transfer. ▪ In case of erroneous behavior (RESPONSE is neither DONE or NO_ERROR), the field is not updated, but keeps its value to ease debugging. At descriptor initialization, this field should be set to zero. This is done by CyDmaSetConfiguration(), CyDmaValidateDescriptor(), and DMA_Start() API functions. This field allows software to read the progress of the data transfer. Note that descriptor source and destination addresses represent base addresses and are not modified during data transfer. However, this field is modified during data transfer and provides an offset wrt. the base addresses.	[15:0]
CYDMA_RESPONSE	Response/status codes for the descriptor, see table below for more detail.	[18:16]
CYDMA_VALID	Indicates if descriptor is valid. '0' - Invalid. Cannot be used for a data transfer. An attempt to use invalid descriptor will result in channel disabling, an INVALID_DESCR response code and the interrupt bit set. DMA transfer engine sets this field to '0' when a descriptor is done, but only if Invalidate descriptor option is selected. At descriptor initialization, this field should be set to zero. This is done by CyDmaSetConfiguration(), CyDmaValidateDescriptor(), and DMA_Start() API functions.	[31]

Response code meaning, only one response can be set at a time. To test for a response code the following syntax can be used:

```
if((CyDmaGetDescriptorStatus() & CYDMA_RESPONSE) == CYDMA_NO_ERROR)
```

Response code	Description
CYDMA_NO_ERROR	The descriptor is either unused or not yet completed.
CYDMA_DONE	Descriptor is done without errors.
CYDMA_SRC_BUS_ERROR	There was an error loading the value from the source location.
CYDMA_DST_BUS_ERROR	There was an error storing the value at the destination location.

CYDMA_SRC_MISAL	Misalignment of source address. This occurs on a 2-byte transfer (16 bits) that is not 2-byte aligned (2-byte aligned addresses end with 0b0), or on a 4-byte transfer (32 bits) that is not 4-byte aligned (4-byte aligned addresses end with 0b00). If this is set, the channel is disabled and the descriptor invalidated.
CYDMA_DST_MISAL	Misalignment of destination address. This occurs on a 2-byte transfer (16 bits) that is not 2-byte aligned (2-byte aligned addresses end with 0b0), or on a 4-byte transfer (32bits) that is not 4-byte aligned (4-byte aligned addresses end with 0b00). If this is set the channel is disabled and the descriptor invalidated.
CYDMA_INVALID_DESCR	Active channel has an invalid descriptor. The channel is disabled and interrupt bit is set for this error response.

Side Effects: None

*void CyDmaSetSrcAddress(int32 channel, int32 descriptor, void * srcAddress)*

Description: Configures the source address for the specified descriptor.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

srcAddress: Address of DMA transfer source.

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

*void * CyDmaGetSrcAddress(int32 channel, int32 descriptor)*

Description: Returns the source address for the specified descriptor, set by CyDmaSetSrcAddress().

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) used by this function.

Return Value: Source address written to specified descriptor.

Side Effects: None

*void CyDmaSetDstAddress(int32 channel, int32 descriptor, void * dstAddress)*

Description: Configures the destination addresses for the specified descriptor.

Parameters: channel: Channel used by this function.

Return Value: descriptor: Descriptor (0 or 1) modified by this function.

dstAddress: Address of DMA transfer destination.

None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

*void * CyDmaGetDstAddress(int32 channel, int32 descriptor)*

Description: Returns the destination address for the specified descriptor set by CyDmaSetDstAddress().

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) used by this function.

Return Value: Destination address written to specified descriptor.

Side Effects: None

void CyDmaSetDataElementSize(int32 channel, int32 descriptor, uint32 dataElementSize)

Description: Sets the data element size for the specified descriptor.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

dataElementSize: Specifies the size of the data element. The DMA transfer engine transfers one data element at a time. How these transfers occur is controlled by the transfer mode.

Parameter Value	Description
CYDMA_BYTE	Each data element is one byte.
CYDMA_HALFWORD	Each data element is a halfword (2 bytes).
CYDMA_WORD	Each data element is a word (4 bytes).

Note The size of the source and destination can be configured to use the data element size, or a word, this is set in CyDmaSetSrcDstTransferWidth().

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

uint32 CyDmaGetDataElementSize(int32 channel, int32 descriptor)

Description: Returns the data element size configured for the specified descriptor, set by CyDmaSetDataElementSize() or CyDmaSetConfiguration().

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

Return Value: See CyDmaSetDataElementSize() for return value meaning.

Side Effects: None

void CyDmaSetNumDataElements(int32 channel, int32 descriptor, int32 numDataElements)

Description: Sets the number of data elements to transfer for specified descriptor.

Parameters: channel: Channel used by this function

descriptor: Descriptor (0 or 1) modified by this function.

numDataElements: Total number of data elements this descriptor transfers. Valid ranges are 1 to 65536.

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

int32 CyDmaGetNumDataElements(int32 channel, int32 descriptor)

Description: Returns the number of data elements to be transferred. Only reflects the value written by CyDmaSetNumDataElements() or CyDmaSetConfiguration().This does not reflect how many elements have been transferred. For that information use the CyDmaGetDescriptorStatus() function.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

Return Value: Number of data elements to transfer.

Side Effects: None

void CyDmaSetSrcDstTransferWidth(int32 channel, int32 descriptor, uint32 transferWidth)

Description: Sets the width of the source and destination. The DMA can either read and write data from the source and destination at the size specified by CyDmaSetDataElementSize() or by a word (4bytes). This also determines how many bytes the addresses are incremented if increment source and destination address are turned on.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

transferWidth: Specifies the width of the source and destination.

Parameter Value	Description
CYDMA_ELEMENT_ELEMENT	Source and destination widths are set by the data element size.
CYDMA_ELEMENT_WORD	Source width is set by data element size. Destination width is a word (4bytes). If the source width is smaller than the destination width the upper bytes of the destination are written with zeros.
CYDMA_WORD_ELEMENT	Source width is a word (4 bytes). Destination width is set by data element size. If the source width is larger than the destination width, the upper bytes of the source are ignored during the transaction.
CYDMA_WORD_WORD	Both source and destination widths are words. However, the data element size still has an effect in this mode. For example, if the data element size is set to a byte, then the upper three bytes of destination will be padded with zeros, and the upper three bytes of the source will be ignored during the transaction.

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

uint32 CyDmaGetSrcDstTransferWidth(int32 channel, int32 descriptor)

Description: Returns the width of the source and destination, as set by CyDmaSetSrcDstTransferWidth(), or CyDmaSetConfiguration().

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

Return Value: Width of source and destination. See CyDmaSetSrcDstTransferWidth() for more information.

Side Effects: None

void CyDmaSetAddressIncrement(int32 channel, int32 descriptor, uint32 addressIncrement)

Description: Sets whether the source or destination addresses are incremented after the transfer of each data element. The amount that the source and destination address are incremented is determined by the CyDmaSetSrcDstTransferWidth() function. The addresses will either be incremented by the data element size or by a word (4 bytes).

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

addressIncrement: Bit field of defines that can be OR'd together.

Parameter Value	Description
CYDMA_INC_SRC_ADDR	Increment the Source address.
CYDMA_INC_DST_ADDR	Increment the Destination address.
CYDMA_INC_NONE	No address increment.

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

uint32 CyDmaGetAddressIncrement(int32 channel, int32 descriptor)

Description: Returns address increment settings as set by CyDmaSetAddressIncrement(), or CyDmaSetConfiguration().

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

Return Value: Address increment settings. Refer to CyDmaSetAddressIncrement() for information on return value meanings.

Side Effects: None

void CyDmaSetTriggerType(int32 channel, int32 descriptor, uint32 triggerType)

Description: Sets the type of input trigger for the DMA. For level sensitive triggers sets how long the DMA waits for the trigger to go low (deactivate) before triggering the channel again.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

triggerType: Type of DMA trigger.

Parameter Value	Description
CYDMA_PULSE	Input trigger is a pulse

CYDMA_LEVEL_FOUR
CYDMA_LEVEL_EIGHT
	Input trigger is a level. Use the pseudo code below to understand how the waiting works.
	`/* Wait for transaction to end. */`
	`while(!end_transaction);`
	
	`/* After transaction finishes. */`
	`/* If trigger is high. */`
	`if (trig == 1)`
	`{`
	`  /* Wait for selected time or until trig goes low. */`
	`  do`
	`  {`
	`  /* Increment on SYSCLK. */`
	`  num++;`
	`  } while(num != trigType && trig == 1)`
	`}`
	`/* Wait for new trigger. */`
	`while(trig == 0);`
	`/* trigger new transaction. */`
	`trigger;`

CYDMA_PULSE_UNKNOWN	Input trigger is a pulse of unknown length. The channel will not trigger again until the pulse goes low. **Note** This setting can cause the DMA to lock up if the trigger signal never goes low.

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

uint32 CyDmaGetTriggerType(int32 channel, int32 descriptor)

Description: Returns the trigger type settings as set by CyDmaSetTriggerType(), or CyDmaSetConfiguration().

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

Return Value: Trigger type settings, see CyDmaSetTriggerType() for return value details.

Side Effects: None

void CyDmaSetTransferMode(int32 channel, int32 descriptor, uint32 transferMode)

Description: Sets the DMA transfer mode. This determines how the DMA reacts to every trigger event.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

transferMode: Specifies how the DMA reacts to a trigger event.

Parameter Value	Description
CYDMA_SINGLE_DATA_ELEMENT	Each trigger causes the DMA to transfer a single data element.
CYDMA_ENTIRE_DESCRIPTOR	Each trigger automatically transfers all of the data elements, set by CyDmaSetNumDataElements(), one data element at a time.
CYDMA_ENTIRE_DESCRIPTOR_CHAIN	Each trigger automatically transfers all of the data elements, set by CyDmaSetNumDataElements(), one data element at a time. Upon completion the next descriptor is automatically triggered. **Note** DMA_CHAIN must be set in CyDmaPostCompletionActions() for this mode to work.

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

uint32 CyDmaGetTransferMode(int32 channel, int32 descriptor)

Description: Returns the transfer mode for the specified descriptor as set by CyDmaSetTransferMode() or CyDmaSetConfiguration().

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

Return Value: DMA transfer mode setting. Refer to CyDmaSetTransferMode() for more information.

Side Effects: None

void CyDmaSetPreemptable(int32 channel, int32 descriptor, uint32 preemptable)

Description: Specifies if the descriptor is preemptable.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

preemptable: Determines if the descriptor is preemptable.

Parameter Value	Description
CYDMA_PREEMPTABLE	The descriptor is preemptable.
CYDMA_NON_PREEMPTABLE	The descriptor is non-preemptable.

If the descriptor is preemptable, it allows channels of higher priority to interrupt this channel transfer. Once the channel has been interrupted, it finishes the current transfer of a single data element, and then goes pending. After it goes pending it must wait until it is the highest priority pending channel before it runs again.

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

uint32 CyDmaGetPreemptable(int32 channel, int32 descriptor)

Description: Returns whether or not the descriptor is preemptable.

Parameters: channel: Channel used by this function

descriptor: Descriptor (0 or 1) modified by this function.

Return Value: DMA transfer mode setting. Refer to CyDmaSetPreemptable() for more information.

Side Effects: None

void CyDmaSetPostCompletionActions(int32 channel, int32 descriptor, uint32 actions)

Description: Specifies what occurs after a descriptor completes.

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

actions: Bit field of defines that can be OR'd together.

Parameter Value	Description
CYDMA_CHAIN	On completion of descriptor chain to the next descriptor.
CYDMA_INVALIDATE	Invalidate the descriptor when it completes.
CYDMA_GENERATE_IRQ	On completion of descriptor generate an interrupt request.
CYDMA_NONE	No actions will occur after a descriptor completes.

Return Value: None

Side Effects: This function should not be called when the specified descriptor is active in the DMA transfer engine. This can be checked by calling CyDmaGetStatus().

uint32 CyDmaGetPostCompletionActions(int32 channel, int32 descriptor)

Description: Returns the post descriptor action settings as set by CyDmaSetPostCompletionActions() or CyDmaSetConfiguration().

Parameters: channel: Channel used by this function.

descriptor: Descriptor (0 or 1) modified by this function.

Return Value: Post descriptor actions. See CyDmaSetPostCompletionActions() for return details.

Side Effects: None

Global Variables

Variable	Description
DMA_initVar	This global variable indicates whether the component has been initialized. The variable is initialized to 0 and set to 1 the first time DMA_Start() is called. This allows the component to restart without re-initialization after the first call to the DMA_Start() routine. If re-initialization of the component is required, then the DMA_Init() function can be called before the DMA_Start() function.

Sample Firmware Source Code

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the "Find Example Project" topic in the PSoC Creator Help for more information.

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The DMA component does not have any specific deviations.

API Memory Usage

The component memory usage varies significantly depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with an associated compiler configured in Release mode with optimization set for Size. For a specific design, the map file generated by the compiler can be analyzed to determine the memory usage.

PSoC 4 (GCC)

Configuration	Flash Bytes	SRAM Bytes
DMA Channel	116	1
DMA Library (cy_dmac)	1118	4 x CH_NR + 5
Typical [2]	324	4 x CH_NR + 1

2. Includes a set of APIs for typical use case. The DMA transfer is configured using Configure dialog and started from user firmware via DMA_Start() API call.

Functional Description

Overview

The DMA controller transfers data between source and destination locations. The source location can be Flash, SRAM or another component, i.e. UART, ADC and many others. The destination location can be SRAM or another component. The transfer is initiated by a trigger that can come from a DMA channel (including itself), a peripheral, or the CPU. The DMA supports multiple channels (consult the part specific datasheet to determine how many channels for a particular part) each of which may have a dedicated trigger and be individually enabled.

The DMA can transfer up to 65,536 data elements of configurable size. The data element size can be a byte, halfword (2 bytes), or word (4 bytes).

The controller supports three operation modes that determine how the DMA operates on a single trigger signal. The modes are:

- Single data element per trigger
- Entire descriptor per trigger
- Entire descriptor chain per trigger

The data transfer specifics, such as source and destination address locations and the size of the transfer, are specified by a descriptor structure. Each channel has dedicated descriptor structure.

The DMA controller provides Active/Sleep functionality and is not available in Deep-Sleep and Hibernate power modes.

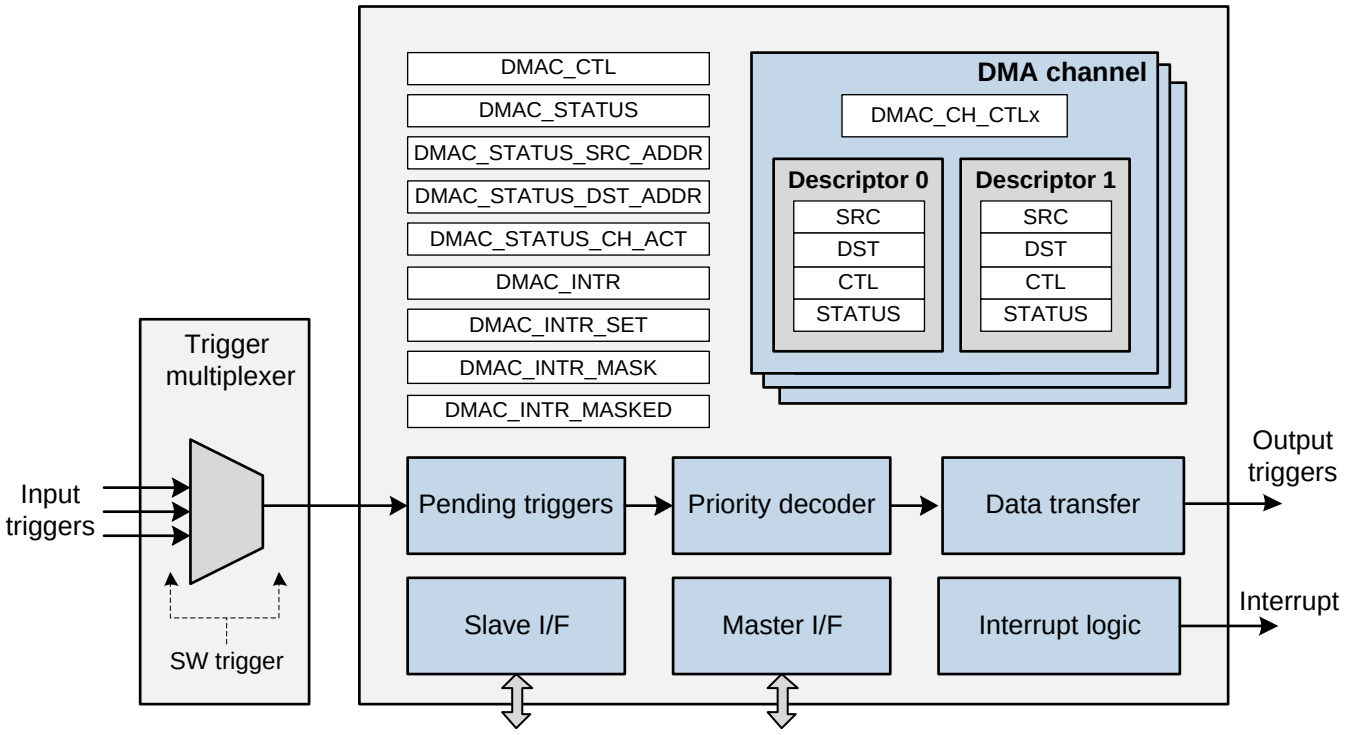
Block Diagram and Configuration

Figure 4 gives an overview of the DMA controller at a block level.

- Trigger multiplexer block is outside the DMA controller and connects each channel to one specific trigger in the design. A logical '1' on a selected trigger line indicates an activated trigger and results in a channel data transfer.
- Pending triggers keep track of activated triggers by locally storing them in pending bits. This is essential, because multiple channel trigger may be activated simultaneously, whereas only one channel can be served by the data transfer engine at a time. This block enables the use of both level-sensitive and pulse-sensitive triggers. See tr_in input description for more information on trigger signal. The pending triggers are registered in the status register (STATUS_CH_ACT).
- Priority decoder determines the highest priority channel with an active trigger. The priorities are set for each channel using the PRIO field of the channel control register (CH_CTL).

- Data transfer engine is responsible for the data transfer from a source location to a destination location. When idle, the data transfer engine is ready to accept the highest priority activated channel. The configuration of the data transfer is specified by the descriptor.
- Descriptor 0/1. Every channel has two descriptor structures for double buffering. A descriptor is a set of four 32-bit registers that describe the configuration of the transfer. The descriptor comprises of information regarding the source and destination address, the mode of transfer to be used and other specifics related to the transfer. See Descriptors section for more details.
- Master I/F is an AHB-Lite bus master, which allows the DMA controller to initiate AHB-Lite data transfers to the source and destination locations.
- Slave I/F is an AHB-Lite bus slave, which allows the PSoC main CPU to access the DMA controller's control/status registers and to access the descriptor structure.
- Interrupt logic includes interrupt status for each of the channels.

Figure 4. DMA controller block diagram



Descriptors

The data transfer between a source and a destination in a channel is described/configured using a descriptor structure. Each DMA channel has two descriptors – descriptor 0 (PING) and descriptor 1 (PONG). The active descriptor is set by the PING_PONG bit in the channel control register (DMAC_CH_CTL).The two descriptors are identical and each structure consists of four 32-bit words:

- Source address (SRC)
- Destination address (DST)
- Control word (CTL)
- Status word (STATUS)

Descriptor Source and Destination Addresses

Source address specifies base address of source location. This is configured by calling DMA_Start(), DMA_SetSrcAddress() or CyDmaSetSrcAddress().

Destination address specifies base address of destination location. This is configured by calling DMA_Start(), DMA_SetDstAddress() or CyDmaSetDstAddress().

To transfer from/to a variable, set source/destination address to the variable address. For example:

```
int16 sample;
...
DMA_SetDstAddress(0, (void *) &sample);
```

For an array, set source/destination address to the address of the first array element. For example:

```
#define BUFFER_SIZE  64
...
uint32 srcBuffer[BUFFER_SIZE];
uint32 dstBuffer[BUFFER_SIZE];
...
DMA_Start((void *)srcBuffer, (void *)dstBuffer);
```

If the source/destination location is another component, i.e. ADC or UART, refer to its datasheet for details which register should be used as a source/destination location for DMA. For example to transfer from a RAM array to SCB UART TX buffer:

```
uint8 uartData[BUFFER_SIZE];
...
DMA_Start((void *)uartData, (void *) UART_TX_FIFO_WR_PTR);
```

Note For readability, the instance name used in all code examples is "DMA".

Descriptor Control Word

Descriptor control word configures the transfer specifics as shown in the following table:

Field	Description
Data element size	Specifies the size of the data element. The data element can be 8-, 16- or 32-bit width. See CyDmaSetDataElementSize() API for more information on configuration values. The field can be configured using Data element size parameter, DMA_SetConfiguration(), DMA_SetDataElementSize(), CyDmaSetConfiguration(), or CyDmaSetDataElementSize() API call.
Number of data elements	Total number of data elements this descriptor transfers before it is completed. Valid ranges are 1 to 65536. In a typical use case, this setting is the buffer size of a transfer. The field can be configured using Number of data elements to transfer parameter, DMA_SetConfiguration(), DMA_SetNumDataElements(), CyDmaSetConfiguration(), or CyDmaSetNumDataElements() API call.
Source / destination widths	Specifies the width of the source and destination locations. The DMA can either read and write data from the source and destination at the size specified by data element size or by a word (4bytes). This also determines how many bytes the addresses are incremented if increment source and destination address are turned on. See CyDmaSetSrcDstTransferWidth() API for more information on configuration values. The field can be configured using Source and destination transfer width parameter, DMA_SetConfiguration(), DMA_SetSrcDstTransferWidth(), CyDmaSetConfiguration(), or CyDmaSetSrcDstTransferWidth() API call.
Address increment	Specifies whether the source or destination addresses are incremented after the transfer of each data element. Enable this option when the source/destination of the data is a buffer and each data element needs to be fetched from subsequent locations in the memory. See CyDmaSetAddressIncrement() API for more information on configuration values. The field can be configured using Increment source address and Increment destination address parameters, DMA_SetConfiguration(), DMA_SetAddressIncrement(), CyDmaSetConfiguration(), or CyDmaSetAddressIncrement() API call.
Trigger type	Specifies the type of input trigger for the DMA. Level sensitive triggers can be configured to wait a number of system clocks (SYSCLK) for the trigger to go low (deactivate) before triggering the channel again. If during this time the trigger goes low, the DMA channel will be retriggered on the next logic HIGH/1 trigger. If the trigger doesn't go low the DMA waits the specified number of clock cycles before retriggering. See tr_in input section for more information on trigger signal behavior. The field can be configured using Trigger type parameter, DMA_SetConfiguration(), DMA_SetTriggerType(), CyDmaSetConfiguration(), or CyDmaSetTriggerType() API call.

Field	Description
Transfer mode	Specifies how the DMA reacts to every trigger event. The DMA can be configured to transfer one data element per trigger, entire descriptor (elements data elements) per trigger or all data elements of current descriptor and automatically trigger next descriptor. See CyDmaSetTransferMode() API for more information on configuration values. The field can be configured using Transfer mode parameter, DMA_SetConfiguration(), DMA_SetTransferMode(), CyDmaSetConfiguration(), or CyDmaSetTransferMode() API call.
Preemptable	Specifies if the descriptor is preemptable. If disabled, the current transfer as defined by transfer mode field is allowed to complete undisturbed. If enabled, the current transfer can be preempted/interrupted by a DMA channel of higher priority. When this channel is preempted, it is set as pending and will run the next time its priority is the highest. See CyDmaSetPreemptable() API for more information on configuration values. The field can be configured using Descriptor 0/1 is preemptable parameter, DMA_SetConfiguration(), DMA_SetPreemptable(), CyDmaSetConfiguration(), or CyDmaSetPreemptable() API call.
Post completion actions	Specifies what occurs after a descriptor completes. The descriptor is defined completed after transfer all data elements as specified by number of data elements field. The settings are (can be selected simultaneously): ▪ Chain to next descriptor ▪ Invalidate descriptor ▪ Generate interrupt See CyDmaSetPostCompletionActions() API for more information on configuration values. The field can be configured using Post completion actions parameter, DMA_SetConfiguration(), DMA_SetPostCompletionActions(), CyDmaSetConfiguration(), or CyDmaSetPostCompletionActions() API call.

Descriptor Status Word

Descriptor status word provides descriptor validity bit, response code and the index of the current data transfer. See CyDmaGetDescriptorStatus() API for more information on descriptor status word fields.

Performance

Single element transfer mode

It requires 12 clock cycles (from “tr_in” to “tr_out”) to complete one data element transfer assuming there are no wait states on AHB-Lite infrastructure for read/write transfers. The equation for one data element is:

No of cycles = 12 + LOAD wait states + STORE wait states

Entire descriptor and entire descriptor chain transfer mode

It requires 12 clock cycles for the first data transfer and 3 cycles per transfer for remaining data elements (This is also assuming no wait states on AHB-Lite infrastructure). So equation looks like the following for transferring 'N' data elements:

No of cycles = (12 + LOAD wait states + STORE wait states) +
(N-1)(3 + LOAD wait states + STORE wait states)

Load wait states are incurred when the transfer source is being accessed by the CPU. Store wait states are incurred when the transfer destination is being accessed by the CPU. The DMA must wait until the CPU finishes before it can transfer data.

Interrupt Service Routine

If all the data transfers as specified by a descriptor channel structure have completed, an interrupt request may be generated by the DMA transfer engine.

This interrupt request is shared among all DMA channels. The preferred method to access this interrupt is to register a user defined callback function to be called by the DMA ISR. Every DMA channel uses dedicated callback function. The DMA ISR determines which channel has triggered the interrupt and invokes the associated function. It also clears the pending interrupts. To set a callback function, use DMA_SetInterruptCallback() or CyDmaSetInterruptCallback().The function should contain code to process an interrupt for the associated DMA channel.

The following is a C language example demonstrating how to define and set a callback function. This example assumes the component has been placed in a design with the instance name "DMA".

Callback function prototype example:

```
void MyIntrCallback(void);
```

Callback function definition example:

```
void MyIntrCallback(void)
{
    /* Code to process the interrupt */
}
```

Setting callback function example:

```
int main()
{

    DMA_SetInterruptCallback(&MyIntrCallback);

    /* Enable interrupts */
    CyIntEnable(CYDMA_INTR_NUMBER);
    CyGlobalIntEnable;
}
```

An alternative method is to use the CyDmaSetInterruptVector() API function. This will allow you to set a customer interrupt vector for the DMA interrupt. To determine which DMA channel generated an interrupt, call CyDmaGetInterruptSourceMasked(). Once asserted, the interrupt request remains high until cleared by CyDmaClearInterruptSource().

Note Other components with embedded DMA may register their callback functions to be called from the DMA ISR. Therefore, changing the ISR will prevent such components from operation. Open DMA Editor tab under Design-Wide Resources (DWR) to view all DMA components that have been directly placed in the design, as well as all the DMA components “inside” placed components.

The following is a C language example of the Interrupt Service Routine for the DMA interrupt:

```
/* ISR Prototype */
CY_ISR_PROTO(DmaInterruptHandler);

/* ISR Implementation */
CY_ISR(DmaInterruptHandler)
{
    uint32 intr;

    /* Determine the interrupt source */
    intr = CyDmaGetInterruptSourceMasked();

    /* Clear interrupt request */
    CyDmaClearInterruptSource(intr);

    /* Code to process the interrupt */
}

int main()
{
    /* Set interrupt vector for the DMA interrupt */
    CyDmaSetInterruptVector(&DmaInterruptHandler);

    /* Enable interrupts */
    CyIntEnable(CYDMA_INTR_NUMBER);
    CyGlobalIntEnable;
}
```

Register Protection

A number of DMA API functions use read/modify/write to update selected bits in registers; Since these APIs do not disable interrupts during the read/modify/write sequence, it is possible for an interrupt service routine to fire up after the read but before the write. If the interrupt service routine were to also modify that register, then the update could get lost when the read/modify/write sequence completes, since it will have read and continue to use the old value.

It is your responsibility to make your application interrupt safe. One way to do that is to disable interrupts during conflicting read/modify/write requests.

The table below lists all APIs that are not interrupt safe along with registers they modify.

Function	Register [3]
CyDmaChEnable()	DMAC_CH_CTLx
CyDmaChDisable()	DMAC_CH_CTLx
CyDmaSetPriority()	DMAC_CH_CTLx
CyDmaSetNextDescriptor()	DMAC_CH_CTLx
CyDmaSetDataElementSize()	DMAC_DESCRx_PING_CTL / DMAC_DESCRx_PONG_CTL
CyDmaSetNumDataElements()	DMAC_DESCRx_PING_CTL / DMAC_DESCRx_PONG_CTL
CyDmaSetAddressIncrement()	DMAC_DESCRx_PING_CTL / DMAC_DESCRx_PONG_CTL
CyDmaSetTransferMode()	DMAC_DESCRx_PING_CTL / DMAC_DESCRx_PONG_CTL
CyDmaSetPreemptable()	DMAC_DESCRx_PING_CTL / DMAC_DESCRx_PONG_CTL
CyDmaSetPostCompletionActions()	DMAC_DESCRx_PING_CTL / DMAC_DESCRx_PONG_CTL
CyDmaSetTriggerType()	DMAC_DESCRx_PING_CTL / DMAC_DESCRx_PONG_CTL

Component Debug Window

PSoC Creator allows you to view debug information about components in your design. Each component window lists the memory and registers for the instance. For detailed hardware registers descriptions, refer to the appropriate device technical reference manual.

To open the Component Debug window:

1. Make sure the debugger is running or in break mode.
2. Choose **Windows > Components…** from the **Debug** menu.
3. In the Component Window Selector dialog, select the component instances to view and click **OK**.

3. x - DMA channel number.

The selected Component Debug window(s) will open within the debugger framework. Refer to the "Component Debug Window" topic in the PSoC Creator Help for more information.

The following registers are displayed in the DMA component debug window.

Register Name	Description [4]
DMAC_CTL	DMA controller control register.
DMAC_STATUS	DMA controller status register.
DMAC_STATUS_SRC_ADDR	Source address currently being used by the DMA controller.
DMAC_STATUS_DST_ADDR	Destination address currently being used by the DMA controller.
DMAC_STATUS_CH_ACT	Channel activation status.
DMAC_CH_CTLx	Channel x control register.
DMAC_DESCRx_PING_SRC	Descriptor 0 source address location for channel x.
DMAC_DESCRx_PING_DST	Descriptor 0 destination address location for channel x.
DMAC_DESCRx_PING_CTL	Descriptor 0 control word for channel x.
DMAC_DESCRx_PING_STATUS	Descriptor 0 status word for channel x.
DMAC_DESCRx_PONG_SRC	Descriptor 0 source address location for channel x.
DMAC_DESCRx_PONG_DST	Descriptor 0 destination address location for channel x.
DMAC_DESCRx_PONG_CTL	Descriptor 0 control word for channel x.
DMAC_DESCRx_PONG_STATUS	Descriptor 0 status word for channel x.
DMAC_INTR	Interrupt register.
DMAC_INTR_SET	Interrupt set register.
DMAC_INTR_MASK	Interrupt mask. Mask for corresponding field in INTR register.
DMAC_INTR_MASKED	Interrupt masked register. Bitwise AND between the interrupt request and mask registers.

Resources

The DMA component utilizes a DMA channel of the device.

4. x - DMA channel number.

Design-Wide cy_dmac Component Changes

This section lists the major changes in the design-wide cy_dmac component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.10	Changed default priority of DMA interrupt from 0 (the highest priority) to 3.	Consistency with the default priority of all other interrupts. Use CyIntSetPriority() to set the desired DMA interrupt priority: `CyIntSetPriority(CYDMA_INTR_NUMBER, priority);`
	Modified CyDmaTrigger() function for new devices support.	New devices support.
1.0.a	Updated API description.	Corrected description of the CyDmaSetPostCompletionActions() function.
1.0	Initial component version.	

DMA Channel Component Changes

This section lists the major changes in the DMA Channel component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.0.b	Updated cy_dmac component to version 1.10.	See Design-Wide cy_dmac Component Changes.
1.0.a	Updated cy_dmac API description.	See Design-Wide cy_dmac Component Changes.
1.0	Initial component version.	

© Cypress Semiconductor Corporation, 2015. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control, or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.