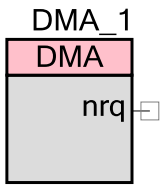
Direct Memory Access (DMA)

1.50



Features

- 24 channels
- Eight priority levels
- 128 Transaction Descriptors (TDs)
- 8-, 16-, and 32-bit data transfers
- Configurable source and destination addresses
- Support for endian compatibility
- Can generate an interrupt when data transfer is complete
- DMA Wizard to assist with application development

General Description

The DMA component allows data transfers to and from memory, components, and registers. The controller supports 8-, 16-, and 32-bit wide data transfers, and can be configured to transfer data between a source and destination that have different endianess. TDs can be chained together for complex operations.

The DMA can be triggered based on a level or rising edge signal. See the Hardware Request parameter selection for more details.

When to Use a DMA component

A DMA component is useful when you want to unburden the CPU of the task of transferring data or when data needs to be transferred in a predictable way that can be set up beforehand. A few basic use cases are:

- Memory to memory
- Memory to peripheral
- Peripheral to memory
- Peripheral to peripheral

TDs can be executed individually or chained together to perform complex transfers.

DMA Wizard

PSoC Creator provides a DMA Wizard to aid in the quick and accurate development of applications that use DMA. The wizard guides you through defining your TDs and generates the necessary C code that you can copy and paste into your application.

Launch the wizard from the PSoC Creator **Tools** menu. Refer to the PSoC Creator Help for more information.

PSoC 3 Addresses

In PSoC 3, all locations that are involved in DMA transfers are within the first 64K of memory except for flash. For all locations except flash, the value provided for the upper 16 bits of the address must be 0. The Keil compiler does not recognize addresses outside of the first 64K and instead uses the upper 16 bits to store other information, resulting in the upper bytes being nonzero. For this reason, the upper 16 bits of a pointer to the location cannot be used directly. Please refer to *Generic Pointers in Keil* for more details. For the case of flash, the proper value to use for the upper 16 bits of the address is:

```
HI16(CYDEV_FLS_BASE)
```

This is a specific handling done by the compiler. In order to create code that functions correctly in both PSoC 3 and PSoC 5, you can use the following code style. Assume “src” is a variable in flash and “dst” is a variable in SRAM:

```
#if (defined(__C51__))
    /* PSoC 3 - Source is Flash */
    dmaChan = DMA_1_DmaInitialize(1, 0, HI16(CYDEV_FLS_BASE), 0);
#else
    /* PSoC 5 */
    dmaChan = DMA_1_DmaInitialize(1, 0, HI16(src), HI16(dst));
#endif
```

PSoC 5 SRAM Access

In PSoC 5, the DMA cannot access SRAM from 0x1FFF8000 to 0x1FFFFFFF, but it can access the same memory at 0x20008000 to 0x2000FFFF.

The CPU accesses:

```
0x1FFF8000 - 0x1FFFFFFF C-BUS 32KB
0x20000000 - 0x20007FFF S-BUS 32KB
```

The DMA accesses:

```
0x20000000 - 0x20007FFF S-BUS 32KB
0x20008000 - 0x2000FFFF C-BUS 32KB
```

This remapping is handled automatically by the APIs used to set up a DMA. The parameters passed to the APIs should be the upper and lower 16 bits of the native CPU address, which will be automatically handled by the DMA API. Note that when the DMA engine increments an address, it is only incrementing the lower 16 bits. Therefore, the next address following 0x2000FFFF will be 0x20000000, which results in the memory space still functioning as a contiguous 64-KB block of memory.

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.

nrq – Output

The nrq terminal may be connected to an interrupt, or to a component to notify the component of the completion of the DMA transfer. The DMA generates a pulse of width two bus clocks at the nrq when the DMA transfer is complete.

drq – Input *

The drq terminal is connected to a component can request a DMA transaction.

The drq input is either level- or edge-sensitive. If the drq is level-sensitive the DMA request will continuously occur when drq is asserted. If the drq is edge-sensitive the DMA request must be at least one bus clock cycle wide.

trq – Input *

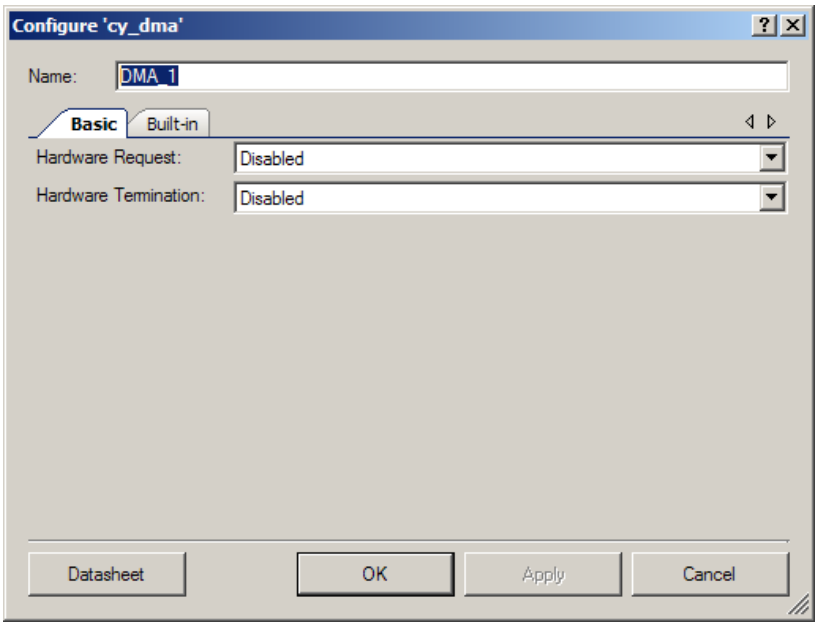
The trq terminal is connected to a component that can terminate a DMA transaction. A component may be asked for data from the DMA when it knows none is available. It uses this signal to terminate the transaction.

When the current TD in the chain is terminated, it will complete as if the transfer count completed. Therefore, whether the transaction is terminated depends on whether there are other TDs in the chain and what type of transaction is defined (for example, ping-pong, circular, auto-repeat, and so on).

This signal is only used when the channel is trying to transfer data. A positive edge on this line is ignored at other times.

Component Parameters

Drag a DMA component onto your design and double click it to open the **Configure** dialog.



The DMA provides the following parameters.

Hardware Request

This parameter configures the type of waveform it should expect to process the trigger to DMA. Any option except **Disabled** adds the drq terminal, which allows a DMA request to be made from hardware. The available options include:

- **Disabled** – The drq terminal is not displayed. In this case, the DMA can be triggered only through the CPU
- **Derived** – Inspects the driver of the drq and, when connected to a fixed function block (I^2C, USB, CAN, and so on), derives the DMA type based on what it is connected to. This automatic assignment is based on information found in the device datasheet. When not connected to a fixed function block, the **Rising Edge** option is used.
- **Rising Edge** – Triggers the DMA on the rising edge of the source signal. Choose this option when the DMA needs to occur based on an event. For example, a DMA that should occur periodically would be configured in rising edge mode and the DRQ signal could be connected to a clock signal set to the appropriate rate.
- **Level** – Selects the source connected to the DMA as a level sensitive request. Choose this option when the DMA should be triggered continuously as long as a particular condition is active. This is typically the case for a DMA that is triggered based on the fill level of a FIFO. This is the typical configuration for use with communication components such as I2S.

Hardware Termination

This parameter can be set to **Enabled** or **Disabled**. **Enabled** adds the trq terminal, which allows a DMA request to be terminated from hardware. When the terminal is disabled, the DMA transfer can be terminated only by CPU request or when the DMA completes the data transfer.

Resources

DRQ	Digital Blocks					API Memory (Bytes)		Pins (per External I/O)
	Datapaths	Macro cells	Status Registers	Control Registers	Counter7	Flash	RAM	
1	N/A	N/A	N/A	N/A	N/A	2332	7	0

Application Programming Interface

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

By default, PSoC Creator assigns the instance name “DMA_1” to the first instance of a component in a given design. You can rename the instance 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.”

APIs per DMA Instance

Function	Description
DMA_DmaInitialize()	Allocates and initializes a DMA channel to be used by the caller.
DMA_DmaRelease()	Frees and disables the DMA channel associated with this instance of the component.

uint8 DMA_DmaInitialize(uint8 burstCount, uint8 requestPerBurst, uint16 upperSrcAddress, uint16 upperDestAddress)

Description: Allocates and initializes a DMA channel to be used by the caller.

Parameters: uint8 burstCount: Specifies the size of bursts (1 to 127) into which this TD should be divided. The burst size should be multiples of the spoke size.

If this value is zero, the whole transfer is done as one burst. In this case, the transfer count parameter for the TD determines the number of bytes to transfer in one burst.

uint8 requestPerBurst: The whole of the data can be split into multiple bursts, if that is required to complete the transaction:

Value	Action
0	All subsequent bursts after the first burst will be automatically requested and carried out
1	All subsequent bursts after the first burst must also be individually requested.

uint16 upperSrcAddress: The upper 16 bits of the source address.

uint16 upperDestAddress: The upper 16 bits of the destination address.

Return Value: uint8: The channel that can be used by the caller for DMA activity. Returns DMA_INVALID_CHANNEL (0xFF) if there are no channels left.

Side Effects: None

void DMA_DmaRelease(void)

Description: Frees the channel associated with this instance of the component. The channel cannot be used again unless DMA_DmaInitialize() is called again.

Parameters: None

Return Value: None

Side Effects: None

DMA Library APIs (Shared by All DMA Instances)

DMA Controller Functions

Function	Description
CyDmacConfigure()	Sets the DMAC Configuration register with the default values.
CyDmacError()	Gets the error bits from the DMAC.
CyDmacClearError()	Clears the error bits in the error register of the DMAC.
CyDmacErrorAddress()	Get the address where the last DMAC error occurred.

void CyDmacConfigure(void)

Description: Creates a linked list of all the TDs to be allocated. This function is called by the startup code; you do not normally need to call it. You could call this function if all of the DMA channels are inactive.

Parameters: None

Return Value: None

Side Effects: None

uint8 CyDmacError(void)

Description: Returns the value of the DMA_ERROR type, which contains the error types for the last failed DMA transaction.

Parameters: None

Return Value: Returns the error data (four bits) from the DMA_ERROR type.

Bit	Define	Description
Bit 3	DMAC_PERIPH_ERR	Set to 1 when a peripheral responds to a bus transaction with an error response. Cleared by writing a 1.
Bit 2	DMAC_UNPOP_ACC	Set to 1 when an access is attempted to an invalid address. Cleared by writing a 1.
Bit 1	DMAC_BUS_TIMEOUT	Set to 1 when a bus timeout occurs. Cleared by writing a 1. Timeout values are determined by the BUS_TIMEOUT field in the PHUBCFG register.

Side Effects: None

void CyDmacClearError(uint8 error)

Description: Clears the error bits in the error register of the DMAC.

Parameters: uint8 error: The bitmask of the error bits to clear in the DMA_ERROR type.

Bit	Define	Description
Bit 3	DMAC_PERIPH_ERR	Set to 1 when a peripheral responds to a bus transaction with an error response. Cleared by writing a 1.
Bit 2	DMAC_UNPOP_ACC	Set to 1 when an access is attempted to an invalid address. Cleared by writing a 1.
Bit 1	DMAC_BUS_TIMEOUT	Set to 1 when a bus timeout occurs. Cleared by writing a 1. Timeout values are determined by the BUS_TIMEOUT field in the PHUBCFG register.

Return Value: None

Side Effects: None

uint32 CyDmacErrorAddress(void)

Description: When a BUS_TIMEOUT, UNPOP_ACC, and PERIPH_ERR occur, the address of the error is written to the error address register and can be read with this function. If there are multiple errors, only the address of the first error is saved.

Parameters: None

Return Value: The address that caused the error.

Side Effects: None

Channel Specific Functions

Function	Description
CyDmaChAlloc()	Allocates a channel of the DMA to be used by the caller.
CyDmaChFree()	Frees a channel allocated by CyDmaChAlloc().
CyDmaChEnable()	Enables the DMA channel for execution.
CyDmaChDisable()	Disables the DMA channel.
CyDmaClearPendingDrq()	Clears a pending DMA data request.
CyDmaChPriority()	Sets the priority of a DMA channel.
CyDmaChSetExtendedAddress()	Sets the high 16 bits of the source and destination addresses.
CyDmaChSetInitialTd()	Set the initial TD for the channel.
CyDmaChSetRequest()	Requests to terminate a chain of TDs or one TD, or start the DMA.
CyDmaChGetRequest()	Checks to see if the CyDmaChSetRequest() request was satisfied.

Function	Description
CyDmaChStatus()	Determines the status of the current TD.
CyDmaChSetConfiguration()	Sets configuration information for the channel.

uint8 CyDmaChAlloc(void)

Description: Allocates a channel from the DMAC to be used in all functions that require a channel handle.

Parameters: None

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

Side Effects: None

cystatus CyDmaChFree(uint8 chHandle)

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

Parameters: uint8 chHandle: The handle previously returned by CyDmaChAlloc(or DMA_DmaInitalize().

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

cystatus CyDmaChEnable(uint8 chHandle, uint8 preserveTds)

Description: Enables the DMA channel. A software or hardware request still must happen before the channel is executed.

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitalize().

uint8 preserveTds: Preserves the original TD state when the TD has completed. This parameter applies to all TDs in the channel.

Value	Action
0	When a TD is completed, the DMAC leaves the TD configuration values in their current state, and does not restore them to their original state.
1	When a TD is completed, the DMAC restores the original configuration values of the TD.

When **preserveTds** is set, the TD slot that equals the channel number becomes RESERVED and that becomes where the working registers exist. So, for example, if you are using CH06 and **preserveTds** is set, you are not allowed to use TD slot 6. That is reclaimed by the DMA engine for its private use.

Note Do not chain back to a completed TD if the **preserveTds** for the channel is set to 0. When a TD has completed **preserveTds** for the channel set to 0, the transfer count will be at 0. If a TD with a transfer count of 0 is started, the TD will transfer an indefinite amount of data.

Take extra precautions when using the hardware request (DRQ) option when the **preserveTds** is set to 0, as you might be requesting the wrong data.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

cystatus CyDmaChDisable(uint8 chHandle)

Description: Disables the DMA channel. Once this function is called, CyDmaChStatus() may be called to determine when the channel is disabled and which TDs were being executed.

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitalize().

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

cystatus CyDmaClearPendingDrq(uint8 chHandle)

Description: Clears pending DMA data request.

Parameters: chHandle: Handle to the dma channel.

Return Value: CYRET_SUCCESS if successful or CYRET_BAD_PARAM.

Side Effects: None

cystatus CyDmaChPriority(uint8 chHandle, uint8 priority)

Description: Sets the priority of a DMA channel. You can use this function when you want to change the priority at run time. If the priority remains the same for a DMA channel, then you can configure the priority in the .cydwr file.

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitalize().

uint8 priority: The priority to set the channel to, 0 to 7.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

cystatus CyDmaChSetExtendedAddress(uint8 chHandle, uint16 source, uint16 destination)

Description: Sets the high 16 bits of the source and destination addresses for the DMA channel (valid for all TDs in the chain).

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitalize().

uint16 source: Upper 16 bit address of the DMA transfer source.

uint16 destination: Upper 16 bit address of the DMA transfer destination.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

cystatus CyDmaChSetInitialTd(uint8 chHandle, uint8 startTd)

Description: Sets the initial TD to be executed for the channel when the CyDmaChEnable() function is called.

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitialize().

uint8 startTd: The index of TD to set as the first TD associated with the channel. Zero is a valid TD index.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

cystatus CyDmaChSetRequest(uint8 chHandle, uint8 request)

Description: Allows the caller to terminate a chain of TDs, terminate one TD, or create a direct request to start the DMA channel.

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitialize().

uint8 request: One of the following constants. Each of the constants is a three-bit value.

Request Values	Description
CPU_REQ	Create a direct request to start the DMA channel
CPU_TERM_TD	Terminate one TD
CPU_TERM_CHAIN	Terminate a chain of TDs

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

cystatus CyDmaChGetRequest(uint8 chHandle)

Description: This function allows the caller of CyDmaChSetRequest() to determine if the request was completed.

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitialize().

Return Value: Returns a three-bit field, corresponding to the three bits of the request, which describes the state of the previously posted request. If the value is zero, the request was completed.

DMA_INVALID_CHANNEL if the handle is invalid.

Side Effects: None

cystatus CyDmaChStatus(uint8 chHandle, uint8 * currentTd, uint8 * state)

Description: Determines the status of the DMA channel.

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitalize().

uint8 * currentTd: The address to store the index of the current TD. Can be NULL if the value is not needed.

uint8 * state: The address to store the state of the channel. Can be NULL if the value is not needed.

Bit 1	STATUS_TD_ACTIVE	0: Channel is not currently being serviced by DMAC
		1: Channel is currently being serviced by DMAC
Bit 0	STATUS_CHAIN_ACTIVE	0: TD chain is inactive; either no DMA requests have triggered a new chain or the previous chain has completed.
		1: TD chain has been triggered by a DMA request

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

cystatus CyDmaChSetConfiguration(uint8 chHandle, uint8 burstCount, uint8 requestPerBurst, uint8 tdDone0, uint8 tdDone1, uint8 tdStop)

Description: Sets configuration information for the channel.

Parameters: uint8 chHandle: A handle previously returned by CyDmaChAlloc() or DMA_DmaInitialize().

uint8 burstCount: Specifies the size of bursts (1 to 127) the data transfer should be divided into. If this value is zero then the whole transfer is done in one burst.

uint8 requestPerBurst: The whole of the data can be split into multiple bursts, if this is required to complete the transaction:

Value	Action
0	All subsequent bursts after the first burst will be automatically requested and carried out
1	All subsequent bursts after the first burst must also be individually requested.

uint8 tdDone0: Selects one of the TERMOUT0 interrupt lines to signal completion. The line connected to the nrq terminal will determine the TERMOUT0_SEL definition and should be used as supplied by *cyfitter.h*

uint8 tdDone1: Selects one of the TERMOUT1 interrupt lines to signal completion. The line connected to the nrq terminal will determine the TERMOUT1_SEL definition and should be used as supplied by *cyfitter.h*

uint8 tdStop: Selects one of the TERMIN interrupt lines to signal to the DMAC that the TD should terminate. The signal connected to the trq terminal will determine which TERMIN (termination request) is used.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **chHandle** is invalid.

Side Effects: None

Transaction Descriptor Functions

Function	Description
CyDmaTdAllocate()	Allocates a TD from the free list for use.
CyDmaTdFree()	Returns a TD back to the free list.
CyDmaTdFreeCount()	Gets the number of free TDs available.
CyDmaTdSetConfiguration()	Sets the configuration for the TD.
CyDmaTdGetConfiguration()	Gets the configuration for the TD.
CyDmaTdSetAddress()	Sets the lower 16 bits of the source and destination addresses.
CyDmaTdGetAddress()	Gets the lower 16 bits of the source and destination addresses.

uint8 CyDmaTdAllocate(void)

Description: Allocates a TD for use with an allocated DMA channel.

Parameters: None

Return Value: Zero-based index of the TD to be used by the caller. Since there are 128 TDs minus the reserved TDs (0 to 23), the value returned would range from 24 to 127 not 24 to 128.

DMA_INVALID_TD is returned if there are no free TDs available.

Side Effects: None

void CyDmaTdFree(uint8 tdHandle)

Description: Returns a TD to the free list.

Parameters: uint8 tdHandle: The TD handle returned by the CyDmaTdAllocate() API

Return Value: None

Side Effects: None

uint8 CyDmaTdFreeCount(void)

Description: Returns the number of free TDs available to be allocated.

Parameters: None

Return Value: The number of free TDs.

Side Effects: None

cystatus CyDmaTdSetConfiguration(uint8 tdHandle, uint16 transferCount, uint8 nextTd, uint8 configuration)

Description: Configures the TD.

Parameters: uint8 tdHandle: A handle previously returned by CyDmaTdAlloc().

uint16 transferCount: The size of the data transfer (in bytes) for this TD. A size of zero will cause the transfer to continue indefinitely. This parameter is limited to 4095 bytes; the TD is not initialized at all when a higher value is passed.

uint8 nextTd: Zero based index of the next Transfer Descriptor in the TD chain. Zero is a valid pointer to the next TD; DMA_END_CHAIN_TD is the end of the chain.

uint8 configuration: Stores the Bit field of configuration bits.

Configuration Options	Description
TD_SWAP_EN	Perform endian swap
TD_SWAP_SIZE4	Swap size = 4 bytes
TD_AUTO_EXEC_NEXT	The next TD in the chain will trigger automatically when the current TD completes.
TD_TERMIN_EN	Terminate this TD if a positive edge on the trq input line occurs. The positive edge must occur during a burst. That is the only time the DMAC will listen for it.
DMA__TD_TERMOUT_EN	When this TD completes, the TERMOUT signal will generate a pulse. Note that this option is instance specific with the instance name followed by two underscores. In this example, the instance name is DMA.
TD_INC_DST_ADR	Increment DST_ADR according to the size of each data transaction in the burst.
TD_INC_SRC_ADR	Increment SRC_ADR according to the size of each data transaction in the burst.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **tdHandle** is invalid.

Side Effects: None

cystatus CyDmaTdGetConfiguration(uint8 tdHandle, uint16 * transferCount, uint8 * nextTd, uint8 * configuration)

Description: Retrieves the configuration of the TD. If a NULL pointer is passed as a parameter, that parameter is skipped. You may request only the values you are interested in.

Parameters: uint8 tdHandle: A handle previously returned by CyDmaTdAlloc().

uint16 * transferCount: The address to store the size of the data transfer (in bytes) for this TD. A size of zero could indicate that the TD has completed its transfer, or that the TD is doing an indefinite transfer.

uint8 * nextTd: The address to store the index of the next TD in the TD chain.

uint8 * configuration: The address to store the Bit field of configuration bits. See CyDmaTdSetConfiguration().

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **tdHandle** is invalid.

Side Effects: If a TD has a transfer count of N and is executed, the transfer count becomes 0. If it is re-executed, the Transfer count of zero will be interpreted as a request for indefinite transfer. Be careful when requesting a TD with a transfer count of zero.

cystatus CyDmaTdSetAddress(uint8 tdHandle, uint16 source, uint16 destination)

Description: Sets the lower 16 bits of the source and destination addresses for this TD only.

Parameters: uint8 tdHandle: A handle previously returned by CyDmaTdAlloc().

uint16 source: The lower 16 address bits of the source of the data transfer.

uint16 destination: The lower 16 address bits of the destination of the data transfer.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if **tdHandle** is invalid.

Side Effects: None

cystatus CyDmaTdGetAddress(uint8 tdHandle, uint16 * source, uint16 * destination)

Description:	Retrieves the lower 16 bits of the source and/or destination addresses for this TD only. If NULL is passed for a pointer parameter, that value is skipped. You may request only the values of interest.
Parameters:	uint8 tdHandle: A handle previously returned by CyDmaTdAlloc().
	uint16 * source: The address to store the lower 16 address bits of the source of the data transfer.
	uint16 * destination: The address to store the lower 16 address bits of the destination of the data transfer.
Return Value:	CYRET_SUCCESS if successful.
	CYRET_BAD_PARAM if **tdHandle** is invalid.
Side Effects:	None

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.

DMA Address Portability

The meaning of the upper address passed to the DMA_DmaInitialize() function or CyDmaChSetExtendedAddress() function is not the same for PSoC 3 and PSoC 5 devices. DMA addresses are not portable between PSoC 3 and PSoC 5 devices. The upper addresses need to be set to the upper 16 bits of the physical source and destination addresses.

The following sections provide example code.

PSoC 3 Devices

For PSoC 3 devices, the upper address values must be set manually because they do not correspond to the upper byte of a pointer. The Keil compiler (used with PSoC 3 devices) uses the upper byte of a pointer to represent the memory type, not the physical address.

To read from flash, the upper address of the source for PSoC 3 devices must be set to this define: `CYDEV_FLS_BASE`.

From RAM to RAM

```
uint8 DMA_1_Chan;
uint8 DMA_1_TD[1];

/* DMA Configuration for DMA_1 */
#define DMA_1_BYTES_PER_BURST 16
#define DMA_1_REQUEST_PER_BURST 1
#define DMA_1_SRC_BASE (CYDEV_SRAM_BASE)
#define DMA_1_DST_BASE (CYDEV_SRAM_BASE)
DMA_1_Chan = DMA_1_DmaInitialize(DMA_1_BYTES_PER_BURST, DMA_1_REQUEST_PER_BURST,
    HI16(DMA_1_SRC_BASE), HI16(DMA_1_DST_BASE));
DMA_1_TD[0] = CyDmaTdAllocate();
CyDmaTdSetConfiguration(DMA_1_TD[0], 128, DMA_INVALID_TD, DMA_1__TD_TERMOUT_EN |
TD_INC_SRC_ADR | TD_INC_DST_ADR);
CyDmaTdSetAddress(DMA_1_TD[0], LO16((uint32)memory1), LO16((uint32)memory2));
CyDmaChSetInitialTd(DMA_1_Chan, DMA_1_TD[0]);
CyDmaChEnable(DMA_1_Chan, 1);
```

From Flash to RAM

```
uint8 DMA_1_Chan;
uint8 DMA_1_TD[1];

/* DMA Configuration for DMA_1 */
#define DMA_1_BYTES_PER_BURST 1
#define DMA_1_REQUEST_PER_BURST 1
#define DMA_1_SRC_BASE (CYDEV_FLS_BASE)
#define DMA_1_DST_BASE (CYDEV_SRAM_BASE)

DMA_1_Chan = DMA_1_DmaInitialize(DMA_1_BYTES_PER_BURST, DMA_1_REQUEST_PER_BURST,
    HI16(DMA_1_SRC_BASE), HI16(DMA_1_DST_BASE));

DMA_1_TD[0] = CyDmaTdAllocate();
CyDmaTdSetConfiguration(DMA_1_TD[0], 128, DMA_INVALID_TD, DMA_1__TD_TERMOUT_EN |
TD_INC_SRC_ADR | TD_INC_DST_ADR);
CyDmaTdSetAddress(DMA_1_TD[0], LO16((uint32)buf1), LO16((uint32)buf2));
CyDmaChSetInitialTd(DMA_1_Chan, DMA_1_TD[0]);
CyDmaChEnable(DMA_1_Chan, 1);
```

From Flash to DAC

```
uint8 DMA_1_Chan;
uint8 DMA_1_TD[1];

/* DMA Configuration for DMA_1 */
#define DMA_1_BYTES_PER_BURST 1
#define DMA_1_REQUEST_PER_BURST 1
#define DMA_1_SRC_BASE (CYDEV_FLS_BASE)
#define DMA_1_DST_BASE (CYDEV_PERIPH_BASE)
DMA_1_Chan = DMA_1_DmaInitialize(DMA_1_BYTES_PER_BURST, DMA_1_REQUEST_PER_BURST,
    HI16(DMA_1_SRC_BASE), HI16(DMA_1_DST_BASE));
DMA_1_TD[0] = CyDmaTdAllocate();
```

```
CyDmaTdSetConfiguration(DMA_1_TD[0], 64, DMA_INVALID_TD, TD_INC_SRC_ADR);
CyDmaTdSetAddress(DMA_1_TD[0], LO16((uint32)fFashMem),
LO16((uint32)VDAC8_1_Data_PTR));
CyDmaChSetInitialTd(DMA_1_Chan, DMA_1_TD[0]);
CyDmaChEnable(DMA_1_Chan, 1);
```

PSoC 5 Devices

For PSoC 5 devices, the upper addresses may be set to the upper 16 bits of the source and destination addresses.

From RAM to RAM

```
uint8 DMA_1_Chan;
uint8 DMA_1_TD[1];

/* DMA Configuration for DMA_1 */
#define DMA_1_BYTES_PER_BURST 1
#define DMA_1_REQUEST_PER_BURST 1
#define DMA_1_SRC_BASE (buffer1)
#define DMA_1_DST_BASE (buffer2)
DMA_1_Chan = DMA_1_DmaInitialize(DMA_1_BYTES_PER_BURST, DMA_1_REQUEST_PER_BURST,
    HI16(DMA_1_SRC_BASE), HI16(DMA_1_DST_BASE));
DMA_1_TD[0] = CyDmaTdAllocate();
CyDmaTdSetConfiguration(DMA_1_TD[0], 128, DMA_INVALID_TD, DMA_1__TD_TERMOUT_EN |
TD_INC_SRC_ADR | TD_INC_DST_ADR);
CyDmaTdSetAddress(DMA_1_TD[0], LO16((uint32)buffer1), LO16((uint32)buffer2));
CyDmaChSetInitialTd(DMA_1_Chan, DMA_1_TD[0]);
CyDmaChEnable(DMA_1_Chan, 1);
```

From Flash to DAC:

```
uint8 DMA_1_Chan;
uint8 DMA_1_TD[1];

/* DMA Configuration for DMA_1 */
#define DMA_1_BYTES_PER_BURST 1
#define DMA_1_REQUEST_PER_BURST 1
#define DMA_1_SRC_BASE (FlashMem)
#define DMA_1_DST_BASE (CYDEV_PERIPH_BASE)
DMA_1_Chan = DMA_1_DmaInitialize(DMA_1_BYTES_PER_BURST, DMA_1_REQUEST_PER_BURST,
    HI16(DMA_1_SRC_BASE), HI16(DMA_1_DST_BASE));
DMA_1_TD[0] = CyDmaTdAllocate();
CyDmaTdSetConfiguration(DMA_1_TD[0], 64, DMA_INVALID_TD, DMA_1__TD_TERMOUT_EN |
TD_INC_SRC_ADR);
CyDmaTdSetAddress(DMA_1_TD[0], LO16((uint32)FlashMem),
LO16((uint32)VDAC8_1_Data_PTR));
CyDmaChSetInitialTd(DMA_1_Chan, DMA_1_TD[0]);
CyDmaChEnable(DMA_1_Chan, 1);
```

Boundary Conditions

A single DMA channel cannot cross a 64-KB boundary. Therefore, if a DMA happens to cross a 64-KB boundary, the operation will fail silently. This is only important for PSoC 5 devices because the memory space of a PSoC 3 device does not span beyond 64 KB of flash or 8 KB of RAM.

When using data structures, you must make sure that they do not cross 64-KB boundaries. This can be done by using the following keywords:

```
__attribute__((section()))
__attribute__((aligned()))
```

Both keywords are described in the help for GCC, in the “Specifying Attributes of Variables” section. Use the section keyword if you need your variables to appear in a particular section and location. Use the aligned keyword to cause the compiler to allocate the variable on a particular boundary.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
1.50.d	Minor datasheet edit.	
1.50.c	Minor datasheet edit.	
1.50.b	Minor datasheet edits and updates	
1.50.a	Added CyDmaClearPendingDrq API	
	Minor datasheet edits and updates	
1.50	Added the ability to pick the DRQ type.	The old functionality (“Derived” in the new version) wasn’t always able to correctly determine the DRQ type so we added the ability to specify it manually. This involved changing the Configure dialog to no longer use the default editor.
	Removed the num_tds parameter.	The number of TDs used depends on the code written. Allowing this number to be specified before equated to a comment and wasn’t guaranteed to reflect the actual number used. This number is no longer displayed in the DWR Editor.
	Added a symbol summary.	Provides overview description of the component.
	Squared the corners of the component shape.	Updated to comply with corporate style guide.
	Updated the CyDmaTdSetConfiguration() function.	API automatically handles the termout signals when the NRQ signal is routed/not routed in the schematic.

Version	Description of Changes	Reason for Changes / Impact
	Added #define DMA__TD_TERMOUT_EN to the header file.	This value may be combined with other TD flags to enable the termout(s) used by the component.
	Removed an assert from uint8 DMA_DmaInitialize()	The assert was obsolete due to other code changes previously made.

© Cypress Semiconductor Corporation, 2010-2016. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC (“Cypress”). This document, including any software or firmware included or referenced in this document (“Software”), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress’s patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage (“Unintended Uses”). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.