AN52705

PSoC® 3 and PSoC 5 - Getting Started with DMA

Author: Anu M D, Lakshmi Natarajan
Associated Project: Yes
Associated Part Family: CY8C38xx/CY8C55xx
Software Version: PSoC® Creator™
Related Application Notes: None

Abstract

AN52705 describes how to use direct memory access (DMA) in PSoC® 3 and PSoC 5. It includes projects that show several different types of DMA configurations that you can use.

Contents

Introduction 2
Basic Concepts of DMA 2
DMA Configuration....3
Channel Configuration....3
TD Configuration 4
Example 1: Point-to-Point Transfer 4
Top Design....5
Channel Configuration....5
TD0 Configuration 6
Operation....6
Example 6
Example 2: Point-to-Array Transfer....6
Top Design....7
Channel Configuration....7
TD0 Configuration 8
Operation....8
Related Resources....8
Example 3: Array-to-Point Transfer....8
Top Design....9
Channel Configuration....10
TD0 Configuration 10
Example 4: Array-to-Array Transfer....10
Top Design....11
Channel Configuration....11
TD0 Configuration 12
Operation....12
Example 5: Ping-Pong Buffer....12
Top Design....13
Channel Configuration....13
TD0 Configuration....13
TD1 Configuration....14
Example 6: Multiplexed Data Buffering....14
Appendix A: DMA Component Hardware Configuration..17
Input/Output Connections of DMA component....17
Setting DMA Channel Priority....17
Appendix B: DMA Wizard Configuration....18
TD Configuration Details....19
Appendix C: DMA Configuration – Important APIs....21
Following are the Major Steps that are used to Configure DMA for Data Transfers.....21
Other Important APIs....22
Appendix D: Example Projects – Test Setup....23
Example 1: Point-to-Point Transfer – Eg1_ADC_DMA_DAC....23
Example 2: Point-to-Array Transfer – Eg2_ADC_DMA_Mem....23
Example 3: Array-to-Point Transfer – Eg3_Mem_DMA_DAC....24
Example 4: Array-to-Array Transfer – Eg4_Mem_DMA_Mem....24
Example 5: Ping-Pong Buffering – Eg5_PingPongBuffer....24

Introduction

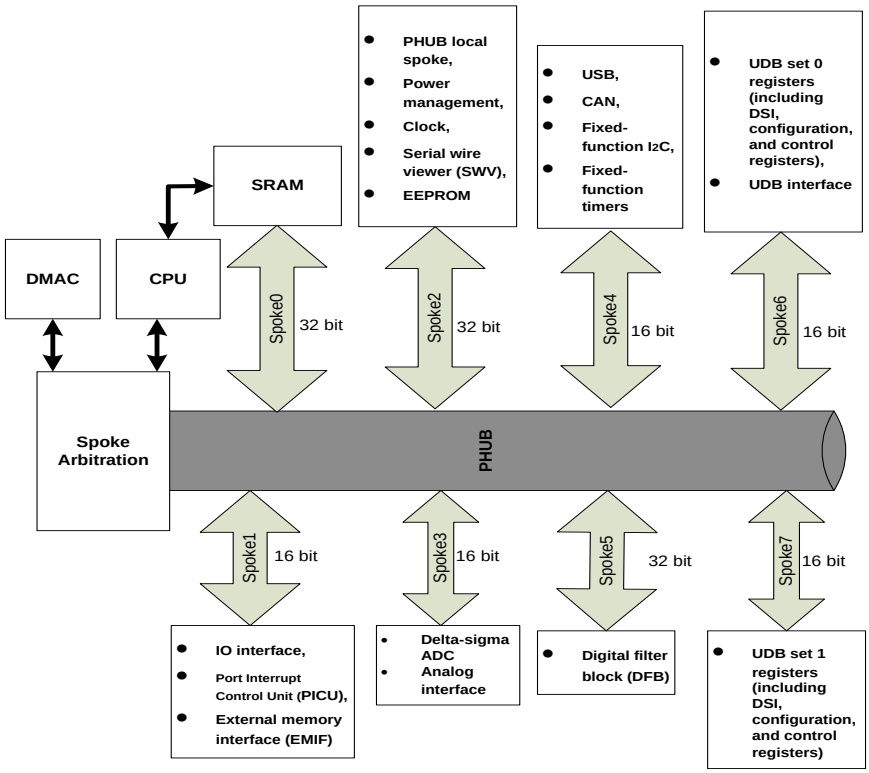
The DMA controller (DMAC) in PSoC 3 and PSoC 5 can send data from a source to a destination without any action by the CPU. This application note describes how a DMAC works and shows the different ways that DMA can be set up. It includes projects that show several different types of DMA configurations, such as:

- Point-to-Point Transfer
- Point-to-Array Transfer
- Array-to-Point Transfer
- Array-to-Array Transfer
- Ping Pong Buffering

Basic Concepts of DMA

Peripheral HUB (PHUB) is a central hub in PSoC 3 and PSoC 5 devices that connects different on-chip peripherals. PHUB contains a DMAC that can move data between on-chip elements without any CPU intervention.

Figure 1. Peripheral HUB



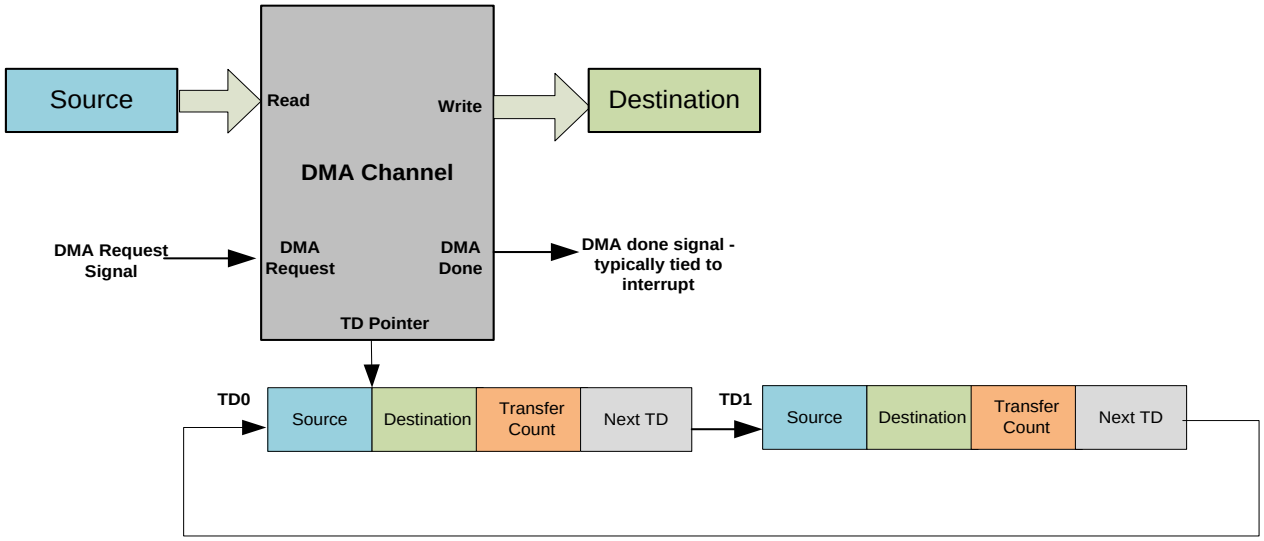
PHUB has eight data buses that are called spokes. Each spoke connects to one or more peripheral blocks as shown in Figure 1. Spokes can have widths of either 16 bits or 32 bits. Peripherals attached to a spoke can have widths of 8 bits, 16 bits, or 32 bits. Both CPU and DMA get access to the peripheral data using PHUB spokes.

The PHUB has two masters the CPU and the DMAC. The CPU and the DMAC can get access to different spokes on the PHUB at the same time. But, if the CPU and DMAC try to get access to the same spoke at the same time, arbitration occurs.

Peripherals on the same spoke can have different widths. The data width of a peripheral is usually less than or equal to the data width of the spoke to which it is attached. However, if a peripheral data width is greater than that of the spoke attached to it, PHUB will transact with the peripheral at the width of the spoke.

The DMAC manages 24 DMA channels and has 128 transaction descriptors (TDs). Each channel is attached to a TD chain, which are one or more TDs attached in a chained configuration. Each TD describes a transfer and contains information such as source address, destination address, transfer count, and next TD pointer.

Figure 2. DMA Channel

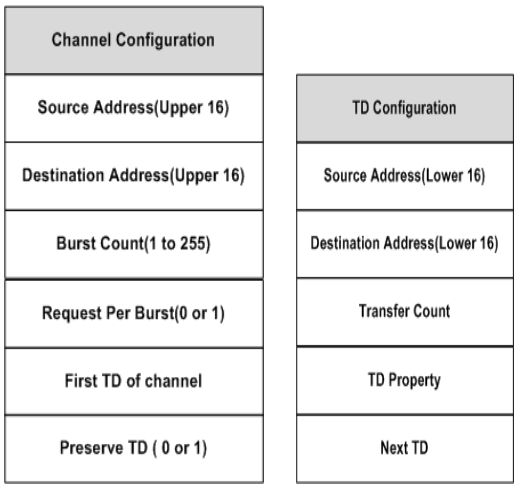


Each DMA channel has a separate DMA request input that activates a transaction for a particular channel. When a DMA transfer is required, a CPU or peripheral asserts the DMA request input of the channel. After it receives a DMA channel request, the DMAC gets access to the spokes attached to the source and destination registers and moves data as configured in channel and TD configuration registers.

DMA Configuration

The various parameters of a DMA transfer are configured using channel configuration and TD configuration registers as described in the following section.

Figure 3. DMA Configuration



Channel Configuration

The source addresses and destination addresses in PSoC 3 and PSoC 5 are 32 bits wide. The upper 16 bits are configured in channel configuration registers and the lower 16 bits are configured in TD configuration registers.

- **Upper Source Address (16 bits)**

  Upper 16-bits of 32-bit source address configured in channel configuration registers

- **Upper Destination Address (16 bits)**

  Upper 16-bits of 32-bit destination address configured in channel configuration registers

- **Lower Source Address (16-bit)**

  Lower 16-bits of 32-bit source address configured in TD configuration registers

- **Lower Destination Address (16-bit)**

  Lower 16-bits of 32-bit destination address configured in TD configuration registers

- **Burst Count (1 to 127)**

  Number of bytes the DMA channel must move from the source to destination before it releases the spoke. The DMAC acquires the spoke for each burst data movement, moves (copies) the specified number of bytes from the source to destination (configured in burst count parameter of channel configuration registers) and then releases the spoke. It re-acquires the spoke during the next burst transfer.

- **Request Per Burst(0 or 1)**

  When multiple bursts are required to finish the DMA data transfer, this parameter determines the nature of the bursts.

  0: All subsequent bursts after the first burst are automatically done without a separate request. (Only the first burst transfer must have a DMA request)

  1: All subsequent bursts after the first burst must have individual requests.

- **Initial TD**

  The channel collects information from the first TD pointer and subsequent TD pointers and keeps it in the TD itself, similar to a linked list. The pointer to the first TD is stored in channel configuration memory and subsequent TD pointers are stored in TD configuration memory, similar to a linked list.

- **Preserve TD(0 or 1)**

  Defines whether to use TD configuration registers or separate PHUB working registers to store intermediate TD states.

  0: Store the intermediate states on top of the original TD chain (TD configuration registers).

  1: Store the intermediate states separately in a working register to keep the original TD configuration.

  Typically TD configurations are preserved so that TD can be repeated.

TD Configuration

- **Transfer Count(0 to 4095)**

  The total number of bytes to be moved from the source to destination.

  For example, if you want to move 100 bytes of data from a 16-bit peripheral to a memory buffer, the burst count is set to 2 and transfer count is set to 100.

- **Next TD**

  Points to the next TD, similar to a linked list

- **TD Property (Configurable from the list below)**

  Increment Source Address

  Increases source address after each burst transfer.

  Increment Destination Address

  Increments destination address after each burst transfer.

Swap Enable

The PSoC 3 Keil Compiler uses big endian format to store 16-bit and 32-bit variables. But the PSoC 3 peripheral registers uses little endian format. A byte swap on 2-byte or 4-byte words must occur to move data between array and peripheral registers. For this reason, the DMA must be configured to swap bytes while it moves data between peripheral registers and memory in PSoC 3.

If this TD property is set, DMA swaps the data bytes while it moves data from the source to destination.

Swap Size: Used with the swap enable setting.

0: Swap size is 2 bytes. Every 2 bytes are endian swapped during the DMA transfer.

1: Swap size is 4 bytes. Every 4 bytes are endian swapped during the DMA transfer.

Auto Execute Next TD

0: The next TD in the chain will be executed only after the next DMA request.

1: The next TD in the chain is automatically executed soon after the current TD transfer is finished.

DMA Completion Event

Generates a DMA “done signal” after the data transfer is finished. This is typically used to create an interrupt after the transfer is finished.

You can set the channel and TD configuration parameters in two ways:

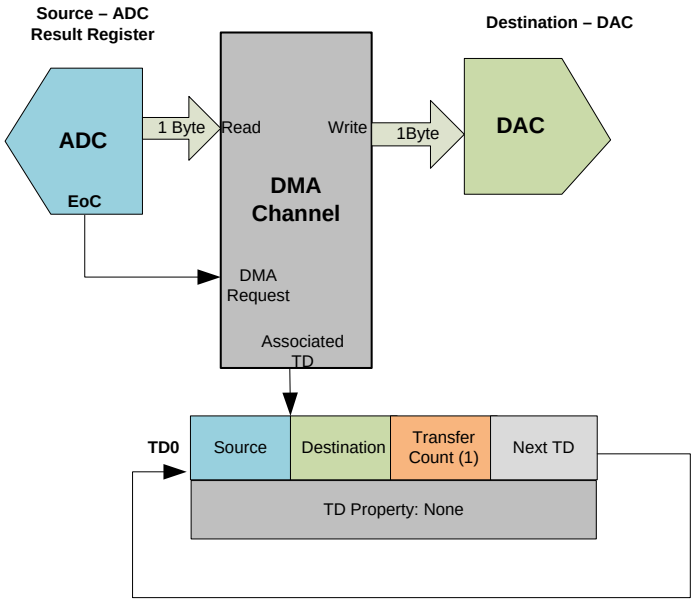
Use the DMA wizard included in PSoC Creator. For more information, see Appendix B: DMA Wizard Configuration.

Use the APIs included in PSoC Creator. For more information, see Appendix C: DMA Configuration – Important APIs.

Example 1: Point-to-Point Transfer

This ADC_DMA_DAC example shows how to use DMA to do a simple point-to-point transfer. The Figure 4 shows an ADC-to-DAC signal chain that uses DMA. The DMA channel moves ADC output data to DAC data register on each DMA request. The End of Conversion (EoC) signal from ADC is configured as a request signal for the DMA channel in this example.

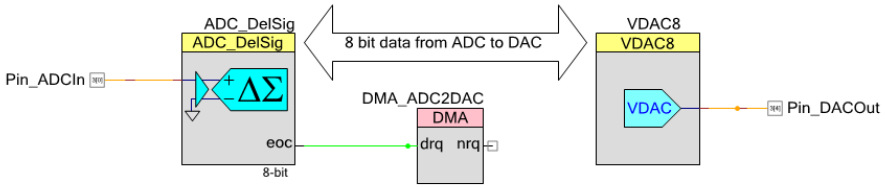
Figure 4. Point-to-Point Transfer Block Diagram



Top Design

ADC is set in 8-bit, single-ended mode to match the data format of the VDAC, which is a single-ended 8-bit voltage DAC. The DMA component instance is given the name DMA_ADC2DAC. To make a request for DMA transfer whenever an ADC result is available, the hardware request terminal of the DMA channel is enabled and connected to the ADC EoC signal.

Figure 5. Top Design



Channel Configuration

Parameter	Project Setting
Upper Source Address	HI16(CYDEV_PERIPH_BASE)
Upper Destination Address	HI16(CYDEV_PERIPH_BASE)
Burst Count	1 Byte
Request Per Burst	True (1)
Initial TD	TD0
Preserve TD	Yes(1)

The upper 16 bits of the 32-bit address for both source and destination address is set to HI16(CYDEV_PERIPH_BASE).

CYDEV_PERIPH_BASE is defined in the header file *cydevice.h* that is created by PSoC Creator and defines the base address of all PSoC 3 and PSoC 5 peripherals, which include ADC and DAC. HI16 is a PSoC Creator macro that returns the upper 16 bits of the 32-bit address.

You can also use HI16(ADC_DEC_OUTSAMP_PTR) as the upper 16 bits of the source address, where ADC_DEC_OUTSAMP_PTR is the address of ADC output register (decimator output). This register pointer is defined in the header file, *ADC_DelSig.h*, which is created by the ADC component. You can also use HI16(VDAC8_DATA_PTR) to identify the upper 16 bits of

the destination address, where VDAC8_DATA_PTR is the pointer for the DAC data register defined in *VDAC8.h*.

The DMA channel must move the one-byte ADC result from the ADC output register to the DAC data register after each DMA request. For this reason, the burst count is set to 1 byte and the request per burst is set to true.

TD0 Configuration

Parameter	Project Setting
Lower Source Address	LO16(ADC_DEC_OUTSAMP_PTR)
Lower Destination Address	LO16(VDAC8_DATA_PTR)
Transfer Count	1 byte
TD property	None
Next TD	Loop back to same TD (TD0)

In the TD, the lower 16 bits of the source address is configured as LO16(ADC_DEC_OUTSAMP_PTR), where ADC_DEC_OUTSAMP_PTR is the address of ADC output register (decimator output). This register pointer is defined in the header file, *ADC_DelSig.h*, which is created by the ADC component. The destination address is configured as LO16(VDAC8_DATA_PTR), where VDAC8_DATA_PTR is the pointer for the DAC data register defined in *VDAC8.h*. LO16 macro returns the lower 16-bits of the 32-bit address.

Because the transaction is to send one byte from ADC to DAC, the transfer count is also set to 1. The next TD to be executed is set to the same TD (looped), so that the same transaction is repeated on each DMA request. To keep the TD settings (source, destination, and transfer count) and make the TD able to be repeated when the transaction is finished, the Preserve TD parameter is set to 1 (TRUE).

Operation

The EoC signal from ADC sets the DMA request for the channel. After it receives the request, DMA reads data from ADC output register and writes to the DAC data register. The transfer count is decreased by 1 (burst count) after the transfer. When the count is zero, the transaction is finished. Because the Preserve TD parameter is set to true, the original TD configuration is kept and reloaded. The DMA channel is ready for the next transaction, which is activated on the next DMA request signal (EoC).

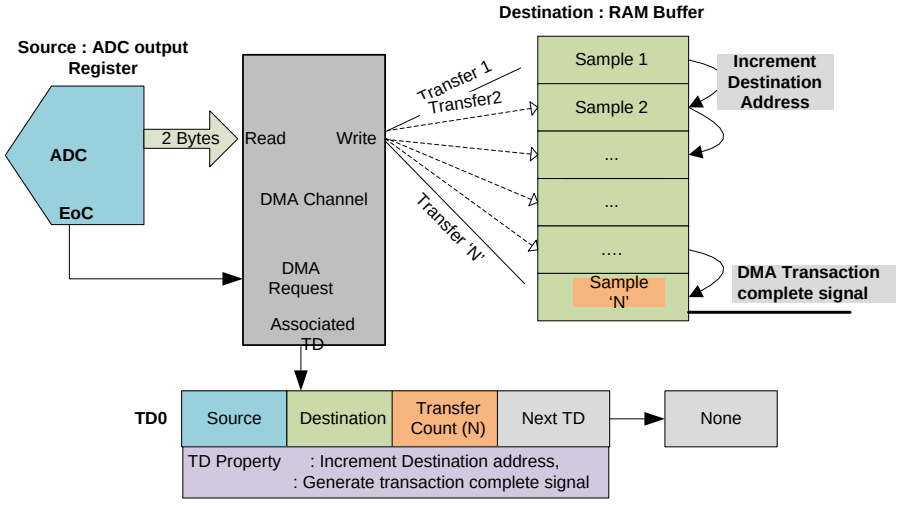
Example

The project Eg1_ADC_DMA_DAC shows the point-to-point example. You can find this project in the AN52705.zip file that is attached to this application note. For more information on the test setup of this project, see Appendix D: Example Projects – Test Setup.

Example 2: Point-to-Array Transfer

This 16-bit ADC data buffering example shows how to use DMA to do a point-to-array transfer.

Figure 6. Point-to-Array Transfer Block Diagram.

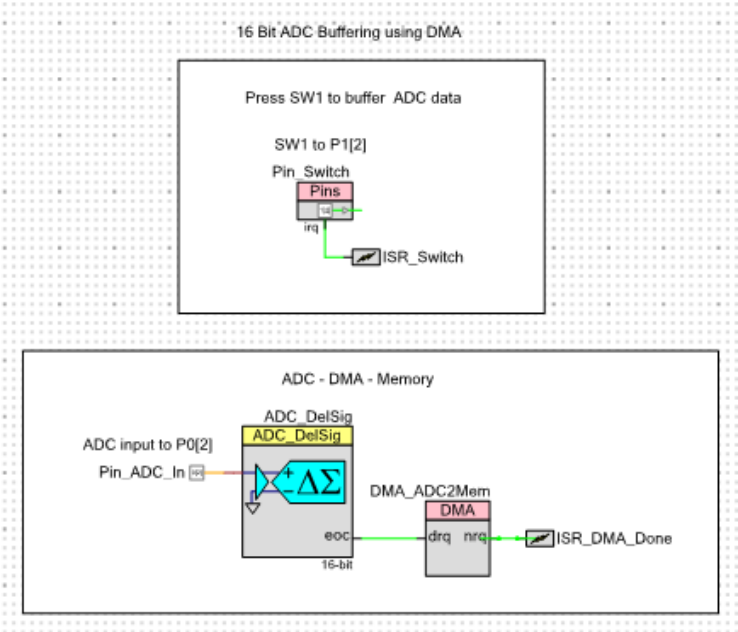


As the Figure 5 shows, the DMA must move the ADC result (2 bytes) from the source ADC to the destination RAM buffer each time it receives a request. The RAM buffer pointer must be increased after each data movement to point to the next sample location. After the specific number of ADC samples is collected, the DMA must send a signal that the transaction is finished.

Top Design

DMA collects a specified number of ADC samples (N) on a switch press. The DMA channel is enabled on each switch press and disabled after it collects the specified number of ADC samples. A DMA Transaction Complete signal on the nrq terminal activates an ISR, which disables the DMA channel.

Figure 7. Top Design



The DMA component instance has the name DMA_ADC2Mem. The hardware request of this DMA channel component is set to rising edge. The hardware request terminal of the DMA channel is connected to the ADC EoC signal so that the DMA channel is requested whenever ADC result is available.

Channel Configuration

Parameter	Project Setting
Upper Source Address	HI16(CYDEV_PERIPH_BASE)
Upper Destination Address	HI16(CYDEV_SRAM_BASE)
Burst Count	2 Bytes
Request Per Burst	True (1)
Initial TD	TD0
Preserve TD	Yes(1)

The upper 16 bits of the source address is set to HI16(CYDEV_PERIPH_BASE).

'CYDEV_PERIPH_BASE is defined in the header file *cydevice.h* that is created generated by PSoC Creator. This gives the 32-bit base address for all PSoC 3 and PSoC 5 peripherals including ADC. The HI16 macro gives the upper 16 bits of this 32-bit address.

The upper 16 bits of RAM variables are given by the macro HI16(CYDEV_SRAM_BASE), where CYDEV_SRAM_BASE is the SRAM base address defined in *cydevice.h*. HI16 (&adc_sampleArray) will also work for PSoC 5 but not for PSoC 3 – Keil compiler. This is because the upper 16 bits of the address of RAM variables is zero for PSoC 3, but the Keil compiler stores Keil-specific information in the upper 16 bits of the variable address. For this reason, HI16 (adc_sampleArray) returns an incorrect address when used with PSoC 3 – Keil compiler. For this reason, use HI16(CYDEV_SRAM_BASE) to specify the upper 16 bits of RAM variables in PSoC 3.

In this example, a 2-byte ADC result must be moved from ADC to memory on each DMA request. For this reason, the burst count is set to 2 and the request per burst is set to true.

To keep the original TD settings (source, destination and transfer count) and make the TD able to be repeated when the transaction is finished, the Preserve TD parameter is set to 1 (TRUE).

TD0 Configuration

Parameter	Project Setting
Lower Source Address	LO16(ADC_DEC_OUTSAMP_PTR)
Lower Destination Address	LO16(adc_sampleArray)
Transfer Count	N×2 (No. of samples × Bytes per sample)
TD property	Increment Destination Address Generate DMA done event Swap Enable required for PSoC 3
Next TD	None/repeat to same TD

The lower 16 bits of the source and destination address are identified by the LO16 macro. The destination is the 16-bit RAM array adc_sampleArray.

The transfer count identifies the total number of bytes to be moved from source to destination to finish the transaction. This is set to 'Number of samples × Bytes per Sample' (2N).

The TD property (TD_INC_DST_PTR) is set to increment the destination address and the RAM buffer pointer after each burst transfer. The TD is also defined to generate a transaction complete signal (DMA__TD_TERMOUT_EN) after the specified number of bytes is moved from ADC to buffer.

When the 16-bit data is moved from ADC to memory in PSoC 3, the bytes must be swapped. This is because PSoC 3 peripheral registers use little endian format and the Keil compiler uses big endian format. For more information, see the KB article Endian format PSoC 3 device vs PSoc 3's KEIL Compiler. The TD_SWAP_EN configuration makes the DMA able to swap bytes while it moves data from peripheral to memory.

Operation

Each time a switch is pressed; ISR_Switch is activated, which sets the flag to enable the DMA channel (CyDmaChEnable). If the DMA channel is enabled, the EoC signal from ADC activates the DMA channel request.

On each DMA request, the DMA fetches 2 bytes (burst count) from the source - ADC output register, writes it to the destination RAM buffer, and increases the destination address by two. The transfer count is decreased by 2 (burst count) after each burst transfer. This repeats until the transfer count is zero, which generates a transaction complete signal at the NRQ terminal of the DMA component, which activates the ISR_DMA_Done interrupt. This sets the flag to disable the DMA channel. The DMA channel is enabled again the next time the switch is pressed. Because the Preserve TD parameter is set to true, the TD configurations are reloaded and the TD is ready for the next transaction when the channel is re-enabled.

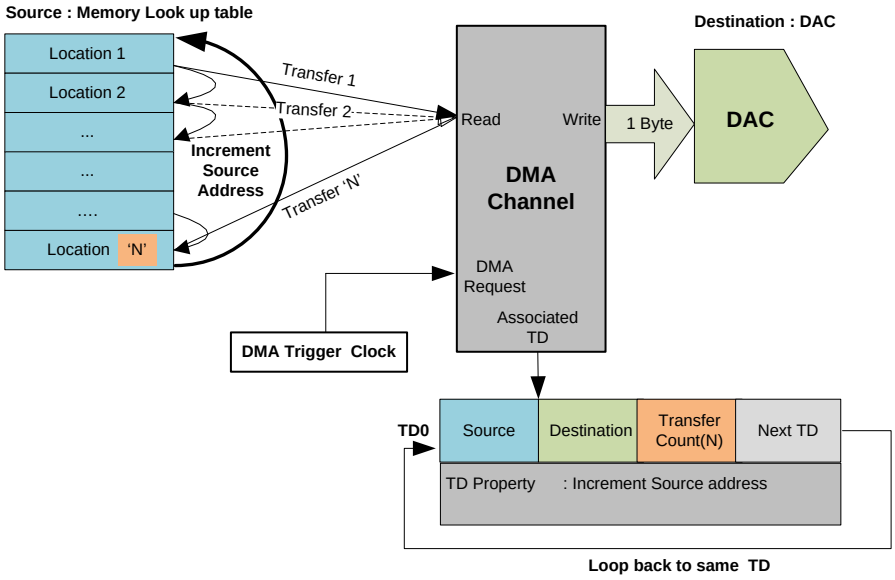
Related Resources

The project Eg2_ADC_DMA_Mem shows the point-to-array example. You can find this project in the AN52705.zip file that is attached to this application note. For more information on the test setup of this project, see Appendix D: Example Projects – Test Setup.

Example 3: Array-to-Point Transfer

This example shows how to use DMA to do an array-to-point transfer. In this example, the DAC is updated at regular intervals with a sequence of values that are kept in the flash array (lookup table). These values create a sine wave at the DAC output. The DMA request comes periodically from a digital clock signal whose frequency determines the update rate of DAC. The update rate (DMA trigger clock) and the number of points in the sine lookup table define the frequency of the output sine wave.

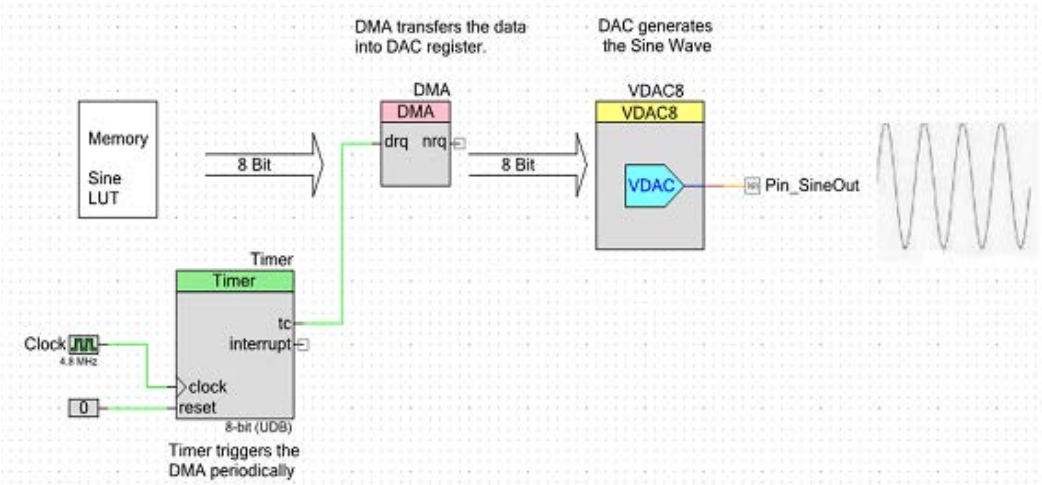
Figure 8. Array-to-Point Transfer



Top Design

Figure 9 shows the top design for the project. A sine lookup table with 128 points is kept in flash memory. These values are updated sequentially to a DAC to create a sine wave. Timer component is used to generate a clock signal to periodically request DMA data movement. The DMA is set to update the values on a hardware trigger signal given from terminal count of the timer. In this example, the frequency of the sine wave generated by the DAC is equal to the DMA trigger clock (Timer TC frequency) divided by number of points in the sine lookup table.

Figure 9. Top Design



Channel Configuration

Parameter	Project Setting
Upper Source Address	HI16(CYDEV_FLS_BASE) for PSoC 3 HI16 (&sineTable) for PSoC 5
Upper Destination Address	HI16(CYDEV_PERIPH_BASE)
Burst Count	1 Byte
Request Per Burst	True (1)
Initial TD	TD0
Preserve TD	Yes(1)

The source for DMA transfer is the sineTable array that is kept in flash memory. The destination is voltage DAC. The HI16 (&sineTable) in PSoC 5 sets the upper 16 bits of the source address. In PSoC 3, HI16 (&sineTable) does not use the correct upper 16-bit address because the Keil compiler keeps Keil-specific information in the upper bits of the array pointer. The macro HI16(CYDEV_FLS_BASE) is used to identify the upper 16 bits of source address for PSoC 3. CYDEV_FLS_BASE, which is defined in the source file *cydevice.h* created in PSoC Creator, identifies the base address for the entire flash memory in PSoC 3. In PSoC 5, the flash address range is '0x0000 0000 to 0x0003 FFFF'. The upper 16 bits of the entire flash address is not a fixed value; it is defined by the location of the lookup table in flash memory and is identified by HI16 (&sineTable).

Similarly, the upper 16 bits of the destination address is defined as HI16(CYDEV_PERIPH_BASE), which is defined in the header file *cydevice.h* that is created by PSoC Creator, along with the HI16 macro to identify the upper 16 bits of any PSoC 3 or PSoC 5 peripheral address.

TD0 Configuration

Parameter	Project Setting
Lower Source Address	LO16 (&sineTable)
Lower Destination Address	LO16(VDAC8_DATA_PTR)
Transfer Count	N (No. of entries in the sine lookup table x no. of bytes per entry)
TD property	Increment source address
Next TD	Loop back to the same TD again

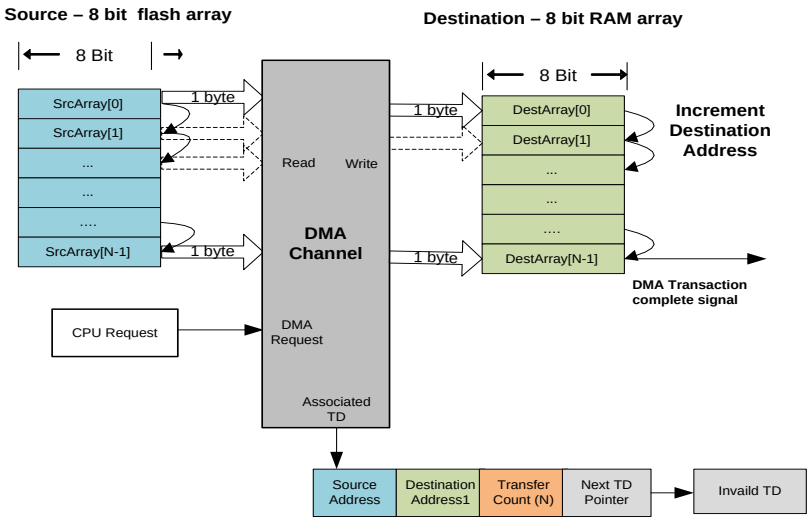
In the TD, the lower 16 bits of source address and destination address are defined with the macro LO16 (&sineTable) and LO16(VDAC8_DATA_PTR), where LO16 macro returns the lower 16 bits of the 32-bit addresses. VDAC8_DATA_PTR is the pointer for the DAC data register that is defined in *VDAC8.h*.

The project Eg3_Mem_DMA_DAC demonstrates the array-to-point example. You can find this project in the AN52705.zip file that is attached to this application note. For more information on the test setup of this project, see Appendix D: Example Projects – Test Setup.

Example 4: Array-to-Array Transfer

This example project shows how to use DMA to do an array-to-array transfer. In this example, an 8-byte flash array is copied to a RAM array using DMA, on a CPU request.

Figure 10. Array-to-Array Transfer

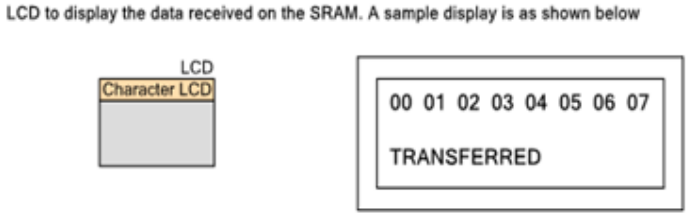


Top Design

Figure 11. DMA Configured to Transfer Data from Flash to SRAM on CPU Request



Figure 12. LCD Display



This example project shows the DMA configuration for bulk data transfer from memory to memory. 8-bytes of data defined in an array in the flash memory are the source for DMA. The destination is another 8-byte array defined in RAM. The DMA moves 8 bytes from the flash array to the RAM array. The transfer is activated by a CPU request. The end of transfer is identified by the NRQ signal of DMA, which activates the ISR_DMADone interrupt and displays the RAM contents on the LCD.

Channel Configuration

Parameter	Project Setting
Upper Source Address	HI16(CYDEV_FLS_BASE) , for PSoC 3 HI16(&sourceArray) for PSoC 5
Upper Destination Address	HI16(CYDEV_SRAM_BASE)
Burst Count	1 Byte
Request Per Burst	False
Initial TD	TD0
Preserve TD	0

The source for DMA transfer is the sourceArrray defined in the flash memory. The destination is 'destinationArray' in RAM. The upper 16 bits of source address is set with HI16(&sourceArray) in PSoC 5 where HI16 macro sends the upper 16 bits of the 32-bit address of the flash array (sourceArray). In PSoC 3, HI16(HI16 (&sourceArray) does not give the correct upper 16-bit address because the Keil compiler keeps Keil-specific information in the upper bits of the array pointer. The macro HI16(CYDEV_FLS_BASE) is used to identify the upper 16 bits of source address for PSoC 3. CYDEV_FLS_BASE, which is defined in the source file *cydevice.h* created in PSoC Creator, identifies the base address for the entire flash memory in PSoC 3. In PSoC 5, the flash address range is '0x0000 0000 to 0x0003 FFFF'. The upper 16 bits of the entire flash address is not a fixed value; it is defined by the location of the lookup table in flash memory and is identified by HI16(&sourceArray).

Similarly, the upper 16 bits of RAM array address is given by the macro HI16(CYDEV_SRAM_BASE), where 'CYDEV_SRAM_BASE' is the SRAM base address defined in *cydevice.h*. HI16(&destinationArray) will also work for PSoC 5 but not for PSoC 3. You should always use CYDEV_SRAM_BASE to identify the upper 16 bits of RAM variables or array in PSoC 3.

In this example, DMA must read byte by byte and write to the destination. For this reason, the burst count is set to 1 byte. The width of the spoke attached to source (system bus) as well as destination (RAM) is 32 bits wide; therefore you can set the burst count to 4 bytes as well for faster data transfers. In this case, the DMA will read and write 4 bytes in one burst.

In this example, the CPU gives only one DMA request and the entire transfer (multiple bursts) must occur after a single request. For this reason, the request per burst parameter is set to 0 (false) because separate requests are not required for each burst transfer.

Because the transaction is done only one time, the TDs do not need to be preserved.

TD0 Configuration

Parameter	Project Setting
Source Address	LO16(&sourceArray)
Destination Address	LO16(&destinationArray)
Transfer Count	N (8 bytes)
TD property	TD_INC_SRC_ADR TD_INC_DST_ADR DMA__TD_TERMOUT_EN
Next TD	None (Terminate TD – 0xFE)

The lower 16 bits of the source address and destination address for the TD configuration are identified by the LO16 macro. Because a total of 8 bytes must be copied from flash array (sourceArray) to RAM array (destinationArray), the transfer count is set to 8. The TD is configured to increase the source address (flash array pointer) and destination address (RAM array pointer) after each burst (TD_INC_SRC_ADR|TD_INC_DST_ADR). The TD is also configured to send a termout pulse (DMA__TD_TERMOUT_EN) after all the 8 bytes in the array are moved from flash to RAM. This pulse is used to activate an ISR when the transfer is finished. The next TD is set to Invalid TD (0xFE) to stop the TD chain after the transfer is finished.

Operation

A CPU request uses the API CyDmaChSetRequest(Channel_handle,CPU_REQ) to activate the DMA transfer.

When it receives a request from the CPU, the DMA transfers one byte(burst) from flash array to RAM array, increases both the source and destination addresses (burst count) by one and decreases the transfer count by one.

Because the request per burst parameter of the channel is set to 0 (false), subsequent burst transfers occur one after the other until all the 8 bytes (transfer count) are moved from flash memory array to the RAM array.

When the transfer count is zero, the termout pulse is generated on the nrq line. This activates an interrupt which sets the flag to indicate that the transfer is complete. The new RAM contents are then displayed on the LCD.

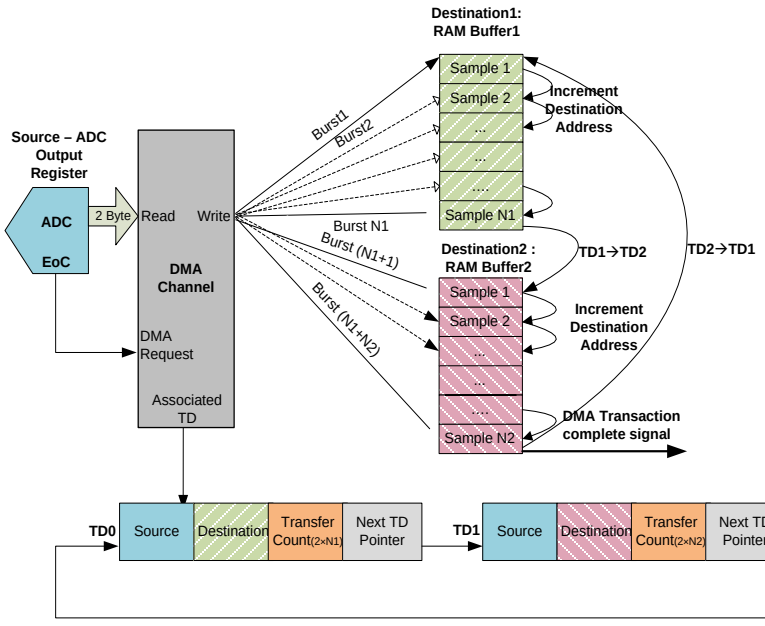
If there are pending requests from another DMA channel while the transfer occurs, the DMA Controller may fulfill other DMA channel requests in between the bursts.

The project Eg4_Mem_DMA_Mem demonstrates the array-to-array example. You can find this project in the AN52705.zip file that is attached to this application note. For more information on the test setup of this project, see Appendix D: Example Projects – Test Setup.

Example 5: Ping-Pong Buffer

This example project shows how to use DMA to do ping-pong buffering.

Figure 13. Ping-Pong Buffer



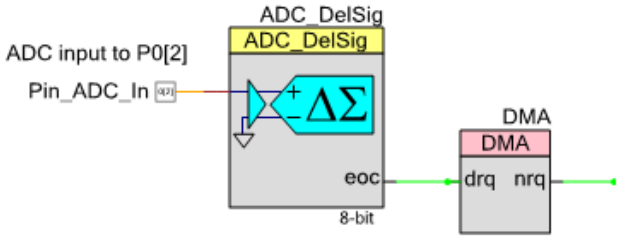
In this example, ADC data is collected in RAM buffer 1. After RAM buffer 1 is full, the destination is changed to RAM buffer 2. This example includes two transactions:

- Transaction 1: ADC to RAM buffer1
- Transaction 2: ADC to RAM buffer2

Because the upper 16-bit address for both the transactions is the same, you need only a single DMA channel and two TDs.

Top Design

Figure 14. ADC-DMA Memory



Channel Configuration

The following table shows the channel configuration, which is similar to that in Example 2.

Parameter	Setting
Upper Source Address	HI16(CYDEV_PERIPH_BASE)
Upper Destination Address	HI16(CYDEV_SRAM_BASE)
Burst Count	2 byte
Request Per Burst	True (1)
Initial TD	TD0
Preserve TD	Yes(1)

Transaction 1 uses TD0, and Transaction 2 uses TD1. To attach the transactions to each other, the **Next TD** parameter of TD0 is set to TD1 and vice versa.

TD0 Configuration

Parameter	Project Setting
Lower Source Address	LO16(ADC_DEC_OUTSAMP_PTR)
Lower Destination Address	LO16(adc_sampleArray1)
Transfer Count	N1×2 (No. of samples × Bytes per sample)
TD property	Increment Destination Address Generate DMA done event Swap Enable required for PSoC 3
Next TD	TD1

TD1 Configuration

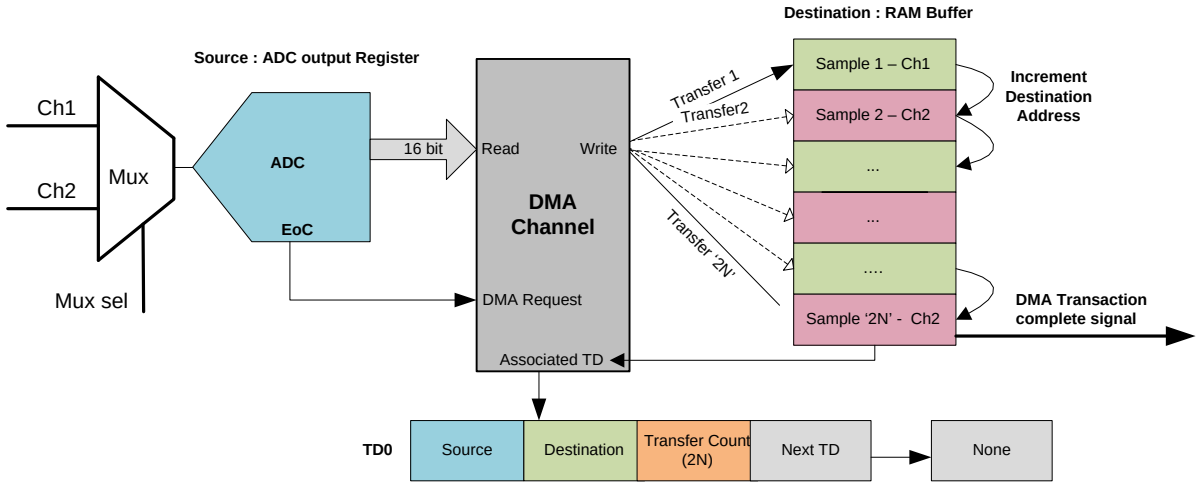
Parameter	Project Setting
Lower Source Address	LO16(ADC_DEC_OUTSAMP_PTR)
Lower Destination Address	LO16(adc_sampleArray2)
Transfer Count	N2×2 (No. of samples × Bytes per sample)
TD property	Increment Destination Address Generate DMA done event Swap Enable required for PSoC 3
Next TD	TD0

The project Eg5_PingPongBuffer demonstrates the ping-pong buffer example. You can find this project in the AN52705.zip file that is attached to this application note. For more information on the test setup of this project, see Appendix D: Example Projects – Test Setup.

Example 6: Multiplexed Data Buffering

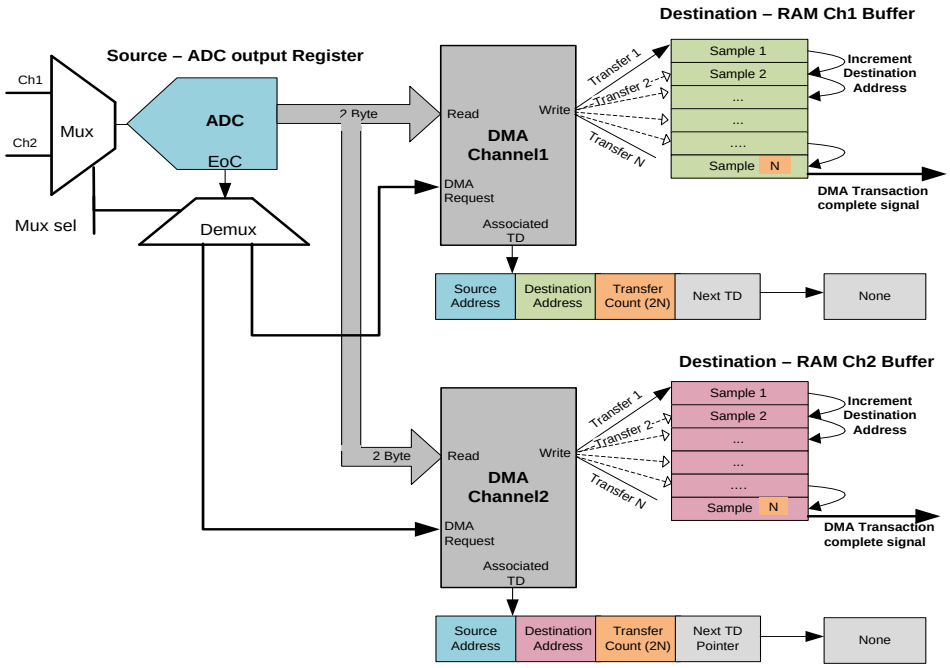
This example shows how to collect ADC data when the input to ADC is multiplexed. If a single DMA channel is used to collect multiplexed ADC data, the buffered data will be as shown in Figure 15 with channel1 and channel2 data combined. But you can also keep the channel1 and channel2 data separate.

Figure 15. Multiplexed Data Buffering Using Single Channel



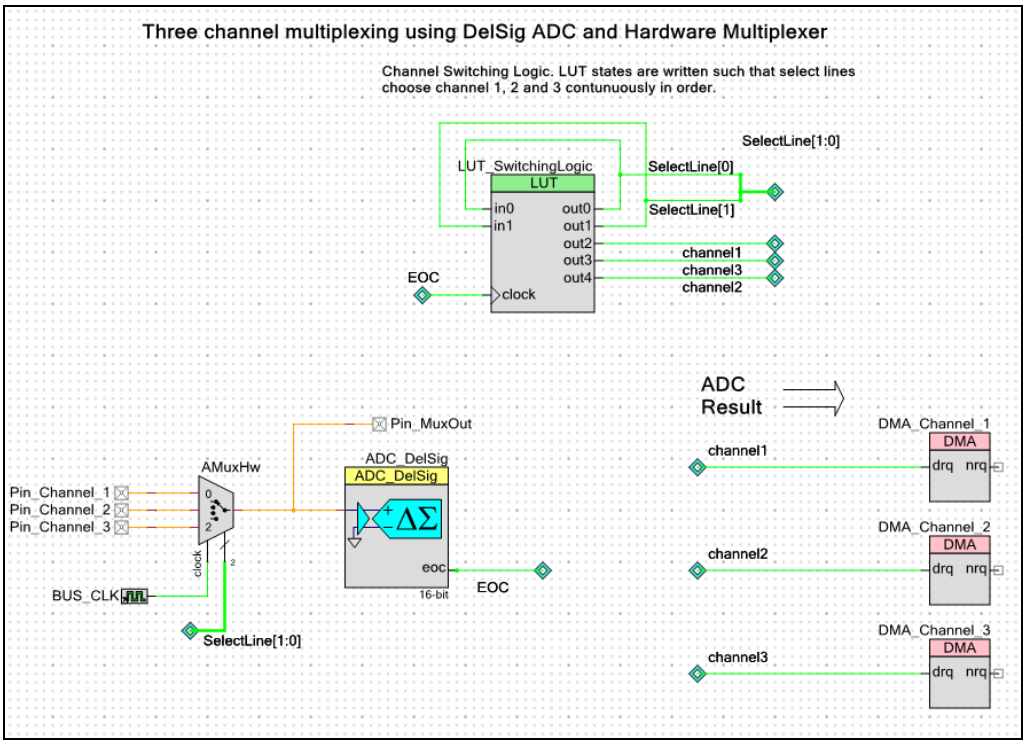
DMA channels cannot change between two transactions until the entire transfer count is finished. For this reason, multiple TDs will not work as well. Figure 16 shows how you can use multiple DMA channels and multiplex them to move the multiplexed channel data into separate buffers.

Figure 16. Multiple DMA Channels for Buffering Multiplexed ADC Data



The EoC signal must be demultiplexed to trigger channel1 and channel2 alternatively. The top design of an actual three channel multiplexed data buffering implementation is shown in Figure 17.

Figure 17. Top Design



The EoC signal changes the output of the LUT state machine whose output activates DMA channels 1–3 in order .The LUT output also controls the hardware multiplexing of the ADC channels. The channel and TD configuration are the similar to Example 2. Multiplexed data buffering is an additional example given to show multiplexing of DMA channels. The application note does not include a project for this example.

Appendix A: DMA Component Hardware Configuration

Figure 18 shows the DMA channel component that PSoC Creator includes under the Systems tab in the Component Catalog. You must use this DMA channel component and a set of APIs to configure DMA for a data transfer.

Figure 18. DMA Channel Component



Input/Output Connections of DMA component

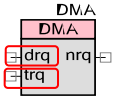
You can configure DMA to create a pulse of width 2 bus clocks at the NRQ terminal when the transfer is finished. The nrq terminal can be connected to an interrupt, or to another component to tell that the DMA transfer is finished.

The transaction descriptor (TD) configuration defines whether or not to generate a signal on the nrq terminal when the transfer is finished.

You can set the following parameters in the component configuration window:

Hardware Request (drq): This setting defines the type of signal (rising edge/level) the DMAC will get as the DMA channel trigger. Any selection for this parameter except "Disabled" adds a drq terminal to the component, which allows a DMA request to be made from any hardware source.

Figure 19. Hardware Request



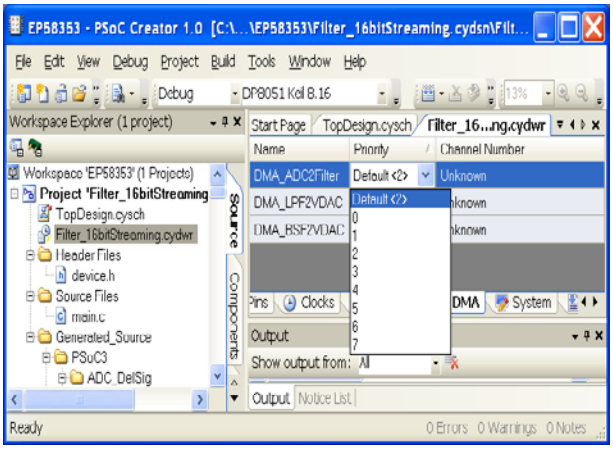
Without drq terminal, the DMA transaction is activated only by the CPU. When this parameter is set to “derived”, the DMA trigger type -edge/level is determined from the source of the DMA trigger. For more information, see the DMA component datasheet.

Hardware Termination (trq)*:* When this option is set to true, another input terminal (trq) is displayed in the component. The DMA transaction can be stopped by giving a hardware request signal (positive edge) to this terminal if TD_TERMIN is enabled. Without this terminal, the DMA transaction is stopped only by the CPU or when the transfer is finished and the transaction is attached in a chained configuration with a terminate TD chain.

Setting DMA Channel Priority

When multiple DMA channel requests are active, the DMA channels are processed by DMAC based on the channel priority settings. Each DMA channel can be given one of the eight different priorities. The DMA channel priority is set in PSoC Creator in **Design Wide Resources (*.cydwr) > DMA** as shown in Figure 20.

Figure 20. Setting DMA Channel Priority



When both the CPU and DMAC request access to the same spoke on PHUB at the same time, the CPU takes the higher priority by default. The PHUB manages the arbitration between the DMA channels and the CPU and DMAC manages arbitration between various DMA channels. For more information, see PSoC® 3, PSoC® 5 Architecture TRM.

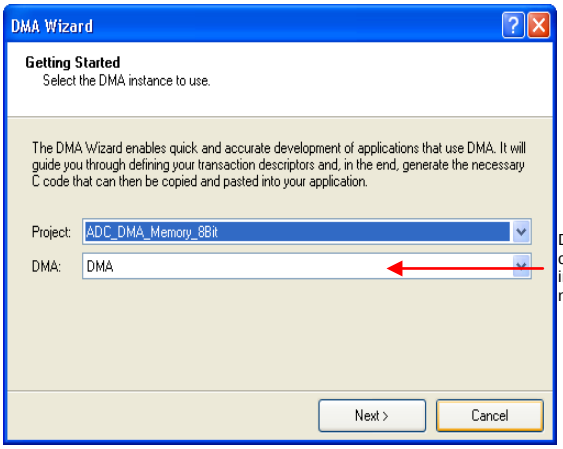
Appendix B: DMA Wizard Configuration

The DMA Wizard can define the firmware configuration of the DMA channel and TD. To start the DMA wizard, go to **PSoC Creator > Tools > DMA Wizard**.

The wizard supports only a few peripherals as source or destination for the DMA channel. The user must use the firmware configuration steps given in Appendix C: DMA Configuration – Important APIs to configure the DMA, if the DMA wizard does not support the peripheral involved in the user's DMA project.

Step 1: Select a DMA channel (DMA component instance)

Figure 21. Selecting DMA Channel



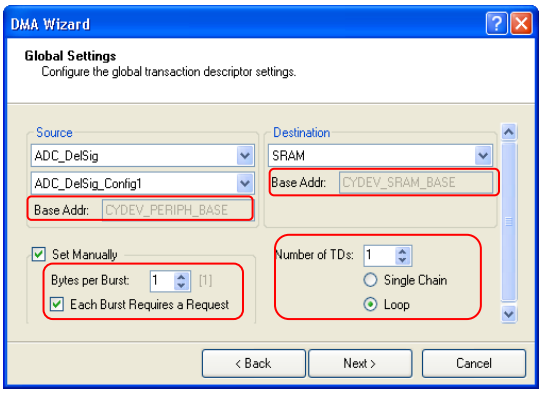
Select the DMA channel to be configured:

- Project: Name of the project
- DMA: DMA component instance name in your project

Click **Next** after you select the channel.

Step 2: Select global settings

Figure 22. Global Settings



Use this window to select the DMA channel configuration parameters such as:

Source and Destination: The base address for the DMA channel's source and destination. The upper 16 bits of the source and destination address for the DMA channel is set in the channel configuration registers based on this setting.

Bytes per Burst: The number of bytes to be moved nin a single burst.

Each Burst Requires a Request**:** Whether each burst requires a separate request.

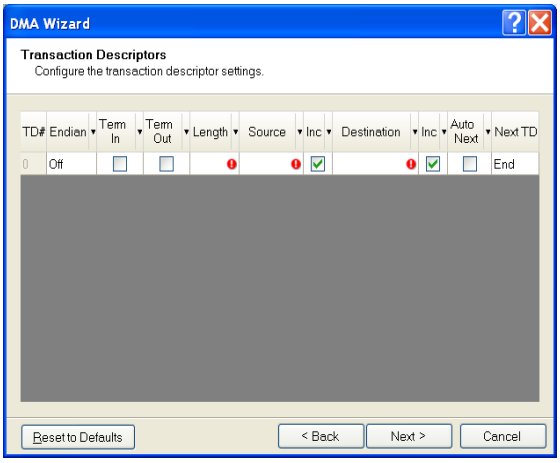
Number of TDs**:** The number of transaction descriptors associated to the DMA channel (between 1 and 128).

Single Chain or Loop**:** This defines what the '**Next TD**' will be for the last TD associated with the DMA channel. If single chain, the next TD will be END TD chain. If loop, it will loop back to the first TD.

After you are finished, click **Next**.

Step 3: Define the transaction descriptors for the channel

Figure 23. Transaction Descriptors



The following table shows information about each TD configuration parameter.

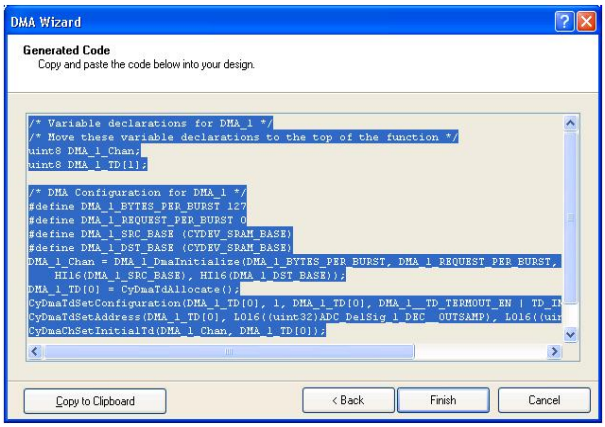
TD Configuration Details

Field	Description
TD#	Displays the logical number for the Transaction Descriptor.
Endian	Enables 2- or 4-byte endian byte swapping. This enables swapping the byte while the data moves from source to destination. The **Bytes per Burst** setting must be set as a multiple of the endian selection. This is usually used for DMA transfers between PSoC 3 memory and peripherals because of the difference in endianess.
Term In	Enables ending the TD transaction on a rising edge of the TERMIN (trq) signal.
Term Out	Enables the creation of the TERMOUT (nrq) signal when the TD finishes.
Length	This specifies the transfer count for the TD in bytes (0 to 4095). This is the total number of bytes that the DMA should transfer to complete the transaction.
Source	The lower 16 bits of the source address for the DMA transfer. A drop-down list of addresses for the source will be given by the DMA wizard if the source selected is a component (not memory). You can also edit or enter the source address manually.
Inc (Source)	Enables an increase of the source address as the DMA does the transaction. If this is enabled, every time the DMA reads the data from source, the source address is increased by the number of bytes that the DMA has read. The DMA will increase the source address until the entire transaction (transfer count) is finished.
Destination	The lower 16 bits of the source address for the DMA transfer. A drop-down list of addresses for the destination will be given by the DMA wizard if the destination selected is a component (not memory). You can also edit or enter the destination address manually.
Inc(Destination)	Enables an increase of the destination address as the DMA does the transaction. The DMA will increase the destination address until the entire transaction (transfer count) is finished.
Auto Next	Automatically execute the next TD without another DMA request.
Next TD	The next logical TD in the chain of TDs. Set to END if this TD chain is finished with this TD.

To go the **Generate Code** page, click **Next**.

Step 4: Add to your firmware the code created by the DMA Wizard

Figure 24. Generated Code



After the DMA channels and TD configuration are finished, the wizard creates code for the DMA channel. This code includes the configuration for the DMA channel and the TDs. Make a copy of the code and pastes it to *main.c*. For more information on the wizard, see the PSoC Creator Help file.

Appendix C: DMA Configuration – Important APIs

The DMA component creates a source file (*DMA_InstanceName_dma.c*) and corresponding header file (*DMA_InstanceName_dma.h*) for each DMA instance. For example, if there is a DMA component instance in your design that has the name DMA_1, then the files - *DMA_1_dma.c* and *DMA_1_dma.h* are created during the build process. These files include the **DmaInitialize** API that is used to initialize the DMA channel. Use other DMA APIs included in *CyDmac.c* and *CyDmac.h* in the **Generated Source** folder for the remaining channel and TD configuration.

Following are the Major Steps that are used to Configure DMA for Data Transfers.

1. Start the DMA channel
   ```
   Channel_Handle =
   DMA_DmaInitialize(DMA_BYTES_PER_BURST,
   DMA_REQUEST_PER_BURST, HI16(Source
   Address), HI16(Destination Address))
   ```
2. Create an instance of TD
   ```
   TD_Handle = CyDmaTdAllocate();
   ```
3. Set TD Configuration
   ```
   CyDmaTdSetConfiguration(TD_Handle,Trans
   fer_Count,Next_TD,TD_Property);
   ```
4. Set TD address
   ```
   CyDmaTdSetAddress(TD_Handle,
   LO16(Source Address), LO16(Destination
   Address))
   ```
5. Identify the first TD with the channel
   ```
   CyDmaChSetInitialTd(Channel_Handle ,
   TD_Handle)
   ```
6. Activate the DMA channel
   ```
   CyDmaChEnable(Channel_Handle, preserve_TD)
   ```

Step 1: DMA Channel Initialization

```
Channel_Handle =
DMA_DmaInitialize(DMA_BYTES_PER_BURST,
DMA_REQUEST_PER_BURST, HI16(Source
Address), HI16(Destination Address))
```

'DmaInitializ' API configures several DMA channel parameters such as the following:

- ***DMA_BYTES_PER_BURST*:** The number of bytes to be read and written by the DMA channel in one burst

  For example, if you want to define DMA to collect 8-bit ADC data, set this parameter to 1 because the DMA channel must move 1 byte from source to destination on each request. Or, if you want to collect 16-bit ADC data, set this parameter to 2.

- ***DMA_REQUEST_PER_BURST*:** Whether each burst must have a separate request.

  If set to 1, each burst transfer must be individually requested. If set to 0, all subsequent bursts after the first burst are automatically carried out without separate request. (Only the first burst transfer must have a DMA request.)

  - ***HI16(Source Address):*** The upper 16 bits of the source address. HI16 is a macro created by PSoC Creator to specify the upper 16 bits of a 32-bit value or address.

    PSoc 3 Keil compiler stores Keil-specific information in the upper 16 bits of the variable address. For this reason, use the following constants defined in *CyDevice.h* along with HI16 macro to configure the upper 16 bits of source and destination address for PSoC 3 especially when the source or destination for the DMA transfer is RAM or flash memory.

Source	DMA_SRC_BASE
Peripheral	CYDEV_PERIPH_BASE
RAM	CYDEV_SRAM_BASE
Flash	CYDEV_FLS_BASE

  - ***HI16(DestinationAddress):*** The upper 16 bits of the destination address. Use macros provided in the previous table to identify the source and destination address in PSoC 3.

Step2: TD allocation

```
TD_Handle = CyDmaTdAllocate();
```

'CyDmaTdAllocate' creates instances of TD and sends the handle to the TD. The TD handle is used by other APIs that configure the TD. To create multiple TDs, call the API multiple times.

Step 3: TD configuration

```
CyDmaTdSetConfiguration(TD_Handle,Transfer_
Count,Next_TD, TD_Property);
```

TD_Handle: A handle previously returned by CyDmaTdAllocate API

- *TD_Handle*: A handle previously returned by CyDmaTdAllocate API
- *Transfer_Count:* The total number of bytes to be moved from source to destination.

Next_TD: Index of the next TD in the TD chain. Use DMA_INVALID_TD (0xFE) as Next_TD to end the TD chain.

- TD_Property: Use the TD Configuration register flags to set the properties of the DMA transaction. The following table shows the TD property flags that are available. Use these flags ORed together to configure the TD property. For example, if you want to configure the TD to swap 4 bytes during the data transfer, use (TD_SWAP_EN| TD_SWAP_SIZE4) as the TD property setting.

Configuration Flag	Function
TD_SWAP_EN	Perform endian swap; swap bytes while moving data from source to destination.
TD_SWAP_SIZE4	Swap size = 4 bytes. Default swap size is 2 bytes.
TD_AUTO_EXEC_NEXT	The next TD in the chain is activated automatically when the current TD finishes.
TD_TERMIN_EN	End 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 listens for it.
DMA__TD_TERMOUT_EN	If this flag is used, a pulse is generated on the nrq line when the TD transfer is complete. This flag is specific to a DMA component instance and is defined in the component instance header file. For example, if the DMA component instance name is DMA_1 in the top design, the termout macro for the instance is '*DMA_1__TD_TERMOUT_EN*' which is included in *DMA_1_dma.h.*
TD_INC_DST_ADR	Increase destination address according to the size of each data burst transaction.
TD_INC_SRC_ADR	Increase source address according to the size of each data burst transaction.

Step 4: Configuring TD source and destination

```
CyDmaTdSetAddress(TD_Handle, LO16(Source),
LO16(destination))
```

- `TD_Handle:` A handle previously returned by CyDmaTdAllocate API
- *LO16(Source):* Lower 16 address bits of the source of the data transfer.
- *LO16(destination):* Lower 16 address bits of the destination of the data transfer.

PSoC is highly programmable and therefore many peripherals are not static with fixed register blocks. They are created from the programmable digital blocks and physical location of peripheral block changes based on design. Therefore, a conventional register map listing all the source and destination addresses is not possible for PSoC 3 or PSoC 5. Instead, you can see the respective component generated header files to identify the register addresses to move data. The registers associated with each component are defined in the respective component header files generated by PSoC Creator during the build process.

Step 5: Attach the TD to the channel

```
CyDmaChSetInitialTd(Channel_Handle ,
TD_Handle)
```

- *Channel_Handle*: The handle of the DMA instance returned by DMA_DmaInitialize () API.
- *TD_Handle:* The index of the TD to be activated first. TD_Handle is returned by the CyDmaTdAllocate API during TD allocation.

Step 6: Enable DMA channel

```
CyDmaChEnable(Channel_Handle, Preserve_TD)
```

- *Channel_Handle*: Handle of the DMA instance returned by DMA_DmaInitialize() API.
- *Preserve_TD* If this option is set to 1, the channel retains the TD configurations set by the user (source, destination and transfer count) and makes the TD able to be repeated.

Other Important APIs

Use the following API if a CPU request is used to activate the DMA channel:

```
CyDmaChSetRequest(Channel_Handle, CPU_REQ);
```

Use the following API to disable the DMA channel:

```
CyDmaChDisable (Channel_Handle);
```

Appendix D: Example Projects – Test Setup

Example 1: Point-to-Point Transfer – Eg1_ADC_DMA_DAC

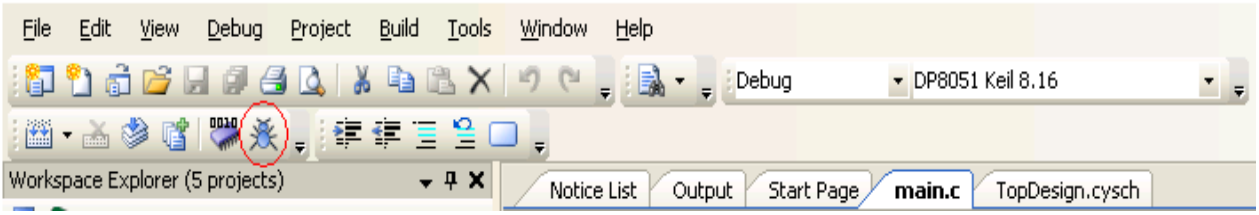
In this example project, the ADC sampling frequency is 10 kHz. The input signal frequency must be less than half of the sampling frequency to satisfy the Nyquist criteria. This project will have the best reconstruction for input signal frequency less than 250 Hz. The test setup is as follows:

1. Connect the Function generator to pin P3[0] - This is the input to the ADC.
2. Set the Function Generator to make a sine wave of 100 Hz.
3. Connect the Oscilloscope probe to pin P3[4] - This is the VDAC output.
4. Build the project and program the device.
5. Look at the output from pin P3[4] on the oscilloscope. The output on the oscilloscope should be a sine wave of frequency 100 Hz, which is the same as the input.

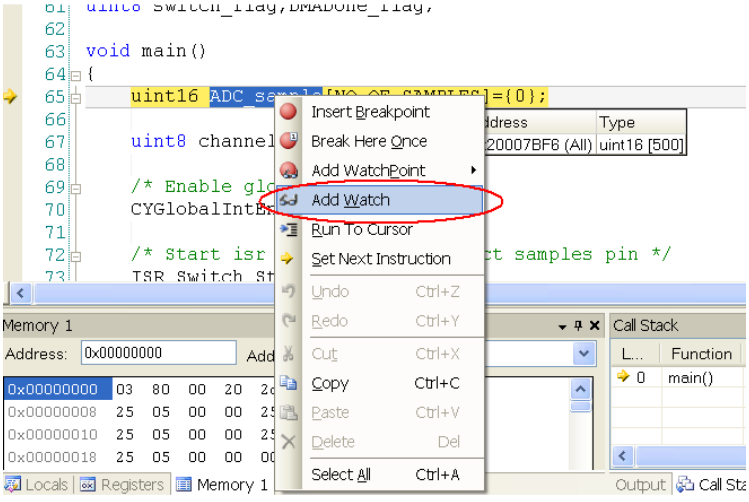
Example 2: Point-to-Array Transfer – Eg2_ADC_DMA_Mem

The test setup is as follows:

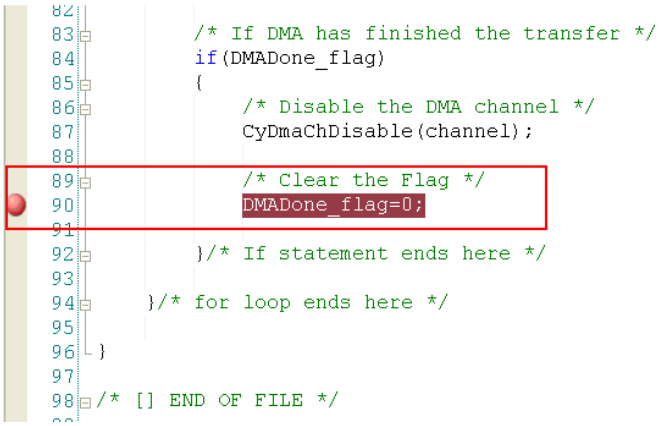
1. Connect the input signal to pin P0[2] (ADC input)on the DVK. Make sure that the input is within the ADC range V_{SSA} to 2.048 V.
2. Connect P1[2] to switch(SW1) on the DVK.
3. Build the project.
4. Press F5 or click the debug icon to download the program and start debugging.



5. Add ADC_sample as a watch variable.



6. Put a breakpoint inside the `if(DMADone_flag)` loop as shown in the figure.



7. Press F5 to run the program. Press the switch (SW1) connected to P1[2] to enable the DMA to start ADC sample buffering. The execution stops at the breakpoint after the DMA has transferred the specified number of samples from ADC to memory. The result can be verified by monitoring the **ADC_sample** array in the watch window.

Watch 1

Name	Value	Address	Type	Radix
ADC_sample	[500]	0x20007BF6 (All)	uint16 [500]	Default
0	0xEAAF	0x20007BF6 (All)	uint16	Default
1	0xEA9F	0x20007BF8 (All)	uint16	Default
2	0xEA97	0x20007BFA (All)	uint16	Default
3	0xEAAC	0x20007BFC (All)	uint16	Default
4	0xEAAF	0x20007BFE (All)	uint16	Default
5	0xEA9C	0x20007C00 (All)	uint16	Default
6	0xEA97	0x20007C02 (All)	uint16	Default
7	0xEAA8	0x20007C04 (All)	uint16	Default
8	0xEAA8	0x20007C06 (All)	uint16	Default
9	0xEAA1	0x20007C08 (All)	uint16	Default
10	0xEA9C	0x20007C0A (All)	uint16	Default

Example 3: Array-to-Point Transfer – Eg3_Mem_DMA_DAC

The test setup is as follows:

1. Connect P3[2] to the oscilloscope - This is the VDAC output.
2. Build the project and program the device.
3. Observe a sine wave of frequency 12.5 kHz on the oscilloscope.

Example 4: Array-to-Array Transfer – Eg4_Mem_DMA_Mem

The test setup is as follows:

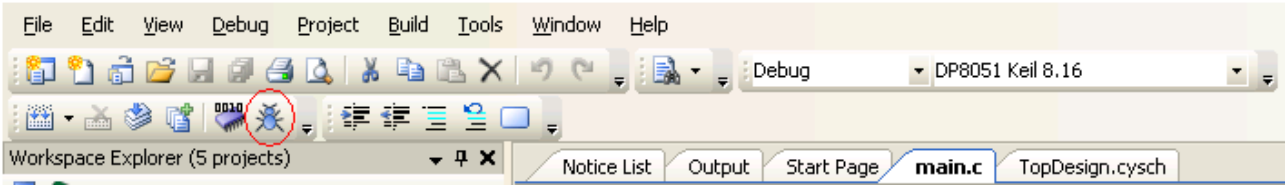
1. Place Character LCD on header P18 (LCD Module - Port 2) of the CY8CKIT-001 PSoC Development Kit.
2. Make sure jumper J12 is in the ON position to power the LCD.
3. Build the project and program the device.
4. Look at the LCD display. The first row displays counts from 00 to 07h. The second row displays the message TRANSFERRED. A sample output LCD display is as follows.

Example 5: Ping-Pong Buffering – Eg5_PingPongBuffer

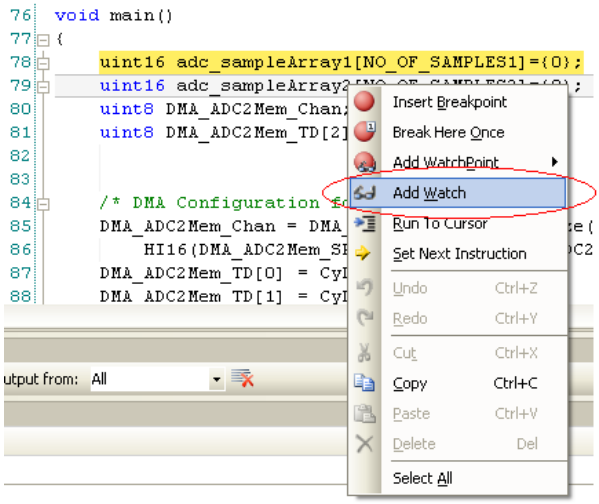
The test setup for this example is same as that of Example 2. The test setup is as follows:

1. Connect the input signal to pin P0[0] (ADC input)on the DVK. Make sure that the input is within the ADC range V_{SSA} to 2.048V.
2. Connect P1[2] to switch(SW1) on the DVK.

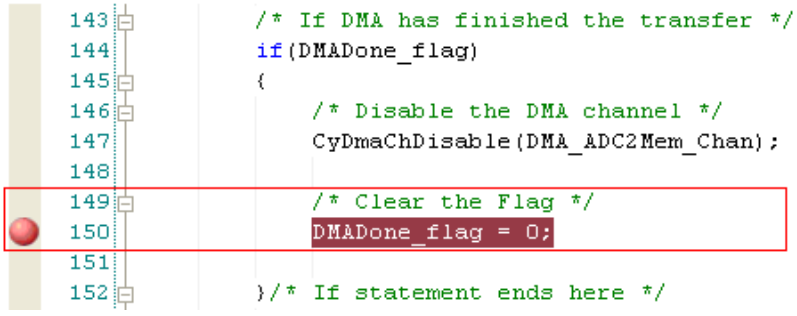
3. Build the project.
4. Press F5 or click the debug icon to download the program and start debugging.



5. Add adc_samplearray1 and adc_samplearray2 as watch variables.



6. Put a breakpoint inside the **if**(DMADone_flag) loop as shown in the figure.



7. Press F5 to run the program. Press the switch (SW1) connected to P1[2] to enable the DMA to start ADC sample buffering.
8. The execution stops at the breakpoint after the DMA has transferred the specified number of samples from ADC to memory. To verify the result, monitor adc_sample array1 and adc_samplearray2 in the watch window.

TopDesign.cysch | main.c | **Watch 1**

Name	Value
adc_sampleArray1	[50]
0	0x6E67
1	0x6E61
2	0x6E5F
3	0x6E60
4	0x6E67
5	0x6E68
6	0x6E65
7	0x6E61
8	0x6E61
9	0x6E60
10	0x6E60

TopDesign.cysch | main.c | **Watch 1**

Name	Value
adc_sampleArray2	[20]
0	0x6E60
1	0x6E61
2	0x6E5E
3	0x6E5D
4	0x6E62
5	0x6E60
6	0x6E5F
7	0x6E5D
8	0x6E5E
9	0x6E60
10	0x6E5F

Document History

Document Title: PSoC® 3 and PSoC 5 - Getting Started with DMA – AN52705

Document Number: 001-52705

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2710860	LNAT	05/25/09	New Application Note.
*A	2768731	LNAT	09/24/09	Updated the projects for PSoC Creator Beta 3 version. Added information about configuring the Termout signals
*B	2951774	LNAT	06/14/10	Updated the projects for PSoC Creator Beta 4.1 Added more information regarding the DMA configuration
*C	2966485	LNAT	08/26/10	Updated the projects for PSoC Creator Beta 5. Used DMA Wizard in the projects.
*D	3269575	LRDK	06/06/11	Rewritten in Simplified English.
*E	3355465	ANUP	08/26/2011	Updated Introduction Updated TD0 Configuration section Updated channel configuration table Updated Figure 8 Updated operation section.
*F	3444066	ANMD	11/22/2011	Project updates for PSoC Creator 2.0. Updated template.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

PSoC is registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2009-2011. 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.

This 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.