

AN84810

PSoC® 3 and PSoC 5LP Advanced DMA Topics

Author: Ranjith M

Associated Project: Yes

Associated Part Family: All PSoC® 3 and PSoC 5LP parts

Software Version: PSoC Creator™ 3.0 SP2

Related Application Notes: [AN52705](#), [AN61102](#)

AN84810 discusses several advanced PSoC® 3 and PSoC 5LP direct memory access (DMA) topics and design challenges. This application note builds upon the fundamental concepts introduced in [AN52705 – Getting Started with DMA](#). Topics covered include indexed transfers, timing and bandwidth considerations, data alignment, and DMA debugging tips.

Contents

Introduction	1
DMA Considerations	2
DMA Terms and Definitions.....	2
DMA Timing.....	3
DMA Channel Priority Handling	7
Terminating a TD Chain	8
Multi-Byte Data Alignment.....	9
Writing to Standard Registers and Components	15
Modifying a TD Dynamically.....	24
Indexed DMA.....	24
Nested DMA	25
Debugging DMA	26
Common Issues	26
Debugging Methods	28
Projects	29
Parallel to Serial Converter Project	29
Nested DMA Project.....	29
Summary.....	30
Appendix A: Memory Maps	31
Appendix B: Termination Request Signal.....	33
Appendix C: DMA Channel Arbitration Flow Diagram	36
Appendix D: Misaligned Data Transfers	37
Worldwide Sales and Design Support.....	41

Introduction

Direct memory access (DMA) controllers transfer data between peripherals and memory without CPU intervention. The DMA controller (DMAC) in PSoC® 3 and PSoC 5LP features 24 channels and 128 transaction descriptors (TDs), making it very versatile for a wide variety of applications. PSoC Creator™, the development environment for PSoC, has tools including a DMA wizard and component APIs that make it easy to design complex DMA functions.

This application note teaches you advanced methods to get maximum performance from the PSoC DMAC. The topics covered include DMA timing, DMA channel priority handling, multi-byte data alignment, and methods to modify a TD dynamically.

This application note assumes that you are already familiar with the topics discussed in the basic DMA application note, [AN52705 – Getting Started with DMA](#). It also assumes that you are familiar with developing applications using PSoC Creator for PSoC 3 or PSoC 5LP. If you are new to these products, you can find introductions in [AN54181 – Getting Started with PSoC 3](#) and [AN77759 – Getting Started with PSoC 5LP](#). If you are new to PSoC Creator, see the [PSoC Creator home page](#).

DMA Considerations

The DMAC is one of the most useful components in PSoC 3 and PSoC 5LP. However, it also exposes a whole new set of design considerations with which you may not be familiar. Here are some points to consider when designing a system with DMA.

DMA Terms and Definitions

Following is a list of terms and definitions that are used in this application note. These terms are described in detail in [AN52705 – Getting Started with DMA](#) but are included here for clarity.

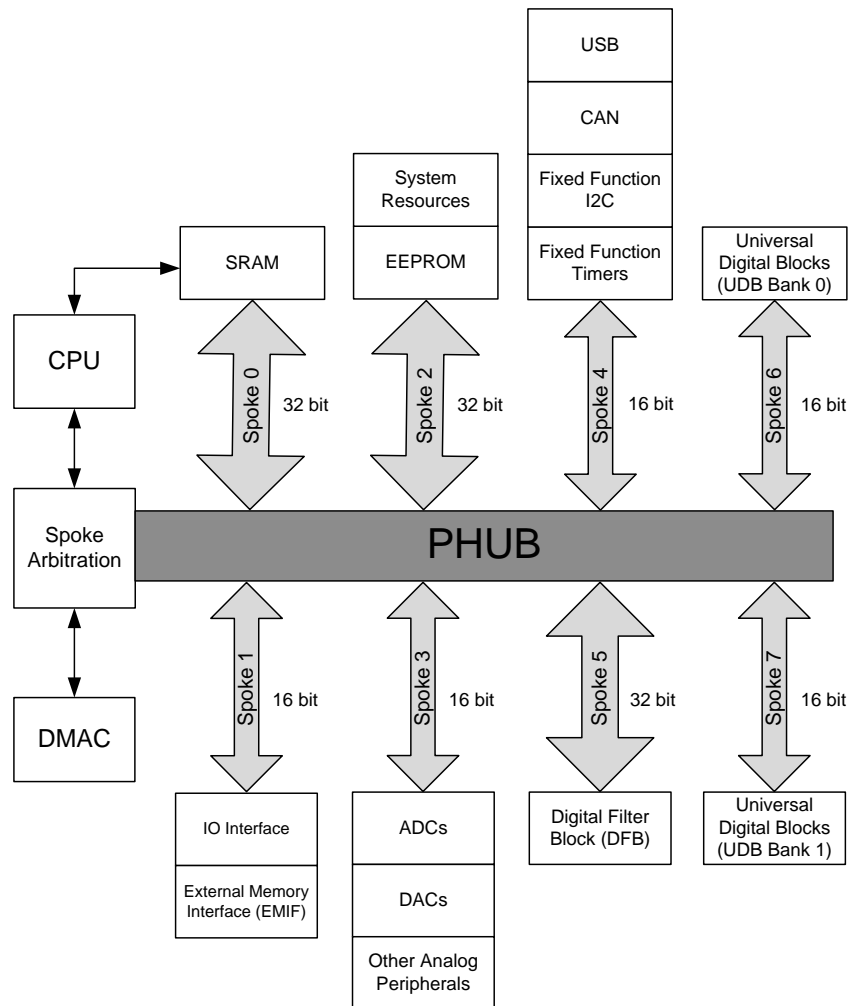
Peripheral HUB (PHUB): The PHUB is the central hub that has data buses connected between the CPU, DMAC, and on-chip peripherals and memory.

Spoke: Spokes are data buses that branch out from the PHUB to peripherals. Spoke widths can be 16 or 32 bits; see the device datasheet and Technical Reference Manual (TRM) for details. [Figure 1](#) shows the PHUB and spoke connections.

Channel: DMA channels use the PHUB to transfer data. A channel fetches transaction descriptors, accesses the PHUB spokes for the source and the destination, and transfers data.

Transaction descriptor: A TD stores all information required for a data transfer, including the source and destination addresses and the number of bytes to transfer. Multiple chained TDs can be allocated to a single DMA channel.

Figure 1. Peripheral HUB



DMA Timing

This section describes how to calculate the best-case timing for a DMA channel. Many factors can cause the DMA to deviate from this best-case timing; those factors are also discussed here.

The PSoC DMAC operates at the same frequency as the CPU, which is the bus clock frequency. Most PSoC family members run at a bus clock frequency as high as 67 MHz. DMA data transfers are either intra-spoke (within the same PHUB spoke) or inter-spoke (between different PHUB spokes).

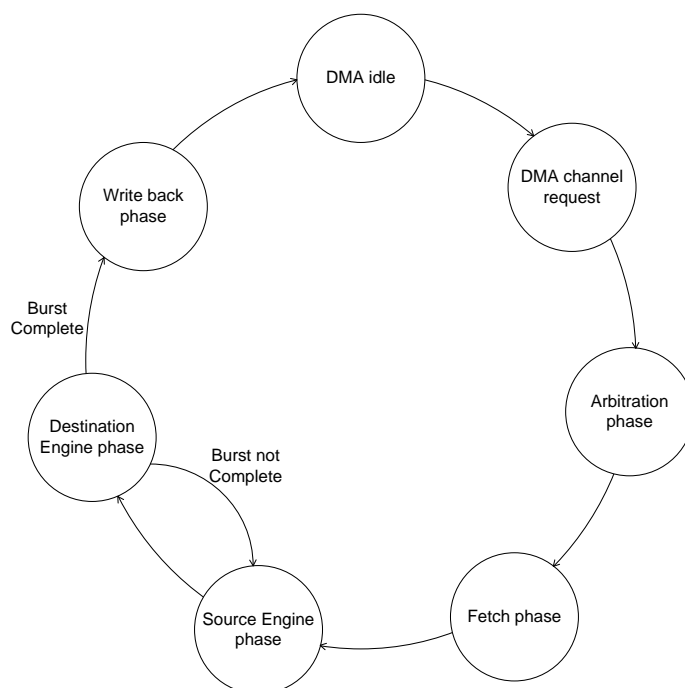
For each data transfer, the DMA implements a set of phases, as [Figure 2](#), [Figure 3](#), and [Figure 4](#) show:

- **DMA request (DRQ) latch phase:** It takes one clock cycle for the DMA request to be latched into the DMAC.
- **Arbitration phase:** This phase is used to arbitrate between simultaneous requests from multiple DMA channels. One clock cycle is required for this phase. If a channel loses arbitration, it reenters the queue and waits for the next arbitration cycle.
- **Fetch phase:** This phase is used to fetch the TD and configuration information for the channel. One clock cycle is required.
- **Source engine phase:** This phase is used to select the spoke to which the source memory or peripheral is connected. If the spoke is being used by another bus master, that is, the CPU, data transfer from the source is delayed until the spoke is available. The source engine phase initially consists of a bus control cycle followed by a data cycle. Then the control and data cycles are pipelined in parallel.

- Destination engine phase: This phase selects the spoke to which the destination peripheral is connected. The data collected in the source engine phase is transferred to the destination peripheral as soon as the spoke is available. The destination engine phase initially consists of a bus control cycle followed by a data cycle. Then the control and data cycles are pipelined in parallel.
- 1. Write back phase: In this phase, the TD and DMA channel configurations are updated after the data transfer. This phase requires one clock cycle.

Figure 2 shows a simple state diagram for a DMA transfer. Figure 3 and Figure 4 on page 5 show the actual timing for an inter-spoke and an intra-spoke data transfer.

Figure 2. DMA Data Transfer State Diagram



The number of clock cycles required for the source engine and destination engine phases is the same for an inter-spoke and an intra-spoke data transfer. However, the two phases can happen in parallel for an inter-spoke transfer, which usually requires fewer clock cycles.

Figure 3. Timing Diagram for DMA Inter-Spoke Data Transfer

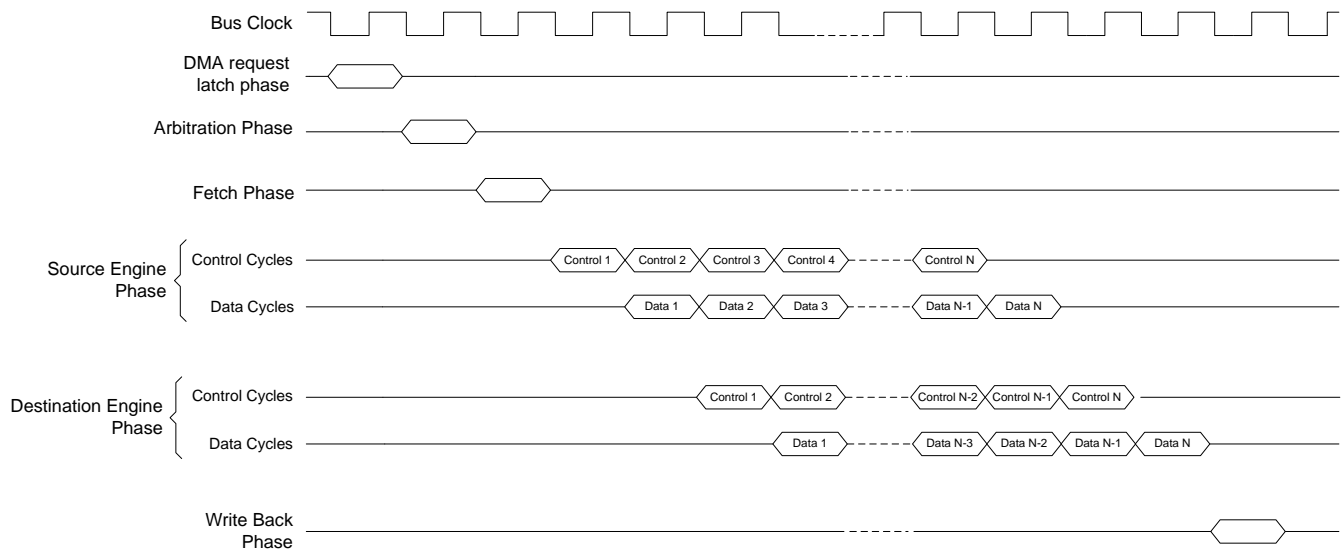
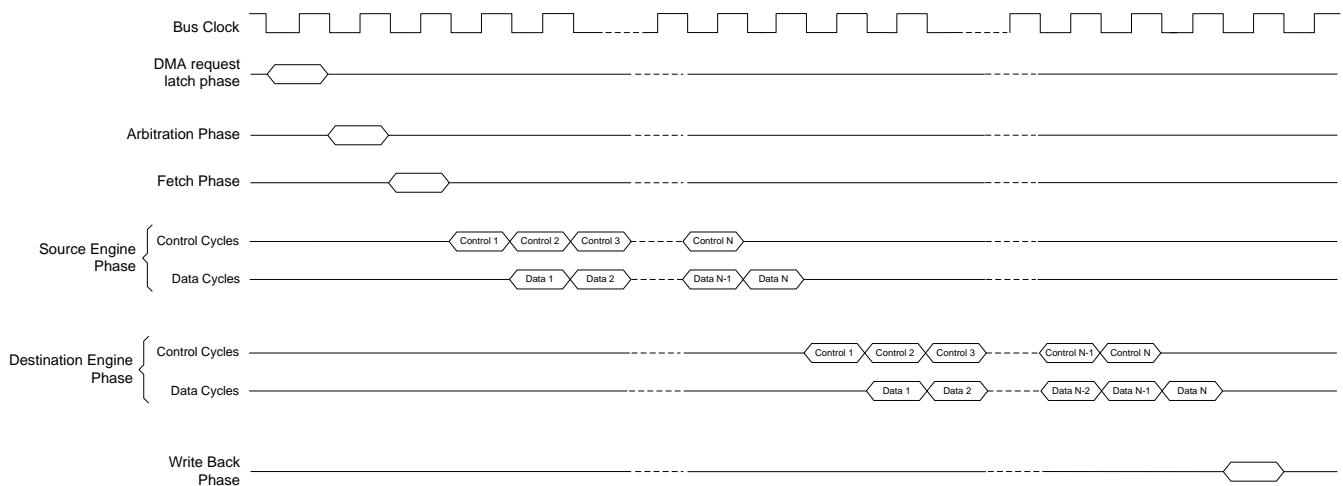


Figure 4. Timing Diagram for DMA Intra-Spoke Data Transfer



When calculating DMA timing, start with the ideal conditions, as follows:

- Only one DMA channel is active, and there is no arbitration between multiple DMA channels.
- There is no arbitration between the DMAC and the CPU. Note that the CPU and the DMAC can access different spokes simultaneously.
- The source and the destination spokes are available when the transfer is to be done.
- The source and destination spoke widths are the same.
- The data length is an even multiple of the spoke width, in bytes.
- The source and destination start addresses are on a spoke width boundary.

The number of bursts required for a DMA transfer N is defined as follows:

$$\text{Number of bursts } [N] = \frac{(\text{Transfer Count})}{(\text{Spoke Width})} \quad (1)$$

where transfer count and spoke width are in bytes. See [AN52705](#) for details on DMA burst transfers.

If the ideal conditions are met, the number of clock cycles required for an inter-spoke data transfer is calculated as follows:

$$\begin{aligned} C = & 1 \text{ clock cycle to latch the drq signal} + \\ & 1 \text{ clock cycle for arbitration phase} + \\ & 1 \text{ clock cycle for fetch phase} + \\ & (N + 3) \text{ clock cycles for source and destination engine phases} + \\ & 1 \text{ clock cycle for write back phase} \end{aligned} \quad C = N + 7 \quad (2)$$

The number of clock cycles required for an intra-spoke data transfer is calculated as follows:

$$\begin{aligned} C = & 1 \text{ clock cycle to assert the drq signal} + \\ & 1 \text{ clock cycle for arbitration phase} + \\ & 1 \text{ clock cycle for fetch phase} + \\ & (N + 1) \text{ clock cycles for source engine phase} + \\ & (N + 1) \text{ clock cycles for destination engine phase} + \\ & 1 \text{ clock cycle for write back phase} \end{aligned} \quad C = 2N + 6 \quad (3)$$

The exact number of clock cycles required for a DMA transaction under non-ideal conditions varies depending on the conditions. Use the following tips to calculate the clock cycles required for a non-ideal DMA transaction:

1. The clock cycle to assert the drq signal is not required for continuous DMA transfers. For example, if the transfer length is set to 100 and the request per burst is set to 0 (complete transfer in a single request), the latching of the request is required only for the first DMA transfer.
2. The arbitration between multiple DMA channels takes only one clock cycle. If the DMAC is already performing a data transfer, arbitration is performed in parallel with the data transfer, and this clock cycle is hidden.
3. If the CPU is competing with the DMAC for a spoke, the DMAC must wait until the CPU releases the spoke if the CPU is assigned a higher priority than the DMAC. The priority of the CPU over the DMAC is controlled using the `spk_cpu_pri[6:0]` bits in the `PHUB_CFG` register. See the “PHUB” section in the PSoC Registers TRM for details.
4. If the source or destination start address is not on a spoke width boundary, the number of clock cycles required for the DMA transfer varies depending upon the spoke width and the addresses. The formulas in Equations (1) and (2) for inter-spoke and intra-spoke transfers can also be used in this case, with a change in the value of N as calculated by Equation (3).

If the source and destination widths are 32 bits, N is unchanged if the source and destination addresses are 32-bit DWORD aligned. N is multiplied by 2 if the source and destination are WORD aligned, and N is multiplied by 3 if the source and destination are BYTE aligned. See the section [Multi-Byte Data Alignment](#) for details.

5. If endian swapping is enabled for an inter-spoke DMA transaction, the destination transfer cannot occur until the last source byte has been put into the DMAC FIFO. The DMAC must write the last source byte to the first location of the destination to do the endian swap. The effective value of N in this case is calculated as $(N \text{ calculated with Source spoke width}) + (N \text{ calculated with Destination spoke width}) - 1$.
6. If the source and destination peripherals have unequal spoke widths, the number of clock cycles required for a transfer is governed by the narrower spoke width. In this case, $N = (\text{Transfer Count}) / (\text{Narrower Spoke Width})$.

Table 1 shows the number of clock cycles required for a DMA transfer in different scenarios to transfer N bursts of data.

Table 1. Number of Clock Cycles for DMA Transfer between PSoC Resources

Use Case	Number of Clock Cycles
Flash to SRAM	$N + 7$
SRAM to SRAM	$2N + 6$
SRAM to peripherals	$N + 7$
Peripherals to SRAM	$N + 7$
Peripherals to peripherals	$2N + 6$

Appendix C: DMA Channel Arbitration Flow Diagram provides a flow diagram for DMA channel arbitration.

DMA Channel Priority Handling

Each of the 24 DMA channels is assigned a priority value ranging from 0 to 7, with 0 being the highest priority and 7 being the lowest. Since there are only 8 different priorities and 24 DMA channels, multiple channels may have the same priority.

For channels having the same priority, two methods are used to decide which channel gets priority:

1. Simple priority: The lowest channel number has a higher priority. This is enabled by default.
2. Round robin: Round robin priority ensures that all channels get an equal opportunity to access the PHUB. This is disabled by default and is set using the API function `CyDmaChRoundRobinEnable()`.

For channels with different priorities, two rules are applied to determine which channel gets priority:

1. Simple priority: The channel with the lowest priority number gets priority.
2. Grant allocation fairness algorithm: The grant allocation fairness algorithm is designed such that even the lowest priority channels get access once in a while. With this method, channels with priorities 0 and 1 always have 100 percent access and are not interrupted, with the exception that 0 is higher priority than 1. The channels with priorities 2 to 7 are given access according to Table 2.

The grant allocation fairness algorithm is enabled by default. You can disable it by setting the simple priority bit (bit 23) in the PHUB_CFG register. See the “PHUB” section of the [PSoC 3 Registers TRM](#) and [PSoC 5LP Registers TRM](#) for details.

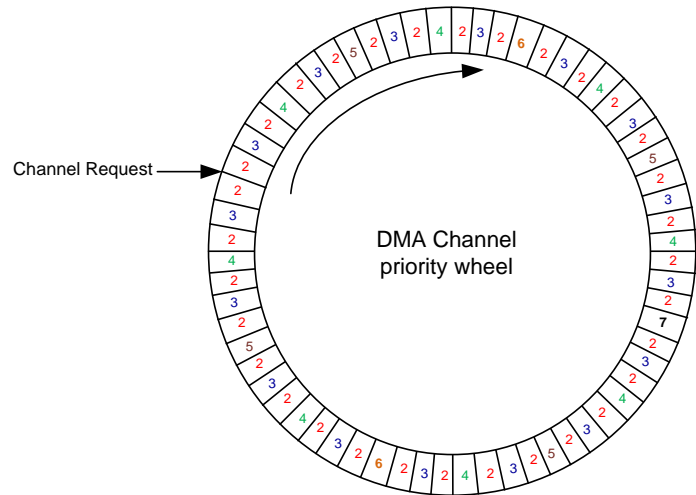
Table 2. Channel Priority Distribution

Channel Priority	Bandwidth (%)
2	50
3	25
4	12.5
5	6.25
6	3
7	1.5

Note that [Table 2](#) applies only if DMA channels with all the priorities are requesting simultaneously. Otherwise, the DMA channel with a higher priority is given more access than [Table 2](#) shows.

[Figure 5](#) shows a channel priority wheel that describes how the next 63 requests are handled if all channels with priorities 2 to 7 are requesting simultaneously.

Figure 5. DMA Channel Priority Wheel



The bandwidth utilization of the DMA channel can be defined as the number of clock cycles utilized by the DMA channel for the data transfer as a percentage of the total number of available clock cycles across sustained DMA requests of burst length N.

$$\text{For inter-spoke data transfers: } BW = \frac{N}{(N+7)} \times 100 \quad (4)$$

$$\text{For intra-spoke data transfers: } BW = \frac{2N}{(2N+6)} \times 100 \quad (5)$$

If a channel with priority 2 to 7 is **not** requesting, the slots of the missing channel priority are used by the channel with the highest priority. In that case, channels with a higher priority get more access than [Figure 5](#) shows.

Terminating a TD Chain

In some cases, a TD must be terminated before data transfer is completed. This is called non-count termination. There are three methods to terminate a TD and abort a DMA transaction:

- API function call to terminate the current TD [CyDmaChSetRequest(channel, CPU_TERM_TD)]
- API function call to terminate the current TD chain [CyDmaChSetRequest(channel, CPU_TERM_CHAIN)]
- Hardware trq signal

Use the API function CyDmaChSetRequest() to terminate either the current TD or the entire TD chain. This API function disables the DMA channel and terminates the chain of TDs if the second parameter is set to CPU_TERM_CHAIN. This API terminates the current TD but does not disable the DMA channel if the second parameter is set to CPU_TERM_TD.

When one of these methods is used, the DMA channel is reconfigured as if the current transaction has completed normally. If enabled, the nrq signal is activated; see the [DMA Component datasheet](#) for details.

Note The DMA channel completes any ongoing transaction before terminating, so termination may require some cycles for the final transaction to be complete. The DMAC may transfer more data bytes before terminating the TD. You should be extremely careful while accessing these memory locations after a terminate TD API request as the function is not blocking and may return before the data transfer has been actually terminated.

Using the Hardware trq Signal

You can also use the hardware signal trq, or termination request. This signal, when asserted during the source engine phase of a transaction, stops the DMA transaction.

This is an effective way to terminate a TD using a hardware signal when the transfer count is set to zero. If the transfer count parameter of a TD is set to zero, the TD runs indefinitely unless it is terminated by a non-count termination. See [Appendix B: Termination Request Signal](#) for an example of how to use the trq signal.

Note The trq signal is used only when the DMA is trying to transfer data. A positive edge on this line is ignored at all other times.

Multi-Byte Data Alignment

One feature of the DMAC is that it can transfer more than one byte in a single bus cycle. This allows more efficient and faster data transfers.

The section [DMA Timing](#) introduced the concept of N:

$$\text{Number of bursts } [N] = \frac{(\text{Transfer Count})}{(\text{Spoke Width})} \quad (6)$$

Here are some examples:

For Transfer Count = 2; Spoke Width = 2: N = 1

For Transfer Count = 4; Spoke Width = 2: N = 2

For Transfer Count = 3; Spoke Width = 2: N = 2 (always round up to the next integer)

For Transfer Count = 4; Spoke Width = 4: N = 1

All of these calculations assume that the source and destination addresses are aligned. There are three possible ways for addresses to be aligned: BYTE, WORD, and DWORD, as [Figure 6](#) shows:

- BYTE = All addresses
- WORD = Even or 16-bit addresses: 0x00, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x0C, 0x0E, and so on
- DWORD = 32-bit addresses: 0x00, 0x04, 0x08, 0x0C, and so on

Figure 6. Notation to Represent Possible Data Alignments in the Memory

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
DWORD				DWORD				DWORD				DWORD			
WORD		WORD		WORD		WORD		WORD		WORD		WORD		WORD	
BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

To reduce the number of clock cycles needed for a data transfer, align the source and destination addresses with the spoke width boundaries. An address is aligned with a 32-bit spoke width boundary if the address is DWORD aligned. An address is aligned with a 16-bit wide spoke if the address is WORD aligned. BYTE-aligned addresses are never aligned with any spoke width boundary.

Note In some cases, if the addresses are not aligned with the spoke width boundaries, the DMAC may fetch incorrect data from the source or write incorrect data to the destination address. [Appendix D: Misaligned Data Transfers](#) details the results of misaligned data transfers.

You can avoid incorrect data transfers by using one of the following methods:

1. Enable the increment source address or increment destination address option in the TD configuration.
2. Force the source and destination addresses to the spoke width boundaries.

[Figure 7](#) and [Figure 8](#) show WORD-aligned and BYTE-aligned arrangements for 16-bit data, respectively.

Figure 7. WORD-Aligned Arrangement of 16-Bit Data

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
			DATA VALUE												
WORD			WORD			WORD			WORD			WORD			

Figure 8. BYTE-Aligned Arrangement of 16-Bit Data

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
			DATA VALUE												
BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

A data transfer from address location “n” is carried out in a single clock cycle if the data at the source is aligned with the spoke boundary, as [Figure 9](#) shows. When the data is BYTE aligned, data transfer takes two clock cycles, as [Figure 10](#) shows.

Figure 9. Data Transfer of WORD-Aligned 16-Bit Data

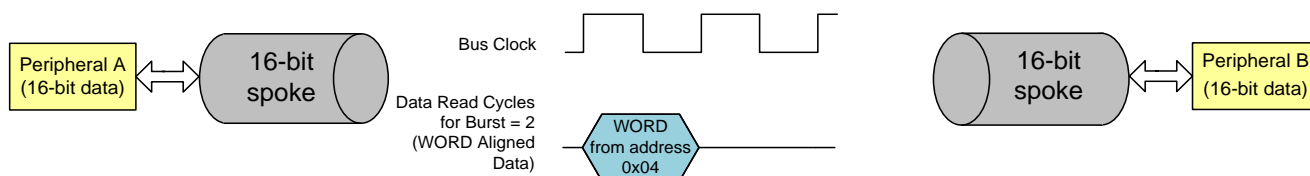
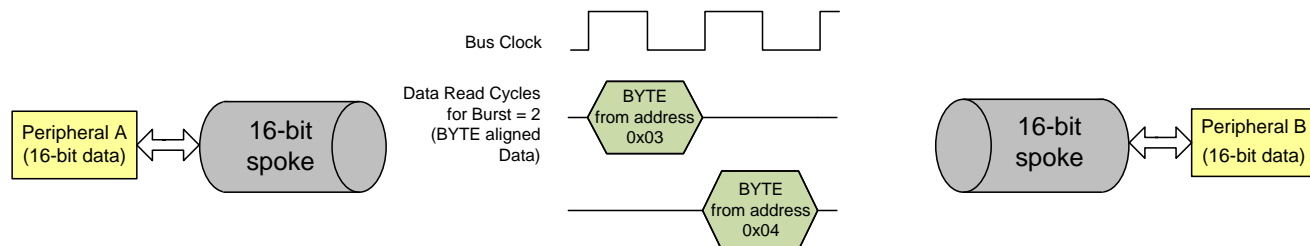


Figure 10. Data Transfer of BYTE-Aligned 16-Bit Data



[Figure 11](#), [Figure 12](#), and [Figure 13](#) show the DWORD-aligned, WORD-aligned, and BYTE-aligned arrangements for a 32-bit data transfer, respectively.

Figure 11. DWORD-Aligned Arrangement of 32-Bit Data

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
				DATA VALUE											
DWORD				DWORD				DWORD				DWORD			

Figure 12. WORD-Aligned Arrangement of 32-Bit Data

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
		DATA VALUE													
WORD		WORD		WORD		WORD		WORD		WORD		WORD		WORD	

Figure 13. BYTE-Aligned Arrangement of 32-Bit Data

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
			DATA VALUE												
BYTE	BYTE	BYTE	BYTE	WORD	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

The data transfer from address location “n” is carried out in a single clock cycle if the data at the source is aligned with the spoke boundary, as Figure 14 shows. When the data is WORD aligned, the data transfer takes two clock cycles, and when the data is BYTE aligned, data transfer takes three clock cycles, as Figure 15 and Figure 16 show.

Figure 14. Data Transfer of DWORD-Aligned 32-Bit Data

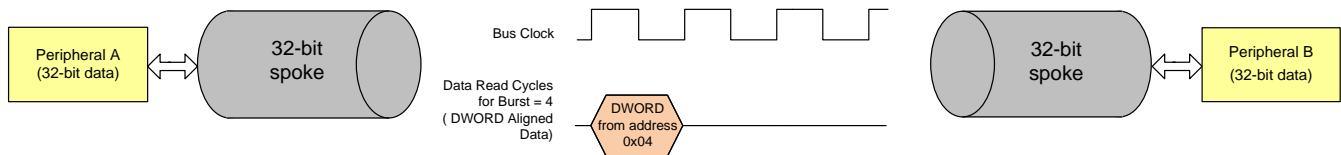


Figure 15. Data Transfer of WORD-Aligned 32-Bit Data

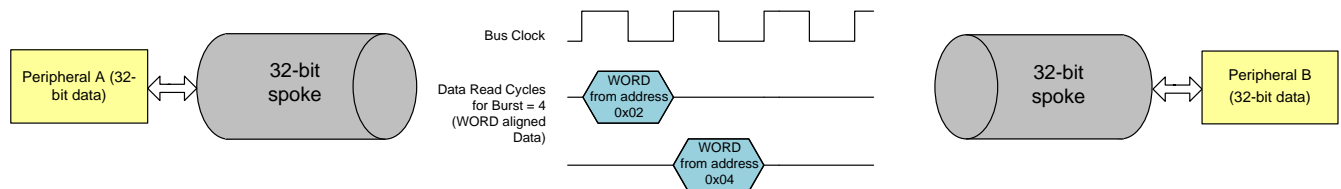
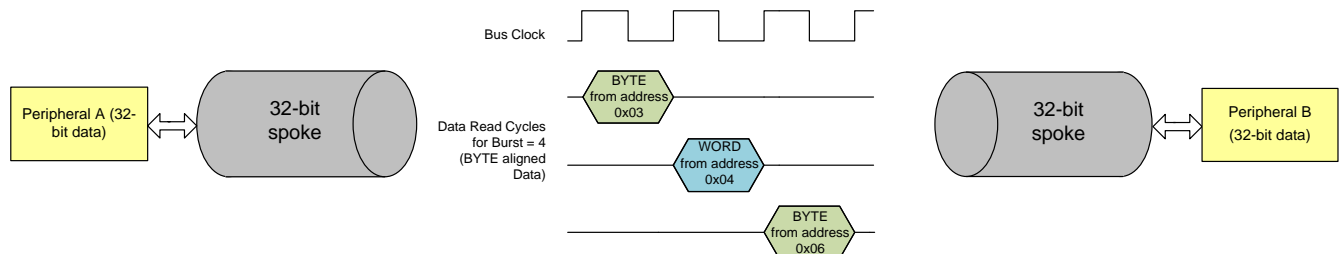


Figure 16. Data Transfer of BYTE-Aligned 32-Bit Data



Note If the transfer count is odd, the number of cycles is different. For example a 3-byte burst on a 32-bit spoke takes two clock cycles, even if the addresses are DWORD aligned: one cycle to transfer two bytes, and another to transfer the third byte. Also, if increment source and destination address are not enabled in the TD configuration, only the first two bytes are transferred.

For the Keil compiler, use the keyword `_at_` to force variables into absolute address locations in SRAM. This ensures that variables are aligned with even address boundaries, for example:

```
uint8 myVariable _at_ 0x1000;
```

Note Variables forced to absolute memory locations cannot be initialized.

For the GCC compiler, use the keyword `__attribute__` to force variables to 16-bit or 32-bit address boundaries. For example:

```
uint32 var __attribute__ ((aligned(32)));
```

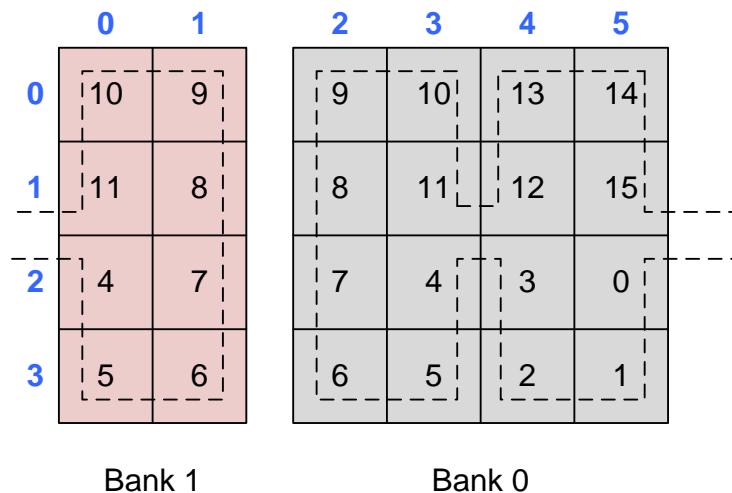
You can replace the number 32 in the previous format with 16 to align the variables with 16-bit address boundaries. You must use the appropriate keywords to align the data with the spoke width boundaries, depending upon your compiler.

Aligning UDBs

You can do DMA transfers to and from the registers in UDB-based PSoC Creator Components, though enabling the increment source or destination address is not always a viable option. To work around this, you may need to force the Component to be aligned with an address boundary. You can use PSoC Creator directives to force a UDB-based Component to start at a specific address.

There are as many as 24 UDBs in a PSoC device, and each set of UDB registers has a particular base address. The base addresses for the UDBs are provided in the [PSoC 3 Registers TRM](#) and [PSoC 5LP Registers TRM](#). The UDBs are distributed in the form of two banks and are numbered in a specific fashion, as [Figure 17](#) shows. The dashed line represents how UDBs are chained.

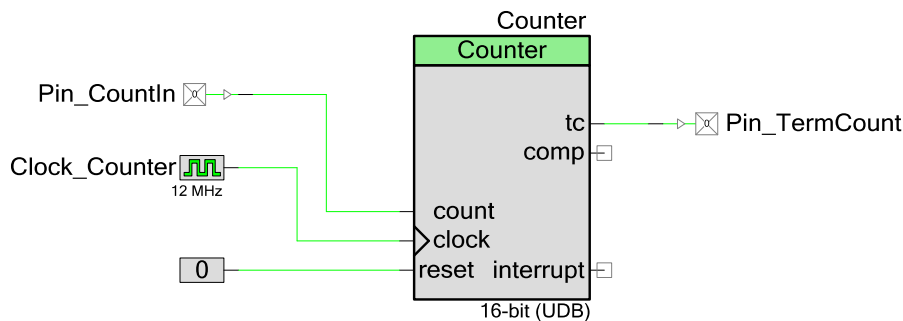
Figure 17. Organization of PSoC UDBs



The numbers in blue on the top and left of the diagram are the indices that PSoC Creator uses to address each of the UDBs. The numbers inside the squares indicate the UDB number. For example, UDB0 is addressed as U(2,5). To find out where PSoC Creator places a Component in the UDB array, follow these steps:

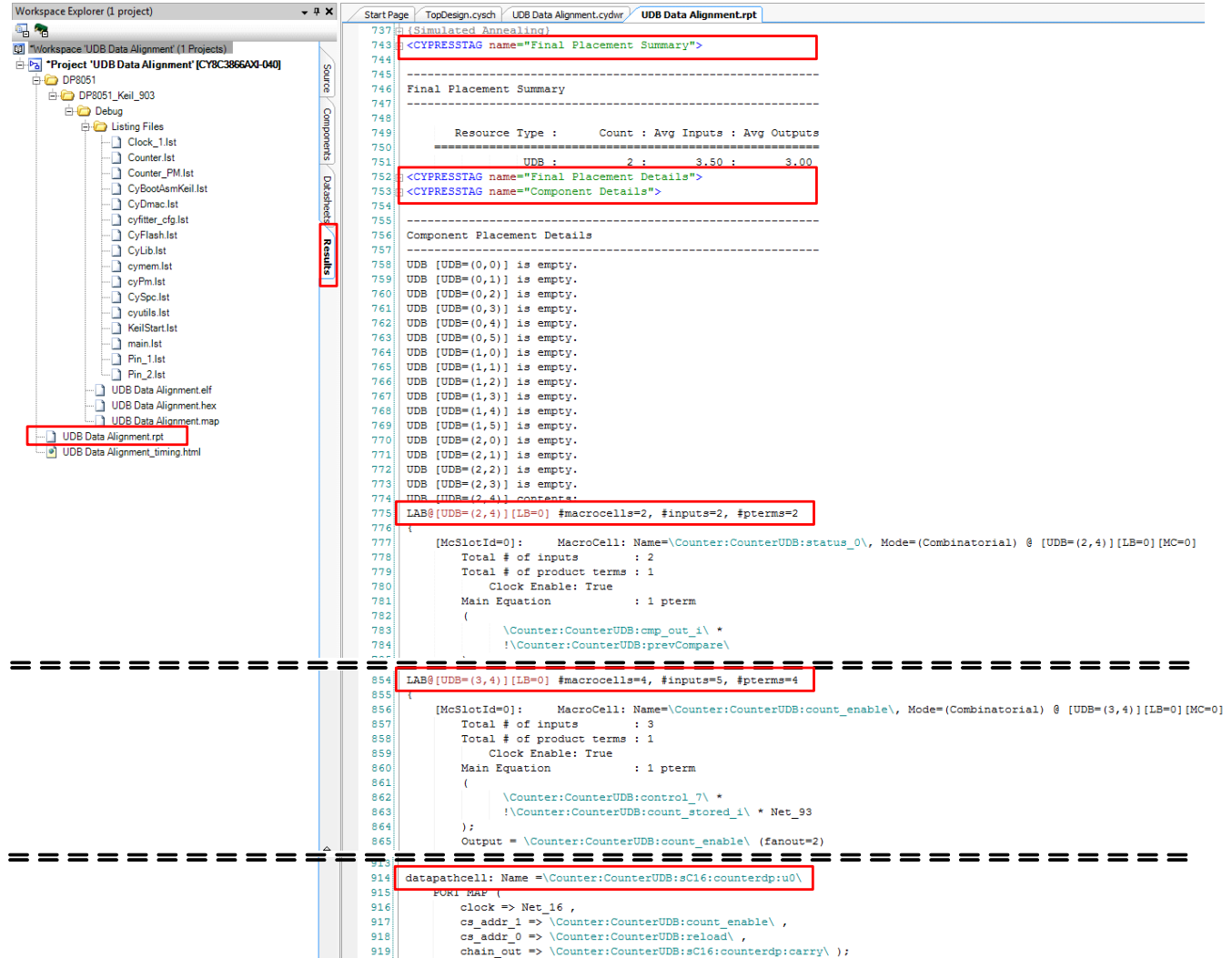
1. Place a 16-bit UDB-based Counter Component in the TopDesign, as [Figure 18](#) shows, and build the project.

Figure 18. Counter Component



- Examine the .rpt file to find the location where PSoC Creator has placed the Component, as Figure 19 shows.

Figure 19. Finding the UDB Location of the Component



The screenshot shows the PSoC Creator workspace with the 'UDB Data Alignment.rpt' file open in the 'Results' tab. The report content is as follows:

```

737 (Simulated Annealing)
743 <CYPRESSTAG name="Final Placement Summary">
744 -----
745 Final Placement Summary
746 -----
747
748 Resource Type : Count : Avg Inputs : Avg Outputs
749 -----
750 UDB : 2 : 3.50 : 3.00
751 <CYPRESSTAG name="Final Placement Details">
752 <CYPRESSTAG name="Component Details">
753 -----
754 Component Placement Details
755 -----
756
757 UDB [UDB=(0,0)] is empty.
758 UDB [UDB=(0,1)] is empty.
759 UDB [UDB=(0,2)] is empty.
760 UDB [UDB=(0,3)] is empty.
761 UDB [UDB=(0,4)] is empty.
762 UDB [UDB=(1,0)] is empty.
763 UDB [UDB=(1,1)] is empty.
764 UDB [UDB=(1,2)] is empty.
765 UDB [UDB=(1,3)] is empty.
766 UDB [UDB=(1,4)] is empty.
767 UDB [UDB=(2,0)] is empty.
768 UDB [UDB=(2,1)] is empty.
769 UDB [UDB=(2,2)] is empty.
770 UDB [UDB=(2,3)] is empty.
771 UDB [UDB=(2,4)] contains:
772 LAB@ [UDB=(2,4)] [LB=0] #macrocells=2, #inputs=2, #pters=2
773 {
774 [McSlotId=0]: MacroCell: Name=\Counter:CounterUDB:status_0\, Mode=(Combinatorial) @ [UDB=(2,4)] [LB=0] [MC=0]
775 Total # of inputs : 2
776 Total # of product terms : 1
777 Clock Enable: True
778 Main Equation : 1 pterm
779 (
780 \Counter:CounterUDB:cmp_out_i\ *
781 !\Counter:CounterUDB:prevCompare\
782 )
783 }
784
785 LAB@ [UDB=(3,4)] [LB=0] #macrocells=4, #inputs=5, #pters=4
786 {
787 [McSlotId=0]: MacroCell: Name=\Counter:CounterUDB:count_enable\, Mode=(Combinatorial) @ [UDB=(3,4)] [LB=0] [MC=0]
788 Total # of inputs : 3
789 Total # of product terms : 1
790 Clock Enable: True
791 Main Equation : 1 pterm
792 (
793 \Counter:CounterUDB:control_7\ *
794 !\Counter:CounterUDB:count_stored_i\ * Net_93
795 )
796 }
797 Output = \Counter:CounterUDB:count_enable\ (fanout=2)
798
799 datapathcell: Name =\Counter:CounterUDB:sC16:counterdp:u0\
800 FORK MAP (
801 clock => Net_16 ,
802 cs_addr_1 => \Counter:CounterUDB:count_enable\ ,
803 cs_addr_0 => \Counter:CounterUDB:reload\ ,
804 chain_out => \Counter:CounterUDB:sC16:counterdp:carry\ );

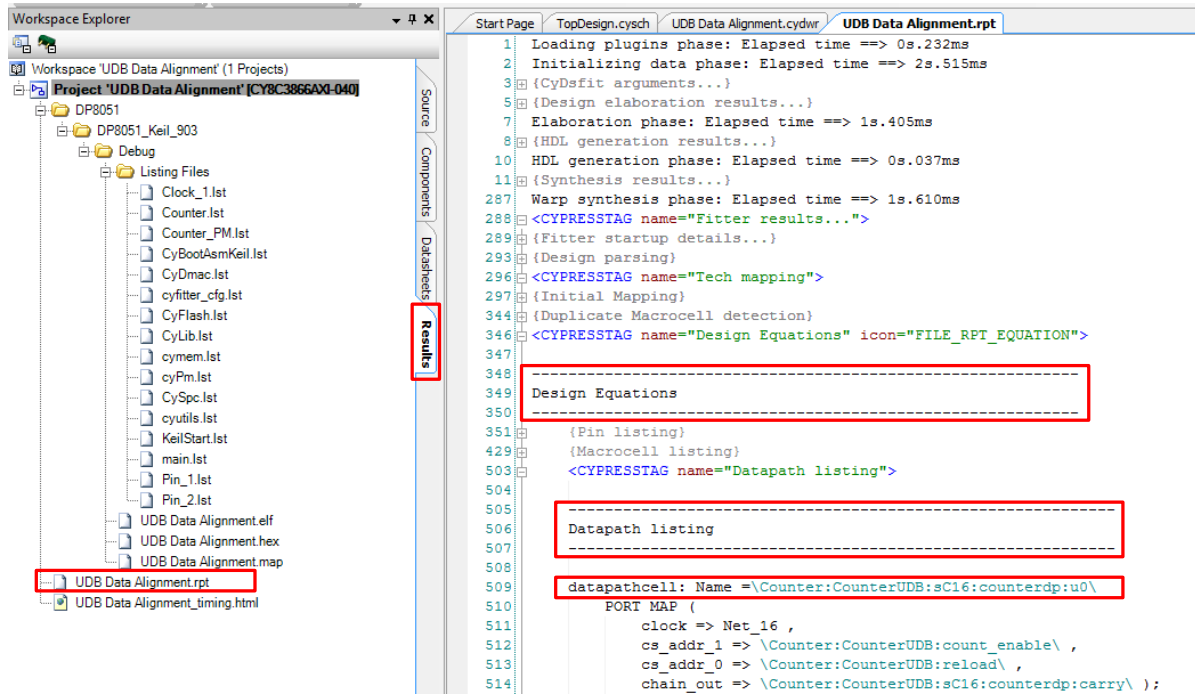
```

Figure 19 shows that PSoC Creator has placed the Component in UDBs U(2,4) and U(3,4) or UDB3 and UDB2. The LSB data path of the Component is located in UDB2. In the Registers TRM, the FIFO 0 (F0) address for UDB2 is 0x6442, which is a WORD-aligned address. Note that the UDB register space is on a 16-bit spoke, and the DMAC can transfer two bytes, or one word, in one clock cycle on this spoke.

If the Component placement was unaligned for some reason, follow these steps to align the counter FIFO 0 addresses with a word address boundary:

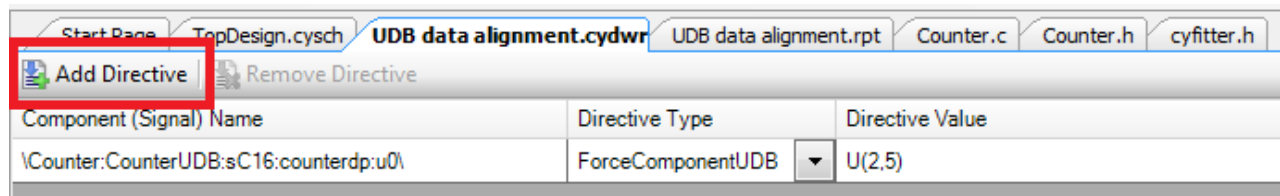
1. Open the .rpt file and locate the line “Design Equations,” as Figure 20 shows. Expand the list by clicking on the [-] symbol left of the line. Locate the line “Datapath listing” to find the fully elaborated name of the Component.

Figure 20. Locating the Fully Elaborated Name of the Component



2. Open the .cydwr file, go to the **Directives** tab, and add a directive using the **Add Directive** button, as Figure 21 shows. In the column “Component (Signal) Name,” type the fully elaborated Component name obtained from the .rpt file. Select “ForceComponentUDB” as the directive type. Enter the required UDB location as mentioned previously. To force the Counter to UDB0, where the FIFO 0 address is 0x6440, which is WORD aligned, enter U(2,5).

Figure 21. Adding the Directive



3. Rebuild the project, and then check the .rpt file again to confirm that the counter was placed in UDB(2,5).

Writing to Standard Registers and Components

The DMAC can transfer data to and from almost any memory or register location in PSoC. This includes the memory, UDBs, and registers of any PSoC Creator Component. However, the DMA Wizard provided with PSoC Creator supports data transfer between only a few Components.

To use DMA to transfer data between Components that are not supported by the DMA Wizard, you must find the addresses of the Components' registers. To do so, use the PSoC register maps in the [PSoC 3 Registers TRM](#) and [PSoC 5LP Registers TRM](#).

For example, the following steps show how to transfer a byte of data from a GPIO port to an SRAM location.

1. Place a Pins Component in the TopDesign and change the number of pins to 8, as [Figure 22](#) shows. Deselect the option hardware connection (HW Connection).
2. Enable the port interrupt control unit (PICU) interrupt, as [Figure 23](#) shows. The PICU interrupt is used to trigger the DMA channel to initiate a data transfer.

Figure 22. Configuring GPIO Pins (Type Tab)

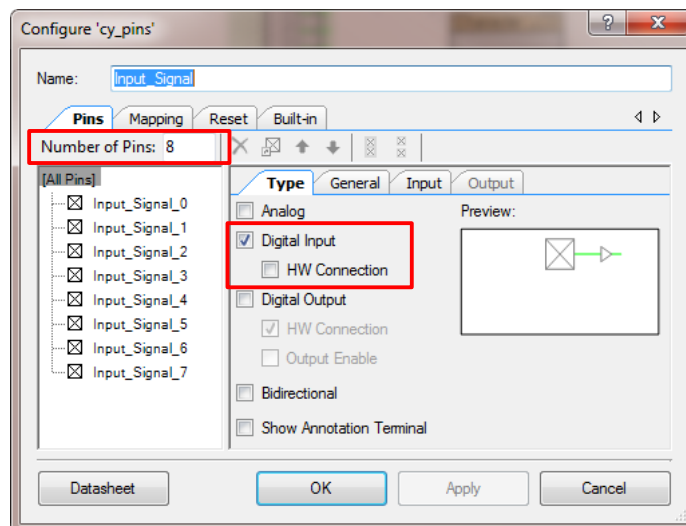
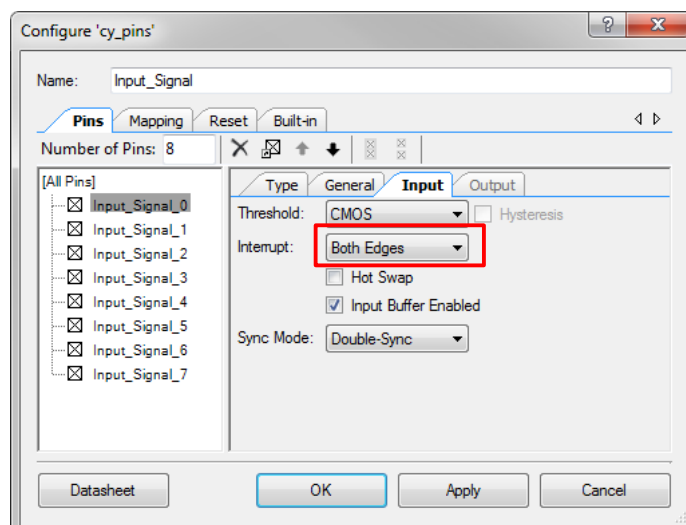
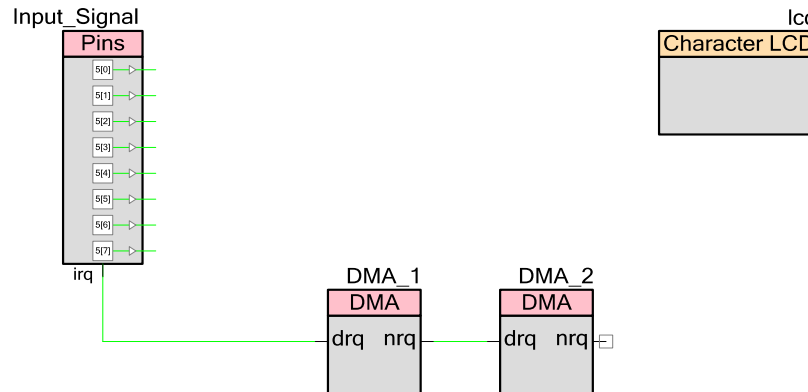


Figure 23. Configuring GPIO Pins (Input Tab)



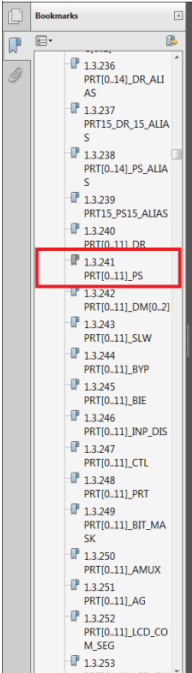
- Place two DMA Components and a Character LCD Component in the TopDesign. Wire the Components as Figure 24 shows.

Figure 24. Completed TopDesign



- Assign a suitable port for the GPIO using the .cydwr file. Then, if you selected port 5, for example, locate the address of port 5 in the register map, as Figure 25 shows.

Figure 25. Locate the Address From the Register Map



The screenshot shows a list of registers in a tree view. The register 'PRT[0..11]_PS' is highlighted with a red box.

1.3.241 PRT[0..11]_PS

Port Pin State Register1

Reset: System reset for retention flops [reset_all_retention]

Register : Address

PRT0_PS: 0x5101	PRT1_PS: 0x5111	PRT2_PS: 0x5121
PRT3_PS: 0x5131	PRT4_PS: 0x5141	PRT5_PS: 0x5151
PRT6_PS: 0x5161		

Bits	7	6	5	4	3	2	1	0
SW Access:Reset	R:00000000							
HW Access	R/W							
Name	PinState							

The Port Pin State Registers PRTxPS read the logical pin state for the corresponding GPIO port. Writes to this register have no effect. If the drive mode for the pin is set to High-Z Analog, the state will read 0 independent of the voltage on the pin.

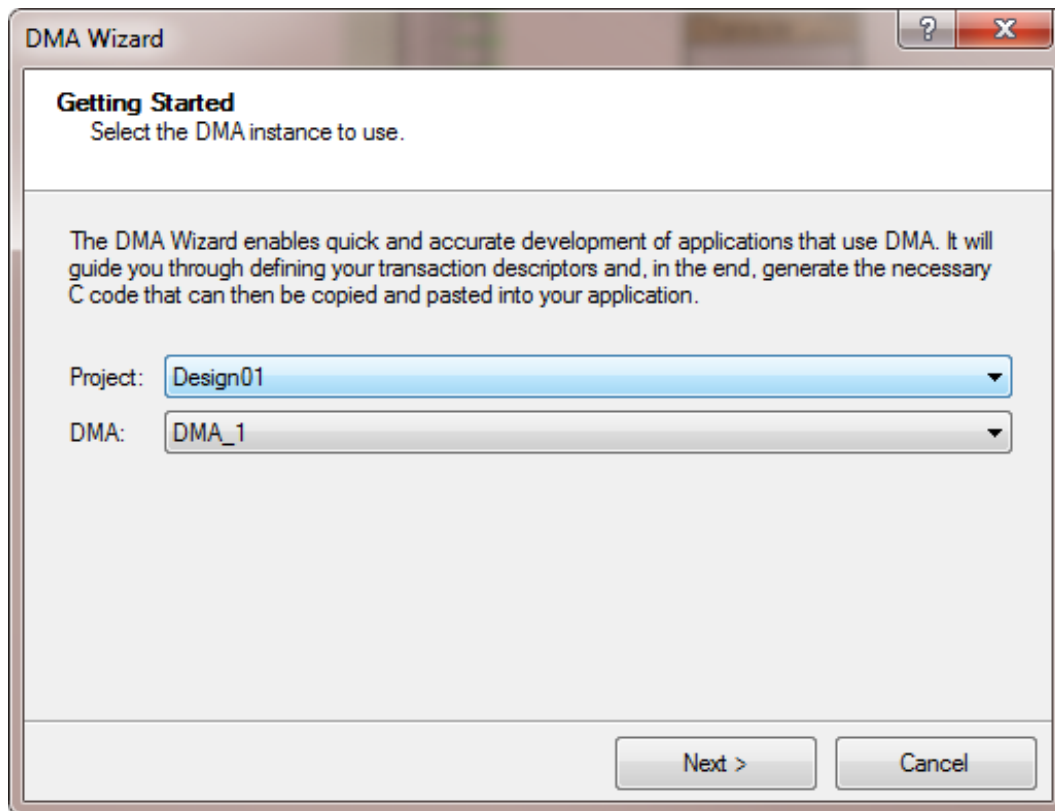
Bits	Name	Description
7:0	PinState[7:0]	Reads of this register return the logical state of the corresponding I/O pin. The data read from this register specifies the logical state of the pin: 1'b1 Reads HIGH if the pin voltage is above the input buffer threshold, logic high. 1'b0 Reads LOW if the pin voltage is below that threshold, logic low. If the drive mode for the pin is set to High-Z Analog, the pin state will read 0 independent of the voltage on the pin.

- You can use either the absolute address (0x5151) or the alias (Input_Signal_PS) in your code to address the port pin state register.

You can find out the alias names for the Component registers from the ".h" generated file of the Component. These files are automatically generated by PSoC Creator when you build the project. In this case, the alias name Input_Signal_PS is defined in the file *Input_Signal.h*.

6. The PICU interrupt must be cleared each time it is triggered. All interrupts from the port are masked until the interrupt is cleared. The PICU interrupt is cleared by reading the interrupt status register of the port.
7. This example uses two DMA channels: one to transfer the GPIO input value to SRAM and the other to clear the PICU interrupt.
8. Open the DMA Wizard from the PSoC Creator menu via **Tools > DMA Wizard**, as [Figure 26](#) shows.

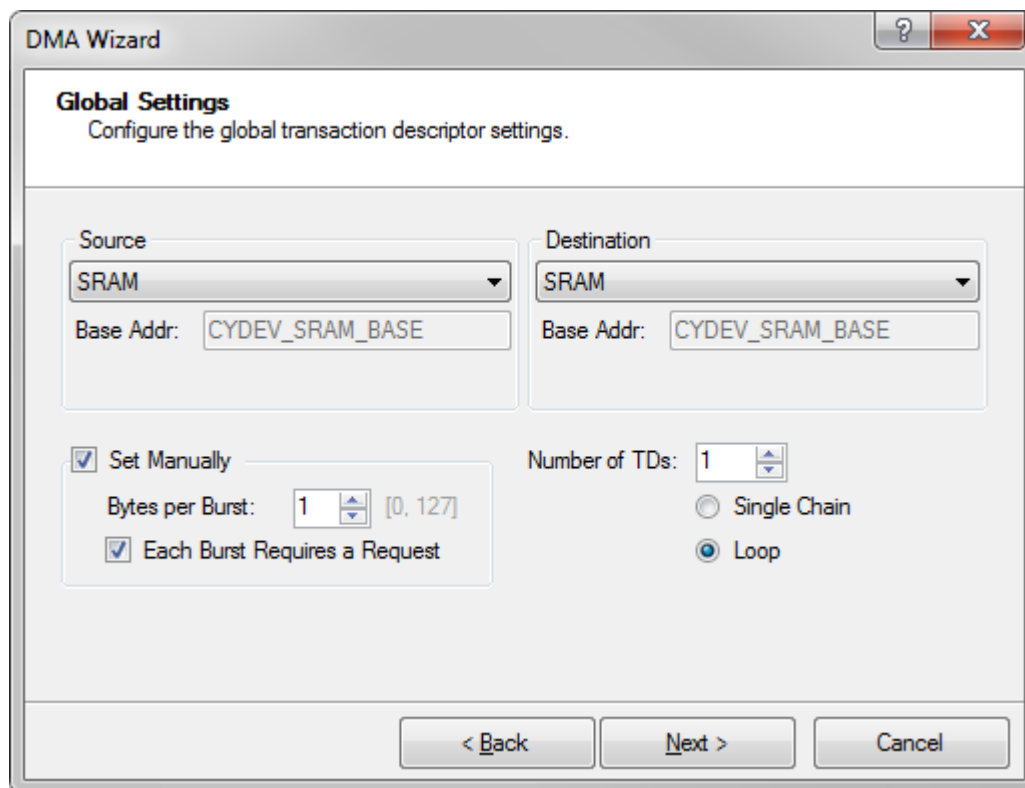
Figure 26. Configure DMA_1



9. Configure the options as [Figure 27](#) shows, and click **Next**.

Leave the source and destination at the default values (SRAM) for now. The DMA Wizard supports only a specific set of Components by default. You must edit the source value in the code later to suit your requirement. The SRAM location should be updated each time the value on the GPIO pins changes. This requires only a single TD that runs when a DMA request is received.

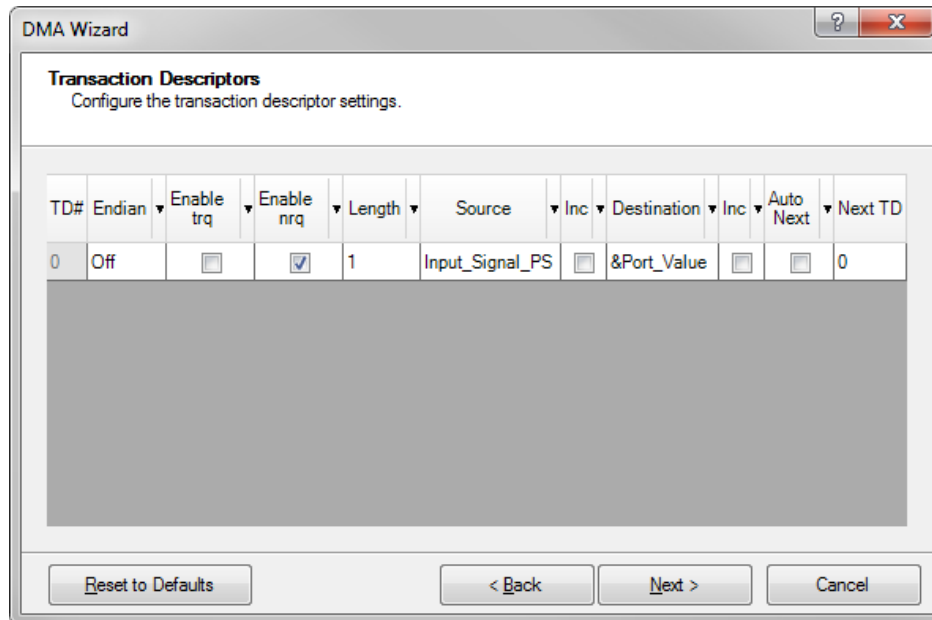
Figure 27. DMA Channel Global Settings



The image shows a screenshot of the 'DMA Wizard' dialog box, specifically the 'Global Settings' tab. The dialog box has a title bar with a question mark and a close button. The main content area is titled 'Global Settings' and contains the instruction 'Configure the global transaction descriptor settings.' Below this, there are two columns of settings. The left column has a 'Source' dropdown menu set to 'SRAM' and a 'Base Addr:' text box containing 'CYDEV_SRAM_BASE'. The right column has a 'Destination' dropdown menu set to 'SRAM' and a 'Base Addr:' text box containing 'CYDEV_SRAM_BASE'. Below these, there are two sections. The left section has a checked checkbox 'Set Manually' and a 'Bytes per Burst:' spinner box set to '1' with a range '[0, 127]' next to it. Below this is another checked checkbox 'Each Burst Requires a Request'. The right section has a 'Number of TDs:' spinner box set to '1' and two radio buttons: 'Single Chain' (unselected) and 'Loop' (selected). At the bottom of the dialog box, there are three buttons: '< Back', 'Next >', and 'Cancel'.

10. Enable the nrq signal. The nrq signal of DMA_1 is used to trigger DMA_2. Set the source address value as described in step 5. The destination is the address of the SRAM location to which the data from the GPIO pins is stored. [Figure 28](#) shows the TD configuration window after setting these values.

Figure 28. Configure the TDs



DMA Wizard

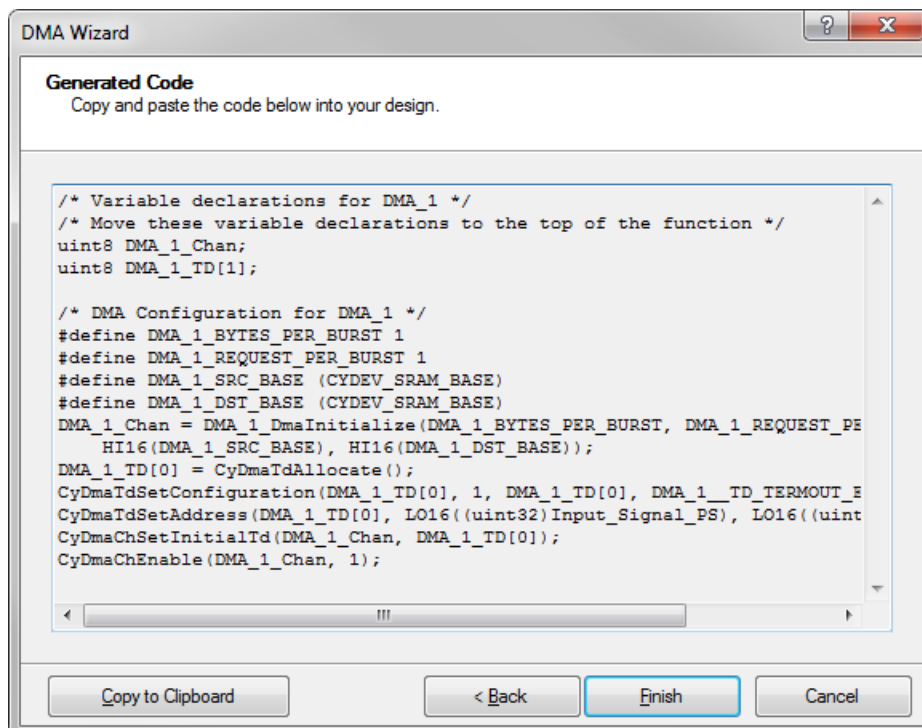
Transaction Descriptors
Configure the transaction descriptor settings.

TD#	Endian	Enable trq	Enable nrq	Length	Source	Inc	Destination	Inc	Auto Next	Next TD
0	Off	<input type="checkbox"/>	<input checked="" type="checkbox"/>	1	Input_Signal_PS	<input type="checkbox"/>	&Port_Value	<input type="checkbox"/>	<input type="checkbox"/>	0

Reset to Defaults < Back Next > Cancel

11. Copy the generated code, as [Figure 29](#) shows, into the clipboard, and paste it into the *main.c* file.

Figure 29. Generated Code After Configurations



DMA Wizard

Generated Code
Copy and paste the code below into your design.

```

/* Variable declarations for DMA_1 */
/* Move these variable declarations to the top of the function */
uint8 DMA_1_Ch;
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_SRAM_BASE)
#define DMA_1_DST_BASE (CYDEV_SRAM_BASE)
DMA_1_Ch = 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], 1, DMA_1_TD[0], DMA_1_TD_TERMOUT_E);
CyDmaTdSetAddress(DMA_1_TD[0], LO16((uint32)Input_Signal_PS), LO16((uint32)&Port_Value));
CyDmaChSetInitialTd(DMA_1_Ch, DMA_1_TD[0]);
CyDmaChEnable(DMA_1_Ch, 1);
  
```

Copy to Clipboard < Back Finish Cancel

12. Configure the second DMA channel similarly, as Figure 30 through Figure 33 show.

Figure 30. Configure DMA_2

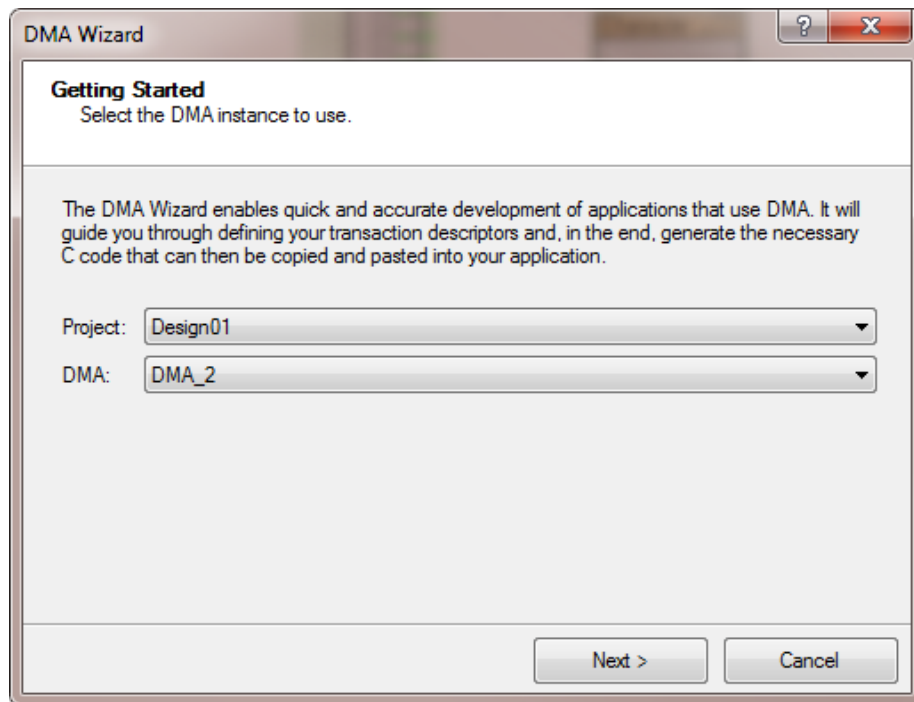


Figure 31. DMA Channel Global Settings

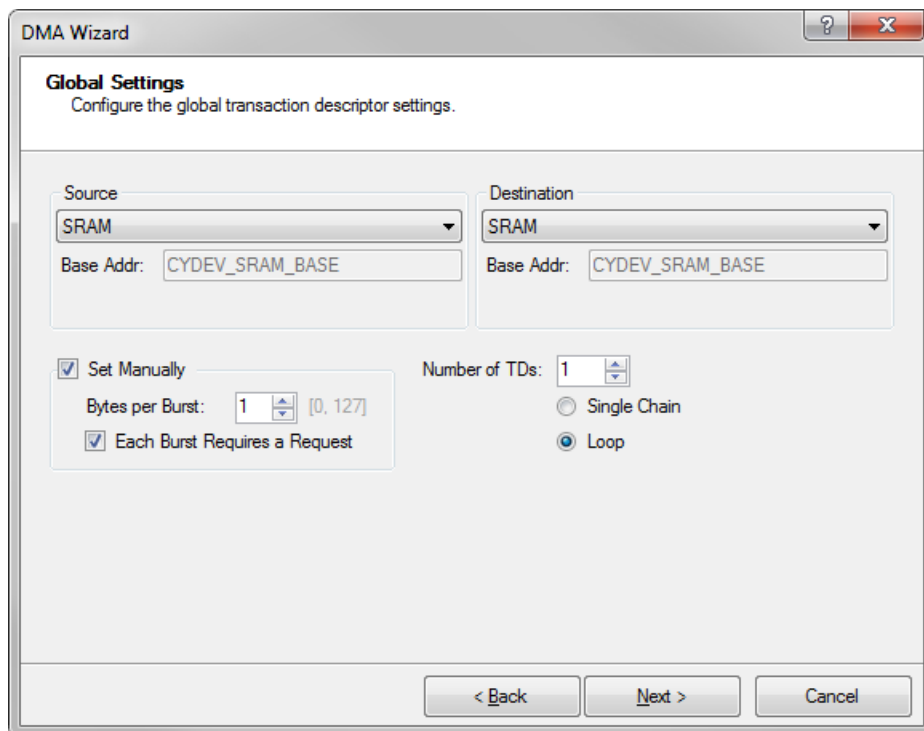
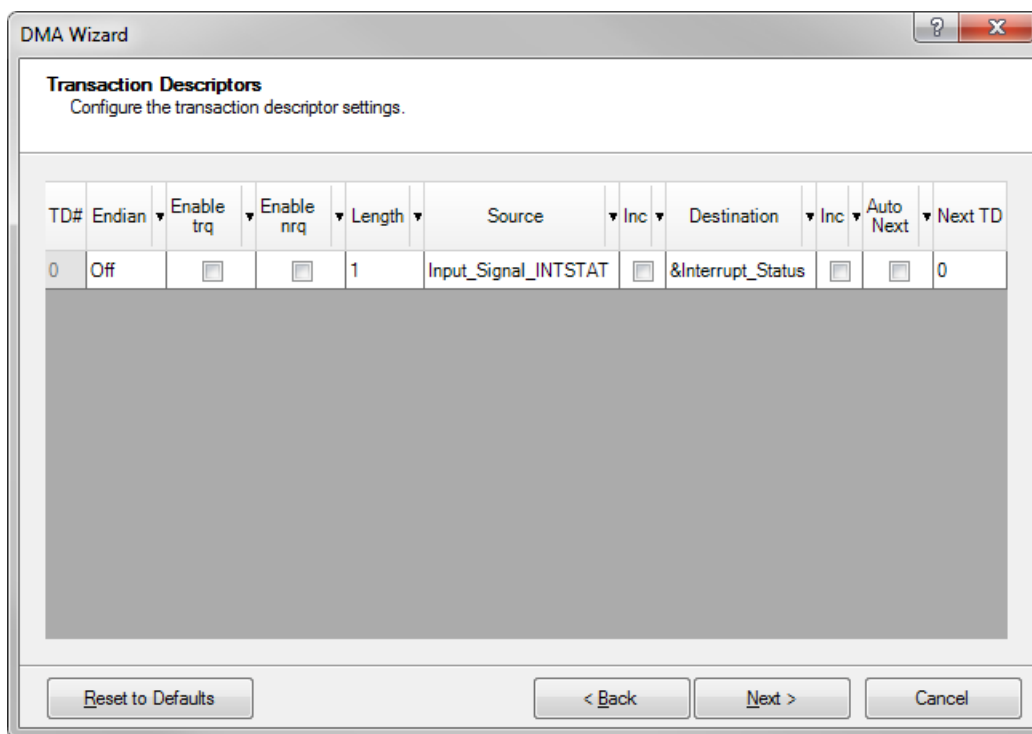


Figure 32. Configure the TD

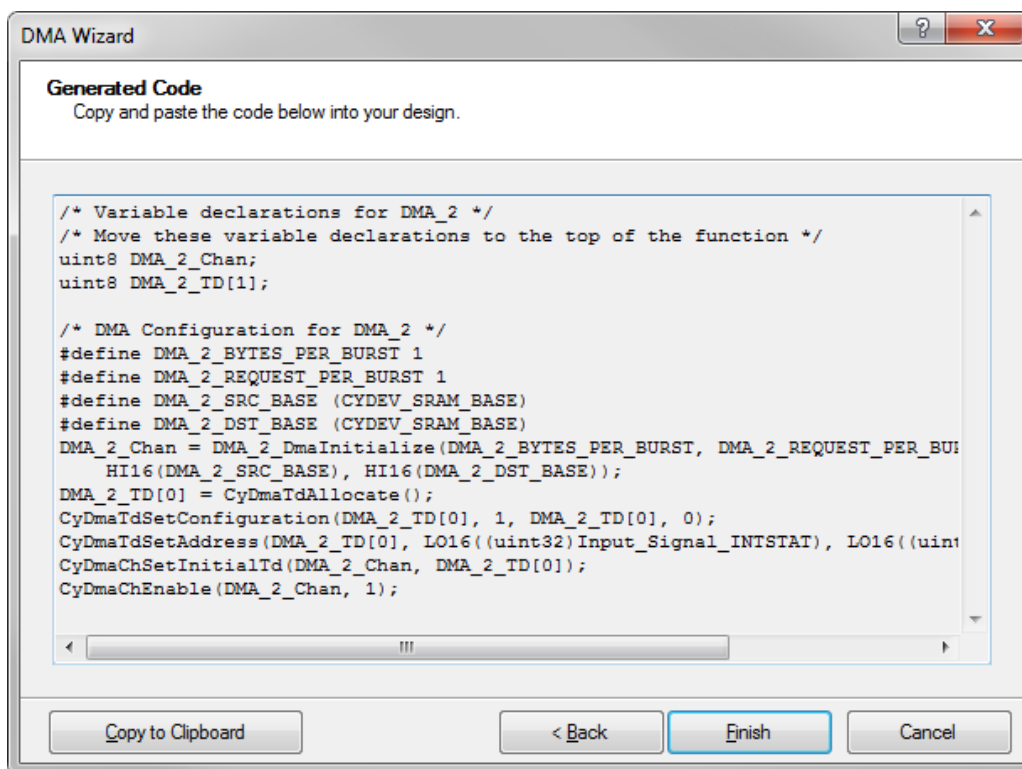


Transaction Descriptors
Configure the transaction descriptor settings.

TD#	Endian	Enable trq	Enable nrq	Length	Source	Inc	Destination	Inc	Auto Next	Next TD
0	Off	<input type="checkbox"/>	<input type="checkbox"/>	1	Input_Signal_INTSTAT	<input type="checkbox"/>	&Interrupt_Status	<input type="checkbox"/>	<input type="checkbox"/>	0

Reset to Defaults < Back Next > Cancel

Figure 33. Generated Code After Configurations



Generated Code
Copy and paste the code below into your design.

```

/* Variable declarations for DMA_2 */
/* Move these variable declarations to the top of the function */
uint8 DMA_2_Ch;
uint8 DMA_2_TD[1];

/* DMA Configuration for DMA_2 */
#define DMA_2_BYTES_PER_BURST 1
#define DMA_2_REQUEST_PER_BURST 1
#define DMA_2_SRC_BASE (CYDEV_SRAM_BASE)
#define DMA_2_DST_BASE (CYDEV_SRAM_BASE)
DMA_2_Ch = DMA_2_DmaInitialize(DMA_2_BYTES_PER_BURST, DMA_2_REQUEST_PER_BURST,
    HI16(DMA_2_SRC_BASE), HI16(DMA_2_DST_BASE));
DMA_2_TD[0] = CyDmaTdAllocate();
CyDmaTdSetConfiguration(DMA_2_TD[0], 1, DMA_2_TD[0], 0);
CyDmaTdSetAddress(DMA_2_TD[0], LO16((uint32)Input_Signal_INTSTAT), LO16((uint32)&Interrupt_Status));
CyDmaChSetInitialTd(DMA_2_Ch, DMA_2_TD[0]);
CyDmaChEnable(DMA_2_Ch, 1);
    
```

Copy to Clipboard < Back Finish Cancel

13. Edit *main.c*. In the `#define` statements, change `DMA_1_SRC_BASE` and `DMA_2_SRC_BASE` to `CYDEV_PERIPH_BASE`, since the GPIO is a peripheral. Add code to start the Character LCD Component. The resulting code is similar to [Code 1](#).
14. You can verify the functionality of the project by changing the data input to the GPIO pins. The character LCD displays the decimal value of the data input applied to the GPIO pins.

This procedure is similar for any other standard Component, or for PSoC registers. Be sure to edit the generated code appropriately so that the base address for the source and the destination is correct. The base addresses for different sources and destinations are available in the PSoC Creator generated file *cydevice_trm.h* after building a project. [Table 3](#) shows a list of commonly used base addresses.

Table 3. Commonly Used Base Addresses

Source/Destination	PSoC 3	PSoC 5LP
Flash	CYDEV_FLS_BASE	CYDEV_FLASH_BASE
SRAM	CYDEV_SRAM_BASE	CYDEV_SRAM_DATA_MBASE
Peripherals	CYDEV_PERIPH_BASE	CYDEV_PERIPH_BASE

Code 1. Completed Code for Transferring Data Bytes from GPIO Pins to SRAM

```
#include <device.h>

/* 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_PERIPH_BASE)
#define DMA_1_DST_BASE (CYDEV_SRAM_BASE)
/* DMA Configuration for DMA_2 */
#define DMA_2_BYTES_PER_BURST 1
#define DMA_2_REQUEST_PER_BURST 1
#define DMA_2_SRC_BASE (CYDEV_PERIPH_BASE)
#define DMA_2_DST_BASE (CYDEV_SRAM_BASE)

uint8 Port_Value;
uint8 Interrupt_Status;

void main()
{
    uint8 DMA_1_Chann;
    uint8 DMA_1_TD[1];
    uint8 DMA_2_Chann;
    uint8 DMA_2_TD[1];

    DMA_1_Chann = 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], 1, DMA_1_TD[0], DMA_1_TD_TERMOUT_EN);
    CyDmaTdSetAddress(DMA_1_TD[0], LO16((uint32)&Input_Signal_PS), LO16((uint32)&Value));
    CyDmaChSetInitialTd(DMA_1_Chann, DMA_1_TD[0]);
    CyDmaChEnable(DMA_1_Chann, 1);

    DMA_2_Chann = DMA_2_DmaInitialize(DMA_2_BYTES_PER_BURST, DMA_2_REQUEST_PER_BURST,
                                      HI16(DMA_2_SRC_BASE), HI16(DMA_2_DST_BASE));
    DMA_2_TD[0] = CyDmaTdAllocate();
    CyDmaTdSetConfiguration(DMA_2_TD[0], 1, DMA_2_TD[0], 0);
    CyDmaTdSetAddress(DMA_2_TD[0], LO16((uint32)&Input_Signal_INTSTAT),
                     LO16((uint32)&Interrupt_Status)); /* Read the interrupt status register
                                                         * to clear the PICU interrupt */
    CyDmaChSetInitialTd(DMA_2_Chann, DMA_2_TD[0]);
    CyDmaChEnable(DMA_2_Chann, 1);

    CyGlobalIntEnable;
    lcd_Start();

    for(;;)
    {
        lcd_Position(0,0);
        lcd_PrintNumber(Port_Value);
        CyDelay(20);
        lcd_ClearDisplay();
    }
}
```

Modifying a TD Dynamically

The PSoC DMAC can access its own registers. This allows a TD to dynamically alter the properties of another TD. Two examples of DMA transfers that use this technique are indexed DMA and nested DMA.

Indexed DMA

With indexed DMA, you can access PSoC memory and register addresses, for example, from an external device using a communication port. You use two TDs to do so. The first TD acts as an address fetch—it writes the destination address of the second TD. Then the second TD is called to carry out the actual data transfer. The external device can then access the PSoC memory locations as if those locations were shared memory. For example, if the actual memory space of the external device is only 256 bytes (0x00 to 0xFF), it can be extended to 512 bytes by using the PSoC memory locations with indexed DMA.

You can configure a DMA channel in the PSoC device for this purpose, as Figure 34 shows. If the external device wants to write to the PSoC memory, it should pass the address of the location followed by the data to be written to the PSoC memory. An interrupt associated with the peripheral through which the external master is connected to the PSoC device can be used to generate a DMA channel request.

Figure 35 shows the timing diagram for an indexed data transfer in this case.

Figure 34. Block Diagram for Indexed DMA Transfer

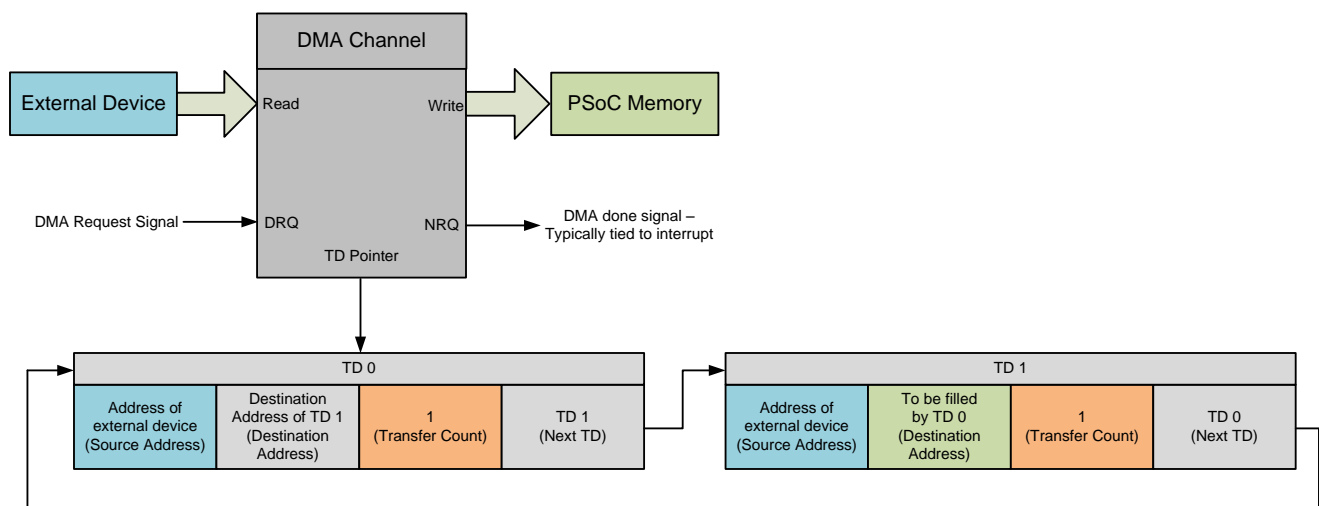
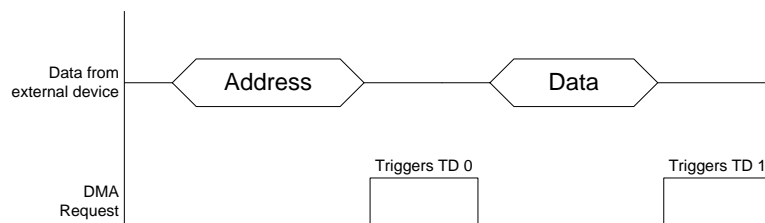


Figure 35. Timing Diagram for an Indexed DMA Transfer

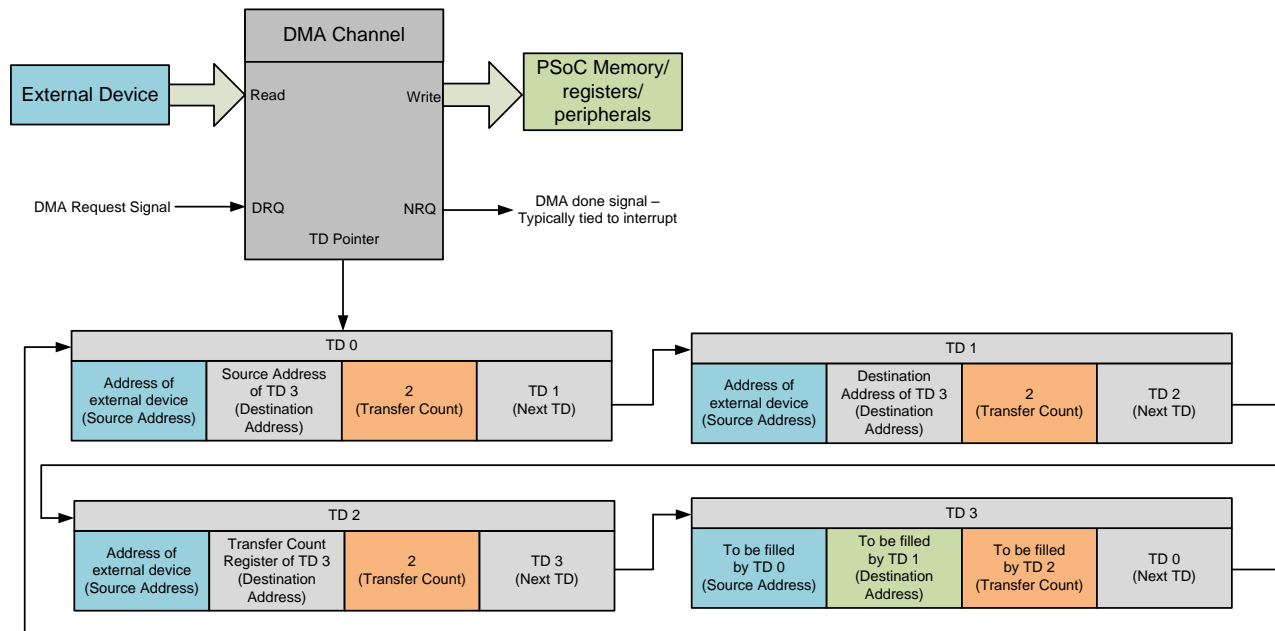


Nested DMA

Nested DMA is similar to indexed DMA, except that it involves multiple TDs to configure any parameters of the TD that does the actual data transfer. These parameters include source address, destination address, and transfer count.

Figure 36 shows the block diagram for a nested DMA data transfer.

Figure 36. Block Diagram for a Nested DMA Transfer



Note If the upper 16 bits of source or destination address are different for TD 3, different DMA channels should be used for updating the TD configuration and actual data transfer.

An example project provided with this application note demonstrates the nested DMA feature.

Debugging DMA

This section explains some common issues that you may encounter while working with DMA, and some debugging techniques.

Common Issues

Base Address Differences

The 8-bit PSoC 3, the 32-bit PSoC 5LP, and the DMAC that exists in both families all have different addressing schemes. It is important to be aware of these differences, especially when porting DMA designs between the PSoC families.

To start, note that the DMAC uses 32-bit addresses (source and destination), stored in separate 16-bit registers. The upper halves of the addresses are specified in a DMA channel:

```
DMA_DmaInitialize(..., upperSrcAddr, upperDestAddr)
```

And the lower halves of the addresses are specified in the TDs allocated to a DMA channel:

```
CyDmaTdSetAddress(..., lowerSrcAddr, lowerDestAddr)
```

For PSoC 5LP, you can easily obtain the upper and lower halves of an address from a (4-byte) pointer variable, by using the HI16 or LO16 macros, defined in the *cytypes.h* file:

```
upperSrcAddr = HI16(srcArray);  
lowerSrcAddr = LO16(srcArray);
```

For PSoC 3, the contents of a pointer variable cannot be used because the Keil 8051 compiler uses a 3-byte pointer. It contains a 16-bit absolute address and a third byte for the memory space used (see [Appendix A](#)). Instead, use one of the following code snippets to obtain the upper half of the address:

Source or destination in SRAM or peripheral register:

```
upperSrcAddr = 0;
```

Source in flash:

```
upperSrcAddr = (CYDEV_FLS_BASE) >> 16
```

A list of commonly used base addresses is provided in [Table 3](#) on page 22.

Conditional compilation can be used:

```
#if (defined(__C51__))  
    upperSrcAddr = 0;  
    lowerSrcAddr = srcArray;  
#else /* PSoC 5 */  
    upperSrcAddr = HI16(srcArray);  
    lowerSrcAddr = LO16(srcArray);  
#endif
```

Endian Formats

Endian format refers to how multi-byte variables are stored in a byte-wide memory. In big endian format, the most significant byte is stored in the first byte (lowest address). In little endian format, the least significant byte is stored in the lowest address.

For PSoC 3, the Keil 8051 compiler uses big endian format. The PSoC 5LP Cortex-M3 and all its compilers use little endian format. Both PSoC 3 and PSoC 5LP multi-byte peripheral registers use little endian format.

DMA TDs can be programmed to have bytes swapped while transferring data. The swap size is set to 2 bytes for 16-bit transfers or 4 bytes for 32-bit transfers. The following examples handle 2- and 4-byte swaps:

```
CyDmaTdSetConfiguration(myTd, 2, myTd, (TD_TERMOUT0_EN | TD_SWAP_EN));  
CyDmaTdSetConfiguration(myTd, 4, myTd, (TD_TERMOUT0_EN | TD_SWAP_EN | TD_SWAP_SIZE4));
```

DMA byte swapping is required only in PSoC 3, when transferring multi-byte parameters between peripherals and variables in memory. It must be disabled for all other uses, including all PSoC 5LP uses.

Note If the transfer count is not an integer multiple of the swap size, incorrect data is transferred to the destination.

Preserving TD Configuration

You can set the preserve the TD configuration using the API function `CyDmaChEnable()`. If the TDs are preserved, the TD whose number is the same as the channel number becomes RESERVED and is used as the working register for the channel. For example, if you are using DMA channel 6 and the TD is preserved, you are not allowed to use TD 6. This TD is now used by the DMA engine for its private use.

If a TD is preserved, the TD is copied into the working register of the channel each time it is executed. The TD updates the source address, destination address, and transfer count only in the working register during the data transfer. If the channel is triggered again, the TD reloads its values to the working register from the original configuration registers.

If the TD is not preserved, the original TD settings are changed during the transfer. The transfer count value is decremented and finally reaches zero when the TD has been completed. An infinite data transfer begins if the TD is triggered again, since the transfer count is set to zero. If the source or destination address for the TD is configured for incrementing during the transfer, it results in erroneous data transfers.

Note Take extra precautions when using the hardware request (drq) option when not preserving TD, as you may be requesting the wrong data.

Address Alignment

If some data bytes are not transferred, a likely reason may be misaligned source and data addresses. Make sure that the source and destination addresses are aligned with spoke width boundaries. Doing so also increases transfer efficiency. A detailed discussion of the methods to align source and destination addresses with even address boundaries appears in the section [Multi-Byte Data Alignment](#).

Debugging Methods

Some methods to debug a DMA transfer are the following:

1. Observe the drq and nrq signals on a scope: If you are using a hardware signal to trigger the DMA Component, observe the drq terminal to make sure that the DMA Component is being triggered correctly. If the termout signal is enabled for the DMA Component, at the completion of a data transfer the DMA Component produces a pulse at the nrq terminal that is two bus clocks wide.
2. View critical DMA registers: The source address, destination address, and DMA configuration registers can be used to debug DMA transfers. Enter debug mode, navigate to the corresponding register location, and make sure that all registers have the expected values. The DMA registers inform the current status of DMA transfer, such as the number of bytes transferred, the TD that is being executed currently, and the status of DMA. The details of these registers are available in the PSoC 3 and PSoC 5LP Registers TRMs.
3. Use the correct method to trigger the DMA: Make sure that the correct trigger option is set for the drq terminal, that is, level or edge. Setting an incorrect trigger method can result in situations in which DMA requests are missed or are generated unnecessarily.

The choice of trigger method depends on the source that generates the trigger signal. The trigger option is generally set to level when the DMA is used to transfer data from a FIFO. For example, the level option is used if you are transferring data from a UART Component to SRAM. This ensures that the DMA begins a transfer as soon as the UART buffer is not empty. The trigger option is set to edge if the data transfer follows the occurrence of an event such as ADC conversion complete.

4. Check the allocation and deallocation of TDs. The TDs of a DMA channel define the behavior of the DMA channel. A TD specifies the source address, destination address, number of bytes to be transferred (transfer count), and the next TD to be executed. Multiple TDs can be allocated to a channel. See the [DMA Component datasheet](#) for details.

Always disable a DMA channel before allocating or deallocating a TD for that channel. Otherwise the channel may behave indeterminately and write to random memory or peripheral addresses, which can be very difficult to debug. The following paragraphs explain the reason that this happens.

Two parameters are used to keep track of the allocation and deallocation of the TDs. One is `CyDmaTdFreeIndex`, which points to the next unallocated TD. The other one is `CyDmaTdCurrentNumber`, which stores the number of free TDs currently available. Byte 0 of the register (`PHUB_TDMEM[xxx]_ORIG_TD0`), pointed by `CyDmaTdFreeIndex` has the next free TD number available after the current free TD is allocated. This acts like a linked list to keep track of the available TDs.

If a TD is being used, `PHUB_TDMEM[xxx]_ORIG_TD0` holds the eight lower bits of the transfer count (`XFRCNT`). This is normally loaded during TD configuration with the total number of bytes to be transferred by that TD. It is updated by the TD after every burst transfer with a new transfer count value (remaining bytes to be transferred). When a new TD is allocated using the API `CyDmaTdAllocate()`, `CyDmaTdFreeIndex` is updated with the value in `PHUB_TDMEM[CyDmaTdFreeIndex]_ORIG_TD0`.

Now consider that the TDs are allocated using the API `CyDmaTdAllocate()` and freed using `CyDmaTdFree()`. When TDs are released with `CyDmaTdFree()`, the general DMA configuration itself is not changed, but instead `CyDmaTdFreeIndex` is updated with the value of the released TD number. If the TD number released using `CyDmaTdFree()` is `Freed_TD_Number`, the `PHUB_TDMEM[Freed_TD_Number]_ORIG_TD0` configuration register is updated with the previous value of `CyDmaTdFreeIndex`.

If a DMA channel request occurs after the TD is freed but with the DMA channel enabled, the `TDMEM.TD0` value that points to the next free TD is corrupted since that value is taken as the transfer count. Now if another TD is allocated, the newly allocated TD may be a TD that is already in use. This TD could now transfer data incorrectly, causing system-wide issues.

Note The DMAC latches a channel request even if the channel is disabled. This latched request is executed as soon as the channel is enabled again. This may cause the DMA Component to transfer incorrect data when the device wakes from sleep or the channel is re-enabled if a channel request occurs when the channel is inactive.

Projects

Two projects are provided with this application note to help you understand the concepts introduced in this application note.

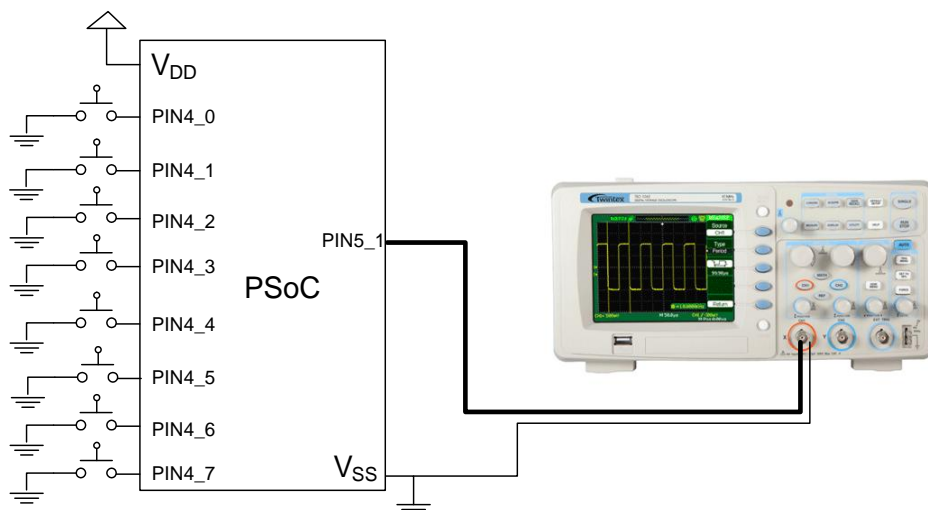
Parallel to Serial Converter Project

This project uses a DMA channel to convert a parallel stream of 8-bit data from a GPIO port to serial data. The data is shifted out at a constant rate through another GPIO pin.

This project shows you how to use DMA to transfer data between standard components that are not supported by the DMA Wizard. This project also demonstrates the method to force UDB Components to required locations.

Figure 37 shows the test setup for the parallel to serial converter project.

Figure 37. Project Setup for Parallel to Serial Converter Project



Nested DMA Project

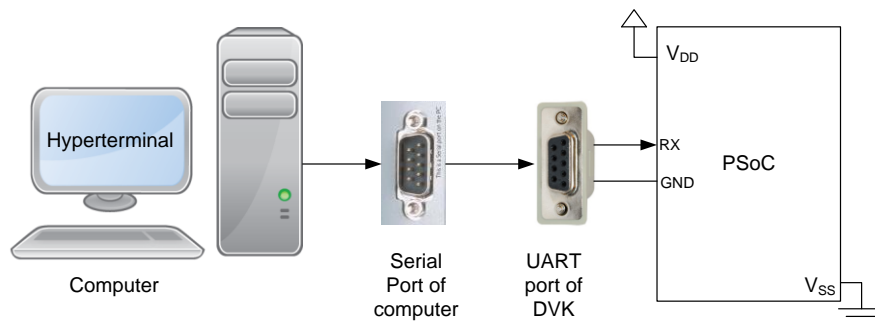
This project demonstrates the method to dynamically modify a TD using another TD.

The project allows an external master with UART transmit capability to transfer variable amounts of data to different locations in the PSoC SRAM. The number of data bytes and the address to which these data bytes are to be written can be changed dynamically.

The first byte of the data stream received by the PSoC device specifies the SRAM address, and the second byte specifies the number of bytes to follow. The remaining bytes in the data stream are written to the SRAM.

Figure 38 shows the test setup for the nested DMA project.

Figure 38. Project Setup for Nested DMA Project



Summary

The PSoC 3 and PSoC 5LP DMA controller allows you to significantly offload the CPU and to transfer data between memory and peripherals. The DMA is powerful enough to modify itself dynamically, which makes this Component unique.

This application note explained some advanced concepts regarding the PSoC DMA and offered projects as examples.

About the Author

Name: Ranjith M
Title: Applications Engineer
Background: Ranjith graduated from Government Engineering College, Thrissur with a Bachelor's Degree in Electronics and Communications Engineering.
Contact: rjmt@cypress.com

Appendix A: Memory Maps

The PSoC 3 and PSoC 5LP memory maps are different due to the architecture of their respective CPUs.

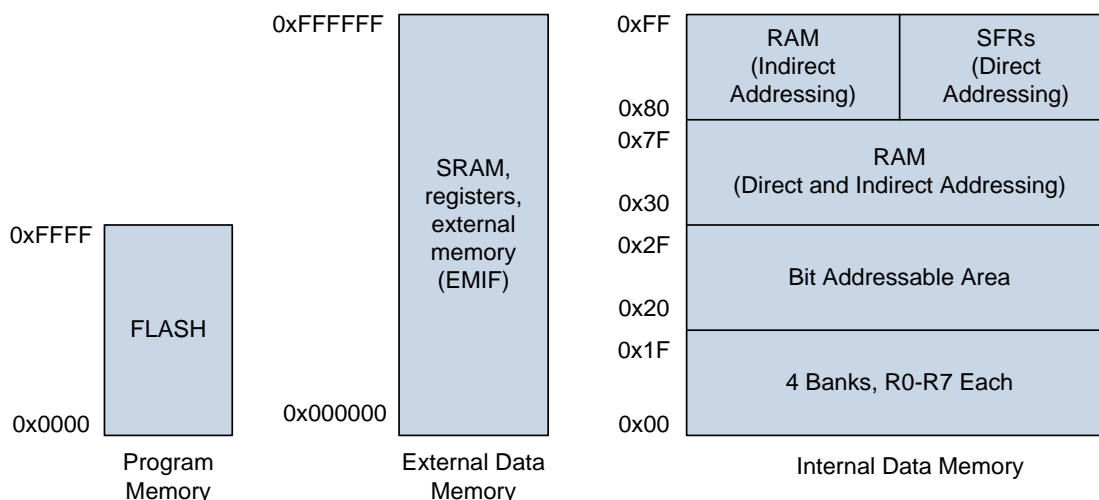
The PSoC 3 8051 memory map consists of three distinct memory spaces, as [Figure 39](#) shows.

- Program memory: This space is 64 KB and is occupied solely by flash memory. The 8051 executes instructions from this space.
- External data memory: This space is “external” to the 8051 core but is “internal” to the PSoC 3 device. All SRAM, registers and EMIF addresses are mapped into this space. Flash memory is also mapped into this space, mainly for DMA data access. This space is 16 MB and requires a 24-bit address to access it.
- 8051 internal data memory: This space is part of the 8051 core. It contains 256 bytes of RAM and several special function registers (SFRs). The 8051 uses fast register and bit instructions to access portions of this space. The 8051 hardware stack also occupies this space; the stack size can be at most 256 bytes.

Note This address space is not included in the PSoC 3 memory map and is not available for DMA access.

For more details, see any PSoC 3 datasheet or the application note [AN60630 – PSoC 3 8051 Code and Memory Optimization](#).

Figure 39. PSoC 3 8051 Memory Map



The PSoC 5LP ARM Cortex-M3 architecture is simpler. It uses a single 32-bit linear memory map, as [Figure 40](#) shows. The SRAM in the PSoC 5LP occupies the addresses 0x1FFF8000 to 0x20007FFF, centered on the boundary between the Cortex-M3 code and SRAM spaces. The remainder of the code space is occupied solely by flash, starting at address 0. The PSoC 5LP registers occupy the Cortex-M3 peripheral space.

Figure 40. PSoC 5LP Cortex-M3 Memory Map

Vendor-specific	0xFFFFFFF
	0xE0100000
Private peripheral bus - External	0xE00FFFF
	0xE0040000
Private peripheral bus - Internal	0xE003FFFF
	0xE0000000
	0xDFFFFFFF
External device 1.0GB	
	0xA0000000
	0x9FFFFFFF
External RAM 1.0GB	
	0x60000000
	0x5FFFFFFF
Peripheral 0.5GB	
	0x40000000
	0x30000000
SRAM 0.5GB	
	0x20000000
	0x1FFFFFFF
Code 0.5GB	
	0x00000000

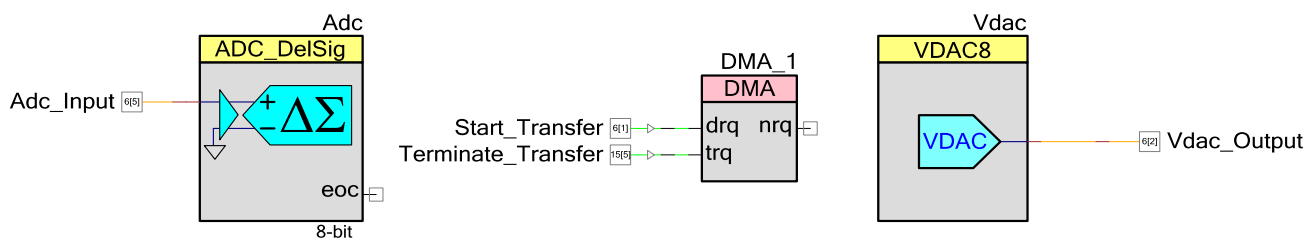
Appendix B: Termination Request Signal

To see how the trq input of the DMA operates, set the transfer count to zero to start an infinite data transfer and then trigger the DMA channel. A positive edge on the trq terminal stops the data transfer by a non-count termination.

The following steps explain a scenario where the trq terminal is used for a non-count termination:

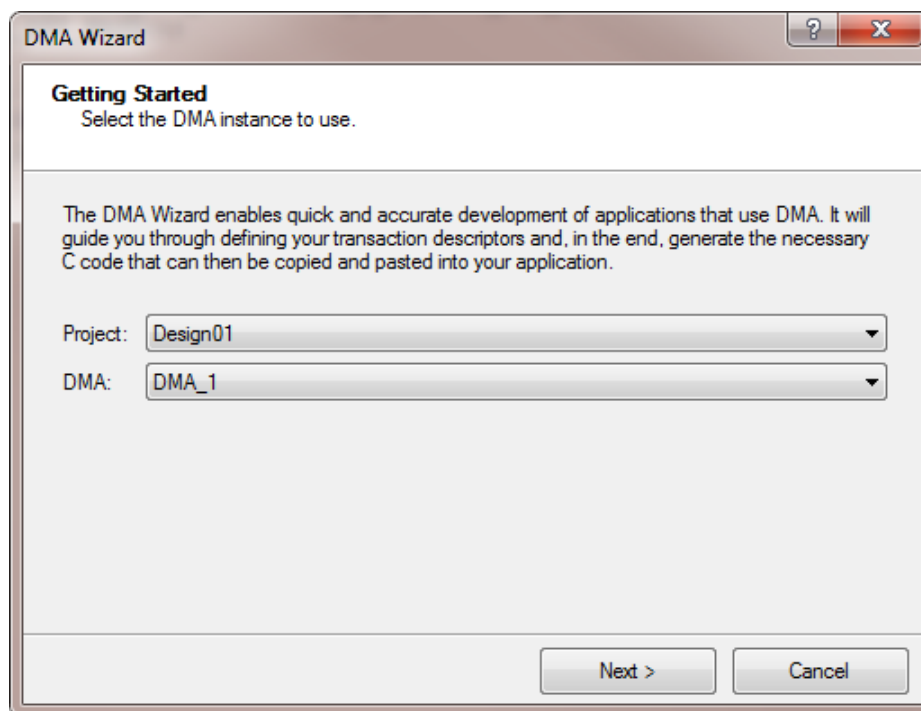
1. A DMA channel is used to transfer an 8-bit data output from the ADC to a VDAC. Figure 41 shows the TopDesign schematic.

Figure 41. Completed TopDesign for the trq Demonstration Example



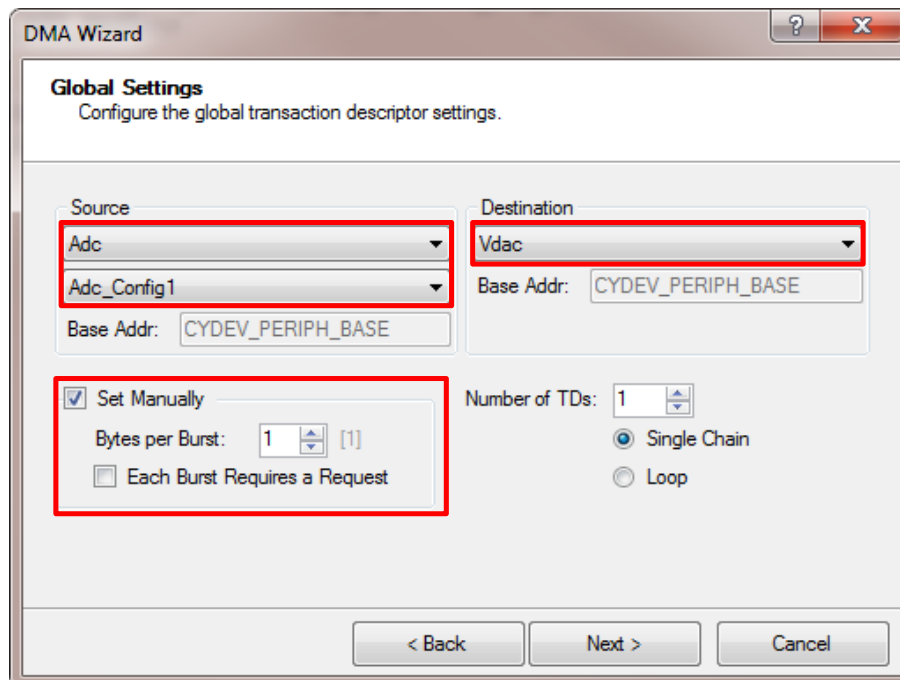
2. Connect a variable voltage source to the Adc_Input and an oscilloscope or an LED to the Vdac_Output. Connect two switches to the pins Start_Transfer and Terminate_Transfer.
3. Open the PSoC Creator DMA Wizard from the menu **Tools > DMA Wizard**, as Figure 42 shows. Select the DMA Component from your design and click **Next**.

Figure 42. Selecting the DMA Channel



- In the next window, set the configurations as [Figure 43](#) shows, and click **Next**.

Figure 43. Configure General DMA Channel Settings



DMA Wizard

Global Settings
Configure the global transaction descriptor settings.

Source: **Adc** (dropdown), **Adc_Config1** (dropdown), Base Addr: CYDEV_PERIPH_BASE

Destination: **Vdac** (dropdown), Base Addr: CYDEV_PERIPH_BASE

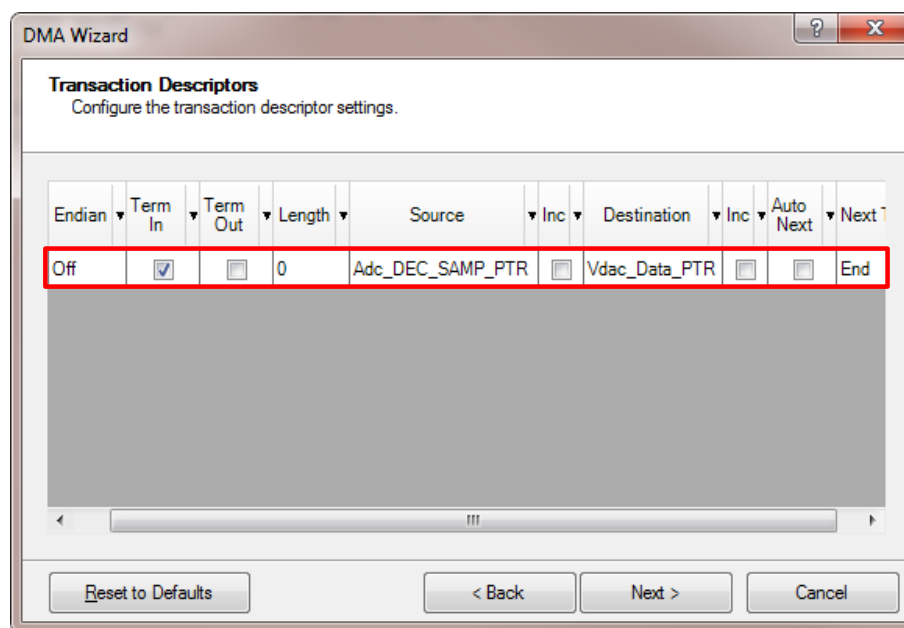
☒ Set Manually
 Bytes per Burst: 1 [1]
☐ Each Burst Requires a Request

Number of TDs: 1
☒ Single Chain
☐ Loop

< Back Next > Cancel

- In the next window, select the option “Term In” and set the “Length” field (the transfer length) to zero, as [Figure 44](#) shows. This initiates an infinite transfer. Click **Next**.

Figure 44. Configure the TDs



DMA Wizard

Transaction Descriptors
Configure the transaction descriptor settings.

Endian	Term In	Term Out	Length	Source	Inc	Destination	Inc	Auto Next	Next
Off	<input checked="" type="checkbox"/>	<input type="checkbox"/>	0	Adc_DEC_SAMP_PTR	<input type="checkbox"/>	Vdac_Data_PTR	<input type="checkbox"/>	<input type="checkbox"/>	End

Reset to Defaults < Back Next > Cancel

6. In the final window, click **Copy to Clipboard** to copy the resultant code. Paste this code into your project *main.c* file.
7. Add code to *main.c* to start the ADC and VDAC Components, and start ADC conversion, as [Code 2](#) shows.

Program the PSoC device and run the code. Press the Start_Transfer switch to transfer ADC output to the VDAC. Similarly, press the Terminate_Transfer switch to terminate data transfer to the VDAC.

Code 2. Code in the File *main.c*

```
#include <device.h>

/* DMA Configuration for DMA_1 */
#define DMA_1_BYTES_PER_BURST 1
#define DMA_1_REQUEST_PER_BURST 0
#define DMA_1_SRC_BASE (CYDEV_PERIPH_BASE)
#define DMA_1_DST_BASE (CYDEV_PERIPH_BASE)

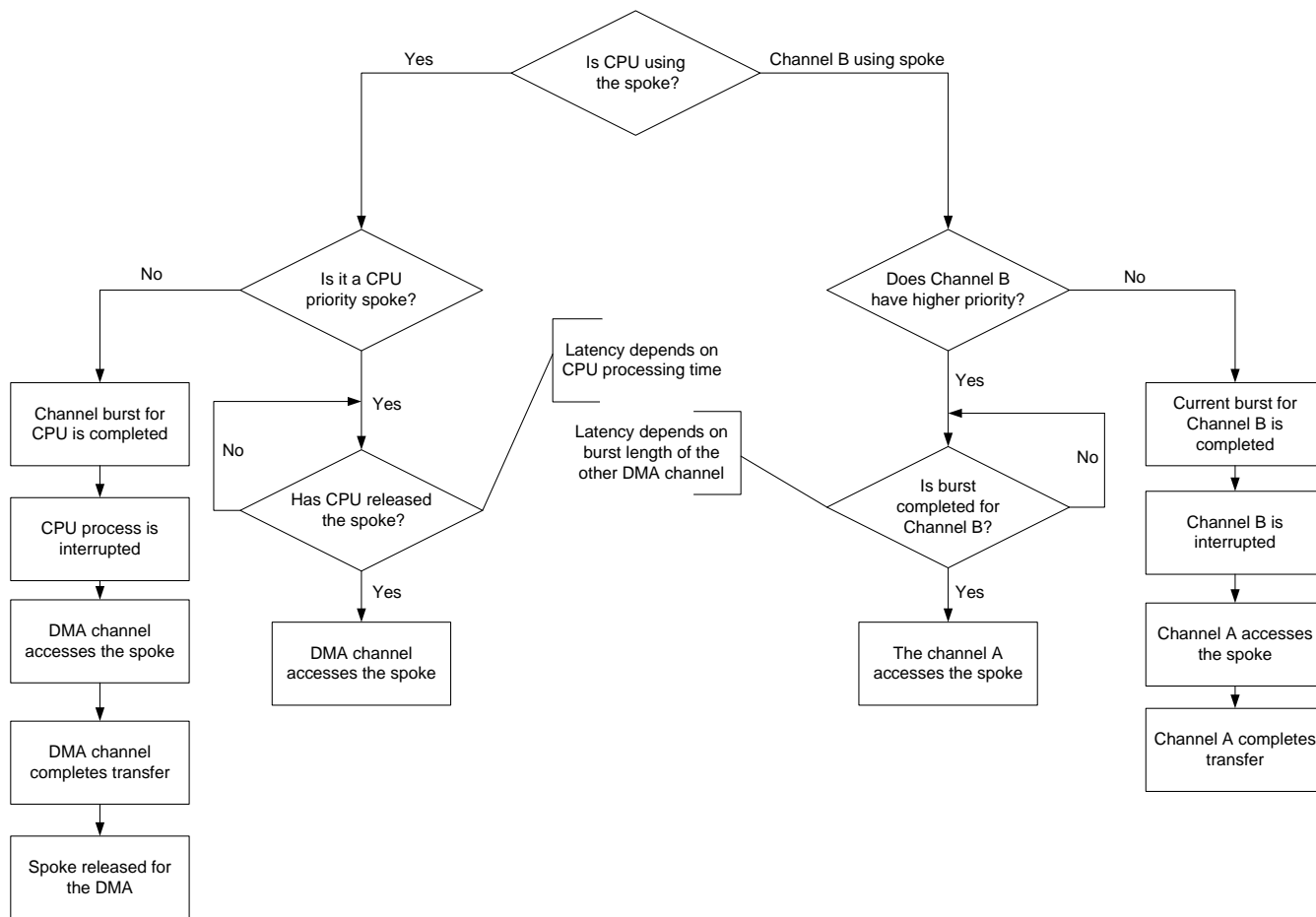
void main()
{
    uint8 DMA_1_Chان;
    uint8 DMA_1_TD[1];

    DMA_1_Chان = 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], 0,
        DMA_INVALID_TD, TD_TERMIN_EN);
    CyDmaTdSetAddress(DMA_1_TD[0],
        LO16((uint32)Adc_DEC_SAMP_PTR),
        LO16((uint32)Vdac_Data_PTR));
    CyDmaChSetInitialTd(DMA_1_Chان,
        DMA_1_TD[0]);
    CyDmaChEnable(DMA_1_Chان, 1);

    Adc_Start();
    Adc_StartConvert();
    Vdac_Start();

    for(;;)
    {
    }
}
```

Appendix C: DMA Channel Arbitration Flow Diagram



Appendix D: Misaligned Data Transfers

The DMAC performs incorrect data transfers if the source or destination addresses are misaligned and the increment source address and increment destination address are not enabled. Figure 45 shows a valid DMA transfer when the source and destination addresses are aligned. Figure 46 through Figure 51 illustrate different scenarios where misaligned data results in incorrect DMA transactions for 16-bit data.

Figure 45. Aligned Source and Destination Addresses – Valid DMA Transfer

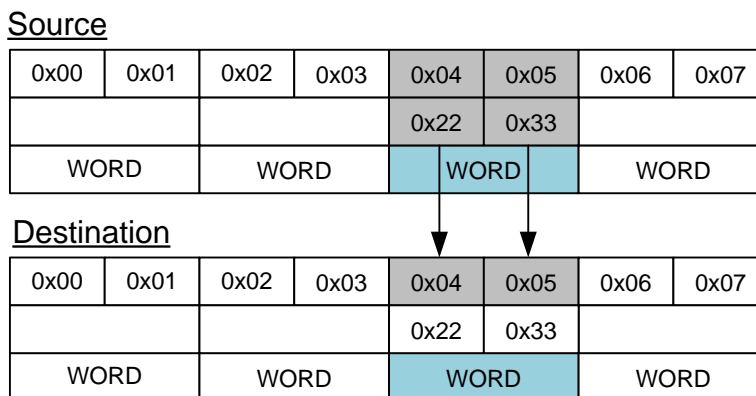


Figure 46. Misaligned Destination Address – Destination Data Overwrite

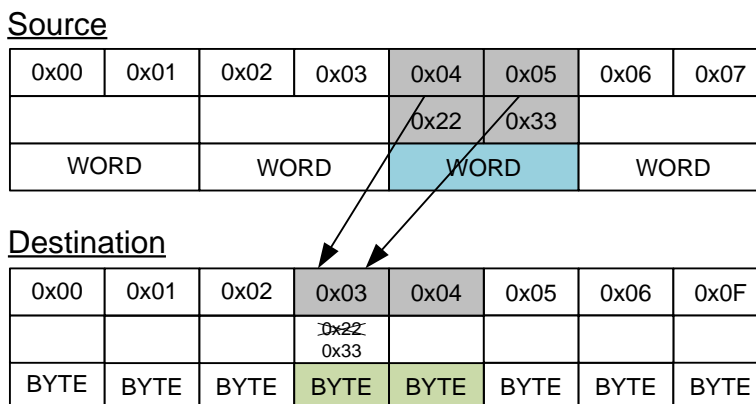


Figure 47. Misaligned Destination Address – Destination Data Overwrite

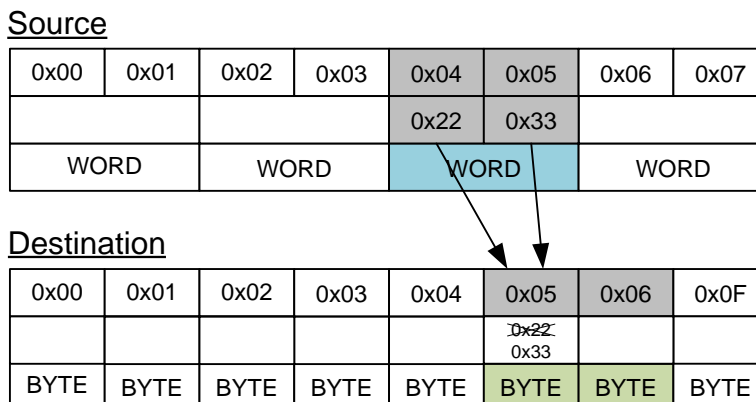


Figure 48. Misaligned Source Address – Destination Data Duplicate

Source

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x0F
					0x22	0x33	
BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

Destination

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
				0x22	0x22		
WORD		WORD		WORD		WORD	

Figure 49. Misaligned Source Address – Destination Data Duplicate

Source

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x0F
			0x22	0x33			
BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

Destination

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
				0x22	0x22		
WORD		WORD		WORD		WORD	

Figure 50. Misaligned Source Address and Destination Address – Single-Byte Transfer

Source

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x0F
			0x22	0x33			
BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

Destination

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x0F
			0x22				
BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

Figure 51. Misaligned Source Address and Destination Address – Single-Byte Transfer

Source

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x0F
					0x22	0x33	
BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

Destination

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x0F
			0x22		0x22		
BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE	BYTE

Document History

Document Title: PSoC® 3 and PSoC 5LP Advanced DMA Topics – AN84810

Document Number: 001-84810

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4015646	RNJT	05/30/2013	New Application Note.
*A	4602407	RNJT	12/19/2014	Added Table 1 with the number of clock cycles required for DMA transfer between PSoC resources. Updated the section Debugging DMA.

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
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 4](#) | [PSoC 5LP](#)

Cypress Developer Community

[Community](#) | [Forums](#) | [Blogs](#) | [Video](#) | [Training](#)

Technical Support

cypress.com/go/support

PSoC is registered trademark and PSoC Creator is a 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, 2013-2014. 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.