How to use direct memory access (DMA) controller in TRAVEO™ II family

About this document

Scope and purpose

This application note describes how to use DMA controllers (P-DMA and M-DMA) in TRAVEO™ II family MCUs. DMA controllers can transfer data from a source to a destination without CPU intervention. The application note illustrates how to configure DMA for peripheral-to-memory, memory-to-peripheral, and memory-to-memory data transfers.

Associated part family

TRAVEO™ II family CYT2/CYT3/CYT4 series.

Intended audience

This document is intended for anyone using TRAVEO™ II family CYT2/3/4 series.

Table of contents

1 Introduction

This application note describes how to use the direct memory access (DMA) controller in TRAVEO™ II family MCUs.

DMA controllers can seamlessly transfer data between memory and on-chip peripherals, or between memories without CPU intervention. This allows the CPU to handle other tasks while the DMA controller transfers data.

Both P-DMA and M-DMA have multiple independent DMA channels. Each DMA channel has a separate DMA request input that initiates the transaction and can independently transfer data. See the **device datasheet** for the number of DMA channels available for each device.

This series supports two types of DMA controllers: Peripheral DMA (P-DMA) and Memory DMA (M-DMA). P-DMA is used for peripheral-to-memory and memory-to-peripheral low-latency data transfers for many channels. M-DMA is used for memory-to-memory high-memory-bandwidth data transfer for a small number of channels.

These DMA controllers have a descriptor that specifies the transfer operation, and it corresponds flexibly to various applications. Descriptors can be chained; it is possible to have circular lists.

This application note explains the functioning of DMA controllers in the series, initial configuration, and data transfer operations with use cases.

To understand the functionality described and terminology used in this application note, see the “Direct Memory Access” chapter of the **architecture technical reference manual (TRM)**.

1.1 Features

Table 1 compares P-DMA with M-DMA, which have similar registers and descriptor structures.

Table 1 P-DMA/M-DMA Features

Feature	P-DMA	M-DMA
Focuses on	Low latency	High memory bandwidth
Useful for	Transfer between peripheral and memory	Transfer between memories
Transfer engine	Shared all channels	Dedicated for each channel
Transfer size	8-bit, 16-bit, 32-bit	8-bit, 16-bit, 32-bit
Channel priority	• Four levels • Preemptable	Four levels
Descriptor type	• Single transfer • 1D/2D transfer • CRC transfer	• Single transfer • 1D/2D transfer • Memory copy • Scatter
Descriptor	• Source and destination address • Transfer size • Descriptor type • Trigger-in type (four types) • Trigger-out type (four types) • Interrupt type (four types)	• Source and destination address • Transfer size • Descriptor type • Trigger-in type (four types) • Trigger-out type (four types) • Interrupt type (four types)

Feature	P-DMA	M-DMA
	• Descriptor chaining	• Descriptor chaining
Trigger input	• Hardware trigger • Software trigger • Trigger output (tr_out)	• Software trigger • Trigger output (tr_out)

P-DMA can be also used for transfers between memories, but the transfer bandwidth may not be enough when compared with M-DMA. M-DMA can also be used for transfers between memory and peripherals, but the transfer latency may not be low when compared with P-DMA.

In P-DMA, when preemptable, a higher-priority pending channel can preempt the current channel between single transfers. M-DMA does not have the preemptable functionality because it would degrade the overall memory bandwidth.

The descriptor determines the DMA transfer specification. The descriptor type determines the type of DMA transfer operation. Both DMAs support single transfer, 1D transfer, and 2D transfer as descriptor types. In addition, P-DMA supports CRC transfer, while M-DMA supports memory copy and scatter. See section **2.3.1 Descriptor type** for details of each descriptor type. Descriptors can be chained by storing the pointer of the next descriptor in the current descriptor. A descriptor chain is also referred to as a descriptor list.

Trigger inputs such as hardware trigger, software trigger, and trigger output (tr_out) are input via the trigger multiplexer, which is a peripheral function outside DMA. The trigger multiplexer routes triggers from potential sources to destinations. See the “Trigger Multiplexer” chapter of the **architecture TRM** for more details.

P-DMA supports hardware trigger, software trigger, and trigger output (tr_out) as trigger inputs, while M-DMA supports only software trigger and trigger output (tr_out). See the **device datasheet** for hardware triggers available. The software trigger is implemented by the trigger multiplexer function. Both DMAs can use the trigger output as their own input trigger. See section **2.3.2 Trigger functionality** for each trigger functionality.

1.2 Block diagram

Figure 1 shows the P-DMA block diagram.

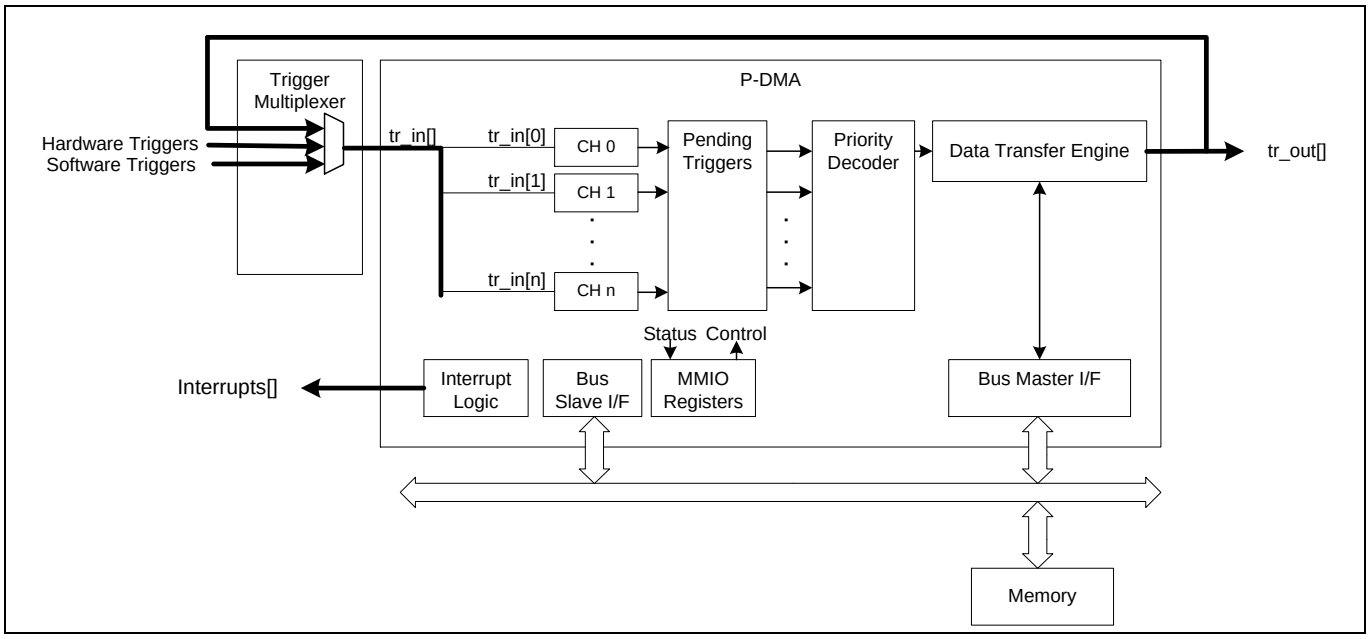


Figure 1 P-DMA block diagram

Introduction

P-DMA consists of channels (CH0 – CH*n*), a pending trigger block, priority decoder, data transfer engine, and the interrupt logic. The P-DMA transfer engine is shared by all channels. See the **architecture TRM** for details of each block.

As mentioned earlier, P-DMA trigger inputs can be a hardware trigger, software trigger, or trigger output (tr_out). These triggers are input via the trigger multiplexer.

The trigger output (tr_out) can be used as its own trigger input, or it can be used as the trigger input to trigger different transfers of other channels.

The memory that is used to store descriptors is outside the DMA block. When the transfer engine activates the next pending channel, the transfer engine reads the descriptor corresponding to the channel from the memory and starts the transfer.

Figure 2 shows M-DMA block diagram.

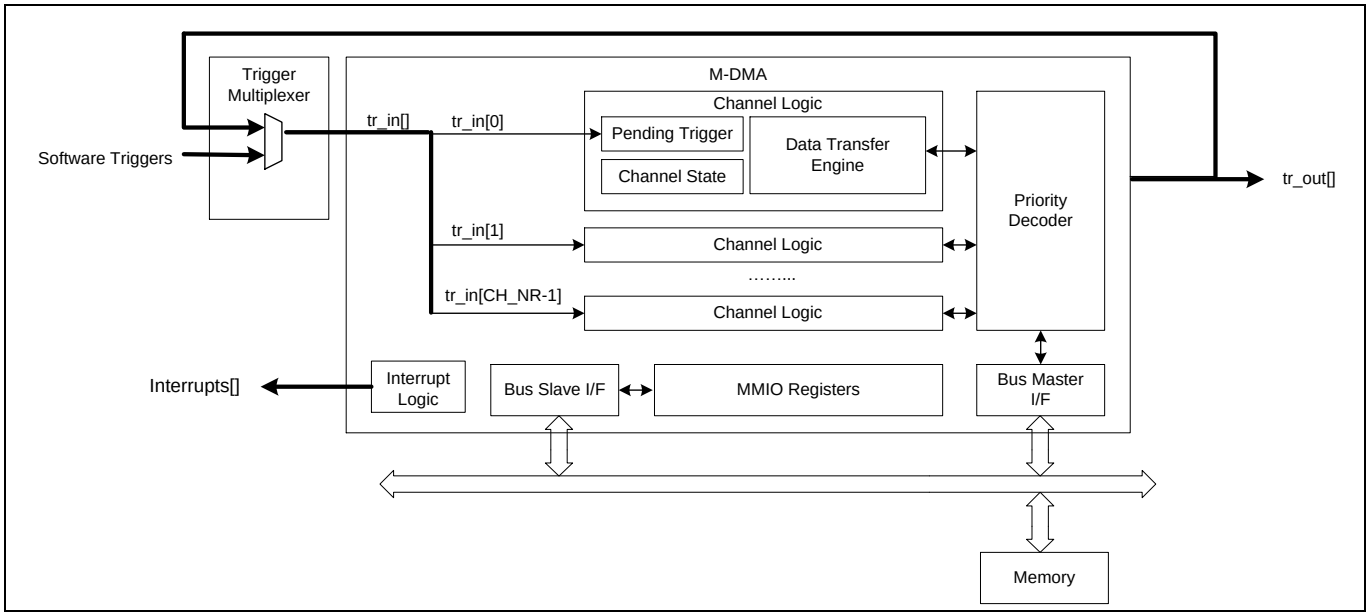


Figure 2 M-DMA block diagram

The M-DMA block consists of the channel logic, priority decoder, and registers. The channel logic itself stores the pending trigger and hosts the current channel state and data transfer engine. M-DMA has transfer engines dedicated for each channel. See the **architecture TRM** for details of each block.

As trigger inputs, M-DMA supports software trigger and its own trigger output (tr_out). These trigger inputs are input via the trigger multiplexer. Note that unlike P-DMA, M-DMA does not support hardware triggers.

2 Operation overview

Figure 3 shows how to configure P-DMA and M-DMA.

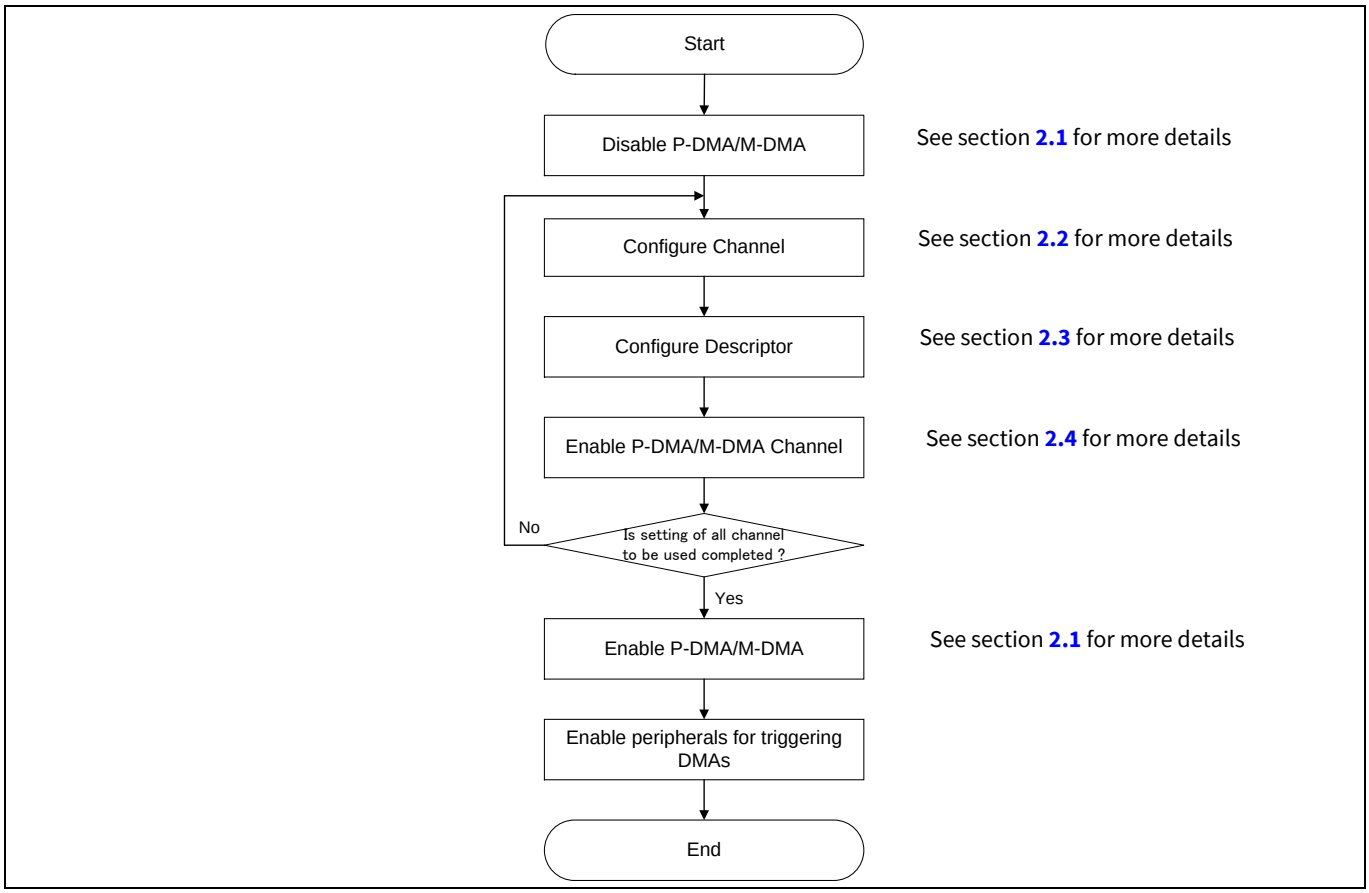


Figure 3 General configuration of P-DMA/M-DMA

In this example, channel, descriptor, and channel enable are configured for each channel. It is also possible to configure all channels to be used within each step.

A peripheral trigger is required after setting the corresponding DMA channel.

2.1 Disable/enable P-DMA/M-DMA

P-DMA and M-DMA can be enabled/disabled using the respective bits as shown in **Table 2**. The default setting after reset is '0' (Disabled).

Table 2 P-DMA/M-DMA Disable/Enable

DMA type	Register (bit)	Description
P-DMA	DW_CTL.ENABLED (bit31)	0: Disable, 1: Enable
M-DMA	DMAC_CTL.ENABLED (bit31)	0: Disable, 1: Enable

2.2 Configure channel

In this step, P-DMA/M-DMA channel settings such as the channel priority and pointer address of the descriptor corresponding to the channel are configured.

In addition, in P-DMA, the preemptable function and CRC calculation mode for CRC transfer are configured, if necessary.

Table 3 and **Table 4** show the registers that are used for configuring a channel in P-DMA and M-DMA, respectively. The registers corresponding to the channel number are configured. See the **architecture TRM** and **registers technical reference manual (registers TRM)** for more details.

Table 3 Channel configuration for P-DMA

Register (bit)	Description
DW_CH_STRUCT_CH_CURR_PTR	Sets the channel current descriptor pointer. Software needs to initialize this register.
DW_CH_STRUCT_CH_CTL.PREEMPTABLE (bit11)	Specifies whether the channel is preemptable.
DW_CH_STRUCT_CH_CTL.PRIO (bit9:8)	Sets the channel priority.
DW_CH_STRUCT_CH_IDX.X_IDX (bit7:0)	Sets the X indices of the channel into the current descriptor. Software needs to initialize this register.
DW_CH_STRUCT_CH_IDX_Y_IDX (bit15:0)	Sets the Y indices of the channel into the current descriptor. Software needs to initialize this register.
Required only for CRC transfer:	
DW_CRC_CTL.DATA_REVERSE (bit0)	Specifies the bit order (MSb or LSb first) in which a data byte is processed.
DW_CRC_CTL.REM_REVERSE (bit8)	Specifies whether the remainder is bit reversed.
DW_CRC_DATA_CTL.DATA_XOR (bit7:0)	Sets the byte mask with which each data byte is XORed. You can choose this 8-bit value randomly.
DW_CRC_POL_CTL.POLYNOMIAL	Sets the CRC polynomial.
DW_CRC_LFSR_CTL.LFSR32	Sets the seed value for CRC calculation.
DW_CRC_REM_CTL.REM_XOR	Sets a mask with which the CRC_LFSR_CTL.LFSR32 register is XORed.

Table 4 Channel configuration for M-DMA

Register (bit)	Description
DMAC_CH_CH_CURR_PTR	Sets the channel current descriptor pointer. Software needs to initialize this register.
DMAC_CH_CH_CTL.PRIO (bit9:8)	Sets the channel priority.

2.3 Configure descriptor

In this step, the descriptor is configured. The descriptor specifies the DMA channel's transfer details. A descriptor is stored in the memory outside DMA and read by the transfer engine. The transfer engine transfers the data according to the descriptor. The descriptor pointer position for each channel is stored in the descriptor pointer register (see section **2.2 Configure channel**).

Operation overview

Figure 4 shows the descriptor structure for P-DMA and M-DMA. The P-DMA descriptor consists of six 32-bit words, while the M-DMA descriptor consists of eight 32-bit words. However, descriptors of both DMAs have similar functions.

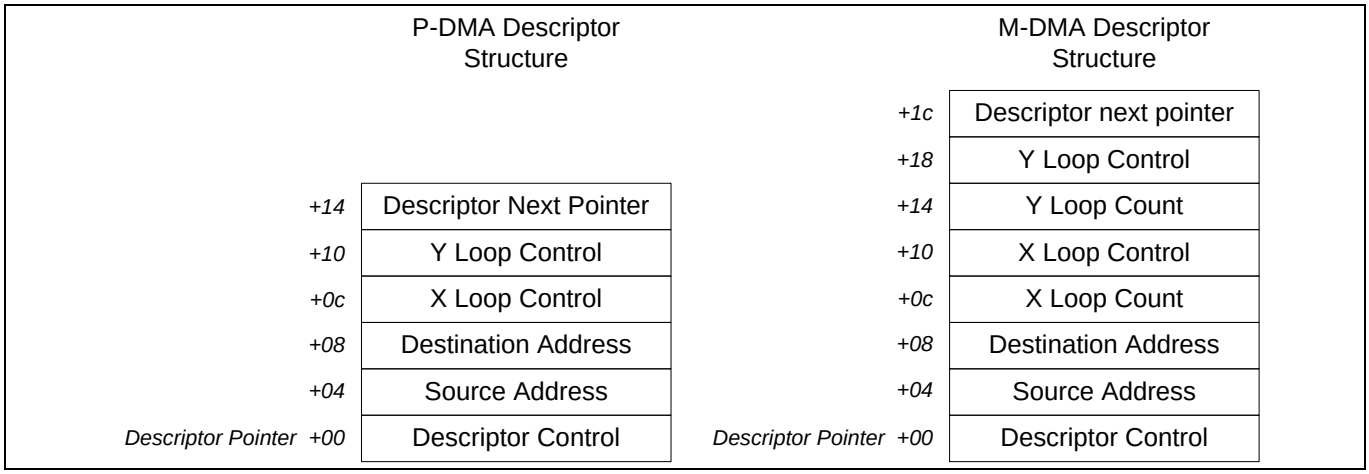


Figure 4 P-DMA/M-DMA descriptor structure

Configure descriptor parameters

- **Descriptor control:** This word describes DMA parameters such as descriptor type, transfer size, trigger-in/out, and interrupt setting.
- **Source address and destination address:** These words specify the base addresses of the source and destination locations.
- **X loop control and X loop count:** These words control the loop in 1D transfer or the inner loop in 2D transfer. The X loop control specifies the increment of the source and destination addresses for each X loop iteration. The X loop count specifies the number of iterations of the X loop.
- **Y loop control and Y loop count:** These words control the outer loop in 2D transfer. The Y loop control specifies the increment of the source and destination addresses for each Y loop iteration. The Y loop count specifies the number of iterations of the Y loop.
- **Descriptor next pointer:** This word specifies the address of the next descriptor. Descriptors can be chained by storing the descriptor of the next pointer in the current descriptor. The last descriptor in the descriptor list has '0' (NULL) in this word.

See the **architecture TRM** and **registers TRM** for descriptor details.

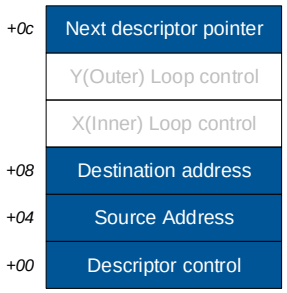
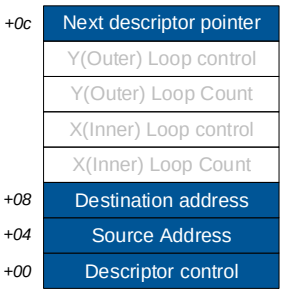
The number of descriptor words used varies depending on the descriptor type. A word address is shifted forward if there is any unused descriptor word.

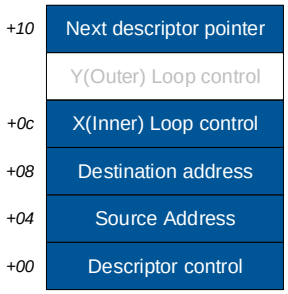
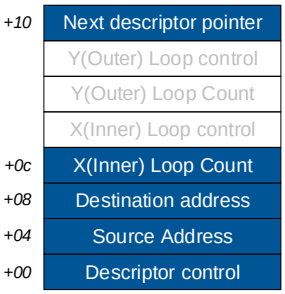
2.3.1 Descriptor type

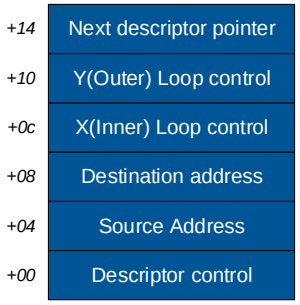
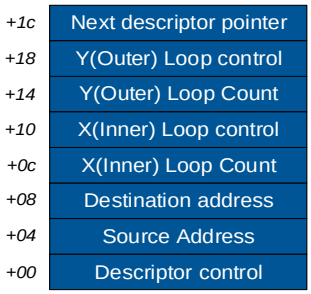
This section explains the descriptor type, which determines the type of DMA transfer.

P-DMA has four descriptor types, and M-DMA has five descriptor types. The number of descriptor words used varies depending on the descriptor type. **Table 5** shows each descriptor type. In **Table 5**, the transfer example column shows the outline and pseudocode of each descriptor type. The Using Descriptor column shows the descriptor word used by the descriptor types in each DMA. Note that a word address is shifted forward if there is any unused descriptor word.

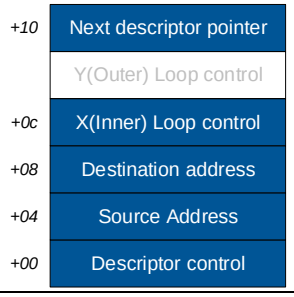
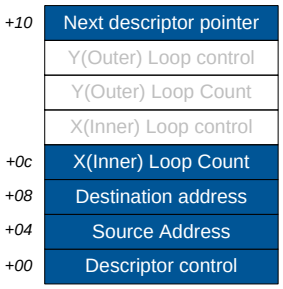
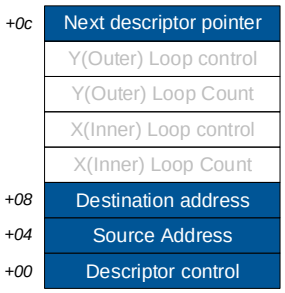
Table 5 P-DMA/M-DMA transfer example and using descriptor for each descriptor type

Descriptor type	Transfer example	Using descriptor	
		P-DMA	M-DMA
Single transfer	**This transfers a single data element:** `DST_ADDR = (DATA_SIZE) SRC_ADDR`		

1D transfer	**One-dimensional “for loop” transfer:**
	`for (X_IDX =0; X_IDX <= COUNT; X_IDX++) { DST_ADDR[DST_INCR] = (DATA_SIZE) SRC_ADDR[SRC_INCR] }`
	`*DST_INCR/SRC_INCR depend on X_INCR DST_ADDR = (DATA_SIZE) SRC_ADDR`
			

2D transfer	**Two-dimensional “for loop” transfer:**
	`for (Y_IDX =0; Y_IDX <= Y_COUNT; Y_IDX++) { for (X_IDX =0; X_IDX <= X_COUNT; X_IDX++) { DST_ADDR[DST_INCR] = (DATA_SIZE) SRC_ADDR[SRC_INCR] } }`
	`*DST_INCR/SRC_INCR depend on X/Y_INCR`
			

CRC transfer	**Calculate CRC of the specified area.**		Not supported

Descriptor type	Transfer example	Using descriptor	
		P-DMA	M-DMA
	Note that for CRC transfer, CRC must be configured with memory mapped I/O (MMIO) registers.		
Memory copy	**One-dimensional "for loop" transfer:** `for (X_IDX =0; X_IDX <= X_COUNT; X_IDX+) { DST_ADDR[IDX] = SRC_ADDR[IDX] }`	Not supported	
Scatter	**Write a set of 32-bit data elements, whose addresses are "scattered" around one-dimensional "for loop" transfer.** `for (X_IDX =0; X_IDX <= X_COUNT; X_IDX +=2) { address = SRC_ADDR[IDX] data = SRC_ADDR[IDX+1] *address = data }`	Not supported	

2.3.2 Trigger functionality

This section explains the trigger function. Trigger input, trigger output (tr_out), and interrupt are controlled by the descriptor.

A trigger input activates DMA channel transfer. The trigger output (tr_out) and interrupt are output when the transfer is complete. The trigger operation is specified by TR_IN_TYPE, TR_OUT_TYPE, and INTR_TYPE in the descriptor. Trigger input, trigger output (tr_out), and interrupt can be configured independently for each channel. See the **registers TRM** for descriptor details.

- There are four types of trigger input operations:
  - Type 0: Trigger results in the execution of a single transfer.
  - Type 1: Trigger results in the execution of a single 1D transfer.
  - Type 2: Trigger results in the execution of the current descriptor.
  - Type 3: Trigger results in the execution of a descriptor list.
- There are four types of trigger outputs and interrupt timing:
  - Type 0: Output trigger or interrupt is generated after a single transfer.
  - Type 1: Output trigger or interrupt is generated after a single 1D transfer.
  - Type 2: Output trigger or interrupt is generated after the execution of the current descriptor.
  - Type 3: Output trigger or interrupt is generated after the execution of a descriptor list.

2.3.2.1 Examples for trigger functionality

This section provides examples for different trigger functions. The descriptor list for the following examples is composed of chaining two descriptors (Descriptor 0 and Descriptor 1) in a 2D transfer.

Example 1:

This example describes the operation of a Type 0 trigger. **Figure 5** shows the operation of the trigger input and trigger output or interrupt in Type 0 trigger.

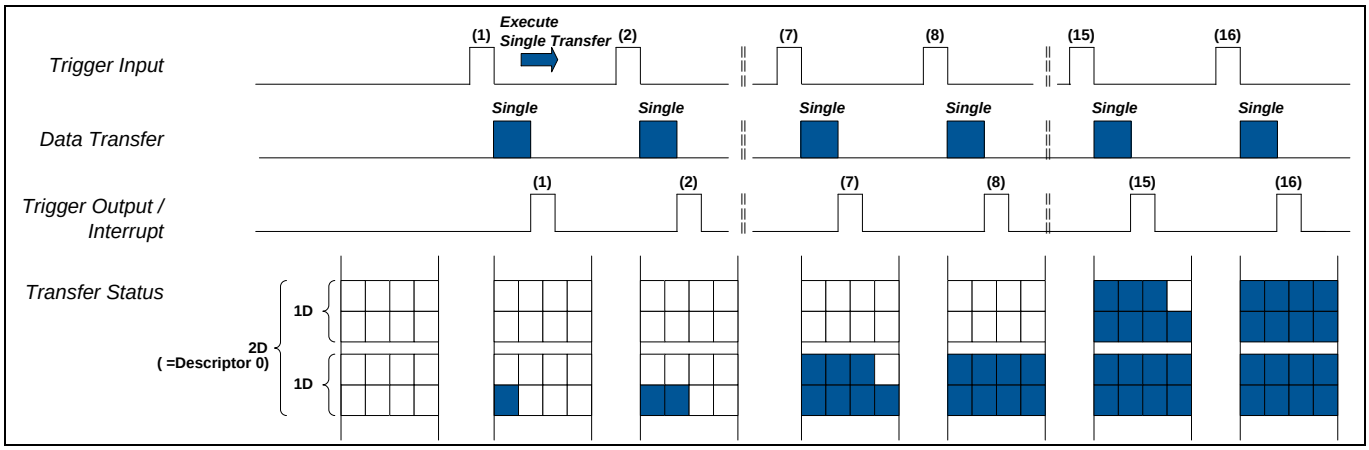


Figure 5 Trigger operation example 1

When the trigger input is set in Type 0 trigger, P-DMA/M-DMA performs a single transfer with every trigger input. Therefore, 16 trigger inputs are required to complete Descriptor 0.

Operation overview

When trigger outputs and interrupts are set in Type 0 trigger, the trigger output, interrupt, or both are output each time a single transfer is completed. Therefore, trigger output, interrupt, or both are output 16 times with the completion of Descriptor 0.

Example 2:

This example describes the Type 1 trigger operation. **Figure 6** shows the operation of trigger input and trigger output or interrupt in Type 1 trigger.

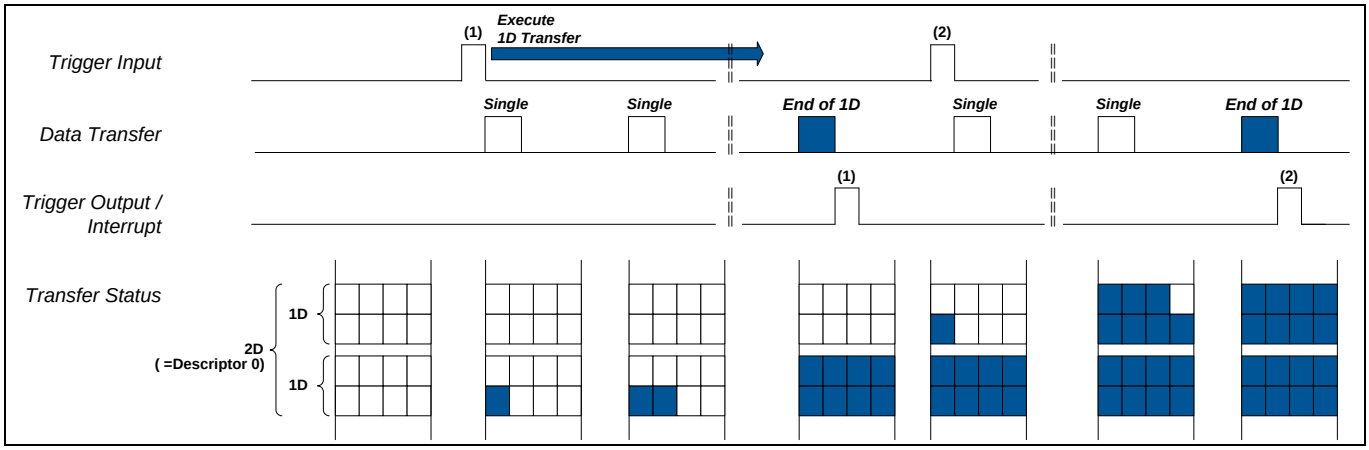


Figure 6 Trigger operation example 2

When trigger input is set in Type 1 trigger, P-DMA/M-DMA performs a 1D transfer with the trigger input. If the next trigger occurs again, P-DMA/M-DMA performs a 1D transfer. Therefore, two trigger inputs are required to complete Descriptor 0.

When trigger outputs and interrupts are set in Type 1 trigger, trigger output, interrupt, or both are output each time a 1D transfer is completed. Therefore, trigger output, interrupt, or both are output twice with the completion of Descriptor 0.

Example 3:

This example describes the operation of Type 2 trigger. **Figure 7** shows the operation of the trigger input and trigger output or interrupt in Type 2 trigger.

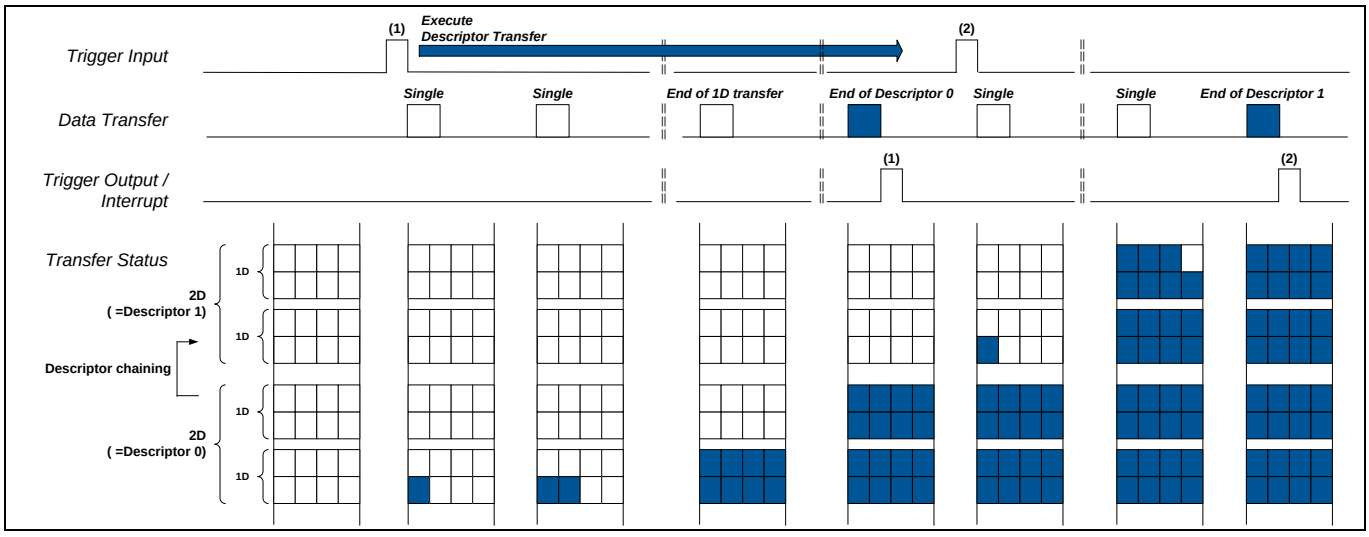


Figure 7 Trigger operation example 3

Operation overview

When the trigger input is set in Type 2 trigger, P-DMA/M-DMA executes the current descriptor (here: Descriptor 0) with the trigger input. If the next trigger input occurs, P-DMA/M-DMA executes the next descriptor (here: Descriptor 1). Therefore, two trigger inputs are required to complete the descriptor list.

When trigger outputs and interrupts are set in Type 2 trigger, trigger output, interrupt, or both are output each time a current descriptor transfer is completed. Therefore, trigger output, interrupt, or both are output twice with the completion of the descriptor list.

Example 4:

This example describes the operation of Type 3 trigger. **Figure 8** shows the operation of the trigger input and trigger output or interrupt in Type 3 trigger.

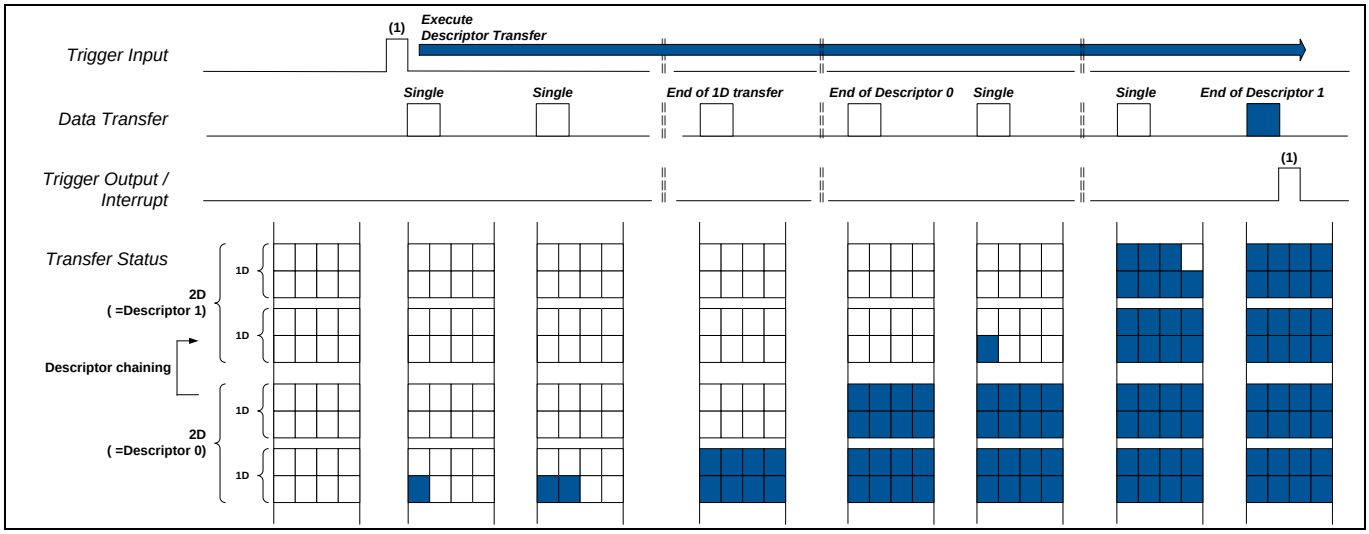


Figure 8 Trigger operation example 4

When the trigger input is set in Type 3 trigger, P-DMA/M-DMA executes the complete descriptor list with each trigger input. Therefore, one trigger input is required to complete the descriptor list.

When trigger outputs and interrupts are set in Type 3 trigger, the trigger output, interrupt, or both are output when descriptor list transfer is completed. Therefore, the trigger output and/or interrupt are output once with the completion of the descriptor list.

2.4 Disable/enable P-DMA/M-DMA channel

P-DMA and M-DMA can be configured/programmed to execute multiple independent data transfers. Each data transfer is managed by a channel.

In DMA channel configuration, the DMA channel must be disabled during the configuration, and enabled after configuring the channel. **Table 6** shows the registers used for enabling or disabling a DMA channel.

The number of channels varies for different part numbers. See the **device datasheet** for the number of available channels.

Table 6 Disable/enable P-DMA/M-DMA channel

DMA type	Register (bit)	Description
P-DMA	DW_CH_STRUCT_CH_CTL.ENABLED (bit31)	0: Disable, 1: Enable
M-DMA	DMAC_CH_CH_CTL.ENABLED (bit31)	0: Disable, 1: Enable

3 P-DMA use cases

This section describes how to use Smart I/O using the sample driver library (SDL). The code snippets in this application note are part of SDL. See **Other references** for the SDL.

SDL has a configuration part and a driver part. The configuration part mainly configures the parameter values for the desired operation. The driver part configures each register based on the parameter values in the configuration part. You can configure the configuration part according to your system.

In this example, CYT2B7 series is used.

3.1 1D transfer (memory-to-peripheral)

3.1.1 Overview

This is an example of transferring the transmit data from the memory to TX-FIFO by DMA when using the SCB UART mode. In this case, the UART uses an 8-bit data frame, and starts the transmission when the data is written to the FIFO. The number of bytes to be transmitted is 16 and the TX FIFO setting is 128 elements-deep with 8-bit data elements. See the “Serial Communications Block (SCB)” chapter of the **architecture TRM** for more details.

P-DMA starts the transfer with a software trigger from the CPU, and generates an interrupt to the CPU after transferring all data into the transmit FIFO.

Figure 9 shows the data transfer.

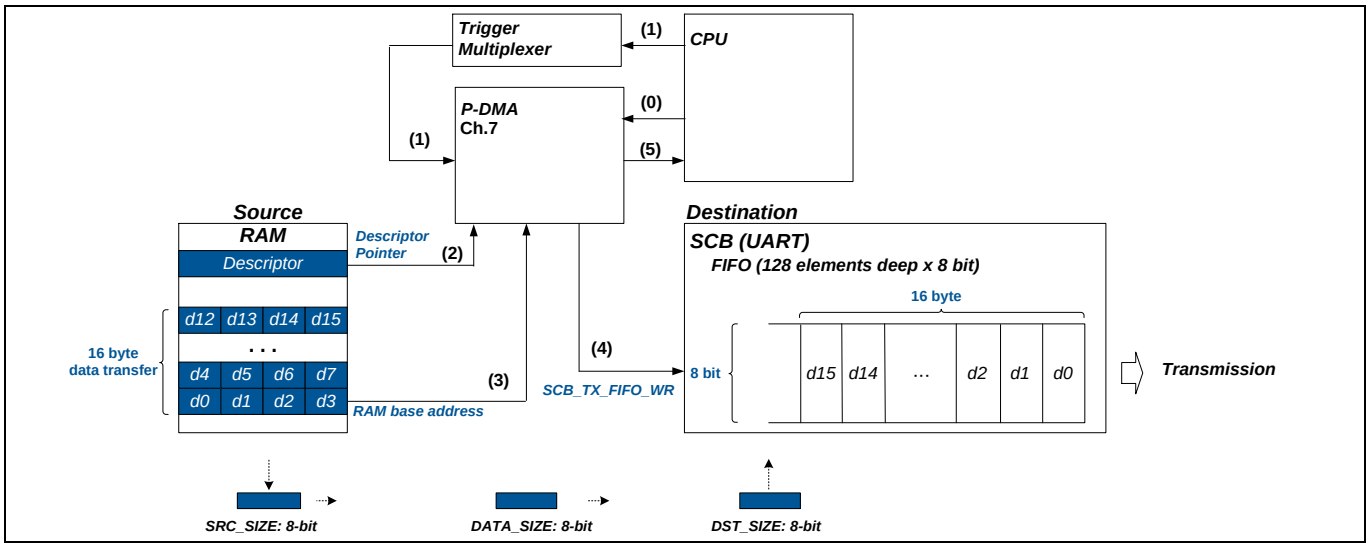


Figure 9 Use case of a memory-to-peripheral (1D transfer) using UART in SCB

(0) Configure P-DMA according to section **3.1.2 Initial configuration of channel registers**, and configure the SCB and trigger multiplexer.

(1) CPU notifies a software trigger to P-DMA via the trigger multiplexer.

(2) P-DMA reads the descriptor from the specified area (Descriptor Pointer) when activating the next pending channel.

(3) P-DMA reads data (d0) from the source address (RAM base address).

(4) P-DMA writes the read data (d0) to the destination address (SCB_TX_FIFO_WR). After that, P-DMA increments the source address by 0x01, but there is no increment of the destination address. Then, P-DMA reads the data (d1) from the source address (RAM base address +0x01) and writes it to the destination address (SCB_TX_FIFO_WR) again. P-DMA repeats from (3) to (4) until d15 data is transferred.

(5) P-DMA notifies the CPU with an interrupt when the transfer of all data is completed.

Initialize the channel registers and set the descriptor as follows.

3.1.2 Initial configuration of channel registers

This section describes the initialization of the P-DMA channel and descriptor of this use case.

Figure 10 shows the setting procedure for P-DMA.

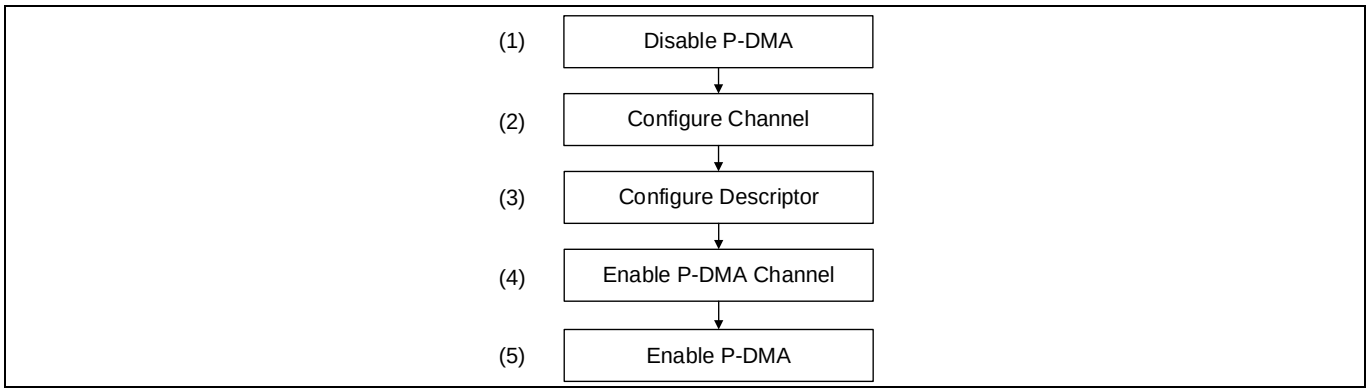


Figure 10 Setting procedure for P-DMA

3.1.3 Configuration and example code

Table 7 lists the parameters and **Table 8** lists the functions of the configuration part in SDL for P-DMA.

Table 7 List of DMA configuration Parameters

Parameters	Description	Value
DW_CHANNEL	Defines P-DMA channel	7ul
DW_CH_INTR	Defines P-DMA channel interrupt	cpuss_interrupts_dw1_7_IRQn
DW_SW_TRIG	Defines P-DMA SW Trigger	TRIG_OUT_MUX_1_PDMA1_TR_IN7
.PDMA_Descriptor	P-DMA current descriptor pointer	&dwUartTxDescr
.preemptable	Channel preemptable	0ul
.priority	Channel priority	0ul
.enable	Channel enable	1ul
.deact	DESCR_CTL WAIT_FOR_DEACT	0ul
.intrType	DESCR_CTL INTR_TYPE	CY_PDMA_INTR_DESCR_CMPLT
.trigoutType	DESCR_CTL TR_OUT_TYPE	CY_PDMA_TRIGOUT_DESCR_CMPLT
.chStateAtCmplt	DESCR_CTL CH_DISABLE	CY_PDMA_CH_DISABLED
.triginType	DESCR_CTL TR_IN_TYPE	CY_PDMA_TRIGIN_DESCR
.dataSize	DESCR_CTL DATA_SIZE	CY_PDMA_BYTE

Parameters	Description	Value
.srcTxfrSize	DESCR_CTL SRC_TRANSFER_SIZE	0ul
.destTxfrSize	DESCR_CTL DST_TRANSFER_SIZE	1ul
.descrType	DESCR_CTL DESCR_TYPE	CY_PDMA_1D_TRANSFER
.srcAddr	DESCR_SRC	(void *)g_uart_tx_buffer
.destAddr	DESCR_DST	(void *)&CY_USB_SCB_TYPE->unTX_FIFO_WR.u32Register
.srcXincr	DESCR_X_CTL SRC_X_INCR	1ul
.destXincr	DESCR_X_CTL DST_X_INCR	0ul
.xCount	DESCR_X_CTL X_COUNT	UART_TX_BUFFER_SIZE
.descrNext	DESCR_NEXT_PTR	NULL

Table 8 List of DMA configuration functions

Functions	Description	Remarks
DW1_Ch_IntHandler()	Handler for P-DMA1 interrupts	See **Code Listing 3**
Cy_PDMA_Disable()	Configures P-DMA disable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 6**
Cy_PDMA_Chnl_DeInit()	Resets P-DMA to default values	Clears all the content of registers corresponding to the channel. See **Code Listing 7**
Cy_PDMA_Descr_Init()	Configures P-DMA descriptor initialize	See **Code Listing 8**
Cy_PDMA_Chnl_Init()	Configures P-DMA channel initialize	See **Code Listing 9**
Cy_PDMA_Chnl_Enable()	Configures P-DMA channel enable	See **Code Listing 10**
Cy_PDMA_Chnl_SetInterruptMask()	Configures P-DMA channel interrupt mask	See **Code Listing 11**
Cy_PDMA_Enable()	Configures P-DMA enable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 12**
Cy_SysInt_InitIRQ()	Configures interrupt	-
Cy_SysInt_SetSystemIrqVector()	Configures Interrupt vector	-
NVIC_EnableIRQ()	NVIC Enable interrupt	-
Cy_TrigMux_SwTrigger()	Generates a Software trigger	-
Cy_PDMA_Chnl_GetInterruptStatusMasked()	Returns logical and of corresponding INTR and INTR_MASK fields in a single load operation	See **Code Listing 4**
Cy_PDMA_Chnl_ClearInterrupt()	Configures P-DMA channel clears the interrupt status	See **Code Listing 5**

Code Listing 1 demonstrates an example program to 1D Transfer (Memory-to-Peripheral). See the **architecture TRM** and **application note** for GPIO, UART, and clock configuration.

The following description will help you understand the register notation of the driver part of SDL:

- base signifies the pointer to the DMA register base address.
- ***base-> CH_STRUCT[idx].unCH_CTL.u32Register*** is the DWx_CH_STRUCT[idx] register mentioned in the **registers TRM**. Other registers are also described in the same manner. "***x***" signifies the port suffix number and "***idx***" signifies the register index number.
- To improve the register setting performance, the SDL writes a complete 32-bit data to the register. Each bit field is generated and written to the register as the final 32-bit data.

```
pstcPDMA->CH_STRUCT[chNum].unCH_CTL.u32Register;
pstcPDMA->CH_STRUCT[chNum].unCH_IDX.u32Register;
pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register;
pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register;
pstcPDMA->CH_STRUCT[chNum].unINTR_SET.u32Register;
```

See *cyip_dw_v2.h* under *hdr/rev_x/ip* for more information on the union and structure representation of registers.

Code Listing 1 Example to 1D transfer (memory-to-peripheral)

```
#define DW_CHANNEL                  7ul
#define DW_CH_INTR                  cpuss_interrupts_dw1_7_IRQn
#define DW_SW_TRIG                  TRIG_OUT_MUX_1_PDMA1_TR_IN7

void DW1_Ch_IntHandler(void);

int main(void)
{
    :

    /* Initialize & Enable DMA */
    Cy_PDMA_Disable(DW1);
    Cy_PDMA_Chnl_DeInit(DW1, DW_CHANNEL);
    Cy_PDMA_Descr_Init(&dwUartTxDescr, &dw1ChUartTxConfig);
    Cy_PDMA_Chnl_Init(DW1, DW_CHANNEL, &dwUartTxConfig);
    Cy_PDMA_Chnl_Enable(DW1, DW_CHANNEL);
    Cy_PDMA_Chnl_SetInterruptMask(DW1, DW_CHANNEL);
    Cy_PDMA_Enable(DW1);

    /* Prepare source buffer data */
    for(uint32_t i = 0ul; i < UART_TX_BUFFER_SIZE; i++)
    {
        g_uart_tx_buffer[i] = (i + 48ul);    /* ASCII Code '0','1','
    }

    /* Interrupt Initialization */
    Cy_SysInt_InitIRQ(&stc_sysint_irq_cfg);
    Cy_SysInt_SetSystemIrqVector(stc_sysint_irq_cfg.sysIntSrc, DW1_Ch_IntHandler);
    NVIC_EnableIRQ(stc_sysint_irq_cfg.intIdx);

    /* SW Trigger */
    Cy_TrigMux_SwTrigger(DW_SW_TRIG, TRIGGER_TYPE_EDGE, 1ul);

    for(;;);

}v
```

Define DW channel
Define DW channel interrupt
Define DW SW Trigger

See **Code Listing 3**.

(1) Disable P-DMA. See **Code Listing 6**.
(2) Resets P-DMA to default values. See **Code Listing 7**.
(3) Configures P-DMA descriptor initialize. See **Code Listing 8**.
(4) Enable P-DMA channel. See **Code Listing 10**
(5) Enable P-DMA. See **Code Listing 12**.

Configures interrupt

Generates a software trigger

Code Listing 2 P-DMA channel, descriptor configuration

```
/**
 * \var uint8_t g_uart_tx_buffer
 * \brief UART TX buffer
 */
uint8_t  g_uart_tx_buffer[UART_TX_BUFFER_SIZE] = { 0ul };

/**
 * \var cy_stc_pdma_descr_t dwUartTxDescr
 * \brief PDMA descriptor
 */
static  cy_stc_pdma_descr_t         dwUartTxDescr;
```

Variable for source address TX_buffer

Variable for P-DMA descriptor

Code Listing 2 P-DMA channel, descriptor configuration

```
/**
 * \var cy_stc_pdma_chnl_config_t dwUartTxConfig
 * \brief PDMA configuration
 */
const   cy_stc_pdma_chnl_config_t   dwUartTxConfig =
{
    .PDMA_Descriptor = &dwUartTxDescr,
    .preemptable     = 0ul,
    .priority        = 0ul,
    .enable          = 1ul,
};

/**
 * \var cy_stc_pdma_descr_config_t dw1ChUartTxConfig
 * \brief PDMA descriptor configuration
 */
static  cy_stc_pdma_descr_config_t  dw1ChUartTxConfig =
{
    .deact          =   0ul,
    .intrType       =   CY_PDMA_INTR_DESCR_CMPLT,
    .trigoutType    =   CY_PDMA_TRIGOUT_DESCR_CMPLT,
    .chStateAtCmplt =   CY_PDMA_CH_DISABLED,
    .triginType     =   CY_PDMA_TRIGIN_DESCR,
    .dataSize       =   CY_PDMA_BYTE,
    .srcTxfrSize    =   0ul, // same as "dataSize"
    .destTxfrSize   =   1ul, // 32bit width
    .descrType      =   CY_PDMA_1D_TRANSFER,
    .srcAddr        =   (void *)g_uart_tx_buffer,
    .destAddr       =   (void *)&CY_USB_SCB_TYPE->unTX_FIFO_WR.u32Register,
    .srcXincr       =   1ul,
    .destXincr      =   0ul,
    .xCount         =   UART_TX_BUFFER_SIZE,
    .descrNext      =   NULL
};

/**
 * \var cy_stc_sysint_irq_t stc_sysint_irq_cfg
 * \brief IRQ DMA configuration
 */
static const cy_stc_sysint_irq_t stc_sysint_irq_cfg =
{
    .sysIntSrc = DW_CH_INTR,
    .intIdx    = CPUIntIdx2_IRQn,
    .isEnabled = true,
};
```

Configure P-DMA channel

Configure P-DMA descriptor

Configure P-DMA descriptor

Configure interrupt

Code Listing 3 DW1_Ch_IntHandler()

```
void DW1_Ch_IntHandler(void)
{
    uint32_t masked;

    masked = Cy_PDMA_Chnl_GetInterruptStatusMasked(DW1, DW_CHANNEL);
    if ((masked & CY_PDMA_INTRCAUSE_COMPLETION) != 0ul)
    {
        /* Clear Complete DMA interrupt flag */
        Cy_PDMA_Chnl_ClearInterrupt(DW1, DW_CHANNEL);
    }
    else
    {
        CY_ASSERT(false);
    }
}
```

See **Code Listing 4**.

See **Code Listing 5**.

Code Listing 4 Cy_PDMA_Chnl_GetInterruptStatusMasked()

```
__STATIC_INLINE uint32_t Cy_PDMA_Chnl_GetInterruptStatusMasked(const volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    return (pstcPDMA->CH_STRUCT[chNum].unINTR_MASKED.u32Register);
}
```

Code Listing 5 Cy_PDMA_Chnl_ClearInterrupt()

```
void Cy_PDMA_Chnl_ClearInterrupt(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unINTR.u32Register = CY_PDMA_INTR_BIT_MASK;
```

Code Listing 5 Cy_PDMA_Chnl_ClearInterrupt()

```
    /* Readback of the register is required by hardware. */
    (void) pstcPDMA->CH_STRUCT[chNum].unINTR.u32Register;
}
```

Code Listing 6 Cy_PDMA_Disable()

```
void Cy_PDMA_Disable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       0ul;
}
```

Write to P-DMA_CTL_ENABLED bit

Code Listing 7 Cy_PDMA_Chnl_DeInit()

```
void Cy_PDMA_Chnl_DeInit(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unCH_CTL.u32Register            = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unCH_IDX.u32Register            = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register       = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register         = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unINTR_SET.u32Register          = 0ul;
}
```

Resets P-DMA to default value

Code Listing 8 Cy_PDMA_Descr_Init()

```
cy_en_pdma_status_t Cy_PDMA_Descr_Init(cy_stc_pdma_descr_t* descriptor, const cy_stc_pdma_descr_config_t* config)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((descriptor != NULL) && (config != NULL))
    {
        descriptor->unPDMA_DESCR_CTL.stcField.u2WAIT_FOR_DEACT          =       config->deact;
        descriptor->unPDMA_DESCR_CTL.stcField.u2INTR_TYPE               =       config->intrType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_OUT_TYPE             =       config->trigoutType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_IN_TYPE              =       config->triginType;
        descriptor->unPDMA_DESCR_CTL.stcField.u1SRC_TRANSFER_SIZE       =       config->srcTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1DST_TRANSFER_SIZE       =       config->destTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1CH_DISABLE              =       config->chStateAtCmplt;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DATA_SIZE               =       config->dataSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DESCR_TYPE              =       config->descrType;

        descriptor->u32PDMA_DESCR_SRC                                   =       (uint32_t)config->srcAddr;
        descriptor->u32PDMA_DESCR_DST                                   =       (uint32_t)config->destAddr;

        switch(config->descrType)
        {
        case (uint32_t)CY_PDMA_SINGLE_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_1D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_CRC_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       0ul;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_2D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);

            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12SRC_Y_INCR   =       (uint32_t)config->srcYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12DST_Y_INCR   =       (uint32_t)config->destYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u8Y_COUNT       =       (uint32_t)((config->yCount) - 1ul);

            descriptor->u32PDMA_DESCR_NEXT_PTR                      =       (uint32_t)config->descrNext;
```

Code Listing 8 Cy_PDMA_Descr_Init()

```
            break;
        }
        default:
        {
            /* Unsupported type of descriptor */
            break;
        }
        }

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return retVal;
}
```

Code Listing 9 Cy_PDMA_Chnl_Init()

```
cy_en_pdma_status_t Cy_PDMA_Chnl_Init(volatile stc_DW_t *pstcPDMA, uint32_t chNum, const cy_stc_pdma_chnl_config_t*
chnlConfig)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((pstcPDMA != NULL) && (chnlConfig != NULL))
    {
        /* Set current descriptor */
        pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register            =  (uint32_t)chnlConfig->PDMA_Descriptor;

        /* Set if the channel is preemtable */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1PREEMPTABLE      =  chnlConfig->preemptable;

        /* Set channel priority */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u2PRIO             =  chnlConfig->priority;

        /* Set enabled status */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED          =  chnlConfig->enable;

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return (retVal);
}
```

Code Listing 10 Cy_PDMA_Chnl_Enable()

```
__STATIC_INLINE void Cy_PDMA_Chnl_Enable(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED = 1ul;
}
```

Code Listing 11 Cy_PDMA_Chnl_SetInterruptMask()

```
__STATIC_INLINE void Cy_PDMA_Chnl_SetInterruptMask(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register = CY_PDMA_INTR_BIT_MASK;
}
```

Code Listing 12 Cy_PDMA_Enable()

```
void Cy_PDMA_Enable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       1ul;
}
```

Write to P-DMA_CTL_ENABLED bit

3.2 1D transfer (peripheral-to-memory)

3.2.1 Overview

This is an example of transferring input states of multiple ports to the memory (RAM) with a periodic trigger. A TCPWM is used for generating the periodic trigger. See the “Timer, Counter, and PWM” chapter of the **architecture TRM** for more details.

The port input state is stored in the memory with the periodic timing set in TCPWM. As a result, the CPU can know the port state at a specified time by reading the memory without the CPU having to access the port register. This is useful when storing the port state before transitioning to a low-power mode.

In this case, P-DMA transfers the GPIO_PRT_IN register data to RAM. The number of registers to be transferred is 23. (from Port 0 to Port 22). GPIO_PRT_IN registers exist at intervals of 0x80. P-DMA generates an interrupt after data is transferred from all registers. **Figure 11** shows the data transfer.

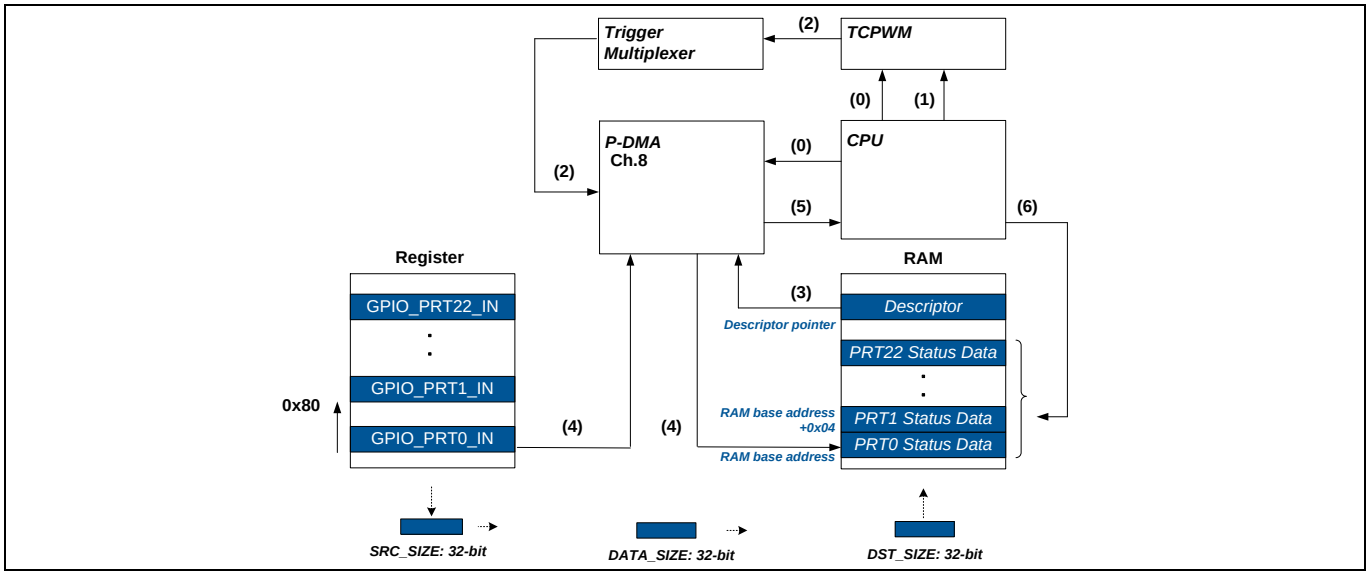


Figure 11 Operation of 1D transfer (peripheral-to-memory)

(0) Configure P-DMA according to section **3.2.2 Initial configuration for P-DMA**. In addition, configure the TCPWM and trigger multiplexer.

(1) The CPU starts the TCPWM timer.

(2) TCPWM outputs the trigger to P-DMA via the trigger multiplexer when it reaches the specified count value (overflow).

(3) P-DMA reads the descriptor from the specified area (Descriptor Pointer) when accepting the transfer.

(4) P-DMA reads the data from the source address (GPIO_PRT0_IN), and writes the read data to the destination address (RAM base address). After that, P-DMA increments the source address by 0x80 and the destination address by 0x04. Then, P-DMA reads the data from the source address (GPIO_PRT1_IN) and writes it to the destination address (RAM base address +0x04) again.

(5) P-DMA notifies the CPU with an interrupt when all transfers are completed.

(6) The CPU accepts the interrupt and reads the port states in the RAM.

When the TCPWM outputs the trigger again, steps (3) to (6) are repeated.

3.2.2 Initial configuration for P-DMA

This section describes the initialization of the P-DMA channel and descriptor of this use case.

Figure 12 shows the setting procedure for P-DMA.

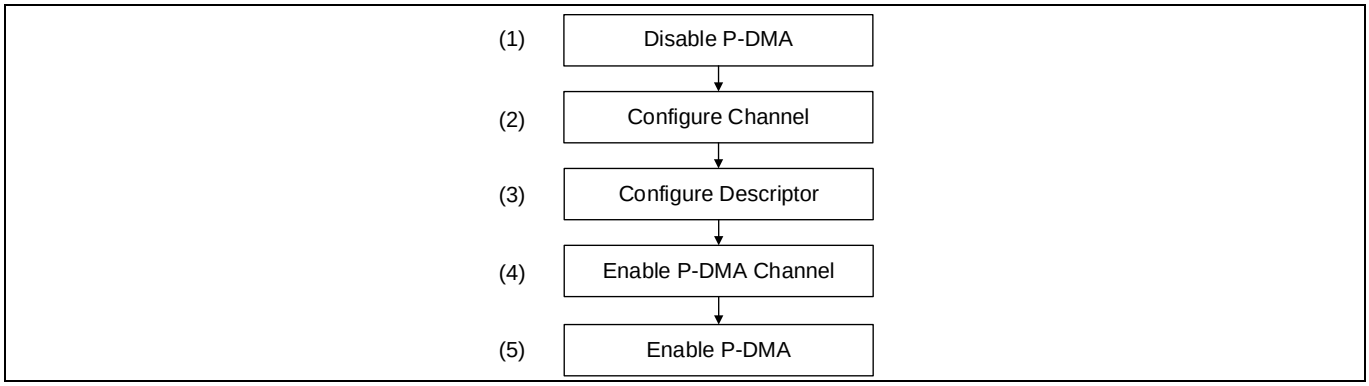


Figure 12 Setting procedure for P-DMA

3.2.3 Configuration and example code

Table 9 lists the parameters and **Table 10** lists the functions of the configuration part in SDL for DMA.

Table 9 List of DMA configuration parameters

Parameters	Description	Value
TCPWMx_GRPx_CNTx_COUNTER	Defines TCPWM using counter	TCPWM0_GRP0_CNT0
DW_CHANNEL	Defines P-DMA channel	8ul
TCPWM_TO_MUX_TRIG	Defines trigger TCPWM to multiplexer	TRIG_IN_MUX_3_TCPWM_16_TR_OUT00
MUX_TO_PDMA_TRIG	Defines trigger multiplexer to P-DMA	TRIG_OUT_MUX_3_PDMA0_TR_IN8
DW_CH_INTR	Defines P-DMA channel interrupt	cpuss_interrupts_dw0_8_IRQn
DST_BUFFER_SIZE	Defines destination buffer size	23ul
.PDMA_Descriptor	P-DMA current descriptor pointer	&dwTcpwmDescr
.preemptable	Channel preemptable	0ul
.priority	Channel priority	0ul
.enable	Channel enable	1ul
.deact	DESCR_CTL WAIT_FOR_DEACT	0ul
.intrType	DESCR_CTL INTR_TYPE	CY_PDMA_INTR_X_LOOP_CMPLT
.trigoutType	DESCR_CTL TR_OUT_TYPE	CY_PDMA_TRIGOUT_X_LOOP_CMPLT
.chStateAtCmplt	DESCR_CTL CH_DISABLE	CY_PDMA_CH_ENABLED
.triginType	DESCR_CTL TR_IN_TYPE	CY_PDMA_TRIGIN_DESCR
.dataSize	DESCR_CTL DATA_SIZE	CY_PDMA_WORD

P-DMA use cases

Parameters	Description	Value
.srcTxfrSize	DESCR_CTL SRC_TRANSFER_SIZE	0ul
.destTxfrSize	DESCR_CTL DST_TRANSFER_SIZE	0ul
.descrType	DESCR_TYPE	CY_PDMA_1D_TRANSFER
.srcAddr	DESCR_SRC	(void *)&GPIO_PRT0->unIN.u32Register
.destAddr	DESCR_DST	(void *)g_DestBuffer
.srcXincr	DESCR_X_CTL SRC_X_INCR	32ul
destXincr	DESCR_X_CTL DST_X_INCR	1ul
.xCount	DESCR_X_CTL X_COUNT	DST_BUFFER_SIZE
.descrNext	DESCR_NEXT_PTR	&dwTcpwmDescr

Table 10 List of DMA configuration functions

Functions	Description	Remarks
DW0_Ch_IntHandler()	Handler for P-DMA0 interrupts	See **Code Listing 15**
Counter_Handler()	Handler for TCPWM Counter interrupts	See **Code Listing 18**
Cy_SysInt_InitIRQ()	Configures interrupt	-
Cy_SysInt_SetSystemIrqVector()	Configures interrupt vector	-
NVIC_SetPriority()	NVIC set priority	-
NVIC_EnableIRQ()	NVIC enable interrupt	-
Cy_PDMA_Disable()	Configures P-DMA disable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 19**
Cy_PDMA_Chnl_DeInit()	Resets P-DMA to default values	Clears all the content of registers corresponding to the channel See **Code Listing 20**
Cy_PDMA_Descr_Init()	Configures P-DMA descriptor initialize	See **Code Listing 21**
Cy_PDMA_Chnl_Init()	Configures P-DMA channel initialize	See **Code Listing 22**
Cy_PDMA_Chnl_Enable()	Configures P-DMA channel enable	See **Code Listing 23**
Cy_PDMA_Chnl_SetInterruptMask()	Configures P-DMA channel interrupt mask	See **Code Listing 24**
Cy_PDMA_Enable()	Configures P-DMA enable	Write to DW_CTL_ENABLED bit See Code Listing 25
Cy_TrigMux_Connect()	Trigger multiplexer initialization	-
Cy_Tcpwm_TriggerStart()	TCPWM start	-
Cy_PDMA_Chnl_GetInterruptStatusMasked()	Returns logical and of corresponding INTR and	See **Code Listing 16**

P-DMA use cases

Functions	Description	Remarks
	INTR_MASK fields in a single load operation	
Cy_PDMA_Chnl_ClearInterrupt()	Configures P-DMA channel clears the interrupt status	See **Code Listing 17**
Cy_Tcpwm_Counter_ClearCC0_Intr()	Clear TCPWM CC0 interrupt flag	-

Code Listing 13 demonstrates an example program to 1D transfer (peripheral-to-memory). See the **architecture TRM** and **application note** for GPIO, TCPWM and Clock configuration.

Code Listing 13 Example to 1D transfer (peripheral -to- memory)

```
#define TCPWMx_GRPx_CNTx_COUNTER            TCPWM0_GRP0_CNT0


#define DW_CHANNEL                          8ul
#define TCPWM_TO_MUX_TRIG                   TRIG_IN_MUX_3_TCPWM_16_TR_OUT00
#define MUX_TO_PDMA_TRIG                    TRIG_OUT_MUX_3_PDMA0_TR_IN8
#define DW_CH_INTR                          cpuss_interrupts_dw0_8_IRQn
#define DST_BUFFER_SIZE                     23ul


void DW0_Ch_IntHandler(void);
void Counter_Handler(void);

int main(void)
{

:

    /* Initialize & Enable DMA */
    Cy_PDMA_Disable(DW0);
    Cy_PDMA_Chnl_DeInit(DW0, DW_CHANNEL);
    Cy_PDMA_Descr_Init(&dwTcpwmDescr, &dw0ChTcpwmConfig);
    Cy_PDMA_Chnl_Init(DW0, DW_CHANNEL, &dwTcpwmConfig);
    Cy_PDMA_Chnl_Enable(DW0, DW_CHANNEL);
    Cy_PDMA_Chnl_SetInterruptMask(DW0, DW_CHANNEL);
    Cy_PDMA_Enable(DW0);

    /* Trigger Multiplexer Initialization */
    /* TCPWM To DW */
    Cy_TrigMux_Connect(TCPWM_TO_MUX_TRIG, MUX_TO_PDMA_TRIG, CY_TR_MUX_TR_INV_DISABLE, TRIGGER_TYPE_LEVEL, 0ul);

    /* Interrupt Initialization */
    Cy_SysInt_InitIRQ(&stc_sysint_irq_cfg);
    Cy_SysInt_SetSystemIrqVector(stc_sysint_irq_cfg.sysIntSrc, DW0_Ch_IntHandler);
    NVIC_SetPriority(stc_sysint_irq_cfg.intIdx, 0ul);
    NVIC_EnableIRQ(stc_sysint_irq_cfg.intIdx);

    /* TCPWM Start */
    Cy_Tcpwm_TriggerStart(TCPWMx_GRPx_CNTx_COUNTER);

    for(;;);
}
```

Define TCPWM using counter

Define P-DMA channel
Define Trigger TCPWM to MUX
Define Trigger MUX to P-DMA
Define P-DMA channel interrupt
Define DST buffer size

See **Code Listing 15**

See **Code Listing 18**

(1) Disable P-DMA. See **Code Listing 19**.
(2) Resets P-DMA to default values. See **Code Listing 20**.
(3) Configures P-DMA descriptor initialize. See **Code Listing 21**.
(4) Enable P-DMA channel. See **Code Listing 23**.
(5) Enable P-DMA. See **Code Listing 25**.

Configures interrupt

TCPWM start

Code Listing 14 P-DMA Channel, descriptor configuration

```
/**
 * \var cy_stc_pdma_descr_t dwTcpwmDescr
 * \brief PDMA descriptor
 */
static  cy_stc_pdma_descr_t         dwTcpwmDescr;

/**
 * \var cy_stc_pdma_chnl_config_t dwTcpwmConfig
 * \brief PDMA configuration
 */
const   cy_stc_pdma_chnl_config_t   dwTcpwmConfig =
{
    .PDMA_Descriptor = &dwTcpwmDescr,
    .preemptable     = 0ul,
    .priority        = 0ul,
    .enable          = 1ul,
};
```

Variable for source address TX_buffer

Configure P-DMA channel

Code Listing 14 P-DMA Channel, descriptor configuration

```
/**
 * \var cy_stc_pdma_descr_config_t dw0ChTcpwmConfig
 * \brief PDMA descriptor configuration
 */
static  cy_stc_pdma_descr_config_t  dw0ChTcpwmConfig =
{
    .deact          =   0ul,
    .intrType       =   CY_PDMA_INTR_X_LOOP_CMPLT,
    .trigoutType    =   CY_PDMA_TRIGOUT_X_LOOP_CMPLT,
    .chStateAtCmplt =   CY_PDMA_CH_ENABLED,
    .triginType     =   CY_PDMA_TRIGIN_DESCR,
    .dataSize       =   CY_PDMA_WORD,
    .srcTxfrSize    =   0ul, /* Same as dataSize */
    .destTxfrSize   =   0ul, /* Same as dataSize */
    .descrType      =   CY_PDMA_1D_TRANSFER,
    .srcAddr        =   (void *)&GPIO_PRT0->unIN.u32Register,
    .destAddr       =   (void *)g_DestBuffer,
    .srcXincr       =   32ul, /* address +80h */
    .destXincr      =   1ul,  /* address +04h */
    .xCount         =   DST_BUFFER_SIZE,
    .descrNext      =   &dwTcpwmDescr
};

/**
 * \var cy_stc_sysint_irq_t stc_sysint_irq_cfg
 * \brief IRQ DMA configuration
 */
static const cy_stc_sysint_irq_t stc_sysint_irq_cfg =
{
    .sysIntSrc = DW_CH_INTR,
    .intIdx    = CPUIntIdx2_IRQn,
    .isEnabled = true,
};
```

Configure P-DMA descriptor

Configure interrupt

Code Listing 15 DW0_Ch_IntHandler()

```
void DW0_Ch_IntHandler(void)
{
    uint32_t masked;

    masked = Cy_PDMA_Chnl_GetInterruptStatusMasked(DW0, DW_CHANNEL);
    if ((masked & CY_PDMA_INTRCAUSE_COMPLETION) != 0ul)
    {
        /* Clear Complete DMA interrupt flag */
        Cy_PDMA_Chnl_ClearInterrupt(DW0, DW_CHANNEL);

        /* Read the port status */
        for(uint8_t portcnt = 0; portcnt < DST_BUFFER_SIZE; portcnt++)
        {
            g_portStatus[portcnt] = g_DestBuffer[portcnt];
        }
    }
    else
    {
        CY_ASSERT(false);
    }
}
```

See **Code Listing 16**.

See **Code Listing 17**.

Code Listing 16 Cy_PDMA_Chnl_GetInterruptStatusMasked()

```
__STATIC_INLINE uint32_t Cy_PDMA_Chnl_GetInterruptStatusMasked(const volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    return (pstcPDMA->CH_STRUCT[chNum].unINTR_MASKED.u32Register);
}
```

Code Listing 17 Cy_PDMA_Chnl_ClearInterrupt()

```
void Cy_PDMA_Chnl_ClearInterrupt(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unINTR.u32Register = CY_PDMA_INTR_BIT_MASK;

    /* Readback of the register is required by hardware. */
    (void) pstcPDMA->CH_STRUCT[chNum].unINTR.u32Register;
}
```

Code Listing 18 Counter_Handler()

```
void Counter_Handler(void)
{
    if(Cy_Tcpwm_Counter_GetCC0_IntrMasked(TCPWMx_GRPx_CNTx_COUNTER))
    {
        /* Clear TCPWM CC0 interrupt flag Group#0 Counter#0 */
        Cy_Tcpwm_Counter_ClearCC0_Intr(TCPWMx_GRPx_CNTx_COUNTER);

        /* user program here.. */
    }
}
```

Code Listing 19 Cy_PDMA_Disable()

```
void Cy_PDMA_Disable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       0ul;
}
```

Write to P-DMA_CTL_ENABLED bit

Code Listing 20 Cy_PDMA_Chnl_DeInit()

```
void Cy_PDMA_Chnl_DeInit(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unCH_CTL.u32Register            = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unCH_IDX.u32Register            = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register       = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register         = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unINTR_SET.u32Register          = 0ul;
}
```

Resets P-DMA to default value

Code Listing 21 Cy_PDMA_Descr_Init()

```
cy_en_pdma_status_t Cy_PDMA_Descr_Init(cy_stc_pdma_descr_t* descriptor, const cy_stc_pdma_descr_config_t* config)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((descriptor != NULL) && (config != NULL))
    {
        descriptor->unPDMA_DESCR_CTL.stcField.u2WAIT_FOR_DEACT          =       config->deact;
        descriptor->unPDMA_DESCR_CTL.stcField.u2INTR_TYPE               =       config->intrType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_OUT_TYPE             =       config->trigoutType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_IN_TYPE              =       config->triginType;
        descriptor->unPDMA_DESCR_CTL.stcField.u1SRC_TRANSFER_SIZE       =       config->srcTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1DST_TRANSFER_SIZE       =       config->destTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1CH_DISABLE              =       config->chStateAtCmplt;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DATA_SIZE               =       config->dataSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DESCR_TYPE              =       config->descrType;

        descriptor->u32PDMA_DESCR_SRC                                   =       (uint32_t)config->srcAddr;
        descriptor->u32PDMA_DESCR_DST                                   =       (uint32_t)config->destAddr;

        switch(config->descrType)
        {
        case (uint32_t)CY_PDMA_SINGLE_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_1D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_CRC_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       0ul;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_2D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
```

Code Listing 21 Cy_PDMA_Descr_Init()

```
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);

            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12SRC_Y_INCR   =       (uint32_t)config->srcYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12DST_Y_INCR   =       (uint32_t)config->destYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u8Y_COUNT       =       (uint32_t)((config->yCount) - 1ul);

            descriptor->u32PDMA_DESCR_NEXT_PTR                      =       (uint32_t)config->descrNext;
            break;
        }
        default:
        {
            /* Unsupported type of descriptor */
            break;
        }
        }

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return retVal;
}
```

Code Listing 22 Cy_PDMA_Chnl_Init()

```
cy_en_pdma_status_t Cy_PDMA_Chnl_Init(volatile stc_DW_t *pstcPDMA, uint32_t chNum, const cy_stc_pdma_chnl_config_t*
chnlConfig)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((pstcPDMA != NULL) && (chnlConfig != NULL))
    {
        /* Set current descriptor */
        pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register            =  (uint32_t)chnlConfig->PDMA_Descriptor;

        /* Set if the channel is preemtable */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1PREEMPTABLE      =  chnlConfig->preemptable;

        /* Set channel priority */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u2PRIO             =  chnlConfig->priority;

        /* Set enabled status */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED          =  chnlConfig->enable;

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return (retVal);
}
```

Code Listing 23 Cy_PDMA_Chnl_Enable()

```
__STATIC_INLINE void Cy_PDMA_Chnl_Enable(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED = 1ul;
}
```

Code Listing 24 Cy_PDMA_Chnl_SetInterruptMask()

```
__STATIC_INLINE void Cy_PDMA_Chnl_SetInterruptMask(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register = CY_PDMA_INTR_BIT_MASK;
}
```

Code Listing 25 Cy_PDMA_Enable()

```
void Cy_PDMA_Enable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       1ul;
}
```

Write to P-DMA_CTL_ENABLED bit

3.3 Descriptor chaining

3.3.1 Overview

This is an example of descriptor chaining by storing the pointer of the next descriptor (DESCR_NEXT_PTR) in the current descriptor.

This example uses two descriptors: Descriptor 0 and Descriptor 1. Both descriptors are for a 1D transfer that transfers multiple port input states to the memory (RAM) with a periodic trigger from the TCPWM. However, Descriptor 0 has RAM base address 0 as the destination address, and Descriptor 1 has RAM base address 1 as the destination address.

P-DMA transfers multiple port input states to the memory (RAM) with a periodic trigger from the TCPWM. P-DMA notifies the interrupt to the CPU when the transfer is completed. The pointer to the next descriptor (DESCR_NEXT_PTR) in the descriptor (Descriptor 0) of this transfer is set to a pointer to another descriptor (Descriptor 1). As a result, P-DMA can start the Descriptor 1 transfer when Descriptor 0 transfer completes. Descriptor 0 and Descriptor 1 have different destination addresses.

P-DMA allows data of the same port address to be transferred to different RAM addresses. In other words, it can have a double buffer. In this case, Descriptor 0 and Descriptor 1 are chained with each other, and Descriptor 0 and Descriptor 1 have a circular list.

Figure 13 shows 1D transfer with descriptor chaining.

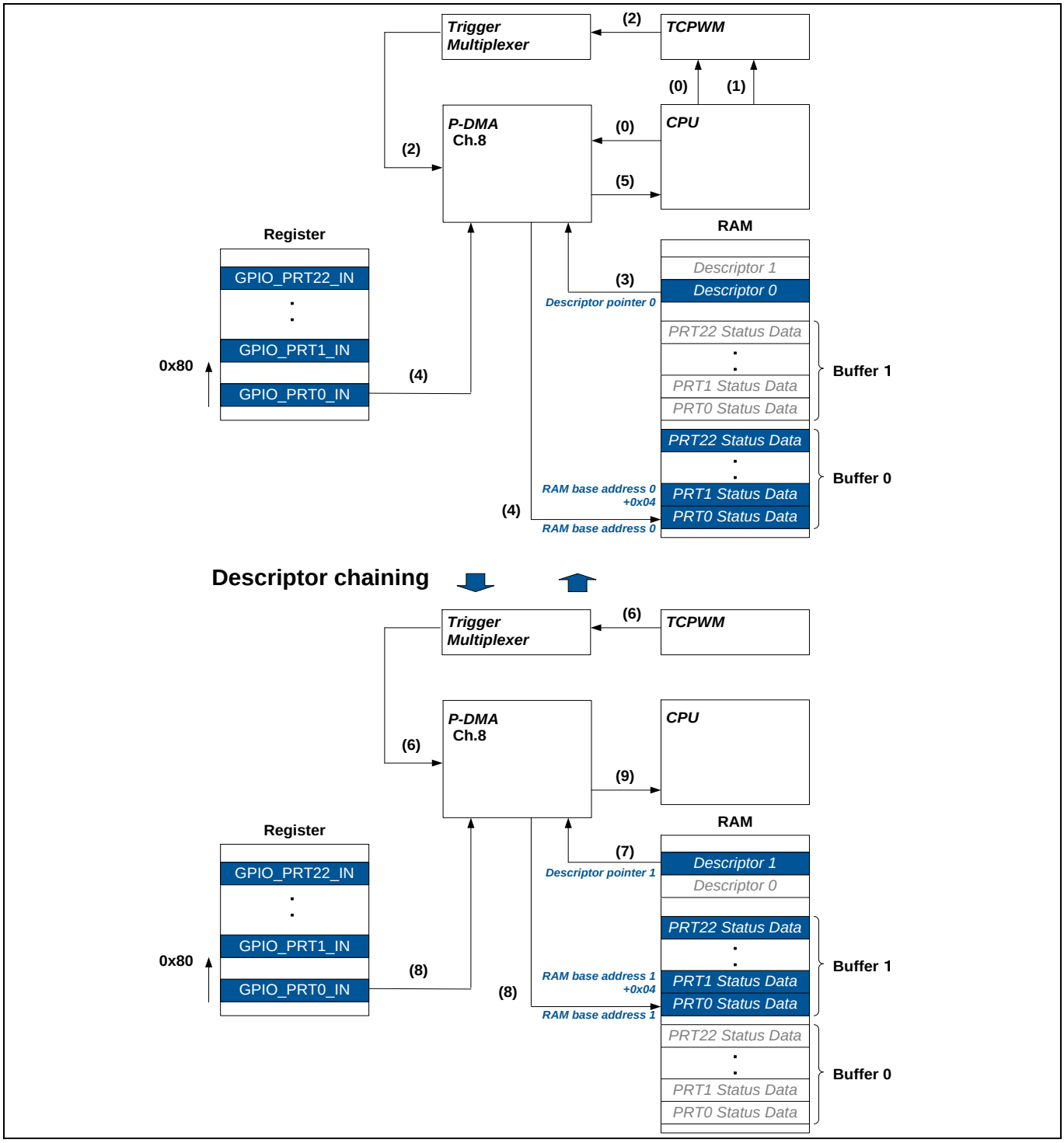


Figure 13 1D transfer with descriptor chaining

(0) Configure P-DMA according to the usage example. In addition, configure the TCPWM and trigger multiplexer.

(1) The CPU starts the TCPWM timer.

(2) The TCPWM outputs the trigger to P-DMA via the trigger multiplexer when it reaches the specified count value (overflow).

(3) P-DMA reads the descriptor from the specified area (Descriptor Pointer 0) when activating the next pending transfer.

(4) P-DMA reads the data from the source address (GPIO_PRT0_IN), and writes the read data to the destination address (RAM base address 0). After that, P-DMA increments the source address by 0x80 and the destination address by 0x04. Then, P-DMA reads the data from the source address (GPIO_PRT1_IN) and writes it to the destination address (RAM base address 0 +0x04) again.

(5) P-DMA notifies the CPU with an interrupt when all transfers are completed.

(6) The TCPWM outputs the trigger to P-DMA when it reaches the specified count value (overflow) again.

(7) P-DMA reads the descriptor from the pointer defined by the next descriptor pointer in its own descriptor (Descriptor Pointer 1) when accepting the transfer.

(8) P-DMA reads the data from the source address (GPIO_PRT0_IN), and writes the read data to the destination address (RAM base address 1). After that, P-DMA increments the source address by 0x80 and the destination address by 0x04. Then, P-DMA reads the data from the source address (GPIO_PRT1_IN) and writes it to the destination address (RAM base address 1 +0x04) again.

(9) P-DMA notifies the CPU through an interrupt when all transfers are completed. Descriptor 1 chains to Descriptor 0. Therefore, when the TCPWM outputs the trigger again, steps from (3) are repeated.

Note: The TCPWM count period must always be longer than the time required by DMA to transfer all data.

3.3.2 Initial configuration

This section describes the initialization of the P-DMA channel and descriptor of this use case.

Figure 14 shows the setting procedure for P-DMA.

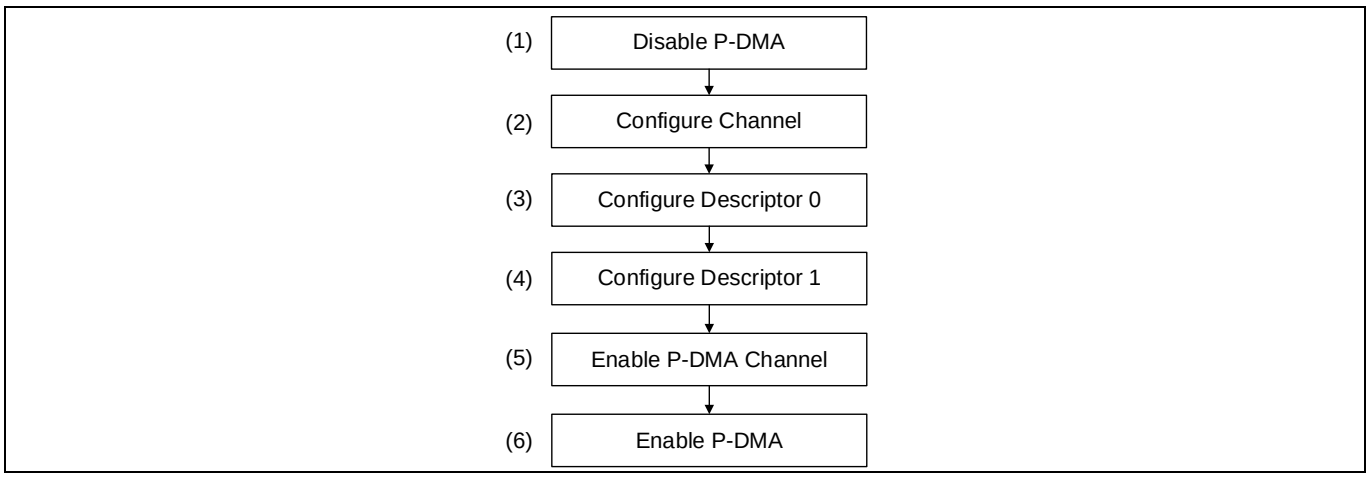


Figure 14 Setting procedure for P-DMA

3.3.3 Configuration and example code

Table 11 lists the parameters and **Table 12** lists the functions of the configuration part in SDL for DMA.

Table 11 List of DMA configuration parameters

Parameters	Description	Value
TCPWMx_GRPx_CNTx_COUNTER	Defines TCPWM using counter	TCPWM0_GRP0_CNT0
DW_CHANNEL	Defines P-DMA channel	8ul
TCPWM_TO_MUX_TRIG	Defines trigger TCPWM to Multiplexer	TRIG_IN_MUX_3_TCPWM_16_TR_OUT00
MUX_TO_PDMA_TRIG	Defines trigger multiplexer to P-DMA	TRIG_OUT_MUX_3_PDMA0_TR_IN8
DW_CH_INTR	Defines P-DMA channel interrupt	cpuss_interrupts_dw0_8_IRQn
DST_BUFFER_SIZE	Defines Destination buffer size	23ul
DST_BUFFER_NUMBER	Defines Destination buffer number	2ul
.PDMA_Descriptor	P-DMA current descriptor pointer	dwTcpwmDescr
.preemptable	Channel preemptable	0ul
.priority	Channel priority	0ul
.enable	Channel enable	1ul
.deact	DESCR_CTL WAIT_FOR_DEACT	0ul
.intrType	DESCR_CTL INTR_TYPE	CY_PDMA_INTR_X_LOOP_CMPLT
.trigoutType	DESCR_CTL TR_OUT_TYPE	CY_PDMA_TRIGOUT_X_LOOP_CMPLT
.chStateAtCmplt	DESCR_CTL CH_DISABLE	CY_PDMA_CH_ENABLED
.triginType	DESCR_CTL TR_IN_TYPE	CY_PDMA_TRIGIN_DESCR
.dataSize	DESCR_CTL DATA_SIZE	CY_PDMA_WORD
.srcTxfrSize	DESCR_CTL SRC_TRANSFER_SIZE	0ul
.destTxfrSize	DESCR_CTL DST_TRANSFER_SIZE	0ul
.descrType	DESCR_CTL DESCR_TYPE	CY_PDMA_1D_TRANSFER
.srcAddr	DESCR_SRC	(void *)&GPIO_PRT0->unIN.u32Register
.destAddr	DESCR_DST	(void *)g_DestBuffer0 .srcXincr
.srcXincr	DESCR_X_CTL SRC_X_INCR	32ul
.destXincr	DESCR_X_CTL DST_X_INCR	1ul
.xCount	DESCR_X_CTL X_COUNT	DST_BUFFER_SIZE
.descrNext	DESCR_NEXT_PTR	NULL

Table 12 List of DMA configuration functions

Functions	Description	Remarks
DW0_Ch_IntHandler()	Handler for P-DMA0 interrupts	See **Code Listing 28**
Counter_Handler()	Handler for TCPWM counter interrupts	
Cy_SysInt_InitIRQ()	Configures interrupt	-
Cy_SysInt_SetSystemIrqVector()	Configures Interrupt vector	-
NVIC_SetPriority()	NVIC set priority	-
NVIC_EnableIRQ()	NVIC enable interrupt	-

Functions	Description	Remarks
Cy_PDMA_Disable()	Configures P-DMA disable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 32**
Cy_PDMA_Chnl_DeInit()	Resets P-DMA to default values	Clears all the content of registers corresponding to the channel. See **Code Listing 33**
Cy_PDMA_Descr_Init()	Configures P-DMA descriptor initialize	See **Code Listing 34**
Cy_PDMA_Chnl_Init()	Configures P-DMA channel initialize	See **Code Listing 35**
Cy_PDMA_Chnl_Enable()	Configures P-DMA channel enable	See **Code Listing 36**
Cy_PDMA_Chnl_SetInterruptMask()	Configures P-DMA channel interrupt mask	See **Code Listing 37**
Cy_PDMA_Enable()	Configures P-DMA enable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 38**
Cy_TrigMux_Connect()	Trigger Multiplexer Initialization	-
Cy_Tcpwm_TriggerStart()	TCPWM Start	--
Cy_PDMA_Chnl_GetInterruptStatusMasked()	Returns logical and of corresponding INTR and INTR_MASK fields in a single load operation	See **Code Listing 29**
Cy_PDMA_Chnl_ClearInterrupt()	Configures P-DMA channel clears the interrupt status	See **Code Listing 30**
Cy_Tcpwm_Counter_ClearCC0_Intr()	Clear TCPWM CC0 interrupt flag	-

Code Listing 26 demonstrates an example program to Descriptor Chaining. See the **architecture TRM** and **application note** for GPIO, TCPWM, and Clock configuration.

Code Listing 26 Example to descriptor chaining

```
#define TCPWMx_GRPx_CNTx_COUNTER            TCPWM0_GRP0_CNT0

#define DW_CHANNEL                          8ul
#define TCPWM_TO_MUX_TRIG                   TRIG_IN_MUX_3_TCPWM_16_TR_OUT00
#define MUX_TO_PDMA_TRIG                    TRIG_OUT_MUX_3_PDMA0_TR_IN8
#define DW_CH_INTR                          cpuss_interrupts_dw0_8_IRQn
#define DST_BUFFER_SIZE                     23ul
#define DST_BUFFER_NUMBER                   2ul

void DW0_Ch_IntHandler(void);
void Counter_Handler(void);

int main(void)
{
:

    /* Initialize & Enable DMA */
    Cy_PDMA_Disable(DW0);
    Cy_PDMA_Chnl_DeInit(DW0, DW_CHANNEL);

    for(uint32_t i = 0ul; i < DST_BUFFER_NUMBER; i++)
    {
        if((i + 1ul) == DST_BUFFER_NUMBER)
        {
            /* Last descriptor */
            dw0ChTcpwmConfig.chStateAtCmplt = CY_PDMA_CH_ENABLED;
            dw0ChTcpwmConfig.destAddr       = &g_DestBuffer1[0];
            dw0ChTcpwmConfig.descrNext      = &dwTcpwmDescr[i - (DST_BUFFER_NUMBER-1)];
        }
        else
        {
            dw0ChTcpwmConfig.chStateAtCmplt = CY_PDMA_CH_ENABLED;
```

Define TCPWM using counter

Define P-DMA channel, Trigger, P-DMA channel interrupt

Define DST buffer size, DST buffer number

See **Code Listing**

See **Code Listing 31**

(1) Disable P-DMA. See **Code Listing 32**.
(2) Resets P-DMA to default values. See **Code Listing 33**.

(4) Configures P-DMA descriptor 1. See **Code Listing 34**.

(3) Configures P-DMA descriptor 0. See **Code Listing 34**.

Code Listing 26 Example to descriptor chaining

```
            dw0ChTcpwmConfig.destAddr       = &g_DestBuffer0[0];
            dw0ChTcpwmConfig.descrNext      = &dwTcpwmDescr[i + 1ul];
        }
        Cy_PDMA_Descr_Init(&dwTcpwmDescr[i], &dw0ChTcpwmConfig);
    }
    Cy_PDMA_Chnl_Init(DW0, DW_CHANNEL, &dwTcpwmConfig);
    Cy_PDMA_Chnl_Enable(DW0, DW_CHANNEL);
    Cy_PDMA_Chnl_SetInterruptMask(DW0, DW_CHANNEL);
    Cy_PDMA_Enable(DW0);

    /* Trigger Multiplexer Initialization */
    /* TCPWM To DW */
    Cy_TrigMux_Connect(TCPWM_TO_MUX_TRIG, MUX_TO_PDMA_TRIG, CY_TR_MUX_TR_INV_DISABLE, TRIGGER_TYPE_LEVEL, 0ul);

    /* Interrupt Initialization */
    Cy_SysInt_InitIRQ(&stc_sysint_irq_cfg);
    Cy_SysInt_SetSystemIrqVector(stc_sysint_irq_cfg.sysIntSrc, DW0_Ch_IntHandler);
    NVIC_SetPriority(stc_sysint_irq_cfg.intIdx, 0ul);
    NVIC_EnableIRQ(stc_sysint_irq_cfg.intIdx);

    /* TCPWM Start */
    Cy_Tcpwm_TriggerStart(TCPWMx_GRPx_CNTx_COUNTER);

    for(;;);
}
```

(5) Enable P-DMA channel. See **Code Listing 36**.
(6) Enable P-DMA. See **Code Listing 38**.

Configures Interrupt

TCPWM start

Code Listing 27 P-DMA Channel, descriptor configuration

```
/**
 * \var cy_stc_pdma_descr_t dwTcpwmDescr
 * \brief PDMA descriptor
 */
static  cy_stc_pdma_descr_t         dwTcpwmDescr[DST_BUFFER_NUMBER];

/**
 * \var cy_stc_pdma_chnl_config_t dwTcpwmConfig
 * \brief PDMA configuration
 */
const   cy_stc_pdma_chnl_config_t   dwTcpwmConfig =
{
    .PDMA_Descriptor = dwTcpwmDescr,
    .preemptable     = 0ul,
    .priority        = 0ul,
    .enable          = 1ul,
};

/**
 * \var cy_stc_pdma_descr_config_t dw0ChTcpwmConfig
 * \brief PDMA descriptor configuration
 */
static  cy_stc_pdma_descr_config_t  dw0ChTcpwmConfig =
{
    .deact          =   0ul,
    .intrType       =   CY_PDMA_INTR_X_LOOP_CMPLT,
    .trigoutType    =   CY_PDMA_TRIGOUT_X_LOOP_CMPLT,
    .chStateAtCmplt =   CY_PDMA_CH_ENABLED,
    .triginType     =   CY_PDMA_TRIGIN_DESCR,
    .dataSize       =   CY_PDMA_WORD,
    .srcTxfrSize    =   0ul, /* Same as dataSize */
    .destTxfrSize   =   0ul, /* Same as dataSize */
    .descrType      =   CY_PDMA_1D_TRANSFER,
    .srcAddr        =   (void *)&GPIO_PRT0->unIN.u32Register,
    .destAddr       =   (void *)g_DestBuffer0,
    .srcXincr       =   32ul, /* address +80h */
    .destXincr      =   1ul,  /* address +04h */
    .xCount         =   DST_BUFFER_SIZE,
    .descrNext      =   NULL /* will be updated in run time */
};

/**
 * \var cy_stc_sysint_irq_t stc_sysint_irq_cfg
 * \brief IRQ DMA configuration
 */
static const cy_stc_sysint_irq_t stc_sysint_irq_cfg =
{
    .sysIntSrc = DW_CH_INTR,
    .intIdx    = CPUIntIdx2_IRQn,
    .isEnabled = true,
};
```

Configure P-DMA channel

Configure P-DMA descriptor

Configure Interrupt

Code Listing 28 DW0_Ch_IntHandler()

```
void DW0_Ch_IntHandler(void)
{
    uint32_t masked;

    masked = Cy_PDMA_Chnl_GetInterruptStatusMasked(DW0, DW_CHANNEL);
    if ((masked & CY_PDMA_INTRCAUSE_COMPLETION) != 0ul)
    {
        /* Clear Complete DMA interrupt flag */
        Cy_PDMA_Chnl_ClearInterrupt(DW0, DW_CHANNEL);

        /* Read the port status */
        for(uint8_t portcnt = 0; portcnt < DST_BUFFER_SIZE; portcnt++)
        {
            g_portStatus[portcnt] = g_DestBuffer[portcnt];
        }
    }
    else
    {
        CY_ASSERT(false);
    }
}
```

See **Code Listing 29**.

See **Code Listing 30**.

Code Listing 29 Cy_PDMA_Chnl_GetInterruptStatusMasked()

```
__STATIC_INLINE uint32_t Cy_PDMA_Chnl_GetInterruptStatusMasked(const volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    return (pstcPDMA->CH_STRUCT[chNum].unINTR_MASKED.u32Register);
}
```

Code Listing 30 Cy_PDMA_Chnl_ClearInterrupt()

```
void Cy_PDMA_Chnl_ClearInterrupt(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unINTR.u32Register = CY_PDMA_INTR_BIT_MASK;

    /* Readback of the register is required by hardware. */
    (void) pstcPDMA->CH_STRUCT[chNum].unINTR.u32Register;
}
```

Code Listing 31 Counter_Handler()

```
void Counter_Handler(void)
{
    if(Cy_Tcpwm_Counter_GetCC0_IntrMasked(TCPWMx_GRPx_CNTx_COUNTER))
    {
        /* Clear TCPWM CC0 interrupt flag Group#0 Counter#0 */
        Cy_Tcpwm_Counter_ClearCC0_Intr(TCPWMx_GRPx_CNTx_COUNTER);

        /* user program here.. */
    }
}
```

Code Listing 32 Cy_PDMA_Disable()

```
void Cy_PDMA_Disable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       0ul;
}
```

Write to P-DMA_CTL_ENABLED bit

Code Listing 33 Cy_PDMA_Chnl_DeInit()

```
void Cy_PDMA_Chnl_DeInit(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unCH_CTL.u32Register            = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unCH_IDX.u32Register            = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register       = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register         = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unINTR_SET.u32Register          = 0ul;
}
```

Resets PDMA to default value

Code Listing 34 Cy_PDMA_Descr_Init()

```
cy_en_pdma_status_t Cy_PDMA_Descr_Init(cy_stc_pdma_descr_t* descriptor, const cy_stc_pdma_descr_config_t* config)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((descriptor != NULL) && (config != NULL))
    {
        descriptor->unPDMA_DESCR_CTL.stcField.u2WAIT_FOR_DEACT          =       config->deact;
        descriptor->unPDMA_DESCR_CTL.stcField.u2INTR_TYPE               =       config->intrType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_OUT_TYPE             =       config->trigoutType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_IN_TYPE              =       config->triginType;
        descriptor->unPDMA_DESCR_CTL.stcField.u1SRC_TRANSFER_SIZE       =       config->srcTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1DST_TRANSFER_SIZE       =       config->destTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1CH_DISABLE              =       config->chStateAtCmplt;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DATA_SIZE               =       config->dataSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DESCR_TYPE              =       config->descrType;

        descriptor->u32PDMA_DESCR_SRC                                   =       (uint32_t)config->srcAddr;
        descriptor->u32PDMA_DESCR_DST                                   =       (uint32_t)config->destAddr;

        switch(config->descrType)
        {
        case (uint32_t)CY_PDMA_SINGLE_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.u32Register               =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_1D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register               =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_CRC_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       0ul;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register               =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_2D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);

            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12SRC_Y_INCR   =       (uint32_t)config->srcYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12DST_Y_INCR   =       (uint32_t)config->destYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u8Y_COUNT       =       (uint32_t)((config->yCount) - 1ul);

            descriptor->u32PDMA_DESCR_NEXT_PTR                      =       (uint32_t)config->descrNext;
            break;
        }
        default:
        {
            /* Unsupported type of descriptor */
            break;
        }
        }

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return retVal;
}
```

Code Listing 35 Cy_PDMA_Chnl_Init()

```
cy_en_pdma_status_t Cy_PDMA_Chnl_Init(volatile stc_DW_t *pstcPDMA, uint32_t chNum, const cy_stc_pdma_chnl_config_t*
chnlConfig)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((pstcPDMA != NULL) && (chnlConfig != NULL))
    {
        /* Set current descriptor */
        pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register            =  (uint32_t)chnlConfig->PDMA_Descriptor;
```

Code Listing 35 Cy_PDMA_Chnl_Init()

```
        /* Set if the channel is preemtable */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1PREEMPTABLE      =  chnlConfig->preemptable;

        /* Set channel priority */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u2PRIO             =  chnlConfig->priority;

        /* Set enabled status */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED          =  chnlConfig->enable;

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return (retVal);
}
```

Code Listing 36 Cy_PDMA_Chnl_Enable()

```
__STATIC_INLINE void Cy_PDMA_Chnl_Enable(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED = 1ul;
}
```

Code Listing 37 Cy_PDMA_Chnl_SetInterruptMask()

```
__STATIC_INLINE void Cy_PDMA_Chnl_SetInterruptMask(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register = CY_PDMA_INTR_BIT_MASK;
}
```

Code Listing 38 Cy_PDMA_Enable()

```
void Cy_PDMA_Enable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       1ul;
}
```

Write to P-DMA_CTL_ENABLED bit

3.4 2D transfer (peripheral-to-memory)

3.4.1 Overview

This example shows how the result of an ADC group conversion is stored to the memory. See the “SAR ADC” chapter of the **architecture TRM** for more details.

The conversion result is grouped for each channel and stored in the memory. In this example, the ADC converts three channels CH_0, CH_1, and CH_2 by group conversion at a specified period. P-DMA stores the three group conversion results in the memory for each channel with the ADC conversion completion trigger.

Figure 15 shows how 2D transfer works.

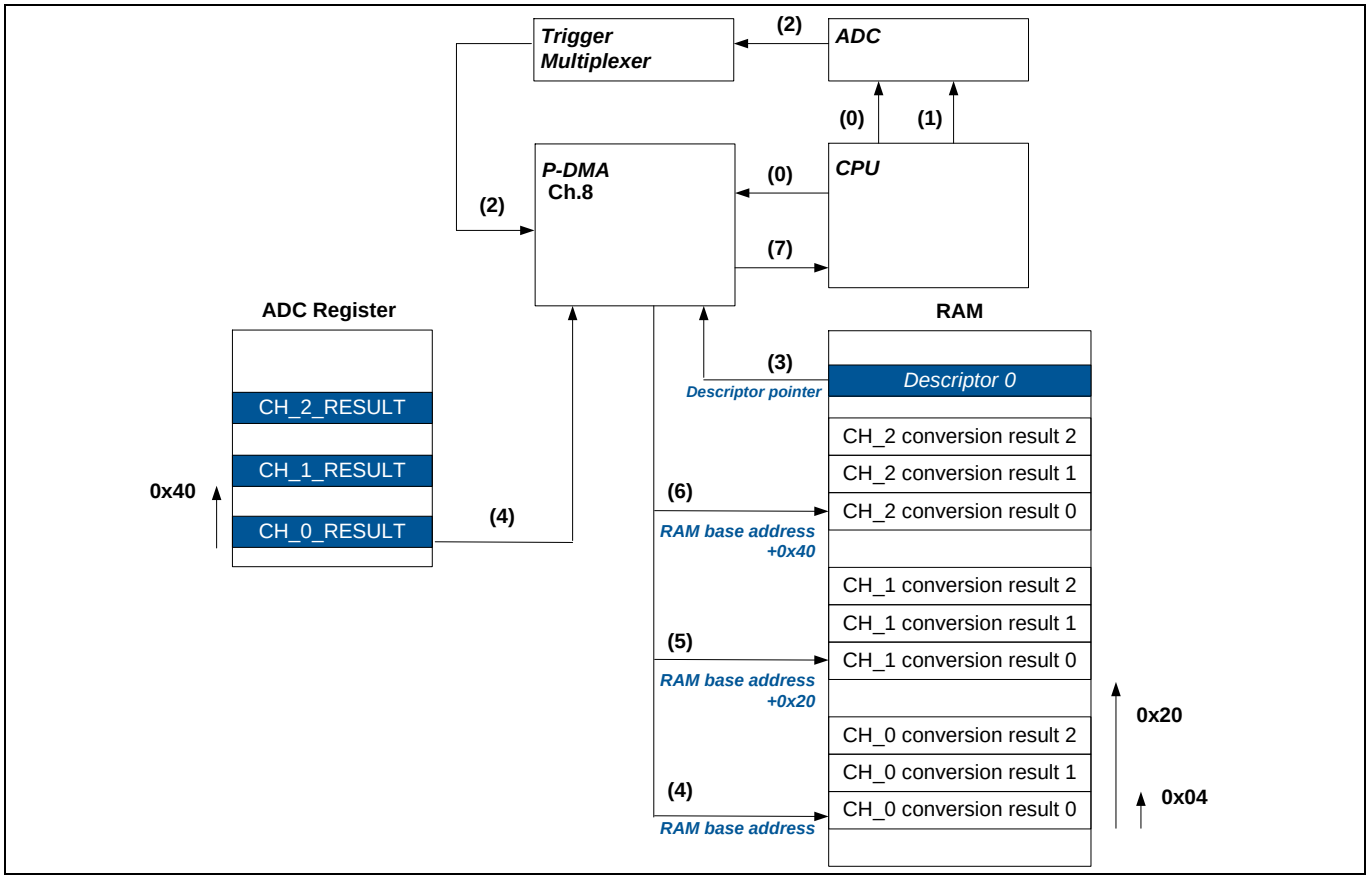


Figure 15 2D transfer (peripheral-to-memory)

(0) Configure P-DMA according to the usage example. In addition, configure the ADC and trigger multiplexer.

(1) The CPU starts ADC group conversion.

(2) The ADC outputs the trigger to P-DMA via the trigger multiplexer when group conversion is completed.

(3) P-DMA reads the descriptor from the specified area (Descriptor Pointer) when accepting the transfer.

(4) P-DMA reads the data from the source address (CH_0_RESULT), and writes the read data to the destination address (RAM base address).

(5) P-DMA increments the source address by 0x40 and the destination address by 0x20. Then, P-DMA reads the data from the source address (CH_1_RESULT) and writes it to the destination address (RAM base address 0 +0x20).

(6) P-DMA increments the source address by 0x40 and the destination address by 0x20. Then, P-DMA reads the data from the source address (CH_2_RESULT) and writes it to the destination address (RAM base address 0 +0x40). When the next trigger occurs, P-DMA increments the RAM base address by 0x04 and repeats from (4).

(7) P-DMA notifies the CPU with an interrupt after transferring CH_ 2 conversion result 2.

In this case, three triggers are required for completing the descriptor.

3.4.2 Initial configuration

This section describes the initialization of the P-DMA channel and descriptor of this use case.

Figure 16 shows the setting procedure for P-DMA.

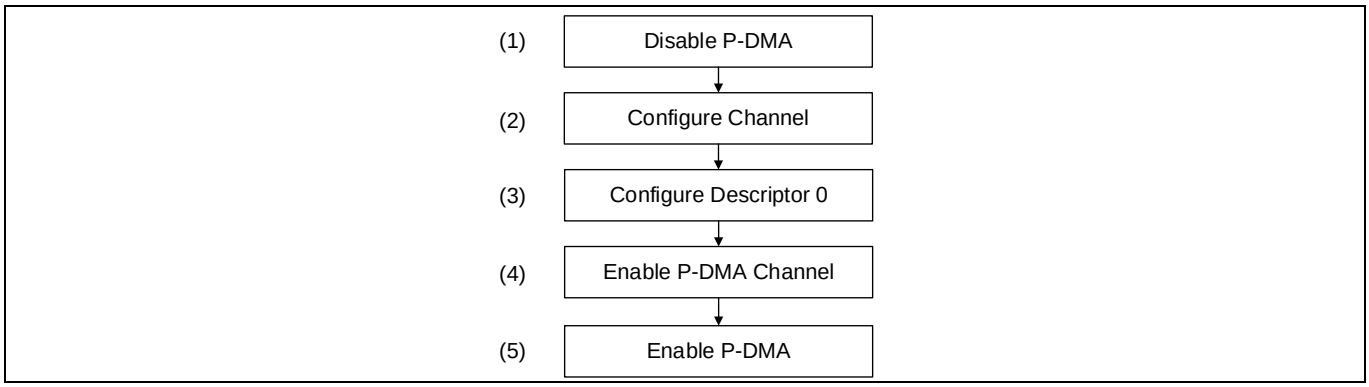


Figure 16 Setting procedure for P-DMA

3.4.3 Configuration and example code

Table 13 lists the parameters and **Table 14** lists the functions of the configuration part in SDL for DMA.

Table 13 List of DMA configuration parameters

Parameters	Description	Value
ADC_MACRO	Defines ADC macro	PASS0_SAR0
ADC_GROUP_NUMBER_OF_CHANNELS	Defines ADC group number	3ul
ADC_LOGICAL_CHANNEL	Defines ADC logical channel	0ul
ADC_GROUP_FIRST_CHANNEL	Defines ADC group first channel	ADC_LOGICAL_CHANNEL
ADC_GROUP_LAST_CHANNEL	Defines ADC group last channel	(ADC_GROUP_FIRST_CHANNEL + ADC_GROUP_NUMBER_OF_CHANNELS - 1ul)
ADC_CH_INTR_BASE	Defines ADC channel interrupt	pass_0_interrupts_sar_0_IRQn
BUFFER_SIZE_IN_WORD	Defines buffer size	25ul
BUFFER_SIZE_IN_BYTE	Defines buffer size	(BUFFER_SIZE_IN_WORD * 4ul)
DW_CHANNEL	Defines P-DMA channel	27ul
DW_CH_INTR	Defines P-DMA channel interrupt	cpuss_interrupts_dw0_27_IRQn
ADC_TO_DMA_TRIG	Defines trigger ADC to DMA	TRIG_OUT_1TO1_2_PASS_CH_DONE_TO_PDMA02
.PDMA_Descriptor	P-DMA current descriptor pointer	&
.preemptable	Channel preemptable	0ul
.priority	Channel priority	0ul
.enable	Channel enable	1ul
.deact	DESCR_CTL WAIT_FOR_DEACT	0ul
.intrType	DESCR_CTL INTR_TYPE	CY_PDMA_INTR_DESCR_CMPLT

P-DMA use cases

Parameters	Description	Value
.trigoutType	DESCR_CTL TR_OUT_TYPE	CY_PDMA_TRIGOUT_DESCR_CMPLT
.chStateAtCmplt	DESCR_CTL CH_DISABLE	CY_PDMA_CH_ENABLED
.triginType	DESCR_CTL TR_IN_TYPE	CY_PDMA_TRIGIN_XLOOP
.dataSize	DESCR_CTL DATA_SIZE	CY_PDMA_WORD
.srcTxfrSize	DESCR_CTL SRC_TRANSFER_SIZE	0ul, /* as specified in DATA_SIZE */
.destTxfrSize	DESCR_CTL DST_TRANSFER_SIZE	0ul, /* as specified in DATA_SIZE */
.descrType	DESCR_CTL DESCR_TYPE	CY_PDMA_2D_TRANSFER
.srcAddr	DESCR_SRC	(void *)&ADC_MACRO->CH[ADC_GROUP_FIRST_CHANNEL].unRESULT.u32Register
.destAddr	DESCR_DST	(void *)g_DestBuffer
.srcXincr	DESCR_X_CTL SRC_X_INCR	16ul
.destXincr	DESCR_X_CTL DST_X_INCR	8ul
.xCount	DESCR_X_CTL X_COUNT	ADC_GROUP_NUMBER_OF_CHANNELS
.srcYincr	DESCR_Y_CTL SRC_Y_INCR	0ul
.destYincr	DESCR_Y_CTL DST_Y_INCR	1ul
.yCount	DESCR_Y_CTL Y_COUNT	3ul
.descrNext	DESCR_NEXT_PTR	&stcDescr

Table 14 List of DMA configuration functions

Functions	Description	Remarks
DW0_Ch_IntHandler()	Handler for P-DMA0 interrupts	See **Code Listing 41**
Cy_SysInt_InitIRQ()	Configures interrupt	-
Cy_SysInt_SetSystemIrqVector()	Configures interrupt vector	-
NVIC_SetPriority()	NVIC set priority	-
NVIC_EnableIRQ()	NVIC enable interrupt	-
Cy_PDMA_Disable()	Configures P-DMA disable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 44**
Cy_PDMA_Chnl_DeInit()	Resets P-DMA to default values	See **Code Listing 45**
Cy_PDMA_Descr_Init()	Configures P-DMA descriptor initialize	See **Code Listing 46**
Cy_PDMA_Chnl_Init()	Configures P-DMA channel initialize	See **Code Listing 47**
Cy_PDMA_Chnl_Enable()	Configures P-DMA channel enable	See **Code Listing 48**
Cy_PDMA_Chnl_SetInterruptMask()	Configures P-DMA channel interrupt mask	See **Code Listing 49**
Cy_PDMA_Enable()	Configures P-DMA enable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 50**
Cy_TrigMux_Connect1To1()	Trigger multiplexer initialization	-

Functions	Description	Remarks
Cy_PDMA_Chnl_GetInterruptStatusMasked()	Returns logical and of corresponding INTR and INTR_MASK fields in a single load operation	See **Code Listing 42**
Cy_PDMA_Chnl_ClearInterrupt()	Configures P-DMA channel clears the interrupt status	See **Code Listing 43**
Cy_Adc_Channel_SoftwareTrigger()	ADC software start trigger	-

Code Listing 39 demonstrates an example program to 2D transfer (peripheral-to-memory). See the **architecture TRM** and **application note** for GPIO, ADC and Clock configuration.

Code Listing 39 Example to 2D transfer (Peripheral-to-Memory)

```
#define ADC_MACRO                       PASS0_SAR0

#define ADC_GROUP_NUMBER_OF_CHANNELS (3ul)


/* ADC logical channel to be used */
#define ADC_LOGICAL_CHANNEL             0ul
#define ADC_GROUP_FIRST_CHANNEL         ADC_LOGICAL_CHANNEL
#define ADC_GROUP_LAST_CHANNEL          (ADC_GROUP_FIRST_CHANNEL + ADC_GROUP_NUMBER_OF_CHANNELS - 1ul)
#define ADC_CH_INTR_BASE                pass_0_interrupts_sar_0_IRQn

#define BUFFER_SIZE_IN_WORD             25ul
#define BUFFER_SIZE_IN_BYTE             (BUFFER_SIZE_IN_WORD * 4ul)
#define DW_CHANNEL                      27ul
#define DW_CH_INTR                      cpuss_interrupts_dw0_27_IRQn
#define ADC_TO_DMA_TRIG                 TRIG_OUT_1TO1_2_PASS_CH_DONE_TO_PDMA02

void DW0_Ch_IntHandler(void);


int main(void)
{
:
    /* Initialize & Enable DW */
    Cy_PDMA_Disable(DW0);
    Cy_PDMA_Chnl_DeInit(DW0, DW_CHANNEL);
    Cy_PDMA_Descr_Init(&stcDescr,&stcDmaDescrConfig);
    Cy_PDMA_Chnl_Init(DW0, DW_CHANNEL, &chnlConfig);
    Cy_PDMA_Chnl_Enable(DW0, DW_CHANNEL);
    Cy_PDMA_Chnl_SetInterruptMask(DW0, DW_CHANNEL);
    Cy_PDMA_Enable(DW0);

    /* Trigger MUX */
    Cy_TrigMux_Connect1To1( ADC_TO_DMA_TRIG,
                            CY_TR_MUX_TR_INV_DISABLE,
                            TRIGGER_TYPE_EDGE,
                            0ul);

    /* Interrupt Initialization */
    Cy_SysInt_InitIRQ(&stc_sysint_irq_cfg);
    Cy_SysInt_SetSystemIrqVector(stc_sysint_irq_cfg.sysIntSrc, DW0_Ch_IntHandler);
    NVIC_SetPriority(stc_sysint_irq_cfg.intIdx, 0ul);
    NVIC_EnableIRQ(stc_sysint_irq_cfg.intIdx);

    /* Issue SW trigger */
    Cy_Adc_Channel_SoftwareTrigger(&ADC_MACRO->CH[ADC_GROUP_FIRST_CHANNEL]);

    for(;;);
}
```

Define ADC macro, ADC group number

Define ADC logical channel, ADC group first channel, ADC group last channel, ADC channel interrupt

Define buffer size, P-DMA channel, P-DMA channel interrupt, Trigger

See **Code Listing 41**.

(1) Disable P-DMA. See **Code Listing 44**.
(2) Resets P-DMA to default values. See **Code Listing 45**.
(3) Configures P-DMA descriptor initialize. See **Code Listing 46**.
(4) Enable P-DMA channel. See **Code Listing 47**.
(5) Enable P-DMA. See **Code Listing 50**.

Configures interrupt

Generates a software trigger

Code Listing 40 P-DMA Channel, Descriptor configuration

```
/**
 * \var uint32_t g_DestBuffer
 * \brief DMA Destination buffer
 */
static uint32_t g_DestBuffer[BUFFER_SIZE_IN_BYTE] = {0ul};

/**
```

Code Listing 40 P-DMA Channel, Descriptor configuration

```
 * \var cy_stc_pdma_descr_t stcDescr
 * \brief PDMA descriptor
 */
static cy_stc_pdma_descr_t stcDescr;

/**
 * \var cy_stc_pdma_chnl_config_t chnlConfig
 * \brief PDMA configuration
 */
static const cy_stc_pdma_chnl_config_t   chnlConfig =
{
    .PDMA_Descriptor = &stcDescr,
    .preemptable     = 0ul,
    .priority        = 0ul,
    .enable          = 1ul,
};

/**
 * \var cy_stc_pdma_descr_config_t stcDmaDescrConfig
 * \brief PDMA descriptor configuration
 */
static const cy_stc_pdma_descr_config_t  stcDmaDescrConfig =
{
    .deact          = 0ul,
    .intrType       = CY_PDMA_INTR_DESCR_CMPLT,
    .trigoutType    = CY_PDMA_TRIGOUT_DESCR_CMPLT,
    .chStateAtCmplt = CY_PDMA_CH_ENABLED,
    .triginType     = CY_PDMA_TRIGIN_XLOOP,
    .dataSize       = CY_PDMA_WORD,
    .srcTxfrSize    = 0ul,  /* as specified in DATA_SIZE */
    .destTxfrSize   = 0ul,  /* as specified in DATA_SIZE */
    .descrType      = CY_PDMA_2D_TRANSFER,
    .srcAddr        = (void *)&ADC_MACRO->CH[ADC_GROUP_FIRST_CHANNEL].unRESULT.u32Register,
    .destAddr       = (void *)g_DestBuffer,
    .srcXincr       = 16ul,  /* address +40h */
    .destXincr      = 8ul,   /* address +20h */
    .xCount         = ADC_GROUP_NUMBER_OF_CHANNELS,
    .srcYincr       = 0ul,   /* Not increments */
    .destYincr      = 1ul,   /* Destination address +4h */
    .yCount         = 3ul,   /* Store results for three times */
    .descrNext      = &stcDescr
};

/**
 * \var cy_stc_sysint_irq_t stc_sysint_irq_cfg
 * \brief IRQ DMA configuration
 */
static const cy_stc_sysint_irq_t stc_sysint_irq_cfg =
{
    .sysIntSrc = DW_CH_INTR,
    .intIdx    = CPUIntIdx2_IRQn,
    .isEnabled = true,
};
```

Configure P-DMA channel

Configure P-DMA descriptor

Configure interrupt

Code Listing 41 DW0_Ch_IntHandler()

```
void DW0_Ch_IntHandler(void)
{
    uint32_t masked;

    masked = Cy_PDMA_Chnl_GetInterruptStatusMasked(DW0, DW_CHANNEL);
    if ((masked & CY_PDMA_INTRCAUSE_COMPLETION) != 0ul)
    {
        /* Clear Complete DMA interrupt flag */
        Cy_PDMA_Chnl_ClearInterrupt(DW0, DW_CHANNEL);

        /* Read the port status */
        for(uint8_t portcnt = 0; portcnt < DST_BUFFER_SIZE; portcnt++)
        {
            g_portStatus[portcnt] = g_DestBuffer[portcnt];
        }
    }
    else
    {
        CY_ASSERT(false);
    }
}
```

See **Code Listing 42**.

See **Code Listing 43**.

Code Listing 42 Cy_PDMA_Chnl_GetInterruptStatusMasked()

```
__STATIC_INLINE uint32_t Cy_PDMA_Chnl_GetInterruptStatusMasked(const volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    return (pstcPDMA->CH_STRUCT[chNum].unINTR_MASKED.u32Register);
}
```

Code Listing 43 Cy_PDMA_Chnl_ClearInterrupt()

```
void Cy_PDMA_Chnl_ClearInterrupt(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unINTR.u32Register = CY_PDMA_INTR_BIT_MASK;

    /* Readback of the register is required by hardware. */
    (void) pstcPDMA->CH_STRUCT[chNum].unINTR.u32Register;
}
```

Code Listing 44 Cy_PDMA_Disable()

```
void Cy_PDMA_Disable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       0ul;
}
```

Write to P-DMA_CTL_ENABLED bit

Code Listing 45 Cy_PDMA_Chnl_DeInit()

```
void Cy_PDMA_Chnl_DeInit(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unCH_CTL.u32Register            = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unCH_IDX.u32Register            = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register       = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register         = 0ul;
    pstcPDMA->CH_STRUCT[chNum].unINTR_SET.u32Register          = 0ul;
}
```

Resets P-DMA to default value

Code Listing 46 Cy_PDMA_Descr_Init()

```
cy_en_pdma_status_t Cy_PDMA_Descr_Init(cy_stc_pdma_descr_t* descriptor, const cy_stc_pdma_descr_config_t* config)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((descriptor != NULL) && (config != NULL))
    {
        descriptor->unPDMA_DESCR_CTL.stcField.u2WAIT_FOR_DEACT          =       config->deact;
        descriptor->unPDMA_DESCR_CTL.stcField.u2INTR_TYPE               =       config->intrType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_OUT_TYPE             =       config->trigoutType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_IN_TYPE              =       config->triginType;
        descriptor->unPDMA_DESCR_CTL.stcField.u1SRC_TRANSFER_SIZE       =       config->srcTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1DST_TRANSFER_SIZE       =       config->destTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1CH_DISABLE              =       config->chStateAtCmplt;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DATA_SIZE               =       config->dataSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DESCR_TYPE              =       config->descrType;

        descriptor->u32PDMA_DESCR_SRC                                   =       (uint32_t)config->srcAddr;
        descriptor->u32PDMA_DESCR_DST                                   =       (uint32_t)config->destAddr;

        switch(config->descrType)
        {
        case (uint32_t)CY_PDMA_SINGLE_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_1D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_CRC_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       0ul;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register              =       (uint32_t)config->descrNext;
```

Code Listing 46 Cy_PDMA_Descr_Init()

```
            break;
        }
        case (uint32_t)CY_PDMA_2D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);

            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12SRC_Y_INCR   =       (uint32_t)config->srcYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12DST_Y_INCR   =       (uint32_t)config->destYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u8Y_COUNT       =       (uint32_t)((config->yCount) - 1ul);

            descriptor->u32PDMA_DESCR_NEXT_PTR                      =       (uint32_t)config->descrNext;
            break;
        }
        default:
        {
            /* Unsupported type of descriptor */
            break;
        }
        }

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return retVal;
}
```

Code Listing 47 Cy_PDMA_Chnl_Init()

```
cy_en_pdma_status_t Cy_PDMA_Chnl_Init(volatile stc_DW_t *pstcPDMA, uint32_t chNum, const cy_stc_pdma_chnl_config_t*
chnlConfig)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((pstcPDMA != NULL) && (chnlConfig != NULL))
    {
        /* Set current descriptor */
        pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register            =  (uint32_t)chnlConfig->PDMA_Descriptor;

        /* Set if the channel is preemtable */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1PREEMPTABLE      =  chnlConfig->preemptable;

        /* Set channel priority */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u2PRIO             =  chnlConfig->priority;

        /* Set enabled status */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED          =  chnlConfig->enable;

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return (retVal);
}
```

Code Listing 48 Cy_PDMA_Chnl_Enable()

```
__STATIC_INLINE void Cy_PDMA_Chnl_Enable(volatile stc_DW_t *pstcPDMA, uint32_t chNum)

    pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED = 1ul;
}
```

Code Listing 49 Cy_PDMA_Chnl_SetInterruptMask()

```
__STATIC_INLINE void Cy_PDMA_Chnl_SetInterruptMask(volatile stc_DW_t *pstcPDMA, uint32_t chNum)
{
    pstcPDMA->CH_STRUCT[chNum].unINTR_MASK.u32Register = CY_PDMA_INTR_BIT_MASK;
}
```

Code Listing 50 Cy_PDMA_Enable()

```
void Cy_PDMA_Enable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       1ul;
}
```

Write to P-DMA_CTL_ENABLED bit

3.5 CRC transfer

3.5.1 Overview

This section describes an example of CRC transfer. CRC transfer is a P-DMA-specific descriptor type. CRC transfer calculates the CRC in the area that is specified by the source address and size, and transfers the result to the destination address.

This is an example of program code validation in the flash with CRC transfer. The CPU calculates the CRC of the program code area with P-DMA before program execution. CRC calculation is performed by using CRC32. When the result of CRC calculation matches the expected value, the CPU starts the execution of the program. In this example, note that it is necessary to prepare for the expected value of the program code.

See the **architecture TRM** for details of CRC parameters that can be set by P-DMA.

Figure 17 shows a use case of a CRC transfer.

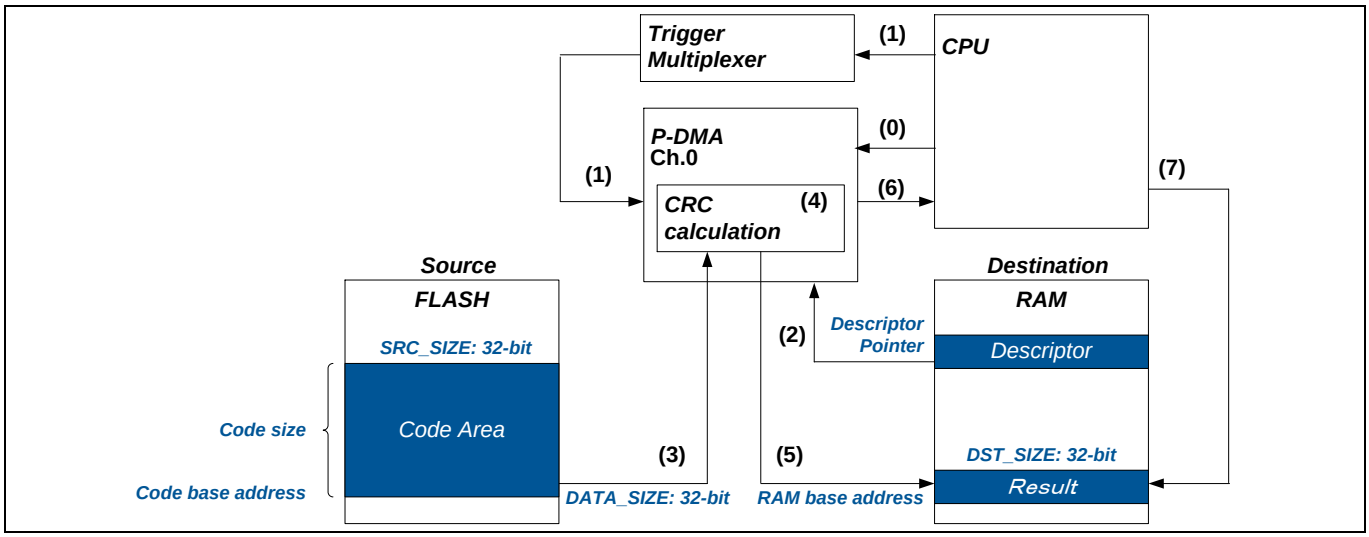


Figure 17 Use case of CRC transfer

(0) Configure P-DMA according to the usage example.

(1) CPU notifies a software trigger to P-DMA via the trigger multiplexer.

(2) P-DMA reads the descriptor from the specified area (Descriptor Pointer) when accepting the transfer.

(3) P-DMA reads the data from the source address (code base address).

(4) P-DMA inputs the read data to the CRC calculator, and reads the data again after incrementing the address. P-DMA repeats (4) until it reaches the area specified by the transfer size (code size).

(5) When CRC calculation of the specified area is completed, the result is transferred to the destination address (RAM base address).

(6) P-DMA notifies an interrupt to the CPU.

When the CPU accepts an interrupt, it compares the result with the expected value. When it matches, CPU starts program execution. If it does not match, CPU transfers to safe operation mode.

3.5.2 Initial configuration

This section describes the initialization of the P-DMA channel and descriptor of this use case.

Figure 18 shows the setting procedure for P-DMA.

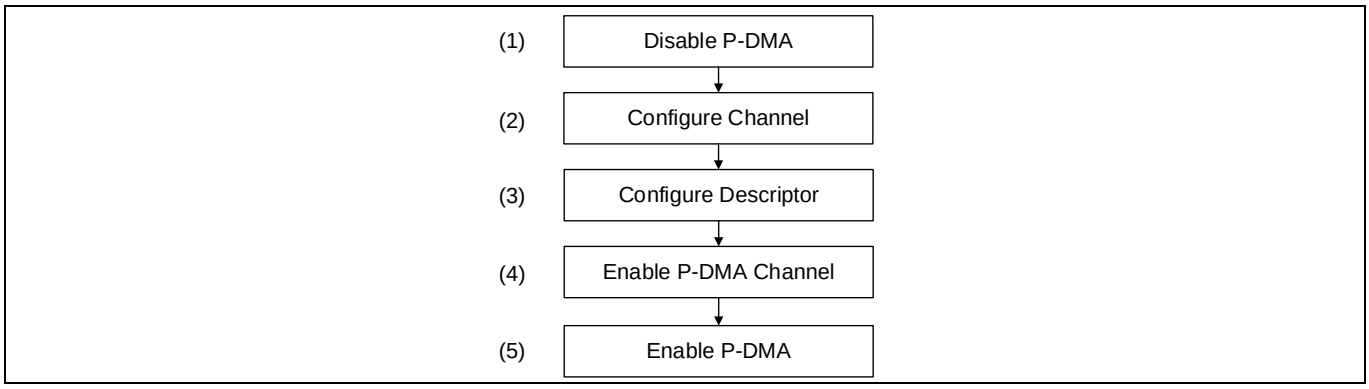


Figure 18 Setting procedure for P-DMA

3.5.3 Configuration and example code

Table 15 lists the parameters and **Table 16** lists the functions of the configuration part in SDL for DMA.

Table 15 List of DMA configuration parameters

Parameters	Description	Value
BUFFER_SIZE	Defines buffer size	5
DW_CHANNEL	Defines DW channel	0
EXPECTED_RESULT_VALUE	Defines result value	(0x3C4687AFUL)
.data_reverse	Least significant bit (bit 0) first	1
.rem_reverse	Remainder is bit reversed	1
.data_xor	Each data byte is XORed with 00h	0
.polynomial	CRC32: POLYNOMIAL	0x04c11db7
.lfsr32	Seed value	0xFFFFFFFF
.rem_xor	CRC_LFSR_CTL.LFSR32 register is XORed with FFFFh	0xFFFFFFFF
.PDMA_Descriptor	P-DMA current descriptor pointer	&stcDescr
.preemptable	Channel preemptable	0
.priority	Channel priority	0
.enable	Channel enable	1
.deact	DESCR_CTL WAIT_FOR_DEACT	0
.intrType	DESCR_CTL INTR_TYPE	CY_PDMA_INTR_1ELEMENT_CMPLT

P-DMA use cases

Parameters	Description	Value
.trigoutType	DESCR_CTL TR_OUT_TYPE	CY_PDMA_TRIGOUT_1ELEMENT_CMPLT
.chStateAtCmplt	DESCR_CTL CH_DISABLE	CY_PDMA_CH_DISABLED
.triginType	DESCR_CTL TR_IN_TYPE	CY_PDMA_TRIGIN_DESCR
.dataSize	DESCR_CTL DATA_SIZE	CY_PDMA_BYTE
.srcTxfrSize	DESCR_CTL SRC_TRANSFER_SIZE	0
.destTxfrSize	DESCR_CTL DST_TRANSFER_SIZE	0
.descrType	DESCR_CTL DESCR_TYPE	CY_PDMA_CRC_TRANSFER
.srcAddr	DESCR_SRC	(void*) au8SrcBuffer
.destAddr	DESCR_DST	0
.srcXincr	DESCR_X_CTL SRC_X_INCR	1
.destXincr	DESCR_X_CTL DST_X_INCR	1
.xCount	DESCR_X_CTL X_COUNT	BUFFER_SIZE
.srcYincr	DESCR_Y_CTL SRC_Y_INCR	0
.destYincr	DESCR_Y_CTL DST_Y_INCR	0
.yCount	DESCR_Y_CTL Y_COUNT	0

Table 16 List of DMA configuration functions

Functions	Description	Remarks
Cy_GPIO_Pin_Init()	Configures GPIO pin	-
Cy_PDMA_Disable()	Configures P-DMA disable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 53**
Cy_PDMA_CRC_Config()	Configures P-DMA CRC	See **Code Listing 54**
Cy_PDMA_Descr_Init()	Configures P-DMA descriptor initialize	See **Code Listing 55**
Cy_PDMA_Chnl_Init()	Configures P-DMA channel initialize	See **Code Listing 56**
Cy_PDMA_Enable()	Configures P-DMA enable	Write to P-DMA_CTL_ENABLED bit. See **Code Listing 57**
Cy_TrigMux_SwTrigger()	Generates a Software trigger	-

Code Listing 51 demonstrates an example program to 2D transfer (peripheral-to-memory). See the **architecture TRM** and **application note** for GPIO and Clock configuration.

Code Listing 51 Example to CRC transfer

```
#define BUFFER_SIZE          5
#define DW_CHANNEL           0

#define EXPECTED_RESULT_VALUE (0x3C4687AFUL)

int main(void)
{
:
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    Cy_GPIO_Pin_Init(CY_LED8_PORT, CY_LED8_PIN, &user_led8_port_pin_cfg);

    Cy_PDMA_Disable(pstcDW);

    Cy_PDMA_CRC_Config( pstcDW,
                        &stcCrcConfig);
```

Define buffer size
Define P-DMA channel
Define result value

Code Listing 51 Example to CRC transfer

```
    stcDmaDescrConfig.destAddr = (void *)&pstcDW->unCRC_LFSR_CTL.u32Register;
    Cy_PDMA_Descr_Init(&stcDescr,&stcDmaDescrConfig);
    Cy_PDMA_Chnl_Init(  pstcDW,
                        DW_CHANNEL,
                        (const cy_stc_pdma_chnl_config_t*) &chnlConfig);


    Cy_PDMA_Enable(pstcDW);

    /*Trigger DMA by SW*/
    Cy_TrigMux_SwTrigger(TRIG_OUT_MUX_0_PDMA0_TR_IN0,
                         TRIGGER_TYPE_CPUSS_DW0_TR_IN__EDGE,
                         1);                            /*output*/

    //Wait for completion
    while(pstcDW->CH_STRUCT[DW_CHANNEL].unCH_CTL.stcField.u1ENABLED)
    {
        u32Time++;
    }

    u32CrcResult                            =   Cy_PDMA_GetCrcRemainderResult(pstcDW);

    CY_ASSERT(u32CrcResult == EXPECTED_RESULT_VALUE);

}
```

(1) Disable P-DMA. See **Code Listing 53**.
(2) Configures P-DMA CRC. See **Code Listing 54**.
(3) Configures P-DMA descriptor initialize. See **Code Listing 55**.
(4) Enable P-DMA channel. See **Code Listing 56**.
(5) Enable P-DMA. See **Code Listing 57**

Code Listing 52 P-DMA Channel, Descriptor configuration

```
static  volatile stc_DW_t*          pstcDW          =   DW0;
static  cy_stc_pdma_descr_t         stcDescr;
const   uint8_t                     au8SrcBuffer[]   =   {0x12,0x34,0x56,0x78,0x9a};

const   cy_stc_pdma_crc_config_t    stcCrcConfig    =   {
                                                        .data_reverse = 1,
                                                        .rem_reverse  = 1,
                                                        .data_xor     = 0,
                                                        .polynomial   = 0x04c11db7,
                                                        .lfsr32       = 0xFFFFFFFF,
                                                        .rem_xor      = 0xFFFFFFFF,
                                                        };
const   cy_stc_pdma_chnl_config_t   chnlConfig      =   {
                                                        .PDMA_Descriptor=   &stcDescr,
                                                        .preemptable    =   0,
                                                        .priority       =   0,
                                                        .enable         =   1,  /*enabled after initialization*/
                                                        };
static  uint32_t                    u32Time         =   0;
static  cy_stc_pdma_descr_config_t  stcDmaDescrConfig=  {
                                                        .deact          =   0,  /*Do not wait for trigger de-
activation*/
                                                        .intrType       =   CY_PDMA_INTR_1ELEMENT_CMPLT,
                                                        .trigoutType    =   CY_PDMA_TRIGOUT_1ELEMENT_CMPLT,
                                                        .chStateAtCmplt =   CY_PDMA_CH_DISABLED,
                                                        .triginType     =   CY_PDMA_TRIGIN_DESCR,
                                                        .dataSize       =   CY_PDMA_BYTE,
                                                        .srcTxfrSize    =   0,  /*= dataSize*/
                                                        .destTxfrSize   =   0,  /*= dataSize*/
                                                        .descrType      =   CY_PDMA_CRC_TRANSFER,
                                                        .srcAddr        =   (void*) au8SrcBuffer,
                                                        .destAddr       =   0,          //below initialized
                                                        .srcXincr       =   1,
                                                        .destXincr      =   1,
                                                        .xCount         =   BUFFER_SIZE,
                                                        .srcYincr       =   0,
                                                        .destYincr      =   0,
                                                        .yCount         =   0,
                                                        };
```

Code Listing 53 Cy_PDMA_Disable()

```
void Cy_PDMA_Disable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       0ul;
}
```

Write to P-DMA_CTL_ENABLED bit

Code Listing 54 Cy_PDMA_CRC_Config()

```
void Cy_PDMA_CRC_Config ( volatile stc_DW_t * pstcPDMA,
                          const cy_stc_pdma_crc_config_t*   pstcCrcConfig)
{
    pstcPDMA->unCRC_CTL.stcField.u1DATA_REVERSE     = pstcCrcConfig->data_reverse;
    pstcPDMA->unCRC_CTL.stcField.u1REM_REVERSE      = pstcCrcConfig->rem_reverse;
    pstcPDMA->unCRC_DATA_CTL.stcField.u8DATA_XOR    = pstcCrcConfig->data_xor;
    pstcPDMA->unCRC_LFSR_CTL.stcField.u32LFSR32     = pstcCrcConfig->lfsr32;
    pstcPDMA->unCRC_POL_CTL.stcField.u32POLYNOMIAL  = pstcCrcConfig->polynomial;
    pstcPDMA->unCRC_REM_CTL.stcField.u32REM_XOR     = pstcCrcConfig->rem_xor;
}
```

Configures P-DMA CRC

Code Listing 55 Cy_PDMA_Descr_Init()

```
cy_en_pdma_status_t Cy_PDMA_Descr_Init(cy_stc_pdma_descr_t* descriptor, const cy_stc_pdma_descr_config_t* config)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((descriptor != NULL) && (config != NULL))
    {
        descriptor->unPDMA_DESCR_CTL.stcField.u2WAIT_FOR_DEACT          =       config->deact;
        descriptor->unPDMA_DESCR_CTL.stcField.u2INTR_TYPE               =       config->intrType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_OUT_TYPE             =       config->trigoutType;
        descriptor->unPDMA_DESCR_CTL.stcField.u2TR_IN_TYPE              =       config->triginType;
        descriptor->unPDMA_DESCR_CTL.stcField.u1SRC_TRANSFER_SIZE       =       config->srcTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1DST_TRANSFER_SIZE       =       config->destTxfrSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u1CH_DISABLE              =       config->chStateAtCmplt;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DATA_SIZE               =       config->dataSize;
        descriptor->unPDMA_DESCR_CTL.stcField.u2DESCR_TYPE              =       config->descrType;

        descriptor->u32PDMA_DESCR_SRC                                   =       (uint32_t)config->srcAddr;
        descriptor->u32PDMA_DESCR_DST                                   =       (uint32_t)config->destAddr;

        switch(config->descrType)
        {
        case (uint32_t)CY_PDMA_SINGLE_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_1D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_CRC_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       0ul;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);
            descriptor->unPDMA_DESCR_Y_CTL.u32Register              =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_PDMA_2D_TRANSFER:
        {
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12SRC_X_INCR   =       (uint32_t)config->srcXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u12DST_X_INCR   =       (uint32_t)config->destXincr;
            descriptor->unPDMA_DESCR_X_CTL.stcField.u8X_COUNT       =       (uint32_t)((config->xCount) - 1ul);

            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12SRC_Y_INCR   =       (uint32_t)config->srcYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u12DST_Y_INCR   =       (uint32_t)config->destYincr;
            descriptor->unPDMA_DESCR_Y_CTL.stcField.u8Y_COUNT       =       (uint32_t)((config->yCount) - 1ul);

            descriptor->u32PDMA_DESCR_NEXT_PTR                      =       (uint32_t)config->descrNext;
            break;
        }
        default:
        {
            /* Unsupported type of descriptor */
            break;
        }
        }

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return retVal;
```

Code Listing 55 Cy_PDMA_Descr_Init()

```
}
```

Code Listing 56 Cy_PDMA_Chnl_Init()

```
cy_en_pdma_status_t Cy_PDMA_Chnl_Init(volatile stc_DW_t *pstcPDMA, uint32_t chNum, const cy_stc_pdma_chnl_config_t*
chnlConfig)
{
    cy_en_pdma_status_t retVal = CY_PDMA_ERR_UNC;

    if ((pstcPDMA != NULL) && (chnlConfig != NULL))
    {
        /* Set current descriptor */
        pstcPDMA->CH_STRUCT[chNum].unCH_CURR_PTR.u32Register            =  (uint32_t)chnlConfig->PDMA_Descriptor;

        /* Set if the channel is preemtable */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1PREEMPTABLE      =  chnlConfig->preemptable;

        /* Set channel priority */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u2PRIO             =  chnlConfig->priority;

        /* Set enabled status */
        pstcPDMA->CH_STRUCT[chNum].unCH_CTL.stcField.u1ENABLED          =  chnlConfig->enable;

        retVal = CY_PDMA_SUCCESS;
    }
    else
    {
        retVal = CY_PDMA_INVALID_INPUT_PARAMETERS;
    }

    return (retVal);
}
```

Code Listing 57 Cy_PDMA_Enable()

```
void Cy_PDMA_Enable(volatile stc_DW_t *pstcPDMA)
{
    pstcPDMA->unCTL.stcField.u1ENABLED  =       1ul;
}
```

Write to P-DMA_CTL_ENABLED bit

4 M-DMA use case

This section describes how to use Smart I/O using the sample driver library (SDL). The code snippets in this application note are part of SDL. See **Other references** for the SDL.

SDL has a configuration part and a driver part. The configuration part mainly configures the parameter values for the desired operation. The driver part configures each register based on the parameter values in the configuration part. You can configure the configuration part according to your system.

In this example, CYT2B7 series is used.

4.1 Memory-to-memory (memory copy)

This section describes an example of memory copy. Memory copy is an M-DMA-specific descriptor type. This is a special 1D transfer. Memory copy transfer data from the area specified by the source address and size to the destination address.

This is an example of data transfer from flash to RAM. This descriptor type is useful for copying the program code for RAM execution and copying the vector table. In the memory copy example, consecutive flash memory areas are transferred to RAM.

Figure 19 shows a use case of memory-to-memory transfer using memory copy.

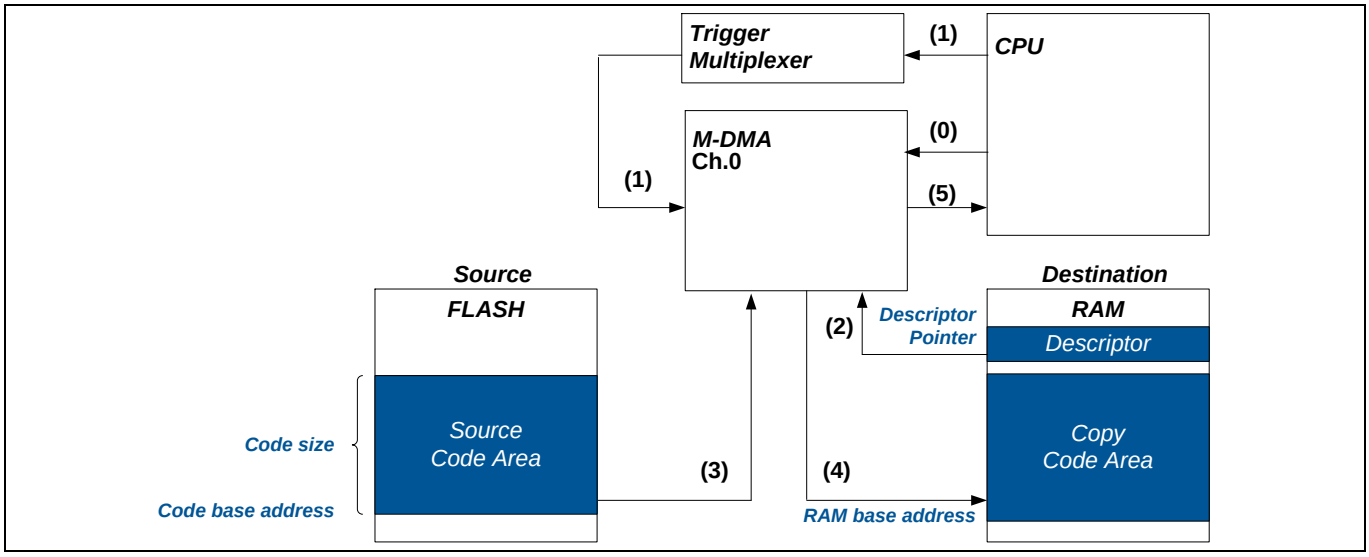


Figure 19 Use case of memory-to-memory using memory copy

(0) Set M-DMA according to the usage example setting.

(1) CPU notifies a software trigger to M-DMA via the trigger multiplexer.

(2) M-DMA reads the descriptor from the specified area (Descriptor Pointer) when accepting the transfer.

(3) M-DMA reads the data from the source address (code base address).

(4) M-DMA writes the read data to the destination address (RAM base address). After that, increment the source address and destination address. M-DMA repeats (3) (4) until it reaches the area specified by the transfer size (code size).

(5) When memory copy of the specified area is completed, M-DMA notifies an interrupt to CPU.

4.1.1 Initial configuration

This section describes the initialization of the M-DMA channel and descriptor of this use case.

Figure 20 shows the setting procedure for M-DMA.

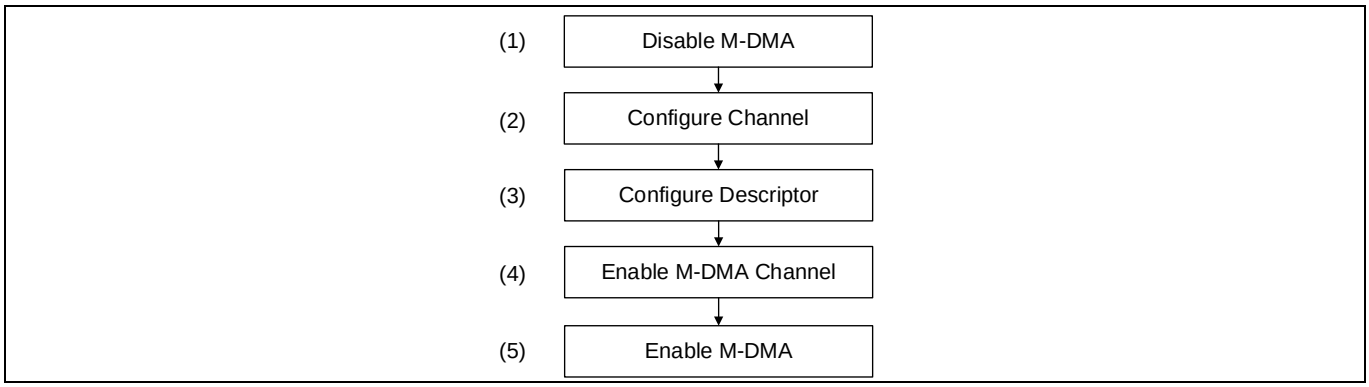


Figure 20 Setting procedure for M-DMA

4.1.2 Configuration and example code

Table 17 lists the parameters and **Table 18** lists the functions of the configuration part in SDL for M-DMA.

Table 17 List of DMA configuration parameters

Parameters	Description	Value
BUFFER_SIZE	Defines buffer size	36
DMAC_CHANNEL	Defines M-DMA channel	0
.MDMA_Descriptor	M-DMA current descriptor pointer	&stcDescr
.preemptable	Channel preemptable	0
.priority	Channel priority	0
.enable	Channel enable	1
.deact	DESCR_CTL WAIT_FOR_DEACT	0
.intrType	DESCR_CTL INTR_TYPE	CY_PDMA_INTR_1ELEMENT_CMPLT
.trigoutType	DESCR_CTL TR_OUT_TYPE	CY_PDMA_TRIGOUT_1ELEMENT_CMPLT
.chStateAtCmplt	DESCR_CTL CH_DISABLE	CY_MDMA_CH_DISABLED
.triginType	DESCR_CTL TR_IN_TYPE	CY_PDMA_TRIGIN_DESCR
.dataSize	DESCR_CTL DATA_SIZE	CY_MDMA_BYTE
.srcTxfrSize	DESCR_CTL SRC_TRANSFER_SIZE	0
.destTxfrSize	DESCR_CTL DST_TRANSFER_SIZE	0
.descrType	DESCR_CTL DESCR_TYPE	CY_MDMA_1D_TRANSFER
.srcAddr	DESCR_SRC	au8SrcBuffer
.destAddr	DESCR_DST	au8DestBuffer
.srcXincr	DESCR_X_INCR SRC_X_INCR	1
.destXincr	DESCR_X_INCR DST_X_INCR	1
.xCount	DESCR_X_SIZE X_COUNT	BUFFER_SIZE

M-DMA use case

Parameters	Description	Value
.srcYincr	DESCR_Y_INCR SRC_Y_INCR	0
.destYincr	DESCR_Y_INCR DST_Y_INCR	0
.yCount	DESCR_Y_SIZE Y_COUNT	0
.descrNext	DESCR_NEXT_PTR	0

Table 18 List of DMA configuration functions

Functions	Description	Remarks
Cy_MDMA_Disable()	Configures M-DMA disable	Write to M-DMA_CTL_ENABLED bit. See **Code Listing 60**
Cy_MDMA_Descr_Init()	Configures M-DMA descriptor initialize	See **Code Listing 61**
Cy_MDMA_Chnl_Init()	Configures M-DMA channel initialize	See **Code Listing 62**
Cy_MDMA_Chnl_Enable()	Configures M-DMA channel enable	See **Code Listing 63**
Cy_MDMA_Enable()	Configures M-DMA enable	Write to M-DMA_CTL_ENABLED bit. See **Code Listing 64**
Cy_TrigMux_SwTrigger()	Generates a Software trigger	-

Code Listing 58 demonstrates an example program to 2D transfer (peripheral-to-memory). See the **architecture TRM** and **application note** for GPIO and clock configuration.

Code Listing 58 Example to M-DMA Transfer

```
#define BUFFER_SIZE             36
#define DMAC_CHANNEL            0

int main(void)
{
:
    uint32_t  i;
    uint8_t   *p_src;

    __enable_irq(); /* Enable global interrupts. */

    /*******/
    /* DMA */
    /*******/

    /* Initialie & Enable DMA */
    Cy_MDMA_Disable(DMAC);
    Cy_MDMA_Descr_Init(&stcDescr,&stcDmaDescrConfig);
    Cy_MDMA_Chnl_Init( DMAC,
                       DMAC_CHANNEL,
                       (const cy_stc_mdma_chnl_config_t*) &chnlConfig);
    Cy_MDMA_Chnl_Enable( DMAC, DMAC_CHANNEL);
    Cy_MDMA_Enable(DMAC);

    /* for test */
    p_src = &au8SrcBuffer[0];
    for (i=0; i<BUFFER_SIZE; i++) {
        *p_src++ = i;
        au8DestBuffer[i] = 0;
    }

    /***************/
    /* Trigger MUX */
    /***************/
    Cy_TrigMux_SwTrigger( TRIG_OUT_MUX_2_MDMA_TR_IN0,
                          TRIGGER_TYPE_CPUSS_DMAC_TR_IN__EDGE,
                          1u);

    for(;;)
    {
    }
}
```

Define buffer size
Define DMAC channel

(1) Disable M-DMA. See **Code Listing 60**
(2) Configures M-DMA channel initialize. See **Code Listing 62**
(3) Configures M-DMA descriptor initialize. See **Code Listing 61**
(4) Enable M-DMA channel. See **Code Listing 63**
(5) Enable M-DMA. See **Code Listing 64**

Code Listing 59 M-DMA Channel, descriptor configuration

```
static  uint8_t                     au8DestBuffer[BUFFER_SIZE] = {0};
static  uint8_t                     au8SrcBuffer[BUFFER_SIZE];
static  cy_stc_mdma_descr_t         stcDescr;
const   cy_stc_mdma_chnl_config_t   chnlConfig      =   {
        /* CURR_PTR                         */     .MDMA_Descriptor =  &stcDescr,
                                                   .preemptable     =   0,
                                                   .priority        =   0,
                                                   .enable          =   1,
                                               };
static  cy_stc_mdma_descr_config_t  stcDmaDescrConfig = {
        /* DESCR_CTL       WAIT_FOR_DEACT   */     .deact           =   0,
        /* DESCR_CTL       INTR_TYPE        */     .intrType        =   CY_PDMA_INTR_1ELEMENT_CMPLT,
        /* DESCR_CTL       TR_OUT_TYPE      */     .trigoutType     =   CY_PDMA_TRIGOUT_1ELEMENT_CMPLT,
        /* DESCR_CTL       CH_DISABLE       */     .chStateAtCmplt  =   CY_MDMA_CH_DISABLED,
        /* DESCR_CTL       TR_IN_TYPE       */     .triginType      =   CY_PDMA_TRIGIN_DESCR,
        /* DESCR_CTL       DATA_SIZE        */     .dataSize        =   CY_MDMA_BYTE,
        /* DESCR_CTL       SRC_TRANSFER_SIZE*/     .srcTxfrSize     =   0,
        /* DESCR_CTL       DST_TRANSFER_SIZE*/     .destTxfrSize    =   0,
        /* DESCR_CTL       DESCR_TYPE       */     .descrType       =   CY_MDMA_1D_TRANSFER,
        /* DESCR_SRC                        */     .srcAddr         =   au8SrcBuffer,
        /* DESCR_DST                        */     .destAddr        =   au8DestBuffer,
        /* DESCR_X_INCR    SRC_X_INCR       */     .srcXincr        =   1,
        /* DESCR_X_INCR    DST_X_INCR       */     .destXincr       =   1,
        /* DESCR_X_SIZE    X_COUNT          */     .xCount          =   BUFFER_SIZE,
        /* DESCR_Y_INCR    SRC_Y_INCR       */     .srcYincr        =   0,
        /* DESCR_Y_INCR    DST_Y_INCR       */     .destYincr       =   0,
        /* DESCR_Y_SIZE    Y_COUNT          */     .yCount          =   0,
        /* DESCR_NEXT_PTR                   */     .descrNext       =   0
                                               };
```

Configure M-DMA channel

Configure M-DMA descriptor

Code Listing 60 Cy_MDMA_Disable()

```
void Cy_MDMA_Disable(volatile stc_DMAC_t *pstcMDMA)
{
    pstcMDMA->unCTL.stcField.u1ENABLED  =       0ul;
}
```

Write to M-DMA_CTL_ENABLED

Code Listing 61 Cy_MDMA_Descr_Init()

```
cy_en_mdma_status_t Cy_MDMA_Descr_Init(cy_stc_mdma_descr_t* descriptor, const cy_stc_mdma_descr_config_t* config)
{
    cy_en_mdma_status_t retVal = CY_MDMA_ERR_UNC;

    if ((descriptor != NULL) && (config != NULL))
    {
        /* Descriptor[0] */
        descriptor->unMDMA_DESCR_CTL.stcField.u2WAIT_FOR_DEACT     =       config->deact;
        descriptor->unMDMA_DESCR_CTL.stcField.u2INTR_TYPE          =       config->intrType;
        descriptor->unMDMA_DESCR_CTL.stcField.u2TR_OUT_TYPE        =       config->trigoutType;
        descriptor->unMDMA_DESCR_CTL.stcField.u2TR_IN_TYPE         =       config->triginType;
        descriptor->unMDMA_DESCR_CTL.stcField.u1DATA_PREFETCH      =       config->dataPrefetch;
        descriptor->unMDMA_DESCR_CTL.stcField.u2DATA_SIZE          =       config->dataSize;
        descriptor->unMDMA_DESCR_CTL.stcField.u1CH_DISABLE         =       config->chStateAtCmplt;
        descriptor->unMDMA_DESCR_CTL.stcField.u1SRC_TRANSFER_SIZE  =       config->srcTxfrSize;
        descriptor->unMDMA_DESCR_CTL.stcField.u1DST_TRANSFER_SIZE  =       config->destTxfrSize;
        descriptor->unMDMA_DESCR_CTL.stcField.u3DESCR_TYPE         =       config->descrType;

        /* Descriptor[1] */
        descriptor->u32MDMA_DESCR_SRC                              =       (uint32_t)config->srcAddr;

        /* after 3rd word of descriptor depends on descriptor type */

        switch(config->descrType)
        {
        case (uint32_t)CY_MDMA_SINGLE_TRANSFER:
        {
            /* Descriptor[2] */
            descriptor->u32MDMA_DESCR_DST                          =       (uint32_t)config->destAddr;

            /* Descriptor[3] -> NEXT_PTR */
            descriptor->unMDMA_DESCR_X_SIZE.u32Register            =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_MDMA_1D_TRANSFER:
```

Code Listing 61 Cy_MDMA_Descr_Init()

```
        {
            /* Descriptor[2] */
            descriptor->u32MDMA_DESCR_DST                               =       (uint32_t)config->destAddr;

            /* Descriptor[3] */
            descriptor->unMDMA_DESCR_X_SIZE.stcField.u16X_COUNT     =       (uint32_t)((config->xCount) - 1ul);

            /* Descriptor[4] */
            descriptor->unMDMA_DESCR_X_INCR.stcField.u16SRC_X_INCR  =       (uint32_t)config->srcXincr;
            descriptor->unMDMA_DESCR_X_INCR.stcField.u16DST_X_INCR  =       (uint32_t)config->destXincr;

            /* Descriptor[5] -> NEXT_PTR */
            descriptor->unMDMA_DESCR_Y_SIZE.u32Register             =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_MDMA_2D_TRANSFER:
        {
            /* Descriptor[2] */
            descriptor->u32MDMA_DESCR_DST                               =       (uint32_t)config->destAddr;

            /* Descriptor[3] */
            descriptor->unMDMA_DESCR_X_SIZE.stcField.u16X_COUNT     =       (uint32_t)((config->xCount) - 1ul);

            /* Descriptor[4] */
            descriptor->unMDMA_DESCR_X_INCR.stcField.u16SRC_X_INCR  =       (uint32_t)config->srcXincr;
            descriptor->unMDMA_DESCR_X_INCR.stcField.u16DST_X_INCR  =       (uint32_t)config->destXincr;

            /* Descriptor[5] */
            descriptor->unMDMA_DESCR_Y_SIZE.stcField.u16Y_COUNT     =       (uint32_t)((config->yCount) - 1ul);

            /* Descriptor[6] */
            descriptor->unMDMA_DESCR_Y_INCR.stcField.u16SRC_Y_INCR  =       (uint32_t)config->srcYincr;
            descriptor->unMDMA_DESCR_Y_INCR.stcField.u16DST_Y_INCR  =       (uint32_t)config->destYincr;

            /* Descriptor[7] */
            descriptor->u32MDMA_DESCR_NEXT_PTR                          =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_MDMA_MEMORY_COPY_TRANSFER:
        {
            /* Descriptor[2] */
            descriptor->u32MDMA_DESCR_DST                               =       (uint32_t)config->destAddr;

            /* Descriptor[3] */
            descriptor->unMDMA_DESCR_X_SIZE.stcField.u16X_COUNT     =       (uint32_t)((config->xCount) - 1ul);

            /* Descriptor[4] -> NEXT_PTR */
            descriptor->unMDMA_DESCR_X_INCR.u32Register             =       (uint32_t)config->descrNext;
            break;
        }
        case (uint32_t)CY_MDMA_SCATTER_TRANSFER:
        {
            /* Descriptor[2] -> X_SIZE */
            descriptor->u32MDMA_DESCR_DST                               =       (uint32_t)((config->xCount) - 1ul);

            /* Descriptor[3] -> NEXT_PTR */
            descriptor->unMDMA_DESCR_X_SIZE.u32Register             =       (uint32_t)config->descrNext;
            break;
        }
        default:
        {
            /* Unsupported type of descriptor */
            break;
        }
        }

        retVal = CY_MDMA_SUCCESS;
    }
    else
    {
        retVal = CY_MDMA_INVALID_INPUT_PARAMETERS;
    }

    return retVal;
}
```

Code Listing 62 Cy_MDMA_Chnl_Init()

```
cy_en_mdma_status_t Cy_MDMA_Chnl_Init(volatile stc_DMAC_t *pstcMDMA, uint32_t chNum, const cy_stc_mdma_chnl_config_t*
chnlConfig)
{
    cy_en_mdma_status_t retVal = CY_MDMA_ERR_UNC;
```

Code Listing 62 Cy_MDMA_Chnl_Init()

```
    if ((pstcMDMA != NULL) && (chnlConfig != NULL))
    {
        /* Set current descriptor */
        pstcMDMA->CH[chNum].unCURR.u32Register   =  (uint32_t)chnlConfig->MDMA_Descriptor;

        /* Set if the channel is preemptable  */
        /* There is no the parameter in MDMA */

        /* Set channel priority */
        pstcMDMA->CH[chNum].unCTL.stcField.u2PRIO    =   chnlConfig->priority;

        /* Set enabled status */
        pstcMDMA->CH[chNum].unCTL.stcField.u1ENABLED =   chnlConfig->enable;

        retVal = CY_MDMA_SUCCESS;
    }
    else
    {
        retVal = CY_MDMA_INVALID_INPUT_PARAMETERS;
    }

    return (retVal);
}
```

Code Listing 63 Cy_MDMA_Chnl_ Enable()

```
__STATIC_INLINE void Cy_MDMA_Chnl_Enable(volatile stc_DMAC_t *pstcMDMA, uint32_t chNum)
{
    pstcMDMA->CH[chNum].unCTL.stcField.u1ENABLED = 1ul;
}
```

Code Listing 64 Cy_MDMA_ Enable()

```
void Cy_MDMA_Enable(volatile stc_DMAC_t *pstcMDMA)
{
    pstcMDMA->unCTL.stcField.u1ENABLED  =       1ul;
}
```

Write to M-DMA_CTL_ENABLED

5 Glossary

Table 19 Glossary

Terms	Description
DMA controller	Direct memory access controller
P-DMA	Peripheral DMA
M-DMA	Memory DMA
Single transfer	This transfers a single data element (8-bit, 16-bit, or 32-bit). See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
1D transfer	This performs a one-dimensional "for loop" (described in C). See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
2D transfer	This performs a two-dimensional "for loop" (described in C). See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
CRC transfer	This performs a one-dimensional “for loop” similar to the 1D transfer. However, the source data is not transferred to a destination. A CRC is calculated over the source data. Only P-DMA is supported. See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
Memory copy	This is a special case of 1D transfer. Only M-DMA is supported. See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
Scatter	This descriptor type is intended to write a set of 32-bit data elements, whose addresses are “scattered” around the address space. Only M-DMA is supported. See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
Descriptor	A descriptor specifies the details of data transfer of DMA channels. See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
Descriptor chain	A DMA channel executes the next descriptor specified in the current descriptor when it completes executing the descriptor. See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
Descriptor list	Same as descriptor chain.
Descriptor pointer	The start address of the memory where the descriptor is stored. See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
Descriptor type	The transfer operation type performed by DMA. See the “Descriptors” section in the Direct Memory Access chapter of the **architecture TRM** for details.
Descriptor word	The **composition element** of the descriptor. There are descriptor control, source/destination address, X/Y loop control, and descriptor next pointer. See the “P-DMA Descriptor Structure” and “M-DMA Descriptor Structure” section in the Direct Memory Access chapter of the **architecture TRM** for details.
Preemptable	P-DMA specific functions. See the “Channels” section in the Direct Memory Access chapter of the **architecture TRM** for details.

MMIO	Memory Mapped I/O
ADC	Analog-to-digital converter. See the “SAR ADC” chapter of the **architecture TRM** for details.
SCB	Serial Communications Block. See the “Serial Communications Block (SCB)” chapter of the **architecture TRM** for details.
TCPWM	Timer, Counter, and Pulse Width Modulator. See the “Timer, Counter, and PWM” chapter of the **architecture TRM** for details.
Trigger multiplexer	A trigger multiplexer routes triggers from a source peripheral to a destination. See the “Trigger Multiplexer” chapter of the **architecture TRM** for details.

References

The following are the TRAVEO™ II family series datasheets and technical reference manuals. Contact **Technical Support** to obtain these documents.

[1] Device datasheets

- **CYT2B7 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ II family**
- **CYT2B9 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ II family**
- **CYT4BF datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ II family**
- CYT4DN datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ II family (Doc No. 002-24601)
- **CYT3BB/4BB datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ II family**
- CYT3DL datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ II family (Doc No. 002-27763)

[2] Technical reference manuals

- Body controller entry family
  - **TRAVEO™ II automotive body controller entry family architecture technical reference manual (TRM)**
  - **TRAVEO™ II automotive body controller entry registers technical reference manual (TRM) for CYT2B7**
  - **TRAVEO™ II automotive body controller entry registers technical reference manual (TRM) for CYT2B9**
- Body controller high family
  - **TRAVEO™ II automotive body controller high family architecture technical reference manual (TRM)**
  - **TRAVEO™ II automotive body controller high registers technical reference manual (TRM) for CYT4BF**
  - **TRAVEO™ II automotive body controller high registers technical reference manual (TRM) for CYT3BB/4BB**
- Cluster 2D family
  - TRAVEO™ II automotive cluster 2D family architecture technical reference manual (TRM) (Doc No. 002-25800)
  - TRAVEO™ II automotive cluster 2D registers technical reference manual (TRM) for CYT4DN (Doc No. 002-25923)
  - TRAVEO™ II automotive cluster 2D registers technical reference manual (TRM) for CYT 3DL (Doc No. 002-29854)

[3] Application notes

- **AN220193 - GPIO usage setup in TRAVEO™ II family**
- **AN225401 - How to use serial communications block (SCB) in TRAVEO™ II family**
- **AN220224 – How to use timer, counter, and PWM (TCPWM) in TRAVEO™ II family**
- **AN219755 - Using a SAR ADC in TRAVEO™ II Family**

Other references

A sample driver library (SDL) including startup as sample software to access various peripherals is provided. SDL also serves as a reference, to customers, for drivers that are not covered by the official AUTOSAR products. The SDL cannot be used for production purposes as it does not qualify to any automotive standards. The code snippets in this application note are part of the SDL. Contact **Technical Support** to obtain the SDL.

Revision history

Document version	Date of release	Description of changes
**	2018-06-09	New application note.
*A	2018-06-12	Updated associated part family as “TRAVEO™ II family CYT2B series”. Changed target part numbers from “CYT2B5/B7 series” to “CYT2B series” in all instances across the document.
*B	2019-03-26	Updated associated part family as “TRAVEO™ II family CYT2B/CYT4B series”. Added target part numbers “CYT4B series” related information in all instances across the document. Updated operation overview: Updated configure channel: Updated “Table 3. channel configuration for P-DMA”. Updated P-DMA use cases: Updated CRC transfer: Updated initial configuration: Updated description (correction of errors in CRC register name).
*C	2019-02-10	Updated associated part family as “TRAVEO™ II family CYT2B/CYT4B/CYT4D series”. Added target part numbers “CYT4D series” related information in all instances across the document. Added the setting flow of DW_CH_STRUCT_CH_IDX into channel configuration for P-DMA in all instances across the document. Updated operation overview: Updated configure channel: Updated “Table 3. channel configuration for P-DMA”. Updated “Table 4. channel configuration for M-DMA”.
*D	2020-03-19	Updated associated part family as “TRAVEO™ II family CYT2/CYT3/CYT4 series”. Changed target part numbers from “CYT2B/CYT4B/CYT4D series” to “CYT2/ CYT4 series” in all instances across the document. Added target part numbers “CYT3 series” related information in all instances across the document.
*E	2021-03-29	Updated to Infineon template.
*F	2021-07-09	Added example of SDL code and description in all instances.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2021-07-09
Published by
Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?
Go to www.cypress.com/support

Document reference
002-20191 Rev. *F

IMPORTANT NOTICE
The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.