XMC4000

32-bit Microcontroller Series for Industrial Applications

General Purposes Direct Memory Access (GPDMA)

AP32290

Application Note

About this document

Scope and purpose

This Application Note describes the usage of the General Purpose Direct Memory Access (GPDMA) in the XMC4000 microcontroller series.

Various transfer modes of the GPDMA are described, showing how to use the GPDMA to transfer data to and from several peripherals.

Applicable Products

- XMC4000 Microcontrollers Family

References

Infineon: Example code: http://www.infineon.com/XMC4000 Tab: Documents

Infineon: XMC Lib, http://www.infineon.com/DAVE

Infineon: DAVE™, http://www.infineon.com/DAVE

Infineon: XMC Reference Manual, http://www.infineon.com/XMC4000 Tab: Documents

Infineon: XMC Data Sheet, http://www.infineon.com/XMC4000 Tab: Documents

Table of Contents

Table of Contents **2**

1 GPDMA Overview **3**
1.1 Block diagram of GPDMA 3
1.2 GPDMA Terminology 4

2 Flow control and Handshaking Interface **5**
2.1 Hardware handshaking 5
2.1.1 Enable of Hardware handshake 6
2.1.2 Routing service request signal to DMA channel 6
2.2 Software handshaking 7

3 DMA transfer block size **8**

4 Source and Destination addressing **9**
4.1 Address increment or decrement 9
4.2 Gather transfer (Source) 10
4.3 Scatter transfer (Destination) 11

5 Block transfer types **12**
5.1 Auto reloading of channel register 12
5.2 Contiguous address between blocks 12
5.3 Block chaining using linked lists 13

6 Service Request Generation **14**

7 Multi block transfer of VADC result registers to RAM for motor control **15**
7.1 PWM period match interrupt to trigger ADC conversion 16
7.2 GPDMA configuration for multiple block transfer 18
7.2.1 Setting the transfer flow and priority 18
7.2.2 Enable the Hardware handshaking 18
7.2.3 Selecting the service request via the DMA Line Router 19
7.2.4 Source configuration 19
7.2.5 Destination configuration 20
7.2.6 Configure the block transfer type 20
7.2.7 Block transfer complete Interrupt 20

8 Single block transfer CRC checking **21**
8.1 Setting up FCE for CRC checking 22
8.2 GPDMA configuration for Single block transfer 22
8.2.1 Setting the transfer flow and priority 23
8.2.2 Source configuration 23
8.2.3 Destination configuration 23
8.2.4 Configure the block transfer type 23
8.2.5 Block transfer complete Interrupt 23

9 Revision History **25**

1 GPDMA Overview

The General Purposes Direct Memory Access (GPDMA) is a module within the XMC4000 series to transfer data without any CPU interference. When a DMA transfer request is generated, the GPDMA transfers data stored at the source address to the destination address.

1.1 Block diagram of GPDMA

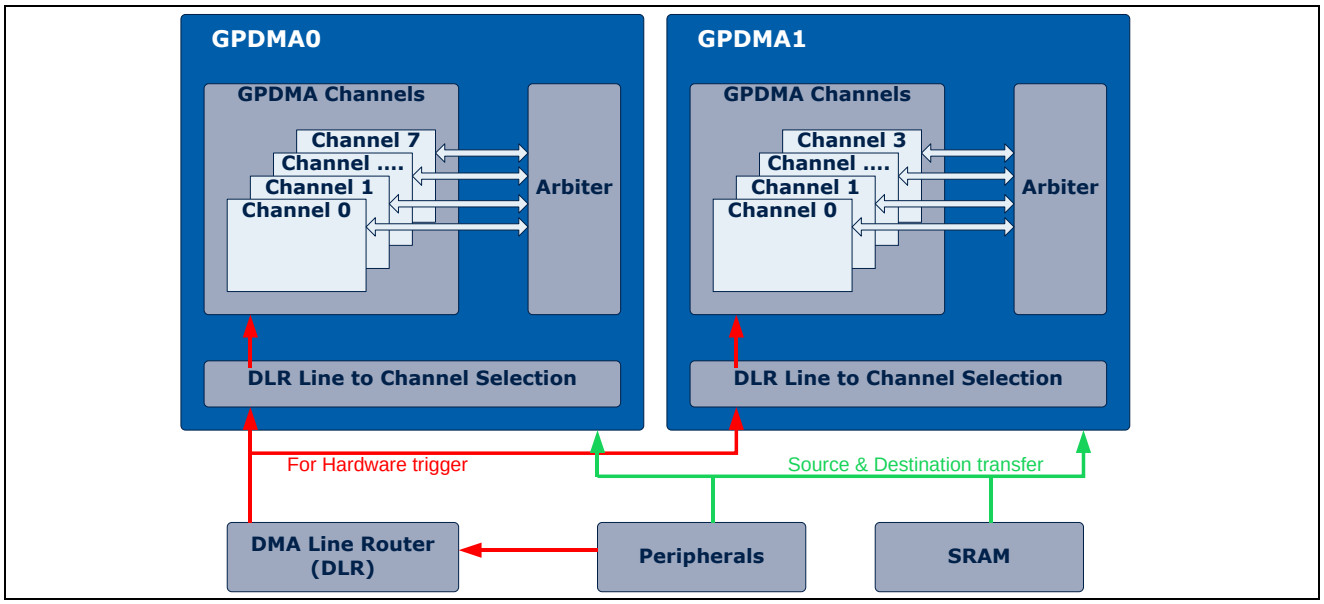


Figure 1 GPDMA Block Diagram

Note: Please note that the number of DMA channels is different for XCM4500, XMC4400, XMC4200 and XMC4100.

Figure 1 shows the block diagram of the GPDMA as implemented in XMC4000 series:

- GPDMA channels: up to 12
- Arbiter
- Single AHB master interface for data transfer
- Single AHB slave interface for register configuration
- DMA Line Router (DLR)

One channel of the GPDMA is required for each source/destination pair. The master interface reads the data from a source peripheral and writes it to a destination peripheral. Two physical transfers which consist of Source to FIFO and FIFO to Destination are therefore required for each DMA transaction.

1.2 GPDMA Terminology

Service partners terms

- The **Source peripheral** is the device from which the GPDMA reads data and stores the data in the channel FIFO. The source peripheral teams up with a destination peripheral to form a channel.
- The **Destination peripheral** is the device to which the GPDMA writes the stored data from the FIFO (previously read from the source peripheral).

The GPDMA0 channels 0 and 1 provide a **FIFO size of 32 Bytes**. These channels can be used to execute burst transfers up to a fixed length burst size of 8. The remaining channels **FIFO size is 8 Bytes**.

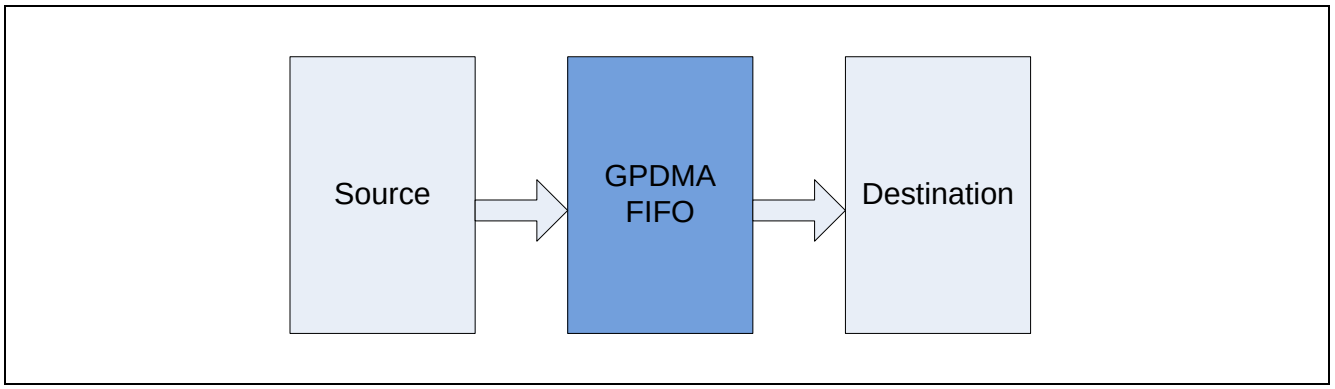


Figure 2 Source and Destination definition

Transfer terms

- A **Single transaction** is made up of a data width size of **8/16/32 bit**.
- A **Burst transaction** can be composed of **1, 4 or 8 transaction**.
- A **Block transfer** shall transfer the number of transaction specified by the block transfer size.

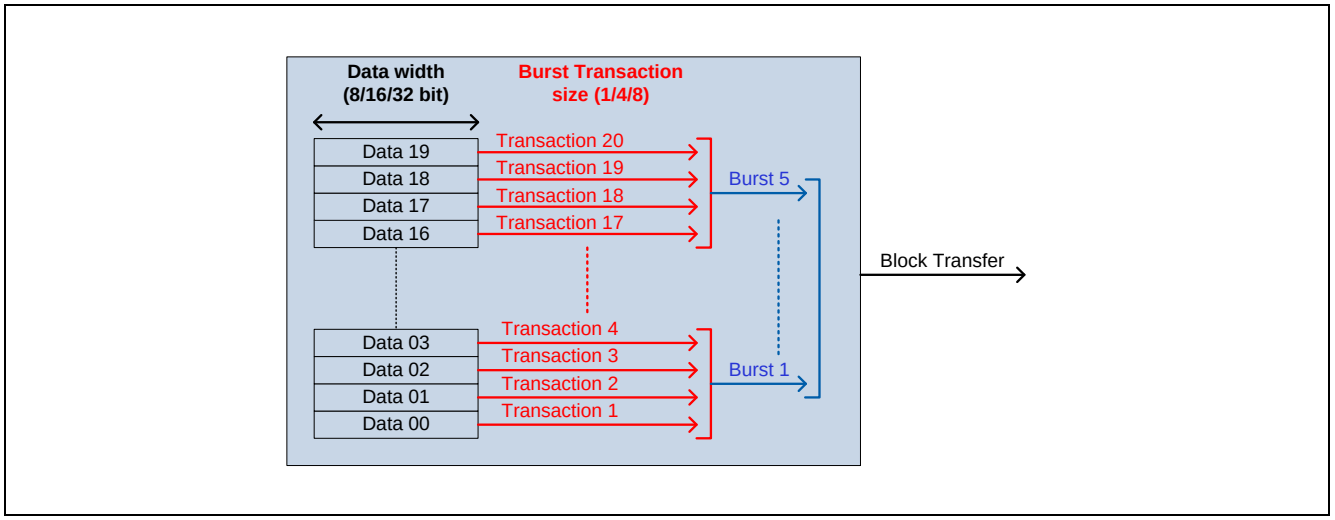


Figure 3 GPDMA transfer definition

2 Flow control and Handshaking Interface

The device that controls the length of a block is known as the flow controller. The GPDMA, the source peripheral, or the destination peripheral can be assigned as the flow controller depends on the selection of the transfer flow in register **CTLL.TT_FC**. Typically if the block size is known, then the GPDMA must be assigned as the flow controller.

Lastly the operation of the handshaking interface depends on whether the peripheral or the GPDMA is the flow controller. The peripheral uses the handshaking interface to indicate to the GPDMA that it is ready to transfer data over the AHB bus. A peripheral can request a DMA transaction through the GPDMA using the Hardware or Software handshaking interface.

Table 1 Transfer flow, Flow Control and Handshake combination

Transfer flow	Flow controller	Handshaking
Memory to Memory	GPDMA	Not required
Memory to Peripheral	GPDMA	Hardware or Software
Peripheral to Memory	GPDMA	Hardware or Software
Peripheral to Peripheral	GPDMA	Hardware or Software
Peripheral to Memory	Peripheral	Software
Peripheral to Peripheral	Source peripheral	Software
Memory to Peripheral	Peripheral	Software
Peripheral to Peripheral	Destination peripheral	Software

2.1 Hardware handshaking

Hardware handshaking means to use the peripheral service request signal (eg. VADC.G0SR0, ERU0.SR0, CCU40.SR0 etc) to request for a DMA transfer via a DMA Line Router (DLR). Before the transfer can begin the GPDMA and DLR units must be set up according to the application requirements.

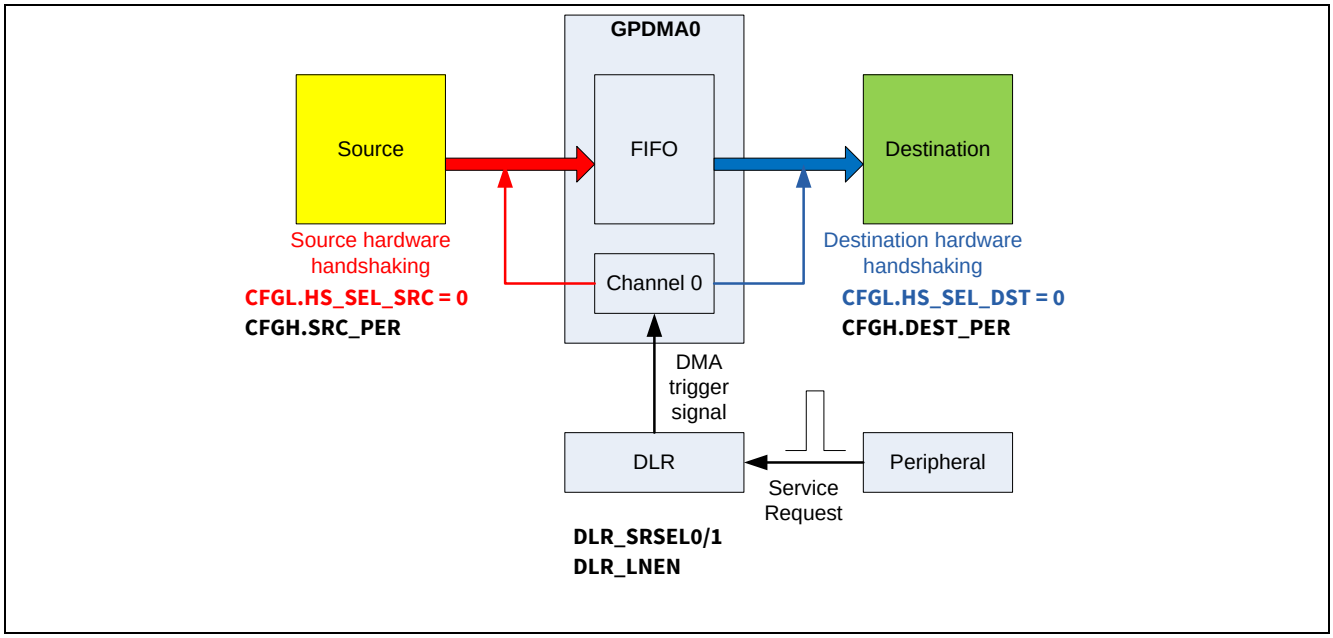


Figure 4 Hardware handshaking interface

2.1.1 Enable of Hardware handshake

The hardware handshake are available for both source and destination of a particular channel which can be enabled by setting the register **CFGL.HS_SEL_SRC = 0** and **CFGL.HS_SEL_DST = 0** respectively**.** And the accepted polarity of the service request signal by the GPDMA is by default active high. However the polarity of the signal can be change if necessary by setting **CFGL.SRC_HS_POL** or **CFGL.DST_HS_POL**.

2.1.2 Routing service request signal to DMA channel

The **DMA Line Router (DLR)** is used to route a required service request signal from a peripheral to a particular GPDMA channel. There are a total of 12 DMA lines (RS0 to RS11) and each DMA line offers selection to certain service request source which can be configured using register **DLR_SRSEL0/1**. (See table 4-5 of reference manual for DMA request source selection)

Then the **DLR_LNEN** register is used to enable selected DLR line and to reset a previously stored and pending service request. Finally individual GPDMA channel configuration register **CFGH.SRC_PER or CFGH.DEST_PER** can select the assigned DLR line.

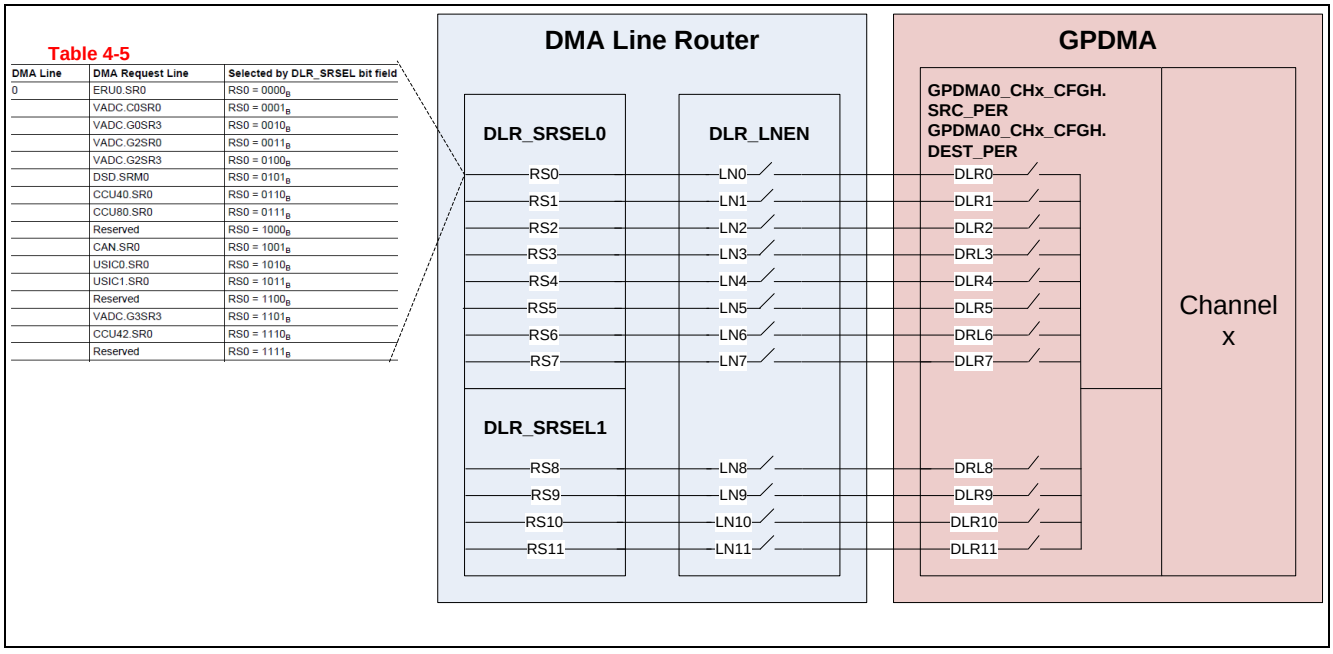


Figure 5 DLR Line to channel selection

2.2 Software handshaking

Additionally the user can trigger a DMA channel using a software trigger. Usually this is done in the ISR responding to a peripheral service request, for example the peripheral being ready to accept new data.

Before the transfer can begin the GPDMA and NVIC units must be set up according to the application requirements. Once the peripheral (source or destination) is ready for a transaction it sends a service request to the CPU. The interrupt service routine then uses the GPDMA software registers, to initiate and control a DMA transaction.

Table 2 Software handshake configuration

Software handshake request	Enable software handshake	Register setting
Source Software Transaction Request	CFGL.HS_SEL_SRC = 1	REQSRCREG.CHx = 1 REQSRCREG.CHx = 1
Destination Software Transaction Request	CFGL.HS_SEL_DST = 1	REQDSTREG.CHx = 1 REQDSTREG.CHx = 1

By setting the software register **CFGL.HS_SEL_SRC = 1** and/or **CFGL.HS_SEL_DST = 1** enables software handshaking on the source and/or destination of channel x.

To start software handshaking to perform a block transfer from source to FIFO on channel 0 simply set **REQSRCREG.CH0 = 1** and **REQSRCREG.WE_CH0 = 1.**

The same process can be done for software handshaking from FIFO to destination by setting **REQDSTREG.CH0 = 1** and **REQDSTREG.WE_CH0 = 1.**

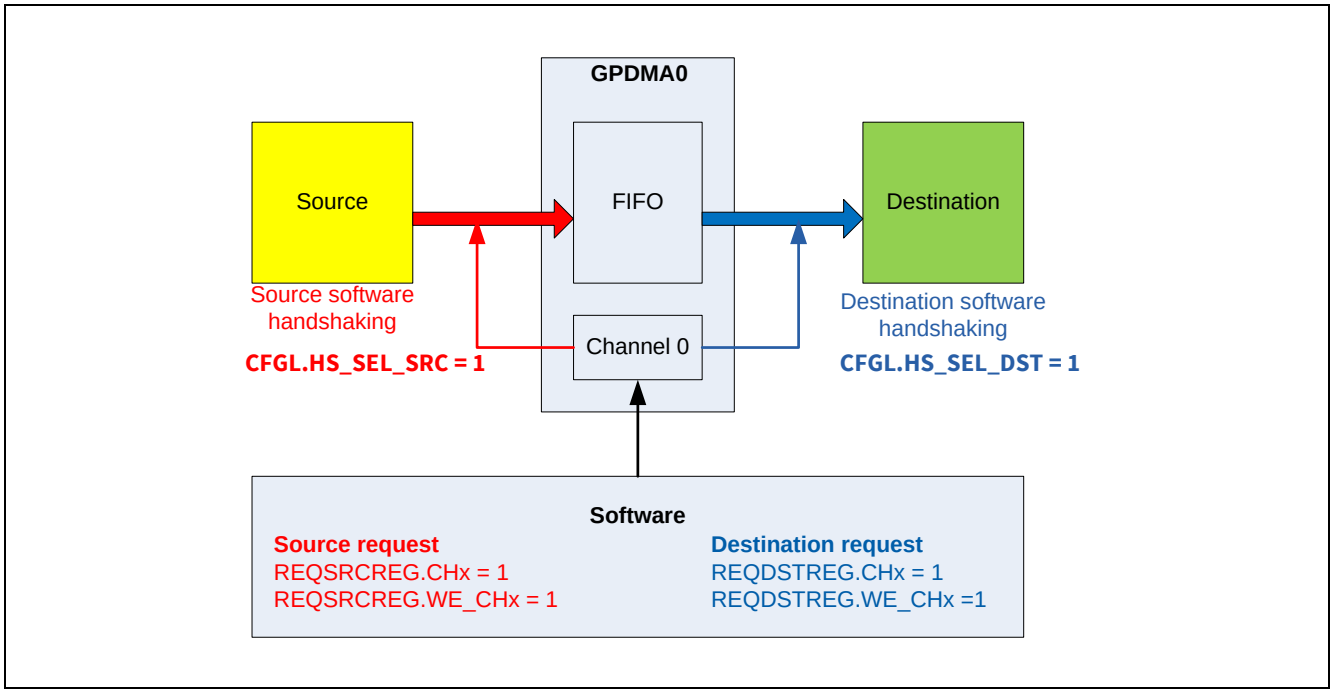


Figure 6 Software handshaking interface

3 DMA transfer block size

Both the source and destination are required to configure the width (8/16/32 bit) of the data by programming the source transfer width **CTLL.SRC_TR_WIDTH** and destination transfer width **CTLL.DST_TR_WIDTH** of the CTLL register respectively. And the total number of data required for a transfer block can be configured in the register **CTLH.BLOCK_TS**.

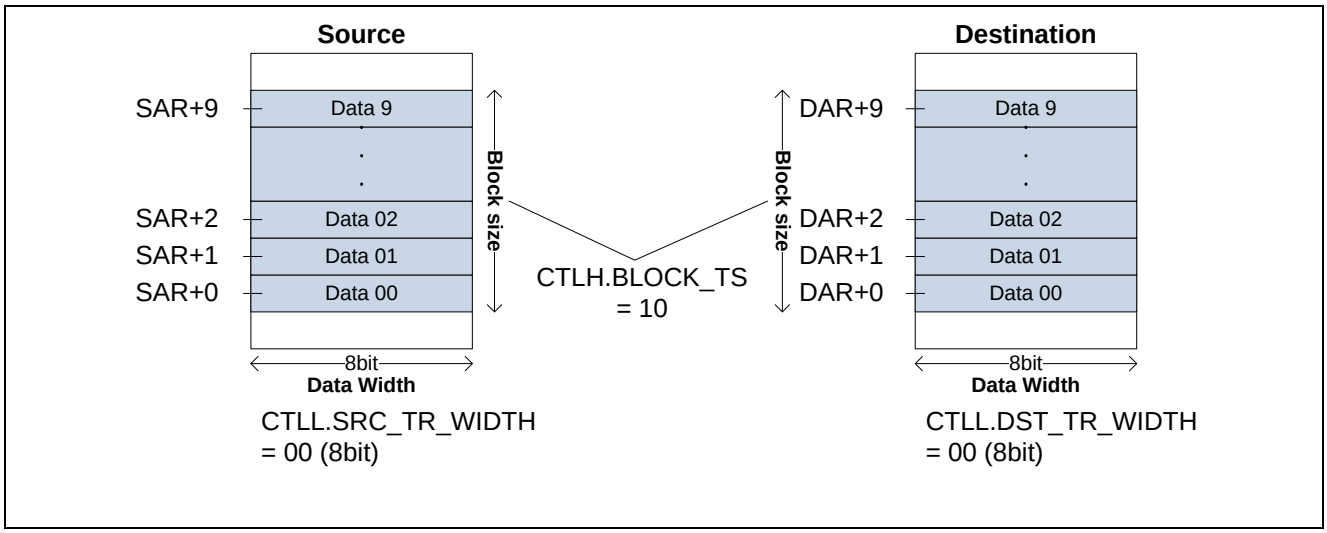


Figure 7 Block size configuration

4 Source and Destination addressing

The GPDMA module needs to know the start address of the source and destination before it can begin any transfer. Therefore the source and destination base address have to be programmed into the Source Address register **SAR** and the Destination Address register **DAR** respectively.

4.1 Address increment or decrement

For every next DMA transaction, the source and destination address can be increment or decrement by setting the Source Address Increment **CTLL.SINC** or Destination Address Increment **CTLL.DINC**. However by default the **SINC** and **DINC** is set to incremental mode.

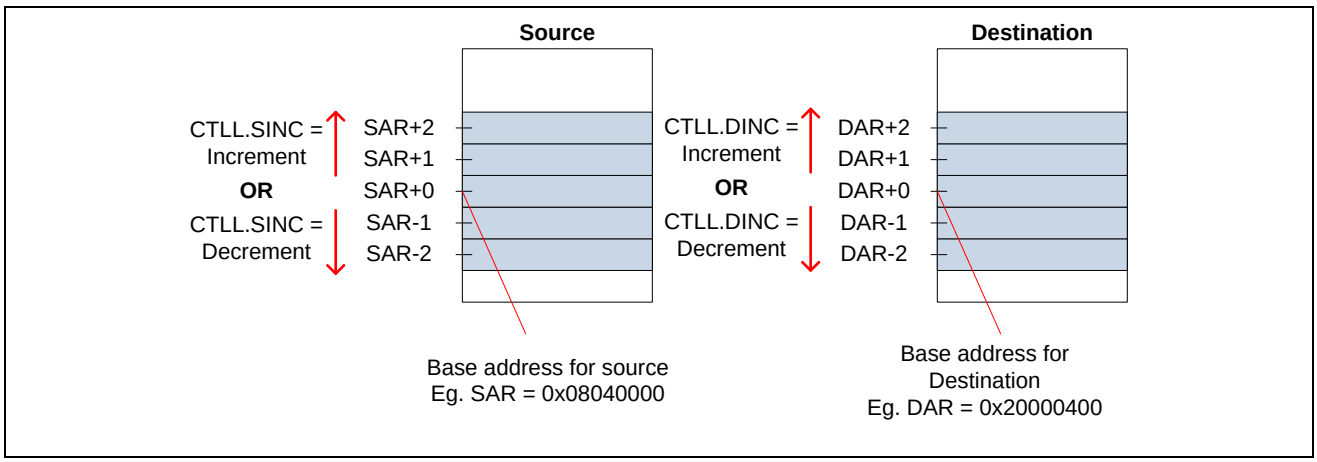


Figure 8 Source and Destination address configuration

4.2 Gather transfer (Source)

Note: Only for GPDMA0 channel 0 to 1.

Gather transfer is relevant to source transfers within a block; we can program the GPDMA to have an interval within a sequential address increment or decrement. This is known as the Gather transfer.

The Gather transfer works by incrementing or decrementing the source address by a programmed amount specified in the Source Gather Count (**SGRx.SGC**) of the **SGR** register, which is also know as the “Gather boundary “. When the Gather boundary is reached, the source address is then offset by the value stored in the source gather interval (**SGRx.SGI**) field.

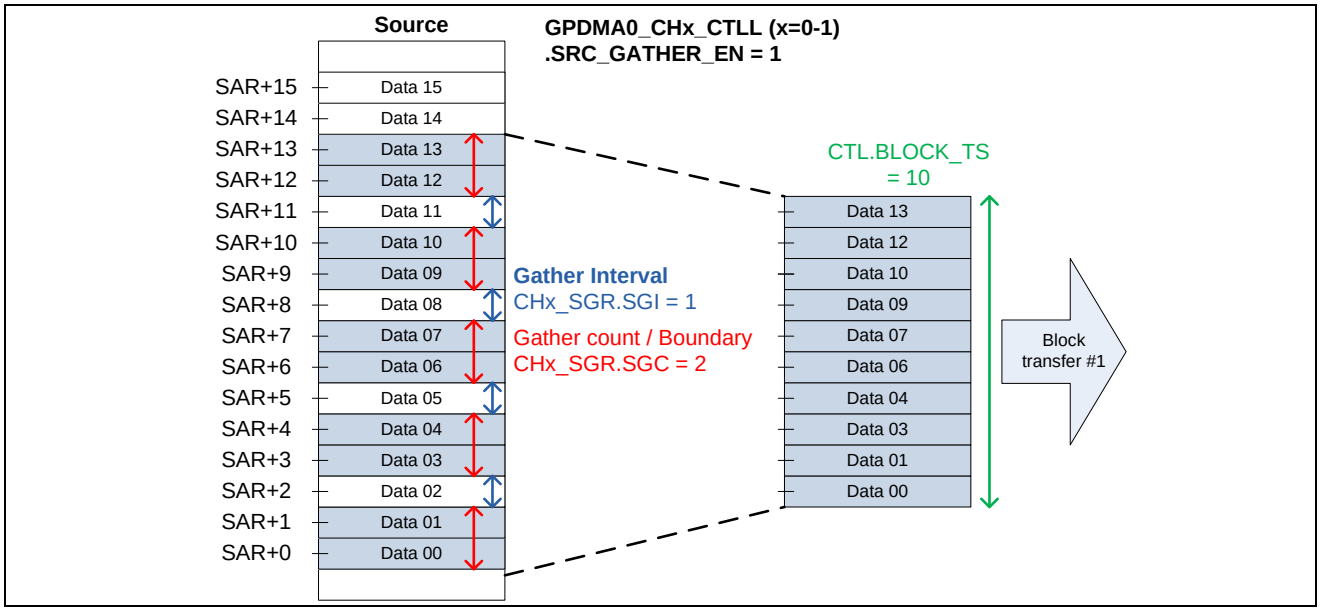


Figure 9 Example of Gather transfer

4.3 Scatter transfer (Destination)

Note: Only for GPDMA0 channel 0 to 1.

Scatter transfer is relevant to destination transfers within a block; we can program the GPDMA to have an interval within a sequential address increment or decrement. This is known as the Scatter transfer.

The Scatter transfer works by incrementing or decrementing the source address by a programmed amount specified in the Destination Scatter Count (DSRx.DSC) of the DSR register, which is also know as the “Scatter boundary “. When the Scatter boundary is reached, the destination address is then offset by the value stored in the destination scatter interval (DSRx.DSI) field.

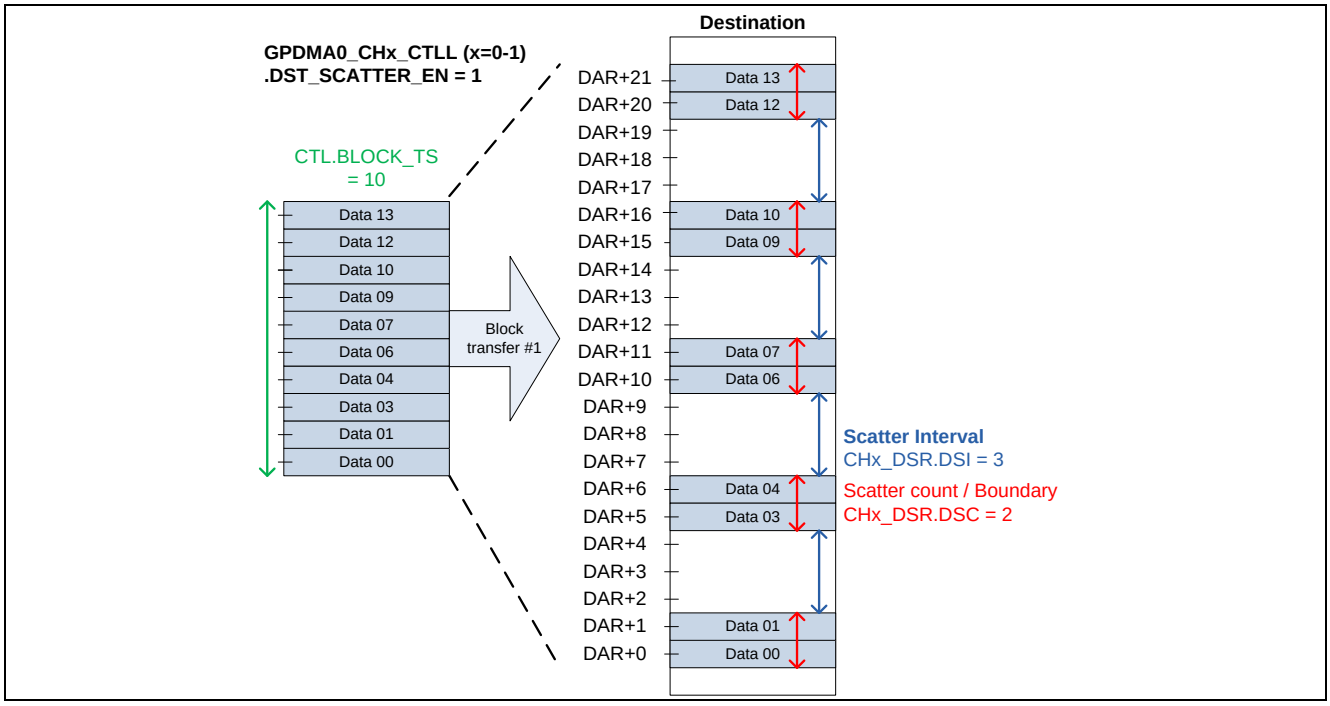


Figure 10 Example of Scatter transfer

5 Block transfer types

The types of DMA block transfer from source to destination are achievable by using the combinations of **Auto Reload method, Contiguous method** or **Linked List method.**

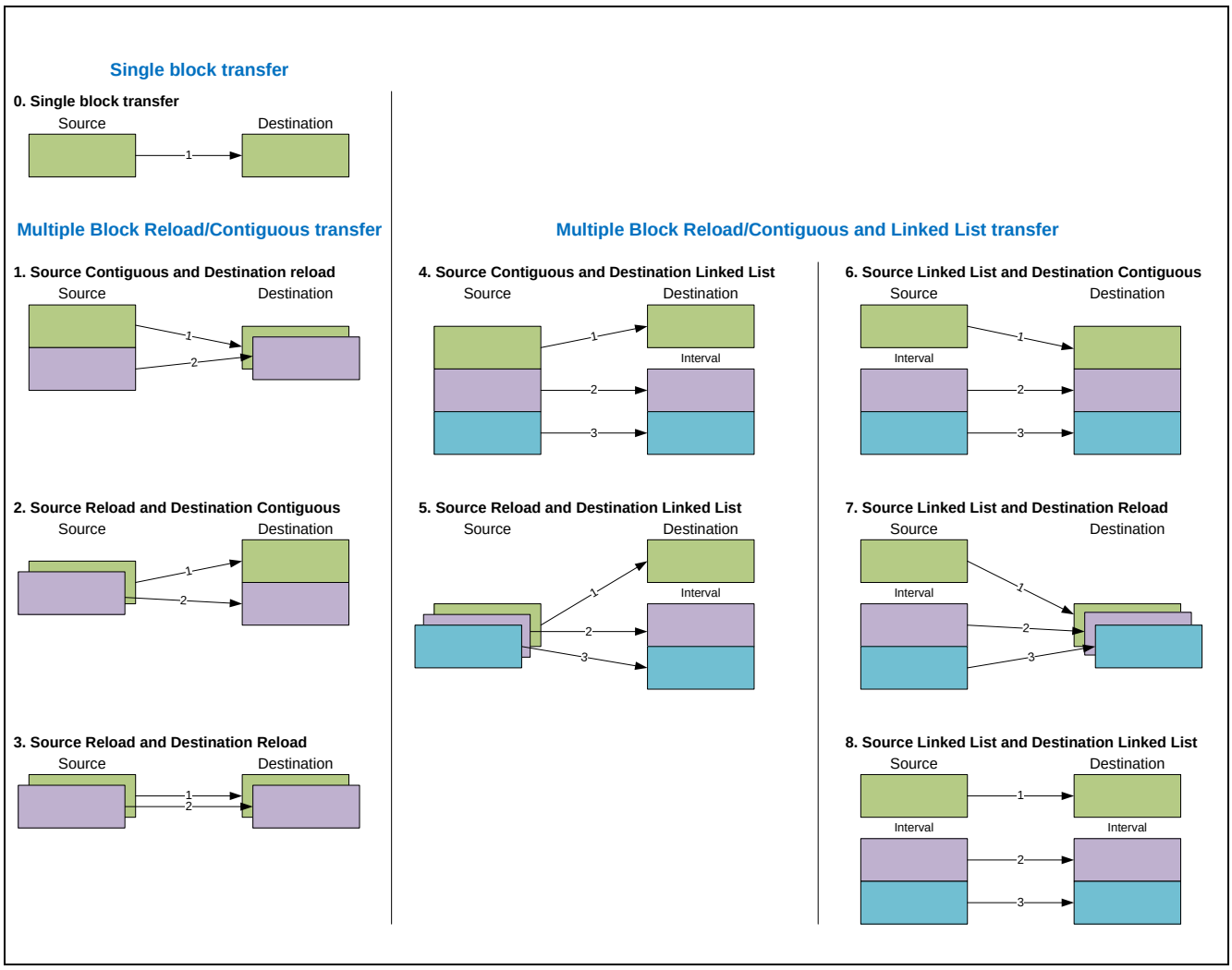


Figure 11 Types of block transfer

5.1 Auto reloading of channel register

For Auto reload method, means that the GPDMA channel registers **SAR, DAR** and **CTL** are reloaded with their initial values at the completion of each block; hence there is no change in the **source** and **destination** address. To configure for auto reload mode set **CFGL.RELOAD_SRC = 1** and/or **CFGL.RELOAD_DST = 1**

5.2 Contiguous address between blocks

In this case the address between successive blocks is selected as a continuation from the end of the previous block. To configure for contiguous method set **CFGL.RELOAD_SRC = 0 or CFGL.RELOAD_DST = 0.** Please note that either the source or destination can be contiguous. However if register is set as **CFGL.RELOAD_SRC = 0 and CFGL.RELOAD_DST = 0** will result to a single block transfer method.

5.3 Block chaining using linked lists

To extend the flexibility of multi block transfer, we can reuse the concept of reloading the **SAR**, **DAR, CTLH** and **CTLL** registers through the use of linked list. A chain of linked lists is pre programmed into the XMC flash memory before the start of DMA transfer and each linked list contains the configuration of the **CTLH**, **CTLL**, **LLP**, **DAR** and **SAR**. During the process of multi block transfer, the GPDMA will reprogram the channel registers (CTLH, CTLL, LLP, DAR and SAR) according to the value stated in the linked list for that block prior to the start of each block transfer. For the next block transfer, the GPDMA will reference the LLP register which points to the next linked list.

Hence linked list transfer offers a very flexible way of DMA block transfer from different locations of the source and destination.

Note: Linked list are only available on channels 0 and 1 of GPDMA0

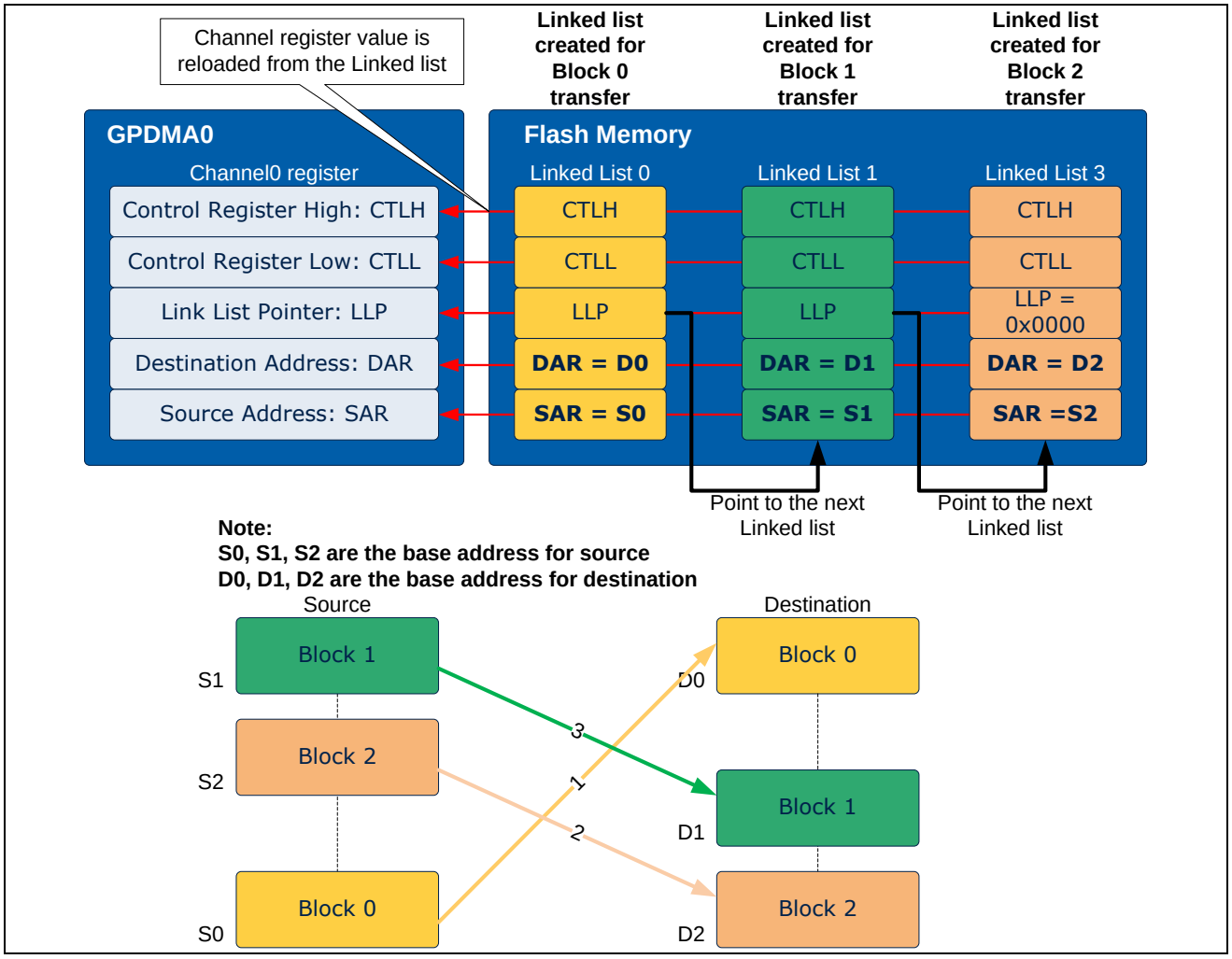


Figure 12 Multi block transfer using Linked List method

6 Service Request Generation

The following DMA Events can be generated for each channel due to DMA activity Block Transfer Complete Interrupt

- DMA Transfer Complete Interrupt
- Destination Transaction Complete Interrupt
- Source Transaction Complete Interrupt.
- Error Interrupt.

Each DMA Event for each channel is directly stored in the according “RAW Status” bit.The user software can control the processing by writing to the according “Mask” and “Clear” bits. Once the event is forwarded to the “Status” bit its occurrence is registered in the Combined Interrupt Status Register and a service request is triggered to the NVIC which is linked to the interrupt nodes of **GPDMA0.SR0 or GPDMA1.SR0**.

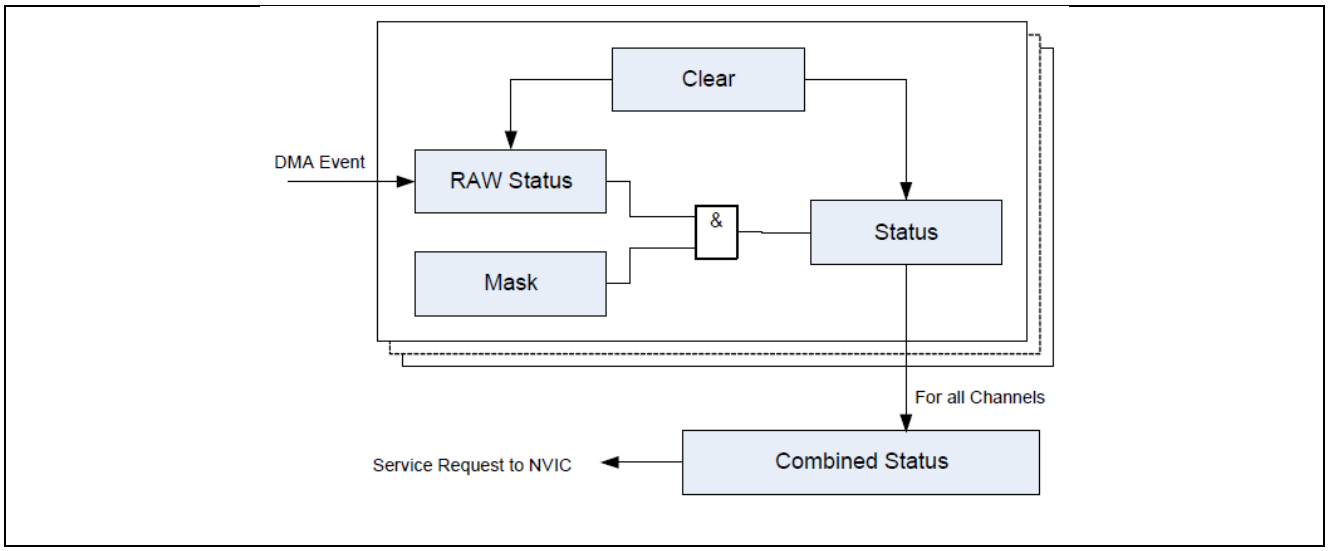


Figure 13 DMA Event to Service Request Flow

7 Multi block transfer of VADC result registers to RAM for motor control

The CCU8 is used to generate the center align PWM for IGBT switching. **[1]** Hence during CCU8 period match interrupt, **CCU80.SR2** signal will be used to trigger VADC queue source conversion to measure the line current and back EMF. **[2]** After completion of the ADC conversion, the result interrupt **VADC.G0SR2** is generated to trigger the GPDMA channel 0 via the DLR. **[3]** Once the assigned channel received the hardware handshaking signal, the GPDMA shall then transfer the values from the VADC result register (Source) to a designated RAM array (Destination). **[4]** Finally a block transfer complete interrupt will be generated by the GPDMA to indicate an update of the ADC result value in the RAM array has been done and ready for futher processing.

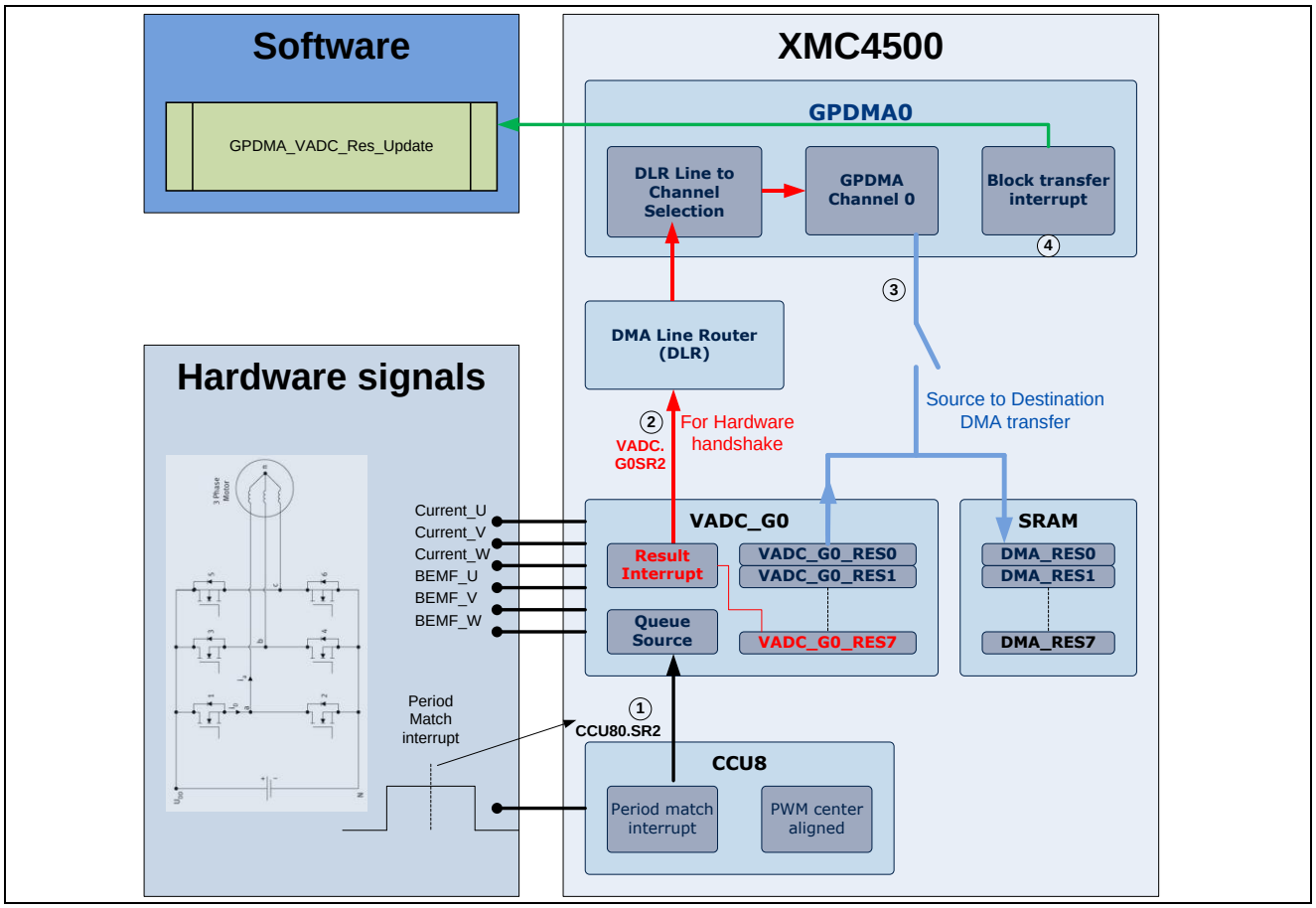


Figure 14 Block diagram for fast VADC result update

Therefore for this use case, we shall make the following configuration for **GPDMA channel 0.**

- Hardware handshake interface: VADC result interrupt signal
- Block transfer size = 8 and Width = 16bit
- Source: VADC result registers [0-7] **VADC_G0_RES0** and Destination: **RAM**
- Gather transfer is used as we only want to extracted 16bit result from the 32bit VADC result register
- Block Transfer type: Source reload and Destination reload
- GPDMA interrupt: Block complete interrupt

7.1 PWM period match interrupt to trigger ADC conversion

Setting up for CCU8 PWM period match interrupt

From “Table 19-13 Digital Connections in the XMC4500” of the reference manual, we understand that the VADC module can be triggered by CCU8 via signals from **CCU80.SR2** or **CCU80.SR3**.

Table 19-13 Digital Connections in the XMC4500 (cont'd)

Signal	Dir.	Source/Destin.	Description
VADC.GxREQTRI	I (s)	CCU80.SR2	Trigger input I
VADC.GxREQTRJ	I (s)	CCU80.SR3	Trigger input J
VADC.GxREQTRK	I (s)	CCU81.SR2	Trigger input K

Figure 15 Table 19-13 Digital Connections in the XMC4500

Therefore for this case we shall configure the CCU8 period match event to be linked to service request node CCU80.SR2 as the triggering signal for VADC conversion.

```
XMC_CCU8_SLICE_EnableEvent(PWM_SLICE_PTR, XMC_CCU8_SLICE_IRQ_ID_PERIOD_MATCH);
XMC_CCU8_SLICE_SetInterruptNode(PWM_SLICE_PTR, XMC_CCU8_SLICE_IRQ_ID_PERIOD_MATCH,
XMC_CCU8_SLICE_SR_ID_2);
NVIC_SetPriority(CCU80_2_IRQn, 3U);
NVIC_EnableIRQ(CCU80_2_IRQn);
```

Setting up for Queue source AD conversion upon external trigger request

This is to enable the ADC for external trigger of XMC_CCU_80_SR2 at the rising edge.

```
XMC_VADC_QUEUE_CONFIG_t g_queue_handle =
{
  .req_src_priority = (uint8_t)3,
  .conv_start_mode  = XMC_VADC_STARTMODE_WFS,
  .external_trigger = (bool) true,
  .trigger_signal   = XMC_CCU_80_SR2,
  .trigger_edge     = XMC_VADC_TRIGGER_EDGE_RISING,
  .gate_signal      = XMC_VADC_REQ_GT_A,
  .timer_mode       = (bool) false,
};
```

The below configuration for Queue entry 0, means that it will start AD conversion on channel number 0 upon receiving an external trigger request.

```
XMC_VADC_QUEUE_ENTRY_t g_queue_entry[VADC_QUEUE_MAX] =
{
      //Queue 0
      {
       .channel_num = 0,
       .refill_needed = true,
       .generate_interrupt = false,
       .external_trigger = true
      },
```

```
}
```

Setting up ADC result event to trigger GPDMA transfer

Next we need to enable the Result Event of the last AD conversion channel in the queue. This is to indicate that the conversion has completed for all channels and the results are ready. Hence we can then route this signal to trigger the GPDMA for transfer.

```
XMC_VADC_RESULT_CONFIG_t g_result_handle[VADC_RES_MAX] =
{

	// Result register 7
	{
		.post_processing_mode   = XMC_VADC_DMM_REDUCTION_MODE,
		.data_reduction_control = 0,
		.part_of_fifo           = false,/* No FIFO */
		.wait_for_read_mode     = true, /* WFS */
		.event_gen_enable       = true  /* Result event */
	},

}
```

For this ADC Result Event, we shall link it to service request node **VADC.G0SR2**.

```
XMC_VADC_GROUP_SetResultInterruptNode(VADC_G0, 7, XMC_VADC_SR_GROUP_SR2);
NVIC_SetPriority(VADC0_G0_2_IRQn, VADC0_G0_2_IRQn_10);
NVIC_EnableIRQ(VADC0_G0_2_IRQn);
```

7.2 GPDMA configuration for multiple block transfer

To begin the GPDMA configuration we had to create a channel configuration structure as shown below. After that the initialization parameters of the DMA channel shall be configured within this structure.

```
XMC_DMA_CH_CONFIG_t GPDMA0_Ch0_config =
{
       .enable_interrupt = true,
       .dst_transfer_width = XMC_DMA_CH_TRANSFER_WIDTH_16,
       .src_transfer_width = XMC_DMA_CH_TRANSFER_WIDTH_16,
       .dst_address_count_mode = XMC_DMA_CH_ADDRESS_COUNT_MODE_INCREMENT,
       .src_address_count_mode = XMC_DMA_CH_ADDRESS_COUNT_MODE_INCREMENT,
       .dst_burst_length = XMC_DMA_CH_BURST_LENGTH_8,
       .src_burst_length = XMC_DMA_CH_BURST_LENGTH_8,
       .enable_src_gather = true,
       .enable_dst_scatter = false,
       .transfer_flow = XMC_DMA_CH_TRANSFER_FLOW_P2M_DMA,
       .src_addr = (uint32_t) &VADC_G0->RES[0],
       .dst_addr = (uint32_t) &verify_ADC_Result[0],
       .src_gather_interval = 1,
       .src_gather_count = 1,
       .dst_scatter_interval = 0,
       .dst_scatter_count = 0,
       .block_size = 8,
       .transfer_type = XMC_DMA_CH_TRANSFER_TYPE_MULTI_BLOCK_SRCADR_RELOAD_DSTADR_RELOAD,
       .priority = XMC_DMA_CH_PRIORITY_7,
       .src_handshaking = XMC_DMA_CH_SRC_HANDSHAKING_HARDWARE,
       .src_peripheral_request = DMA0_PERIPHERAL_REQUEST_VADC_G0SR2_1,
};
```

7.2.1 Setting the transfer flow and priority

As hardware handshaking is required only from the source peripheral (VADC). Therefore we can select the transfer flow from **peripheral to memory** (*_P2M*) with **DMA** *_DMA*) as the flow controller.

Since there is no other DMA channel is configured in this case, therefore the priority is not relevant.

```
.transfer_flow = XMC_DMA_CH_TRANSFER_FLOW_P2M_DMA,
.priority = 7,
```

7.2.2 Enable the Hardware handshaking

When the VADC (Source) is ready, it shall instruct the GPDMA to transfer data from the result registers to the GPDMA FIFO. For this case hardware handshaking is necessary for the source. Next to transfer data from the GPDMA FIFO to memory (Destination), no handshaking is required for the destination as the memory is always in ready state.

```
.src_handshaking = XMC_DMA_CH_SRC_HANDSHAKING_HARDWARE,
```

7.2.3 Selecting the service request via the DMA Line Router

After the hardware handshake is enabled, the next thing is to route the service request signal **VADC.G0SR2 (VADC result event)** to GPDMA channel 0. This can be done by simply searching for **`VADC_G0SR2 in the`** file **xmc_dma_map.h** and assigned the macro to the configuration structure as shown below:-

```
.src_peripheral_request = DMA0_PERIPHERAL_REQUEST_VADC_G0SR2_1,
```

7.2.4 Source configuration

The source consists of **8** VADC result registers from VADC_G0RES0 to VADC_G0RES7 (block size = 8) with individual 16 bit RESULT (transfer width = 16bit) within the 32bit register. From the assigned base address of the VADC_G0_RES_ADDRESS, the GPDMA will increament according to the programmed transfer width to the next address location.

Note: Block size configuration is the same for both the source and destination

```
.block_size = 8,
.src_addr = VADC_G0_RES_ADDRESS,
.src_transfer_width = XMC_DMA_CH_TRANSFER_WIDTH_16,
.src_address_count_mode = XMC_DMA_CH_ADDRESS_COUNT_MODE_INCREMENT,
```

Note: One word is equal to 16bit

As only the 16bit RESULT is required out of the 32bit result register, therefore we can program the GPDMA to transfer one word (Gather count = 1) from the start of the base address followed by interval of one word (Gather Interval =1) for the next word to be transferred.

```
.enable_src_gather = true,
.src_gather_count = 1,
.src_gather_interval = 1,
```

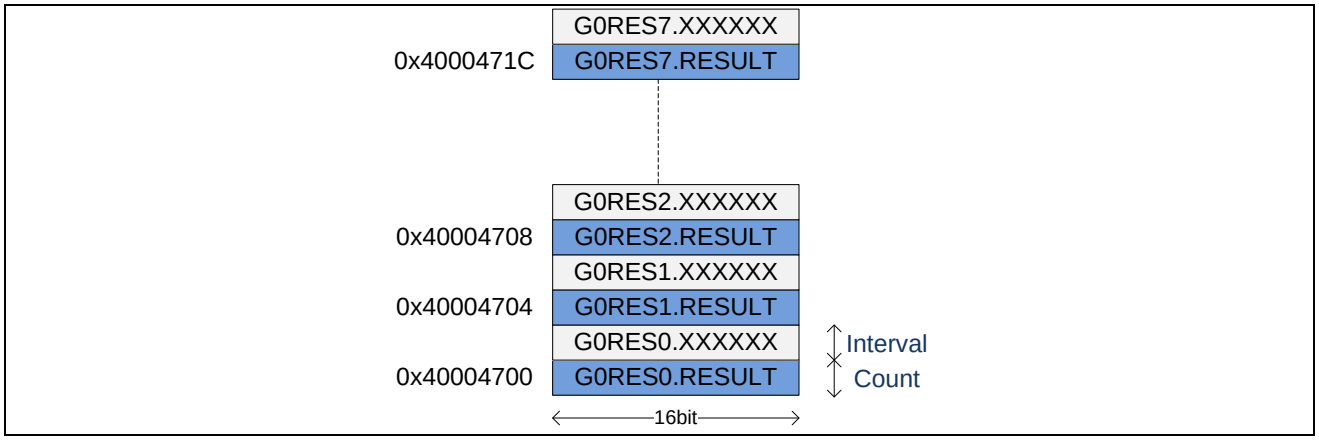


Figure 16 Gather transfer

7.2.5 Destination configuration

An array of 8 words (transfer width = 16bit) is created for the storage of the converstion result. Therefore the destination is configured with the base address (`&DMA_ADC_Result[0]`)of the array with incremental count.

```
.dst_addr = (uint32_t)&DMA_ADC_Result[0],
.dst_transfer_width = XMC_DMA_CH_TRANSFER_WIDTH_16,
.dst_address_count_mode = XMC_DMA_CH_ADDRESS_COUNT_MODE_INCREMENT,
```

7.2.6 Configure the block transfer type

The GPDMA is programmed for multiblock source reload and destination reload.

```
.transfer_type = XMC_DMA_CH_TRANSFER_TYPE_MULTI_BLOCK_SRCADR_RELOAD_DSTADR_RELOAD,
```

7.2.7 Block transfer complete Interrupt

After a successful DMA block transfer of the VADC result registers to the designated RAM location, the user is notified with a block transfer complete interrupt from the GPDMA.

```
.enable_interrupt = true

void GPDMA_Init(void)
{
      XMC_DMA_CH_EnableEvent(XMC_DMA0, CH0, XMC_DMA_CH_EVENT_BLOCK_TRANSFER_COMPLETE);
      NVIC_SetPriority(GPDMA0_0_IRQn, 11); //Priority no.11
      NVIC_EnableIRQ(GPDMA0_0_IRQn);
}
```

User can place their codes in the interrupt routine to process the value of the result registers.

```
#define GPDMA_BlockTransfer_ISR              GPDMA0_0_IRQHandler
void GPDMA_BlockTransfer_ISR(void)
{
  uint32_t event;

  event = XMC_DMA_CH_GetEventStatus(XMC_DMA0,CH0);

  if(event == XMC_DMA_CH_EVENT_BLOCK_TRANSFER_COMPLETE)
  {
      XMC_DMA_CH_ClearEventStatus(XMC_DMA0, CH0,
      XMC_DMA_CH_EVENT_BLOCK_TRANSFER_COMPLETE);
  }

  /* Coding to process array verify_ADC_Result[0] */
}
```

8 Single block transfer CRC checking

In this use case, **[1]** the GPDMA is used to feed the Flexible CRC Engine (FCE) to calculate the CRC-32 on a fictitious frame of data. In this process the CRC-32 of a frame of 256 words is calculated. **[2]** When the block transfer is completed an interrupt will be triggered. **[3]** And in the interrupt handler, the GPDMA shall compare the values of the calculated CRC with a pre determined CRC value stored in the CRC check register. If the calculated CRC matches the expected one, then the status of a variable will be updated as PASS.

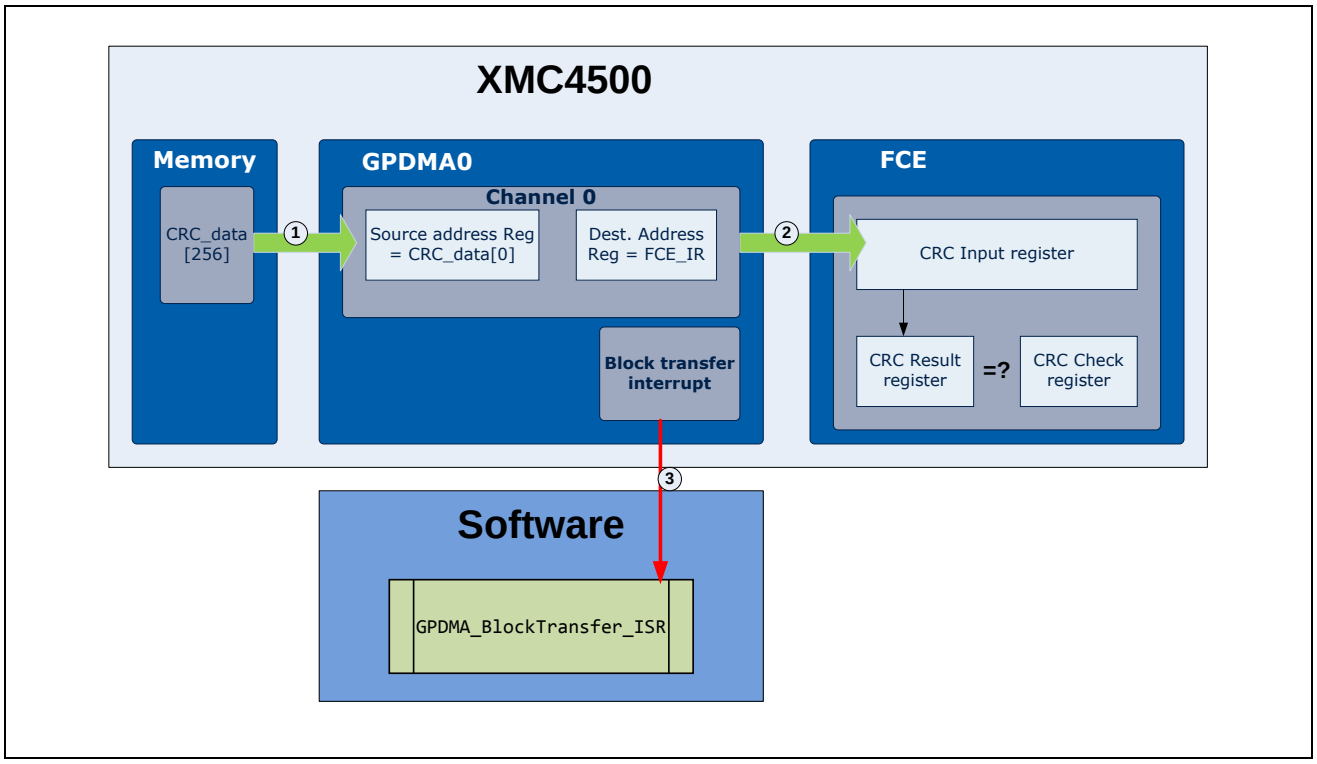


Figure 17 FCE upload with DMA

Therefore for this use case, we shall make the following configuration for **GPDMA channel 0.**

- Block transfer size = 256 and Width = 32bit
- Source: CRC_Data (Memory) and Destination: CRC input register
- Block Transfer type: Single block transfer
- GPDMA interrupt: Block complete interrupt

8.1 Setting up FCE for CRC checking

The FCE is configured to use CRC Kernel 0, which is a CRC32 with ethernet polynomial of 0x04C11DB7. And the initial value (seedvalue) is configured as 0xffffffff.

```
XMC_FCE_t crc_engine =
{
  .kernel_ptr = FCE_KE0,
  .fce_cfg_update.config_xsel = true,
  .fce_cfg_update.config_refin = true,
  .fce_cfg_update.config_refout = true,
  .seedvalue = 0xffffffffU
};
```

The check value is aslo input into the CRC Check register such that the FCE hardware can do a comparison with the CRC Result register to determine if they are matching.

```
#define CRC_CHECK_VALUE                0x99f69cd9
/* Set expected CRC-32 */
XMC_FCE_UpdateCRCCheck(&crc_engine, CRC_CHECK_VALUE);
```

8.2 GPDMA configuration for Single block transfer

For this use case single block DMA transfer is being used with transfer flow from Memory to Memory as handshaking is not required due to the reason that both the memory and FCE are in ready state.

```
XMC_DMA_CH_CONFIG_t GPDMA0_Ch0_config =
{
.enable_interrupt = true,
.dst_transfer_width = XMC_DMA_CH_TRANSFER_WIDTH_32,
.src_transfer_width = XMC_DMA_CH_TRANSFER_WIDTH_32,
.dst_address_count_mode = XMC_DMA_CH_ADDRESS_COUNT_MODE_NO_CHANGE,
.src_address_count_mode = XMC_DMA_CH_ADDRESS_COUNT_MODE_INCREMENT,
.dst_burst_length = XMC_DMA_CH_BURST_LENGTH_8,
.src_burst_length = XMC_DMA_CH_BURST_LENGTH_8,
.enable_src_gather = false,
.enable_dst_scatter = false,
.transfer_flow = XMC_DMA_CH_TRANSFER_FLOW_M2M_DMA,
.src_addr = (uint32_t) &CRC_data,
.dst_addr = (uint32_t) &(FCE_KE0->IR),
.src_gather_interval = 0,
.src_gather_count = 0,
.dst_scatter_interval = 0,
.dst_scatter_count = 0,
.block_size = 256,
.transfer_type = XMC_DMA_CH_TRANSFER_TYPE_SINGLE_BLOCK,
.priority = XMC_DMA_CH_PRIORITY_0,
};
```

8.2.1 Setting the transfer flow and priority

The FCE and memory does not require handshaking, as they are always ready to accept new data. Therefore transfer flow **memory to memory** (*_M2M*) with **DMA** as the flow controller (*_DMA*) is chosen. For priority setting, it is not relevant since there is no other DMA channel is configured in this case.

```
.transfer_flow = XMC_DMA_CH_TRANSFER_FLOW_M2M_DMA,
.priority = XMC_DMA_CH_PRIORITY_0,
```

8.2.2 Source configuration

The source refers to a 32bit array **CRC_data** with a size of 256 words (block size = 256) for CRC checking. Therefore the base address and transfer width can be configured as shown below with address increment after every single word.

```
.block_size = 256,
.src_addr = (uint32_t) &CRC_data,
.src_transfer_width = XMC_DMA_CH_TRANSFER_WIDTH_32,
.src_address_count_mode = XMC_DMA_CH_ADDRESS_COUNT_MODE_INCREMENT,
```

8.2.3 Destination configuration

And the destination refers to the FCE input register which has a data width of 32 bit. As we want to transfer always to the input register of the FCE so there is no change to the address after every transaction.

```
.dst_addr = (uint32_t) &(FCE_KE0->IR),
.dst_transfer_width = XMC_DMA_CH_TRANSFER_WIDTH_32,
.dst_address_count_mode = XMC_DMA_CH_ADDRESS_COUNT_MODE_NO_CHANGE,
```

8.2.4 Configure the block transfer type

The single block transfer is used and after the transfer is completed the DMA channel will be disabled.

```
.transfer_type = XMC_DMA_CH_TRANSFER_TYPE_SINGLE_BLOCK,
```

8.2.5 Block transfer complete Interrupt

After a successful DMA block transfer of the CRC_data to the FCE input register, the user is notified with a block transfer complete interrupt from the GPDMA.

```
.enable_interrupt = true
#define GPDMA0_0_IRQn_11                              11
void GPDMA_Init(void)
{
      XMC_DMA_CH_EnableEvent(XMC_DMA0, CH0, XMC_DMA_CH_EVENT_BLOCK_TRANSFER_COMPLETE);
      NVIC_SetPriority(GPDMA0_0_IRQn, GPDMA0_0_IRQn_11);
      NVIC_EnableIRQ(GPDMA0_0_IRQn);
}
```

Single block transfer CRC checking

With this interrupt, the user can instruct the GPDMA to compare the value of the CRC result register with the CRC check register. If both values are the same the “verify_CRC_check” shall be updated with PASS.

```
#define GPDMA_BlockTransfer_ISR              GPDMA0_0_IRQHandler
void GPDMA_BlockTransfer_ISR(void)
{
      if (XMC_FCE_GetEventStatus(&crc_engine, XMC_FCE_STS_MISMATCH_CRC) == false)
      {
            verify_CRC_check = PASS;
      }
      else
      {
            verify_CRC_check = FAIL;
      }
}
```

9 Revision History

Current Version is V1.0, 2015-07

Page or Reference	Description of change
V1.0, 2015-07	
	Initial Version

Trademarks of Infineon Technologies AG

AURIX™, C166™, CanPAK™, CIPOS™, CIPURSE™, CoolGaN™, CoolMOS™, CoolSET™, CoolSiC™, CORECONTROL™, CROSSAVE™, DAVE™, DI-POL™, DrBLADE™, EasyPIM™, EconoBRIDGE™, EconoDUAL™, EconoPACK™, EconoPIM™, EiceDRIVER™, eupec™, FCOS™, HITFET™, HybridPACK™, ISOFACE™, IsoPACK™, i-Wafer™, MIPAQ™, ModSTACK™, my-d™, NovalithIC™, OmniTune™, OPTIGA™, OptiMOS™, ORIGA™, POWERCODE™, PRIMARION™, PrimePACK™, PrimeSTACK™, PROFET™, PRO-SIL™, RASIC™, REAL3™, ReverSave™, SatRIC™, SIEGET™, SIPMOS™, SmartLEWIS™, SOLID FLASH™, SPOC™, TEMPFET™, thinQ!™, TRENCHSTOP™, TriCore™.

Other Trademarks

Advance Design System™ (ADS) of Agilent Technologies, AMBA™, ARM™, MULTI-ICE™, KEIL™, PRIMECELL™, REALVIEW™, THUMB™, µVision™ of ARM Limited, UK. ANSI™ of American National Standards Institute. AUTOSAR™ of AUTOSAR development partnership. Bluetooth™ of Bluetooth SIG Inc. CAT-iq™ of DECT Forum. COLOSSUS™, FirstGPS™ of Trimble Navigation Ltd. EMV™ of EMVCo, LLC (Visa Holdings Inc.). EPCOS™ of Epcos AG. FLEXGO™ of Microsoft Corporation. HYPERTERMINAL™ of Hilgraeve Incorporated. MCS™ of Intel Corp. IEC™ of Commission Electrotechnique Internationale. IrDA™ of Infrared Data Association Corporation. ISO™ of INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. MATLAB™ of MathWorks, Inc. MAXIM™ of Maxim Integrated Products, Inc. MICROTEC™, NUCLEUS™ of Mentor Graphics Corporation. MIPI™ of MIPI Alliance, Inc. MIPS™ of MIPS Technologies, Inc., USA. muRata™ of MURATA MANUFACTURING CO., MICROWAVE OFFICE™ (MWO) of Applied Wave Research Inc., OmniVision™ of OmniVision Technologies, Inc. Openwave™ of Openwave Systems Inc. RED HAT™ of Red Hat, Inc. RFMD™ of RF Micro Devices, Inc. SIRIUS™ of Sirius Satellite Radio Inc. SOLARIS™ of Sun Microsystems, Inc. SPANSION™ of Spansion LLC Ltd. Symbian™ of Symbian Software Limited. TAIYO YUDEN™ of Taiyo Yuden Co. TEAKLITE™ of CEVA, Inc. TEKTRONIX™ of Tektronix Inc. TOKO™ of TOKO KABUSHIKI KAISHA TA. UNIX™ of X/Open Company Limited. VERILOG™, PALLADIUM™ of Cadence Design Systems, Inc. VLYNQ™ of Texas Instruments Incorporated. VXWORKS™, WIND RIVER™ of WIND RIVER SYSTEMS, INC. ZETEX™ of Diodes Zetex Limited.

Last Trademarks Update 2014-07-17

www.infineon.com

Edition 2015-07

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2015 Infineon Technologies AG. All Rights Reserved.

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference

AP32290

Legal Disclaimer

THE INFORMATION GIVEN IN THIS APPLICATION NOTE (INCLUDING BUT NOT LIMITED TO CONTENTS OF REFERENCED WEBSITES) IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED 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.

Information

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

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office. Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.