

Using the DMA module in Kinetis devices

By: Technical Information Center

1 Introduction

Direct memory access (DMA) allows data transfers between main memory and a peripheral device (such as UARTs) without having each word (byte) handled by the CPU. While the data transfers are being carried out the CPU is free to perform other operations as long as these operations do not require any of the buses used by the DMA controller. Thus DMA provides a way to optimize overall system performance, increase data throughput and offload expensive data operations.

The following example addresses basic DMA controller usage within the K2xx family of microcontrollers. The main goal of this example is to introduce the necessary notions of DMA controlled data transfers, in order to do so a description of the main registers is given as well as two implementation examples.

Contents

Using the DMA module in Kinetis devices.....	1
1 Introduction.....	1
2 eDMA.....	2
3 Direct Memory Access Multiplexer.....	2
4 Functional description.....	3
5 DMA initialization example.....	8
6 DMA Examples.....	15
7 Appendix.....	24
8 References.....	27

2 eDMA

The eDMA module is partitioned into two major modules: the eDMA engine and the transfer-control descriptor local memory. The eDMA block diagram is being shown in the following figure.

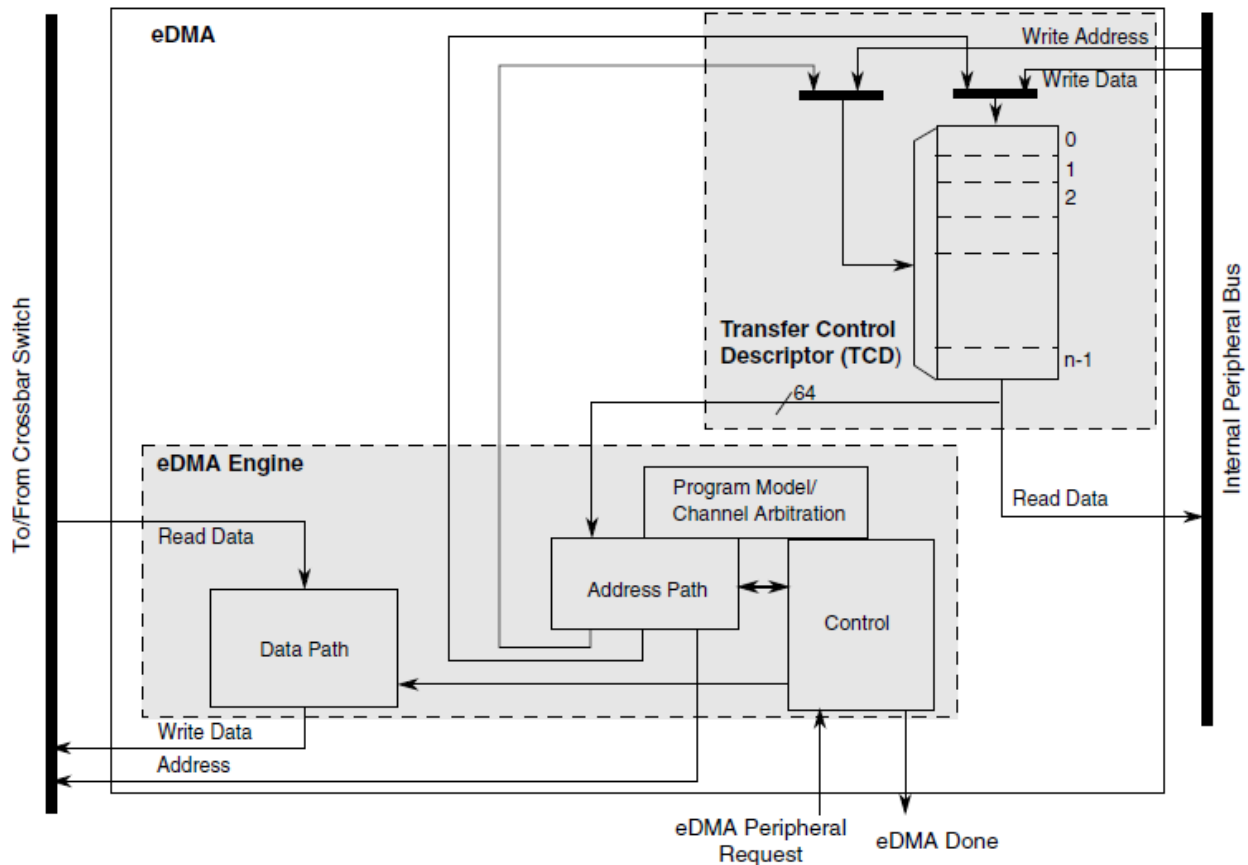


Figure 1- eDMA block diagram

The TCD contains all the necessary information for the transfer to be successful, e.g. the source- and destination- address, number of bytes to be transferred, the size of transference per request, number of total transfers, offset applied to every address read and write and adjustment values for the source- and destination-address once the transfer has been done. Meanwhile the eDMA engine performs all the necessary calculations and controls to read/write the required data.

3 Direct Memory Access Multiplexer

The direct memory access multiplexer (DMAMUX) routes DMA sources to any of the 16 DMA channels. This process is illustrated in the following figure.

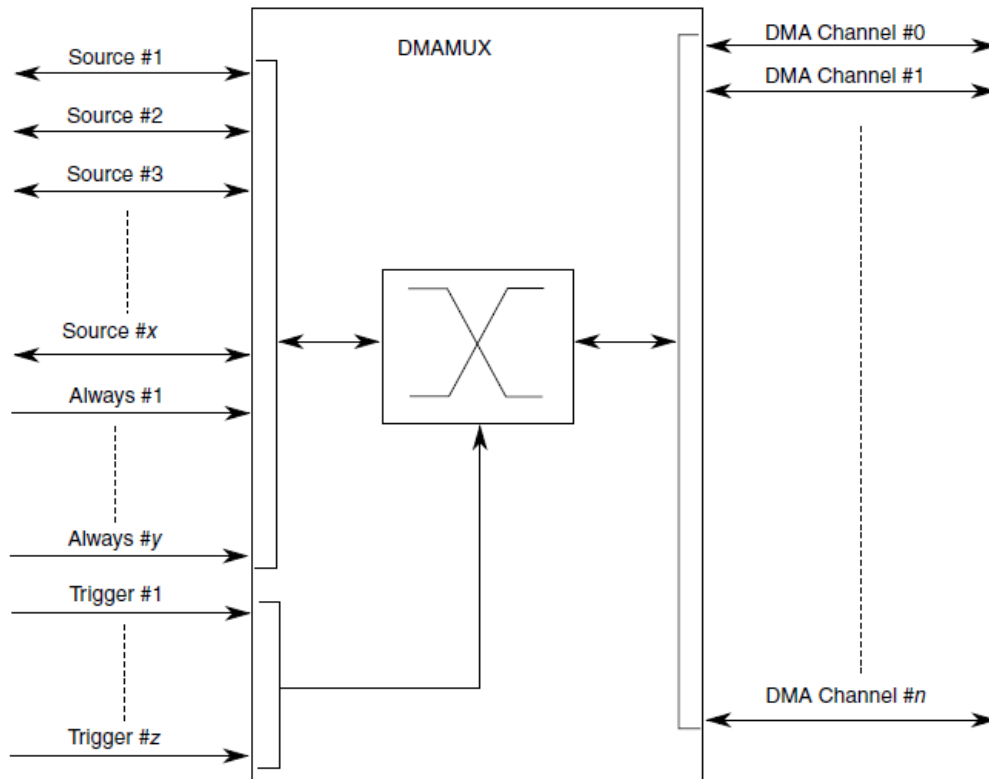


Figure 2 - DMAMUX block diagram

The DMA channel MUX provides these features:

- 52 peripheral slots and 10 always-on slots can be routed to 16 channels.
- 16 independently selectable DMA channel routers.
 - The first 4 channels additionally provide a trigger functionality.
- Each channel router can be assigned to one of the 52 possible peripheral DMA slots or to one of the 10 always-on slots.

4 Functional description

The control of the DMA module can be split into three levels:

- Basic transfer
- Minor loop
- Major loop

Basic transfer is the process in which the required source reads and destination writes occur in order to move the data, as well as source- and destination-address offset adjustments and decreasing of the number of bytes to be transferred happen. The minor loop is in charge of issuing another basic transfer if the number of bytes to be transferred have not been accomplished (bytes to be transferred = 0). The major loop contains the number of minor loops necessary to move the whole data. The number of minor loops in a major loop is specified by the beginning iteration count (BITER). Every time a minor loop terminates the current major loop

iteration (CITER) decreases by one. Once the CITER reaches a value of 0 the whole data have been successfully transferred. These levels are illustrated in the following figure.

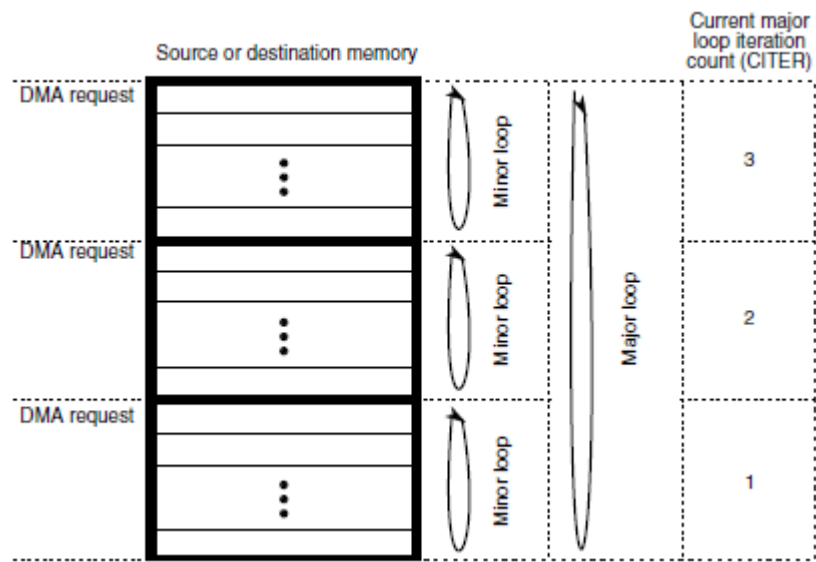


Figure 3- Example of multiple loop iterations

In this example a peripheral request signal is being used to request service for channel n. When a peripheral request signal is issued the control module in the eDMA handles it in conjunction with the address path module in order to get the corresponding TCD descriptor (for the active channel) and load its information to the address path module. This process is illustrated in the following figure.

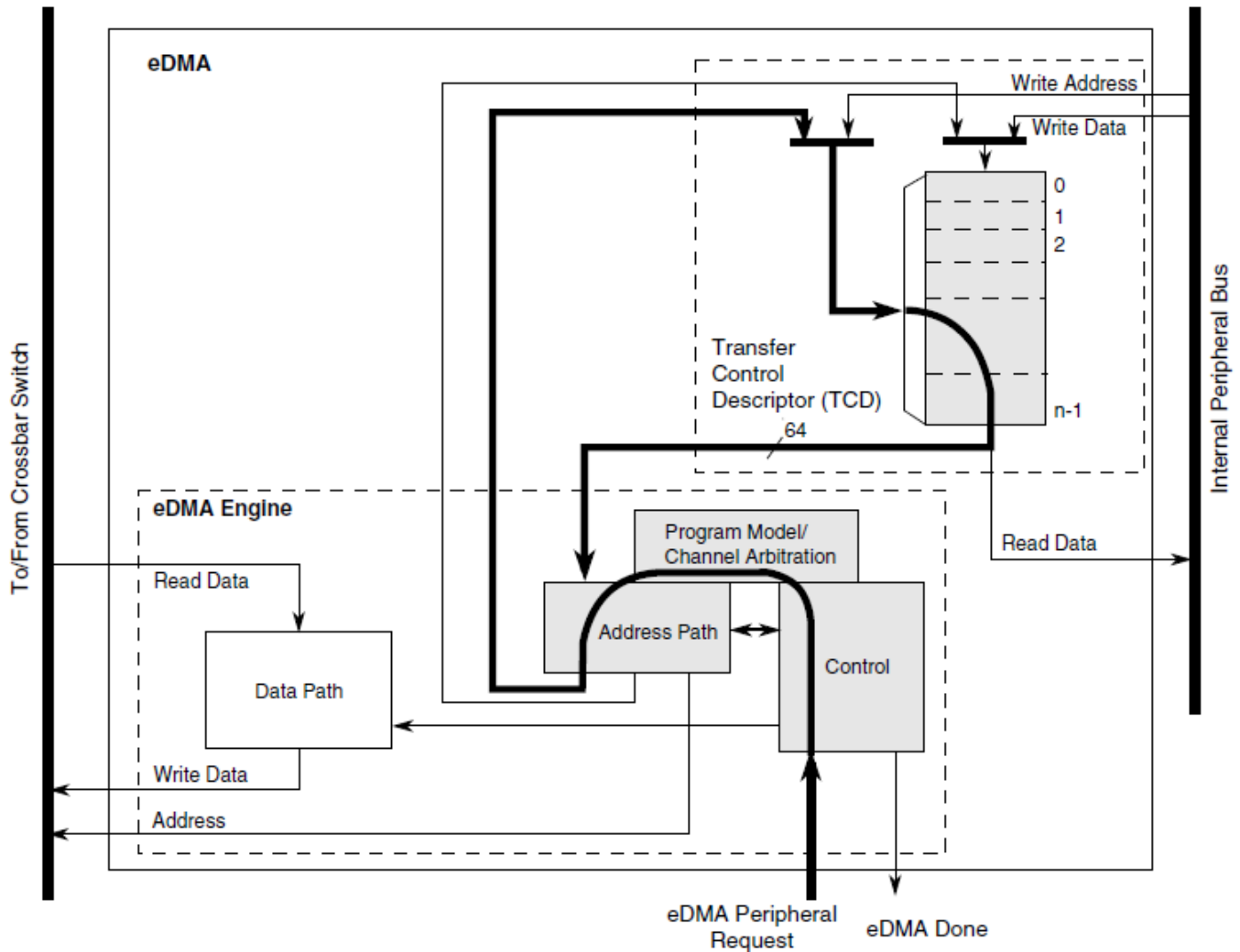


Figure 4- eDMA operation, part 1

Once the source- and destination-address, read/write offsets and number of bytes to be transferred have been successfully loaded into the address path module, the control and data path module perform basic transfers. The source reads are initiated and the fetched data is temporarily stored in the data path block until it is gated onto the internal bus during the destination write. This source read/destination write processing continues until the minor loop terminates. The following figure illustrates this process.

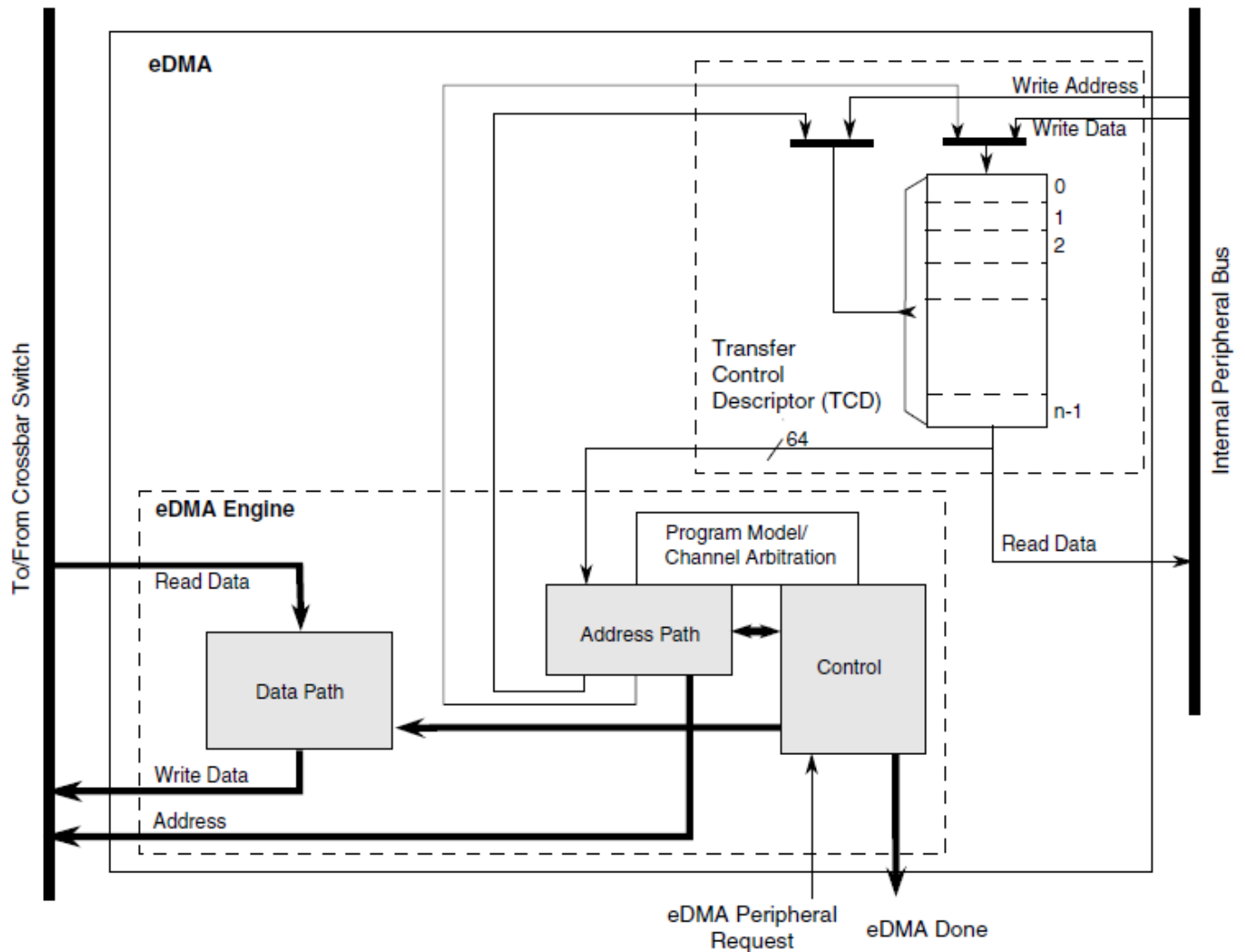


Figure 5- eDMA operation, part 2

Once the minor loop has terminated the address path logic performs the required updates to certain fields in the appropriate TCD, e.g., source- and destination-address and CITER decrement. This process continues (DMA request followed by basic transfers until the minor loop is performed) until the CITER equals zero. Once the CITER equals zero the final phase of the basic data flow is performed. In this segment the following operations are carried out: final address adjustments, reloading of minor loops register (BITER) into major loops register (CITER) and assertion of an optional interrupt request if enabled. This process is illustrated in the following figure.

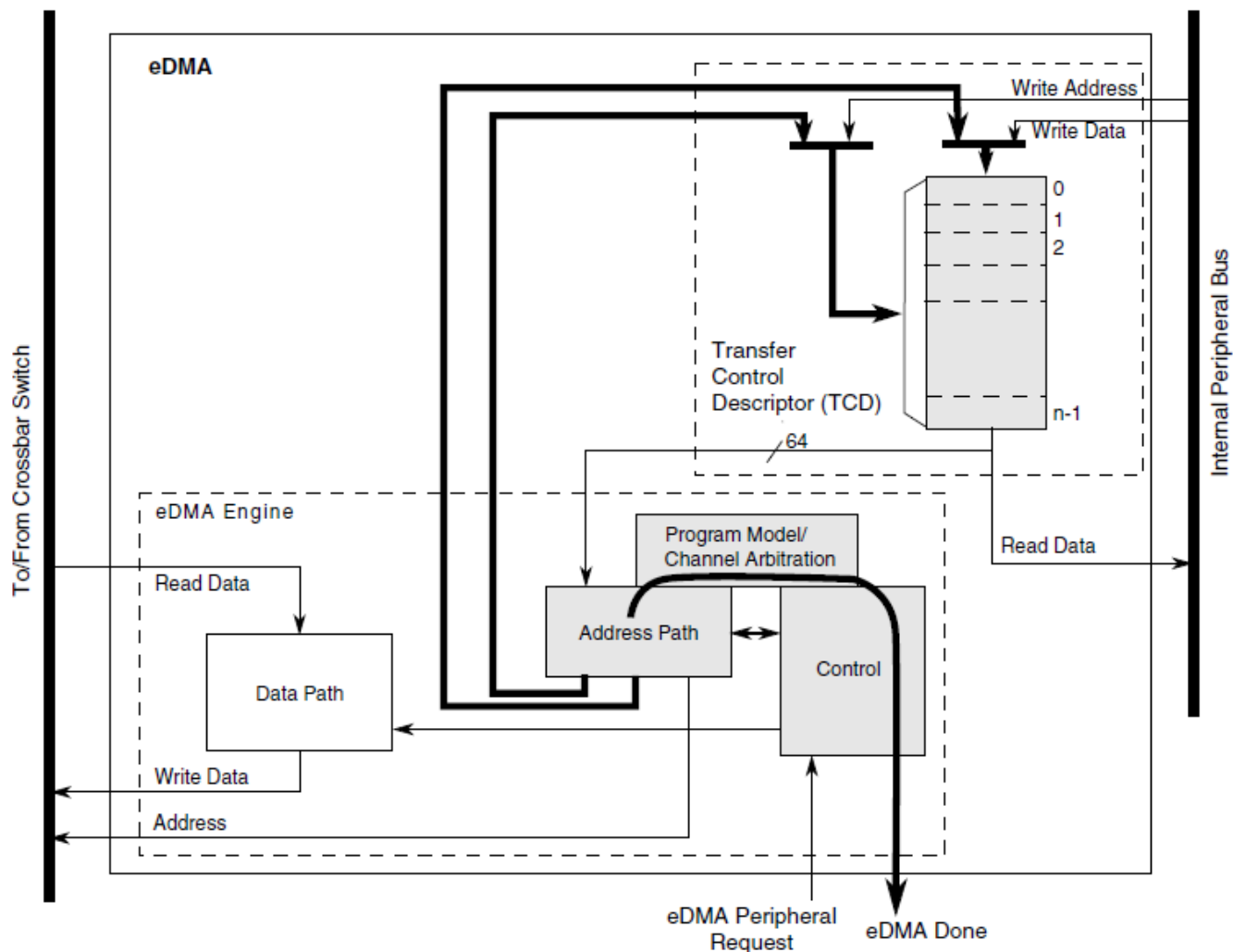


Figure 6- eDMA operation, part 3

The basic function of eDMA is summarized as follows:

1. The DMA channel with right setting of all parameters, waits for the Start control (DMA_TCDn_CSR field start) bit to get asserted.
2. When the Start bit is asserted, the eDMA engine starts the initial Major loop.
3. The Major loop starts the first Minor loop. The Minor loop waits for the peripheral request.
4. Peripheral request starts the Minor loop which in turn, starts the Basic transfer.
5. Basic transfer reads and writes from a source address to a destination address.
6. The Minor loop decreases the number of bytes to be transferred. If the number of bytes to be transferred = 0, the minor loop ends, else the next Basic transfer starts.
7. Minor loop that is finished decreases the CITER counter.
8. If the CITER counter = 0, then the Major loop is finished, otherwise the eDMA waits for asserting the next Start bit.

5 DMA initialization example

In this section initialization of the eDMA and programming considerations are discussed. The following initialization example is based upon both of the examples that can be found in the [Appendix](#). The approach that will be taken may differ from other methods, this is intended as a DMA overview to help the readers familiarize themselves with the eDMA controller.

1. First of all the clock must be enabled for both the DMA multiplexer and the DMA itself. This is done by setting the corresponding field in the system clock gating register (SCGC) that contains the DMA and DMAMUX. The information can be found within the reference manual of the corresponding Kinetis family. In this example a Kinetis K20 is being used, therefore the corresponding registers are SCGC6 for DMAMUX and SCGC7 for DMA as can be seen in the following figures.

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0	1		0		0						0				0
W			RTC		ADC0		FTM1	FTM0	PIT	PDB	USBDCD			CRC		
Reset	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R		0			0	0			0					0		
W	I2S		SPI1	SPI0								FLEXCAN0			DMAMUX	FTFL
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 7- SIM_SCGC6

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0														0	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Figure 8-SIM_SCGC7

The CodeWarrior code that performs this task is the following:

```
SIM_SCGC6 |= SIM_SCGC6_DMAMUX_MASK;
SIM_SCGC7 |= SIM_SCGC7_DMA_MASK;
```

2. Once the clocks have been enabled the channel that will be used must be enabled in the DMAMUX and a source for this channel must be selected. Each channel has a configuration register that features an enable field, trigger enable (if the mode is available for the channel) and a source field to specify if

the DMA is routed to a particular source. The channel configuration register is shown in the following figure.

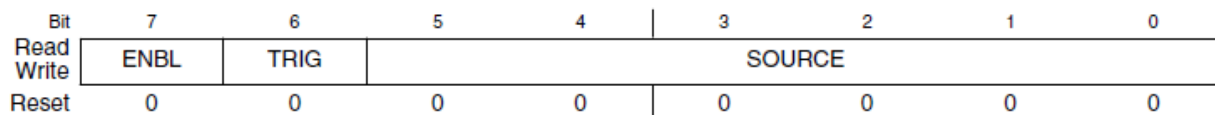


Figure 9- DMAMUX_CHCFG register fields

In this example channel 0 is used with Port C as source. The source number can be found at the reference manual of each sub-family in the chip configuration chapter and DMA MUX request sources section. The corresponding source number for Port C with the K20 is 51 as can be seen in the following table.

Source number	Source module	Source description
48	PDB	—
49	Port control module	Port A
50	Port control module	Port B
51	Port control module	Port C

Table 1- DMA request sources

The CodeWarrior that enables channel 0 with Port C as source is the following.

```
DMAMUX_CHCFG0 |= DMAMUX_CHCFG_ENBL_MASK;    //Enable channel 0
DMAMUX_CHCFG0 |= DMAMUX_CHCFG_SOURCE(51);    //Select Port C as source
```

- In order to be able initiate a DMA transfer when a hardware service request is issued the enable request register (DMA_ERQ) must be set, it is important to notice that the state of the DMA enable request flag does not affect a channel service request made explicitly through software or a linked channel request. The following figure shows the register fields.

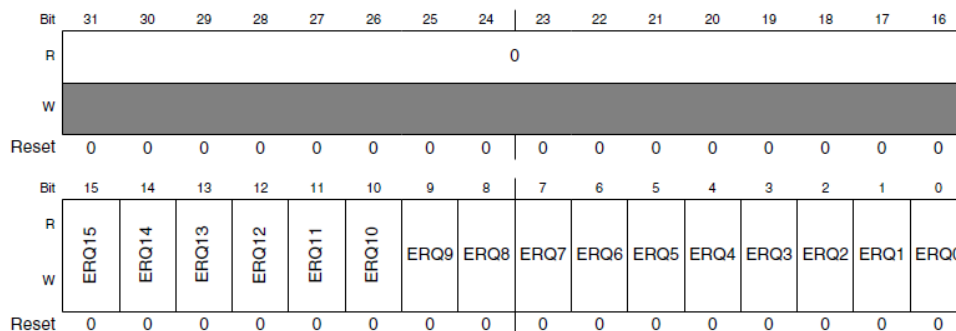
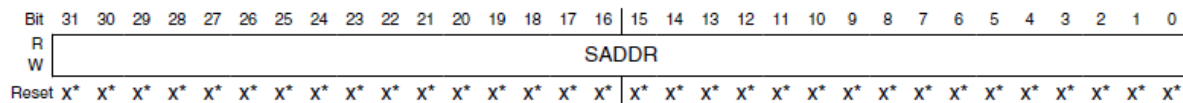


Figure 10-DMA_ERQ fields

To enable the request register that corresponds to channel 0 the ERQ0 field must be set to 1, the CodeWarrior code that does this is the following.

```
DMA_ERQ = DMA_ERQ_ERQ0_MASK;
```

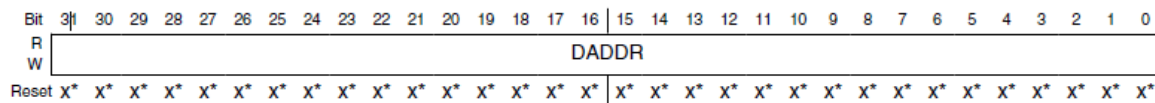
- The source-address and destination-address for the transfer must be specified and set into the corresponding TCD register. The following figures show the register fields for the source- and destination-address.



* Notes:

- x = Undefined at reset.

Figure 11- DMA_TCDn_SADDR fields



* Notes:

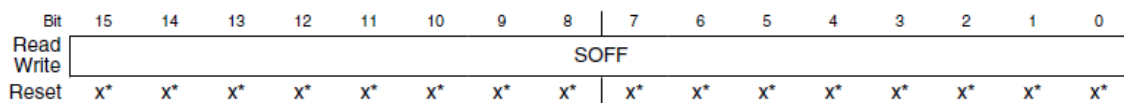
- x = Undefined at reset.

Figure 12-DMA_TCDn_DADDR fields

SADDR holds the memory address that points to the source data and DADDR holds the memory address that points to the destination data. The CodeWarrior code that sets the memory addresses for channel 0 is the following:

```
DMA_TCD0_SADDR = &Source_address;
DMA_TCD0_DADDR = &Destination_address;
```

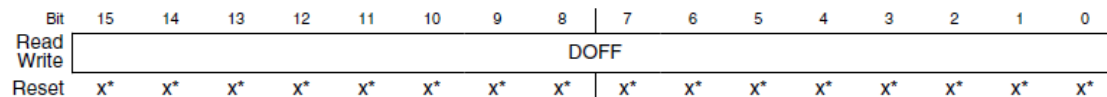
- Once the source- and destination-address have been set the source- and destination-address offset must be specified in the corresponding TCD register. Each address is modified once a basic transfer is completed. The SADDR is updated once a read is completed in the following way: next state SADDR = SADDR + Source Offset (SOFF). The DADDR is updated once a write is completed in the following way: next state DADDR = DADDR + Destination Offset (DOFF). Both registers allow a signed offset in order to perform diverse data manipulations. For instance with a negative offset a backwards reading of the source could be performed or setting the SOFF to zero one could copy several instances of the same byte and so on and so forth. The following figures show the register fields for the source- and destination-address offsets.



* Notes:

- x = Undefined at reset.

Figure 13-DMA_TCDn_SOFF field



* Notes:

- x = Undefined at reset.

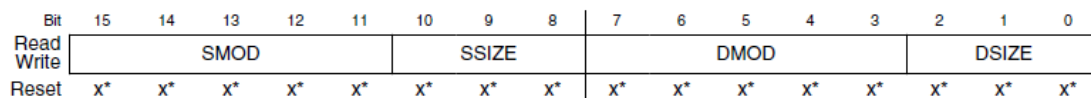
Figure 14--DMA_TCDn_DOFF field

In this example the source-address is being read with 2 bytes intervals meaning that once a byte is read the next one is skipped and the next byte that will be read and written will be the third in the sequence. The destination-address offset is set to one byte per write, thus placing each read byte next to each other. The CodeWarrior code is the following:

```
DMA_TCD0_SOFF = 0x02;
```

```
DMA_TCD0_DOFF = 0x01;
```

- The transfer size must be specified for both the source and the destination, this sets the size of the data to be read and written. The TCD register that keeps this information is the TCD transfer Attributes register (DMA_TCDn_ATTR). This register also features a source modulo and destination modulo field which helps in the implementation of circular data queues. The following figure shows the register fields.



* Notes:

- x = Undefined at reset.

Figure 15- DMA_TCDn_ATTR fields

Field	Description
10–8 SSIZE	Source data transfer size The attempted use of a Reserved encoding causes a configuration error.
000	8-bit
001	16-bit
010	32-bit
011	Reserved
100	16-byte
101	Reserved
110	Reserved
111	Reserved

Figure 16 - SSIZE and DSIZE Field

In this example an 8-bit transfer size for both source and data is being used (see Figure-16). Source- and destination-address modulo are both disabled. The CodeWarrior code is the following:

```
DMA_TCD0_ATTR = DMA_ATTR_SSIZE(0) | DMA_ATTR_DSIZE(0);
```

7. The number of bytes to be transferred in each service request of the channel must be specified. Each minor loop will terminate until the minor byte transfer count has been reached, several basic transfers (read and writes) might be necessary in order to achieve this. Because the normal operation mode is used in this example the corresponding TCD register is the minor byte count (Minor Loop Disabled) or DMA_TCDn_NBYTES_MLNO. If another configuration was chosen the corresponding register might change. The following figure shows the register field.

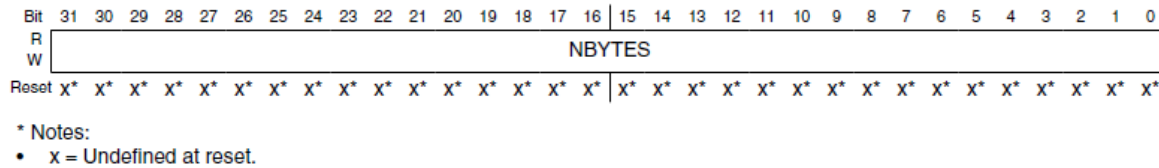


Figure 17- DMA_TCDn_NBYTES_MLNO field

In this example two different configurations can be chosen: 5 bytes per minor loop or 1 byte per minor loop. They must be selected to initialize properly the DMA controller. The following CodeWarrior code is being used in each instance:

```
DMA_TCD0_NBYTES_MLNO = 0x05; // 5 bytes per minor loop
DMA_TCD0_NBYTES_MLNO = 0x01; // 1 byte per minor loop
```

NOTE:

It is important to notice that each configuration is in a separate function and cannot run in parallel.

8. The current major iteration count (CITER) and the beginning iteration count (BITER) must be initialized to the same value. Every time a minor loop is completed the CITER is decreased and once it reaches zero it is reloaded with the value in BITER. If the channel is configured to execute a single service request (a single minor loop), the initial values of BITER and CITER should be 1.

The corresponding registers are TCD Current Minor Loop Link, Major Loop Count (Channel Linking Disabled) or DMA_TCDn_CITER_ELINKNO and TCD Beginning Minor Loop Link, Major Loop Count (Channel Linking Disabled) or DMA_TCDn_BITER_ELINKNO. The following figures show each register fields.

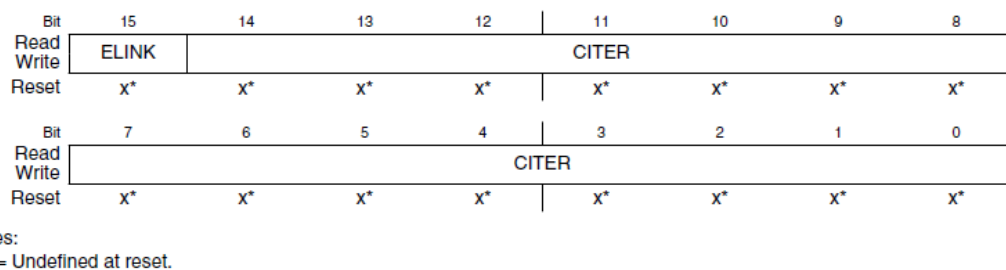
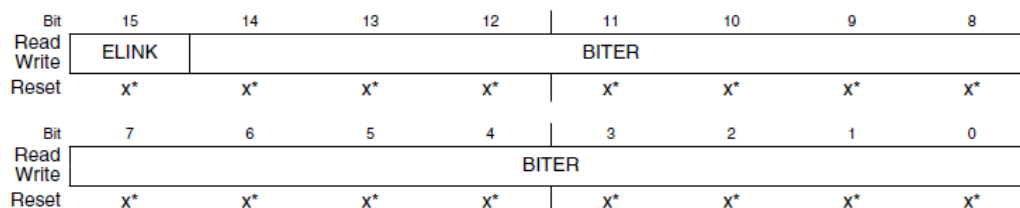


Figure 18-DMA_TCDn_CITER_ELINKNO fields



* Notes:

- x = Undefined at reset.

Figure 19-DMA_TCDn_BITER_ELINKNO fields

When the channel linking is disabled, both CITER and BITER values are extended to 15 bits.

In this example the CITER and BITER initialization will depend upon the configuration previously chosen (5 bytes per minor loop or 1 byte per minor loop).

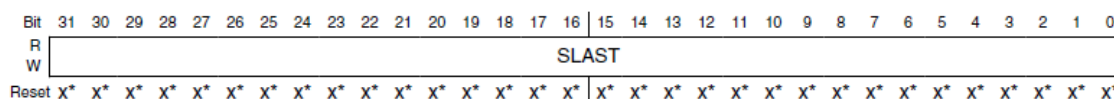
With the 5 bytes configuration a single minor loop is carried out therefore the code used is the following:

```
DMA_TCD0_CITER_ELINKNO = DMA_CITER_ELINKNO_CITER(1);
DMA_TCD0_BITER_ELINKNO = DMA_BITER_ELINKNO_BITER(1);
```

With the single byte configuration 5 minor loops are carried out therefore the code used is the following:

```
DMA_TCD0_CITER_ELINKNO = DMA_CITER_ELINKNO_CITER(5);
DMA_TCD0_BITER_ELINKNO = DMA_BITER_ELINKNO_BITER(5);
```

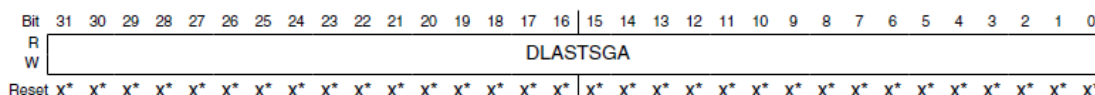
- The source- and destination-address adjustment must be specified. These values will be used once the major loop is completed and their purpose is to restore the addresses to their initial values or set them to reference the next data structure. The corresponding TCD registers are the Last Source Address Adjustment register (DMA_TCDn_SLAST) and the Last Destination Address Adjustment/Scatter Gather Address register (DMA_TCDn_DLASTSGA). The following figures show the register fields for the source- and destination-address adjustment.



* Notes:

- x = Undefined at reset.

Figure 20-DMA_TCDn_SLAST field



* Notes:

- x = Undefined at reset.

Figure 21-DMA_TCDn_DLASTSGA field

In this example since a source offset of 2 bits is being used and 5 bytes are being transferred (no matter what configuration was chosen 5 minor loops or 5 basic transfers will be executed) the required adjustment in order to return to the initial address will be given by $\text{SOFF} * \text{NBYTES} * \text{CITER}$ in both configuration the answer is 10 bits so this must be subtracted to the current address, thus -10 bits (-0x0A in hexadecimal) will be the adjustment value for the source and doing the same thing for the destination ($\text{DOFF} * \text{NBYTES} * \text{CITER} = 5$ bits) gives back an adjustment value of -5. The code that sets these values is the following:

```
DMA_TCD0_SLAST = -0x0A;           // Source address adjustment
DMA_TCD0_DLASTSGA = -0x05;        // Destination address adjustment
```

10. Finally the Control and Status register (DMA_TCDn_CSR) must be setup. The following figure shows the register fields and the following table describes each of the fields.

Bit	15	14	13	12	11	10	9	8
Read	BWC		0		MAJORLINKCH			
Write								
Reset	x*	x*	x*	x*	x*	x*	x*	x*

Bit	7	6	5	4	3	2	1	0
Read	DONE	ACTIVE	MAJORELINK	ESG	DREQ	INTHALF	INTMAJOR	START
Write								
Reset	x*	x*	x*	x*	x*	x*	x*	x*

* Notes:

- x = Undefined at reset.

Figure 22- DMA_TCDn_CSR fields

Field	Description
15-14 BWC	Bandwidth control This field forces the eDMA to stall after the completion of each read/write access to control the bus request bandwidth. 00 No eDMA engine stalls 01 Reserved 10 eDMA engine stalls for 4 cycles after each r/w 11 eDMA engine stalls for 8 cycles after each r/w
13-12 Reserved	This read-only field is reserved and always has the value 0.
11-8 MAJORLINKCH	Link Channel Number
7 Done	This flag indicates the eDMA has completed the major loop.
6 Active	This flag signals the channel is currently in execution. It is set when channel service begins, and the eDMA clears it as the minor loop completes or if any error condition is detected. This bit resets to zero.
5 MAJORELINK	Enable channel-to-channel linking on major loop complete 0 The channel-to-channel linking is disabled 1 The channel-to-channel linking is enabled

4 ESG	Enable Scatter/Gather Processing 0 The current channel's TCD is normal format. 1 The current channel's TCD specifies a scatter gather format. The DLASTSGA field provides a memory pointer to the next TCD to be loaded into this channel after the major loop completes its execution.
3 DREQ	Disable Request 0 The channel's ERQ bit is not affected 1 The channel's ERQ bit is cleared when the major loop is complete
2 INTHALF	Enable an interrupt when major counter is half complete. 0 The half-point interrupt is disabled 1 The half-point interrupt is enabled
1 INTMAJOR	Enable an interrupt when major iteration count completes 0 The end-of-major loop interrupt is disabled 1 The end-of-major loop interrupt is enabled
0 START	Channel Start 0 The channel is not explicitly started 1 The channel is explicitly started via a software initiated service request

Table 2- DMA_TCDn_CSR fields description

This example does not require Bandwidth control, Channel-to-channel linking, scatter/gather processing, interrupts, disable requests (after the major loop completion the hardware request remains active) nor software starts (all DMA request are issued via hardware). Therefore all of these fields must be set to 0. The code that performs this is the following:

```
DMA_TCD0_CSR = 0;
```

6 DMA Examples

Within the provided sample code (Appendix) there are two initializations to choose from, the first one performs a single minor loop with a number of bytes to be transferred of 5 and the second one performs 5 minor loops with a number of bytes to be transferred of 1 per minor loop. From now on the first configuration (single minor loop) will be referred as "Example 1" and the second configuration (five minor loops) will be referred as "Example 2". The following explanation will assume familiarity with the CodeWarrior debug environment and will rely heavily on it.

Both examples features two toggling LEDs for demonstration's sake, this task won't be affected by the DMA transfer as the CPU does not take part in the transfer.

Two buffers are created to illustrate the data transfer, g8bSource represents the data source and g8bDestiny the destination buffer. The arrays are initialized as follows:

```
unsigned char g8bSource[10] = {0,1,2,3,4,5,6,7,8,9};
unsigned char g8bDestiny[10] = {1,0,0,0,0,0,0,0,0,0};
```

These variables are allocated in a space of memory by the Linker, for this example no considerations were taken to place the variables in a special memory allocation. If a specific location is desired this must be specified in the [linker file](#).

The destination array is being initialized so that it will be easier to notice the data movement. Both arrays values as well as their addresses can be seen in the following figures. The figures were taken from CodeWarrior's variables view.

Name	Value	Location
g8bDestiny	0x1fff800c	0x1fff800c
g8bSource	0x1fff8000	0x1fff8000
[0]	0	0x1fff8000
[1]	1	0x1fff8001
[2]	2	0x1fff8002
[3]	3	0x1fff8003
[4]	4	0x1fff8004
[5]	5	0x1fff8005
[6]	6	0x1fff8006
[7]	7	0x1fff8007
[8]	8	0x1fff8008
[9]	9	0x1fff8009

Figure 23-Source array

Name	Value	Location
g8bDestiny	0x1fff800c	0x1fff800c
[0]	1	0x1fff800c
[1]	0	0x1fff800d
[2]	0	0x1fff800e
[3]	0	0x1fff800f
[4]	0	0x1fff8010
[5]	0	0x1fff8011
[6]	0	0x1fff8012
[7]	0	0x1fff8013
[8]	0	0x1fff8014
[9]	0	0x1fff8015

Figure 24-Destination array

6.1 Example 1

By choosing DMA_ONE_TRANSACTION as the initialization parameter in main.h, init_DMA_1Trans will be called and the registers initialization for the single minor loop example will proceed.

The registers initialization can be seen in the following figure.












(x)= Variables Breakpoints 1010 0101 Registers Memory Modules		
Name		Value
 DMA_DCHPRI12		0x0c
 DMA_TCD0_SADDR		0x1fff8000
 DMA_TCD0_SOFF		0x0002
 DMA_TCD0_ATTR		0x0000
 DMA_TCD0_NBYTES_MLNO		0x00000005
 DMA_TCD0_NBYTES_MLOFFNO		0x00000005
 DMA_TCD0_NBYTES_MLOFFYES		0x00000005
 DMA_TCD0_SLAST		0xffffffff6
 DMA_TCD0_DADDR		0x1fff800c
 DMA_TCD0_DOFF		0x0001
 DMA_TCD0_CITER_ELINKNO		0x0001

Figure 25-DMA channel 0 registers

As can be seen in the figure the source address (SADDR) and destination address (DADDR) matches those of the source and destination buffers, the number of bytes to be transferred is 5 (NBYTES), there is a source offset address of 2 bits therefore the values that will be read in the array are 0, 2, 4, 6 and 8, this values will be written with an offset of one byte and only one minor loop will be executed per major loop (CITER = 1).

Once the peripheral request has been activated (SW1 in Port C in this case) the transfer process will begin, the source address will be read and its contents loaded to the destination address until NBYTES equals 0, CITER will be decremented and if it equals 0 the major loop will terminate, CITER will be updated with the contents of BITER, address adjustments will be performed (both addresses back to their initial values) and a flag in the control and status register will be issued to indicate the completion of the major loop. This process is illustrated in the following figures.












 g8bDestiny	0x1fff800c	0x1fff800c
 [0]	0	0x1fff800c
 [1]	2	0x1fff800d
 [2]	4	0x1fff800e
 [3]	6	0x1fff800f
 [4]	8	0x1fff8010
 [5]	0	0x1fff8011
 [6]	0	0x1fff8012
 [7]	0	0x1fff8013
 [8]	0	0x1fff8014
 [9]	0	0x1fff8015

Figure 26-Destination address after the transfer has been completed

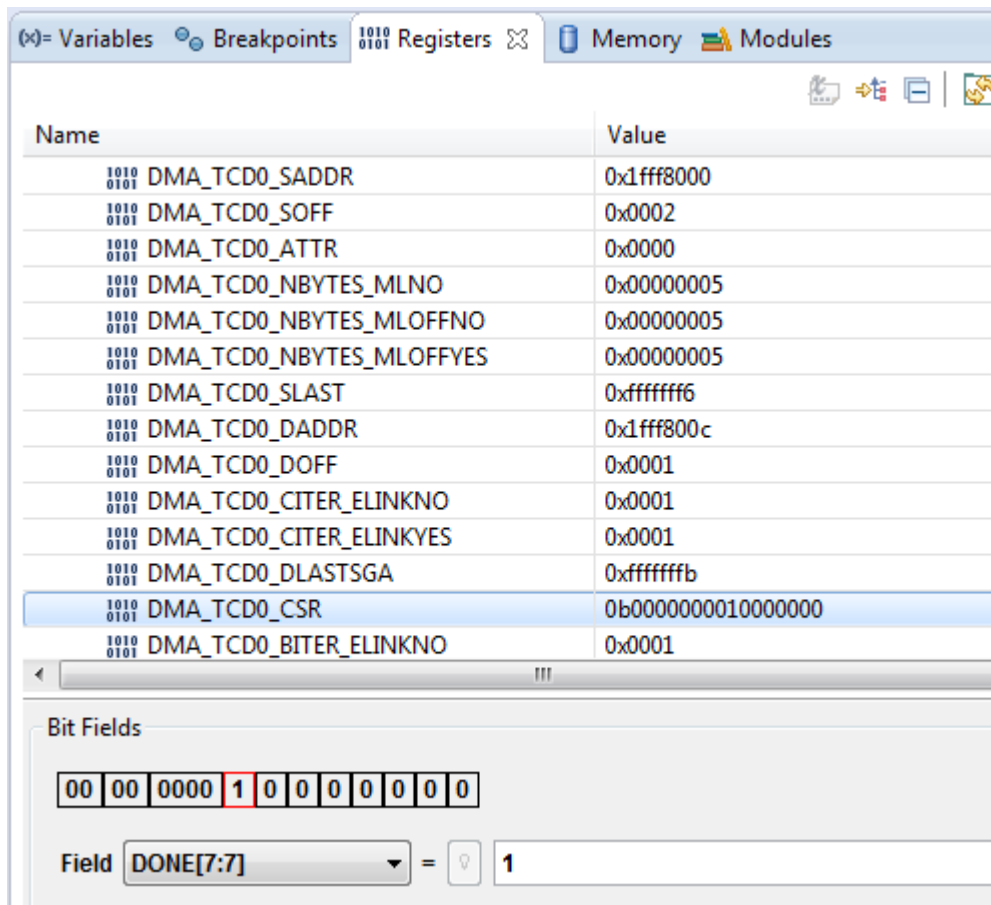


Figure 27-Registers status after the transfer has been completed

By altering certain parameters like address adjustments or offsets the way in which the data will be transferred will change. For instance if the destination address adjustment (DLASTSGA) happened to be zero, once the major loop has finished the destination address will remain unaltered and the next peripheral request will write the following 5 bytes where the previous write left it. This is only for demonstration's sake, because if the process continues it might reach a space in memory that is not allocated for data storage and some data corruptions might occur.

g8bDestiny	0x1fff800c	0x1fff800c
(x) [0]	0	0x1fff800c
(x) [1]	2	0x1fff800d
(x) [2]	4	0x1fff800e
(x) [3]	6	0x1fff800f
(x) [4]	8	0x1fff8010
(x) [5]	0	0x1fff8011
(x) [6]	2	0x1fff8012
(x) [7]	4	0x1fff8013
(x) [8]	6	0x1fff8014
(x) [9]	8	0x1fff8015

Figure 28-Example 1 with DLASTSGA = 0

6.2 Example 2

By choosing DMA_ONE_ELEMENT as the initialization parameter in main.h, init_DMA_1Elem will be called and the registers initialization for the single byte 5 minor loops example will proceed.

The registers initialization can be seen in the following figure.

Name	Value
 DMA_TCD0_SADDR	0x1fff8000
 DMA_TCD0_SOFF	0x0002
 DMA_TCD0_ATTR	0x0000
 DMA_TCD0_NBYTES_MLNO	0x00000001
 DMA_TCD0_NBYTES_MLOFFNO	0x00000001
 DMA_TCD0_NBYTES_MLOFFYES	0x00000001
 DMA_TCD0_SLAST	0xffffffff6
 DMA_TCD0_DADDR	0x1fff800c
 DMA_TCD0_DOFF	0x0001
 DMA_TCD0_CITER_ELINKNO	0x0005
 DMA_TCD0_CITER_ELINKYES	0x0005
 DMA_TCD0_DLASTSGA	0xffffffffb
 DMA_TCD0_CSR	0b0000000000000000
 DMA_TCD0_BITER_ELINKNO	0x0005

Figure 29-DMA channel 0 registers

As can be seen in the figure the source address (SADDR) and destination address (DADDR) matches those of the source and destination arrays, the number of bytes to be transferred per minor loop is 1 (NBYTES), there is a source offset address of 2 bits therefore the values that will be read in the array are 0, 2, 4, 6 and 8, this values will be written with an offset of one bit and only one minor loop will be executed per major loop (CITER = 1).

Once the peripheral request has been activated (SW1 in Port C in this case) the transfer process will begin, the source address will be read and its contents loaded to the destination address until NBYTES equals 0 since NBYTES equals 1 just one minor loop will be executed and CITER will be decreased as can be seen in figure 30

at the end of the first minor loop (transfer) CITER now holds the value 4, both source- and destination- address had been updated with their corresponding offsets.

g8bDestiny	0x1fff800c	0x1fff800c
(x) [0]	0	0x1fff800c
(x) [1]	0	0x1fff800d
(x) [2]	0	0x1fff800e
(x) [3]	0	0x1fff800f
(x) [4]	0	0x1fff8010
(x) [5]	0	0x1fff8011
(x) [6]	0	0x1fff8012
(x) [7]	0	0x1fff8013
(x) [8]	0	0x1fff8014
(x) [9]	0	0x1fff8015

Figure 30-First minor loop












Name	Value
 DMA_TCD0_SADDR	0x1fff8002
 DMA_TCD0_SOFF	0x0002
 DMA_TCD0_ATTR	0x0000
 DMA_TCD0_NBYTES_MLNO	0x00000001
 DMA_TCD0_NBYTES_MLOFFNO	0x00000001
 DMA_TCD0_NBYTES_MLOFFYES	0x00000001
 DMA_TCD0_SLAST	0xffffffff6
 DMA_TCD0_DADDR	0x1fff800d
 DMA_TCD0_DOFF	0x0001
 DMA_TCD0_CITER_ELINKNO	0x0004
 DMA_TCD0_CITER_ELINKYES	0x0004
 DMA_TCD0_DLASTSGA	0xffffffffb
 DMA_TCD0_CSR	0b0000000000000000
 DMA_TCD0_BITER_ELINKNO	0x0005

Figure 31-Register status after first minor loop

The process continues for the next 3 elements before CITER reaches 0, as can be seen in the following figures.

g8bDestiny	0x1fff800c	0x1fff800c
[0]	0	0x1fff800c
[1]	2	0x1fff800d
[2]	4	0x1fff800e
[3]	6	0x1fff800f
[4]	0	0x1fff8010
[5]	0	0x1fff8011
[6]	0	0x1fff8012
[7]	0	0x1fff8013
[8]	0	0x1fff8014
[9]	0	0x1fff8015

Figure 32-Fourth minor loop

Name	Value
DMA_TCD0_SADDR	0x1fff8008
DMA_TCD0_SOFF	0x0002
DMA_TCD0_ATTR	0x0000
DMA_TCD0_NBYTES_MLNO	0x00000001
DMA_TCD0_NBYTES_MLOFFNO	0x00000001
DMA_TCD0_NBYTES_MLOFFYES	0x00000001
DMA_TCD0_SLAST	0xffffffff6
DMA_TCD0_DADDR	0x1fff8010
DMA_TCD0_DOFF	0x0001
DMA_TCD0_CITER_ELINKNO	0x0001
DMA_TCD0_CITER_ELINKYES	0x0001
DMA_TCD0_DLASTSGA	0xffffffffb
DMA_TCD0_CSR	0b0000000000000000
DMA_TCD0_BITER_ELINKNO	0x0005

Figure 33-Register status after the fourth minor loop

For the last minor loop CITER finally reaches zero, both source- and destination-address have been adjusted for the following major loop, CITER has been updated with the value in BITER and the done flag in the control and status register has been activated. This process can be verified in the following figures.

g8bDestiny	0x1fff800c	0x1fff800c
[0]	0	0x1fff800c
[1]	2	0x1fff800d
[2]	4	0x1fff800e
[3]	6	0x1fff800f
[4]	8	0x1fff8010
[5]	0	0x1fff8011
[6]	0	0x1fff8012
[7]	0	0x1fff8013
[8]	0	0x1fff8014
[9]	0	0x1fff8015

Figure 34-Last minor loop


Name	Value
 DMA_TCD0_SADDR	0x1fff8000
 DMA_TCD0_SOFF	0x0002
 DMA_TCD0_ATTR	0x0000
 DMA_TCD0_NBYTES_MLNO	0x00000001
 DMA_TCD0_NBYTES_MLOFFNO	0x00000001
 DMA_TCD0_NBYTES_MLOFFYES	0x00000001
 DMA_TCD0_SLAST	0xffffffff6
 DMA_TCD0_DADDR	0x1fff800c
 DMA_TCD0_DOFF	0x0001
 DMA_TCD0_CITER_ELINKNO	0x0005
 DMA_TCD0_CITER_ELINKYES	0x0005
 DMA_TCD0_DLASTSGA	0xffffffffb
 DMA_TCD0_CSR	0b0000000010000000
 DMA_TCD0_BITER_ELINKNO	0x0005

Figure 35- Register status after last minor loop

If the data transfer is desired to be in a different order, for instance 8, 6, 4, 2 and 0 instead of 0, 2, 4, 6 and 8, this could be achieved only by modifying the source address in order to start at the address of the desired byte (8 in this case) and modifying the source address offset sign, so that it diminishes the source address by 2 bits per basic transfer.












 g8bSource	0x1fff8000	0x1fff8000
 [0]	0	0x1fff8000
 [1]	1	0x1fff8001
 [2]	2	0x1fff8002
 [3]	3	0x1fff8003
 [4]	4	0x1fff8004
 [5]	5	0x1fff8005
 [6]	6	0x1fff8006
 [7]	7	0x1fff8007
 [8]	8	0x1fff8008
 [9]	9	0x1fff8009

Figure 36-Source array

To achieve this the source-address must match the address of number 8 in the array this can be done by adding 8 to the source address that is being currently used. The source offset (SOFF) register must be set to -2 bits and the source-address adjustment is set to 10 bits in order to reestablish the initial value of the source address. This can be verified in the following figure.


Name	Value
 DMA_TCD0_SADDR	0x1fff8008
 DMA_TCD0_SOFF	0xfffe
 DMA_TCD0_ATTR	0x0000
 DMA_TCD0_NBYTES_MLNO	0x00000001
 DMA_TCD0_NBYTES_MLOFFNO	0x00000001
 DMA_TCD0_NBYTES_MLOFFYES	0x00000001
 DMA_TCD0_SLAST	0x0000000a
 DMA_TCD0_DADDR	0x1fff800c
 DMA_TCD0_DOFF	0x0001
 DMA_TCD0_CITER_ELINKNO	0x0005
 DMA_TCD0_CITER_ELINKYES	0x0005
 DMA_TCD0_DLASTSGA	0xfffffff
 DMA_TCD0_CSR	0b0000000000000000
 DMA_TCD0_BITER_ELINKNO	0x0005

Figure 37-Register status for Example 2 modification

The results of this modification are being shown in the following figure.












Name	Value	Location
(x)= i	13814	0x1fffff4
 g8bDestiny	0x1fff800c	0x1fff800c
 [0]	8	0x1fff800c
 [1]	6	0x1fff800d
 [2]	4	0x1fff800e
 [3]	2	0x1fff800f
 [4]	0	0x1fff8010
 [5]	0	0x1fff8011
 [6]	0	0x1fff8012
 [7]	0	0x1fff8013
 [8]	0	0x1fff8014
 [9]	0	0x1fff8015

Figure 38-Descending data movement

7 Appendix

The following appendix contains the code that was used for the implementation example:

main.c

```
#include "derivative.h" /* include peripheral declarations */
#include "dma.h"
#include "main.h"

int main(void)
{
    int i; // Delay variable

    // Enable clock and MUX for LEDs
    SIM_SCGC5 |= SIM_SCGC5_PORTC_MASK;
    PORTC_PCR9 = PORT_PCR_MUX(1);
    PORTC_PCR10 = PORT_PCR_MUX(1);

    // Set LEDs as outputs
    GPIOC_PDDR |= (1 << 10) | (1 << 9);
    GPIOC_PCOR |= (1 << 9); // Turn off Green LED D9
    GPIOC_PSOR |= (1 << 10); // Turn on Blue LED D10

    // Set SW1 as peripheral for DMA request on falling edge
    PORTC_PCR1 |= PORT_PCR_MUX(0x01) |
PORT_PCR_IRQC(0x02) | PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;

    // Select size of data transfer
    #if (TEST)
        init_DMA_1Elem();
    #else
        init_DMA_1Trans();
    #endif

    while(1)
    {
        // Delay
        for(i = 0; i < 500000; i++)
            asm("nop");

        // Toggle Green LED D9 and Blue LED D10
        GPIOC_PTOR = (1 << 9) | (1 << 10);
    }
}
```


main.h

```
#ifndef MAIN_H_
#define MAIN_H_

#define DMA_ONE_TRANSACTION 0    // 5 bytes per DMA request
#define DMA_ONE_ELEMENT      1    // 1 element per DMA request

// Select option
#define TEST DMA_ONE_ELEMENT

#endif /* MAIN_H_ */
```

dma.c

```
#include "dma.h"
#include "derivative.h"

unsigned char g8bSource[10] = {0,1,2,3,4,5,6,7,8,9};
unsigned char g8bDestiny[10] = {1,0,0,0,0,0,0,0,0,0};

/*
 *
 * Initializes channel 0 DMA registers in order to perform 1 data transfer of 5 bytes
 * and sets Port C as DMA request source
 *
 */
void init_DMA_1Trans(void)
{
    // Enable clock for DMAMUX and DMA
    SIM_SCGC6 |= SIM_SCGC6_DMAMUX_MASK;
    SIM_SCGC7 |= SIM_SCGC7_DMA_MASK;

    // Enable Channel 0 and set Port C as DMA request source
    DMAMUX_CHCFG0 |= DMAMUX_CHCFG_ENBL_MASK | DMAMUX_CHCFG_SOURCE(51);

    // Enable request signal for channel 0
    DMA_ERQ = DMA_ERQ_ERQ0_MASK;

    // Set memory address for source and destination
    DMA_TCD0_SADDR = (uint32_t)&g8bSource;
    DMA_TCD0_DADDR = (uint32_t)&g8bDestiny;

    // Set an offset for source and destination address
    DMA_TCD0_SOFF = 0x02; // Source address offset of 2 bits per transaction
    DMA_TCD0_DOFF = 0x01; // Destination address offset of 1 bit per transaction

    // Set source and destination data transfer size
    DMA_TCD0_ATTR = DMA_ATTR_SSIZE(0) | DMA_ATTR_DSIZE(0);

    // Number of bytes to be transfered in each service request of the channel
    DMA_TCD0_NBYTES_MLNO = 0x05;
```

```

    // Current major iteration count (a single iteration of 5 bytes)
    DMA_TCD0_CITER_ELINKNO = DMA_CITER_ELINKNO_CITER(1);
    DMA_TCD0_BITER_ELINKNO = DMA_BITER_ELINKNO_BITER(1);

    // Adjustment value used to restore the source and destiny address to the initial
value
    DMA_TCD0_SLAST = -0x0A;           // Source address adjustment
    DMA_TCD0_DLASTSGA = -0x05; // Destination address adjustment

    // Setup control and status register
    DMA_TCD0_CSR = 0;
}

/*
 *
 * Initializes channel 0 DMA registers in order to perform 5 data transfers of 1 byte
each
 *
 * and sets Port C as DMA request source
 *
 * */
void init_DMA_1Elem(void)
{
    // Enable clock for DMAMUX and DMA
    SIM_SCGC6 |= SIM_SCGC6_DMAMUX_MASK;
    SIM_SCGC7 |= SIM_SCGC7_DMA_MASK;

    // Enable Channel 0 and set Port C as DMA request source
    DMAMUX_CHCFG0 |= DMAMUX_CHCFG_ENBL_MASK | DMAMUX_CHCFG_SOURCE(51);

    // Enable request signal for channel 0
    DMA_ERQ = DMA_ERQ_ERQ0_MASK;

    // Set memory address for source and destination
    DMA_TCD0_SADDR = (uint32_t)&g8bSource;
    DMA_TCD0_DADDR = (uint32_t)&g8bDestiny;

    // Set an offset for source and destination address
    DMA_TCD0_SOFF = 0x02; // Source address offset of 2 bits per transaction
    DMA_TCD0_DOFF = 0x01; // Destination address offset of 1 bit per transaction

    // Set source and destination data transfer size
    DMA_TCD0_ATTR = DMA_ATTR_SSIZE(0) | DMA_ATTR_DSIZE(0);

    // Number of bytes to be transfered in each service request of the channel
    DMA_TCD0_NBYTES_MLNO = 0x01;

    // Current major iteration count (5 iteration of 1 byte each)
    DMA_TCD0_CITER_ELINKNO = DMA_CITER_ELINKNO_CITER(5);
    DMA_TCD0_BITER_ELINKNO = DMA_BITER_ELINKNO_BITER(5);
}

```

```

        // Adjustment value used to restore the source and destiny address to the initial
value
        DMA_TCD0_SLAST = -0x0A;           // Source address adjustment
        DMA_TCD0_DLASTSGA = -0x05; // Destination address adjustment

        // Setup control and status register
        DMA_TCD0_CSR = 0;
    }

```

dma.h

```

#ifndef DMA_H_
#define DMA_H_

/* Prototypes */
void init_DMA_1Trans(void);
void init_DMA_1Elem(void);

#endif /* DMA_H_ */

```

8 References

- [Reference manual K20 sub-family](#)
- For a deeper and more extensive treatise of the topics please refer to [Application Note 4522](#)