





## Agenda

1. Module Overview
2. On-chip Interconnections and Inter-module Dependencies
3. Software Configuration
4. Example Use Case
5. eDMA Frequently Asked Questions

 External Use | 1 

In this presentation we'll cover:

- An overview of the module
- The on-chip interconnections and inter-module dependencies
- Software configuration
- An example use case
- And a few frequently asked questions



Let's first begin with an overview of the module.

## eDMA Module Overview

- The enhanced direct memory access (eDMA) controller is a module capable of performing complex data transfers with minimal intervention from a host processor.
- The eDMA module works in conjunction with the direct memory access multiplexer (DMAMUX), which routes DMA sources, called slots, to any of the DMA channels.

## eDMA Module Overview

- The eDMA module is capable of performing complex data transfers with minimal intervention from a host processor.
- The main goal of this module is to replace the processor for data movement, which allows the CPU to keep working on other tasks while the data is being copied, resulting in better system performance.
- The eDMA module works in conjunction with the direct memory access multiplexer (DMAMUX), which routes DMA sources called slots to any of the DMA channels. The DMAMUX allows Kinetis MCUs to support a large number of peripheral DMA requests using 16 or 32 DMA channels.

## eDMA and DMAMUX Features

### eDMA Features

- 16 or 32 channels, depending on the Kinetis device
- Each channel has its own transfer control descriptor (TCD)
  - Supports two-deep nested transfers called minor and major loops
- Fixed-priority and round-robin channel arbitration

### DMAMUX Features

- One or two muxes, depending on number of DMA channels
- DMA routing is typically static, but the mux can be reprogrammed
- Two modes of operation
  1. Normal mode
  2. Periodic triggering mode available on first 4 channels



External Use | 4



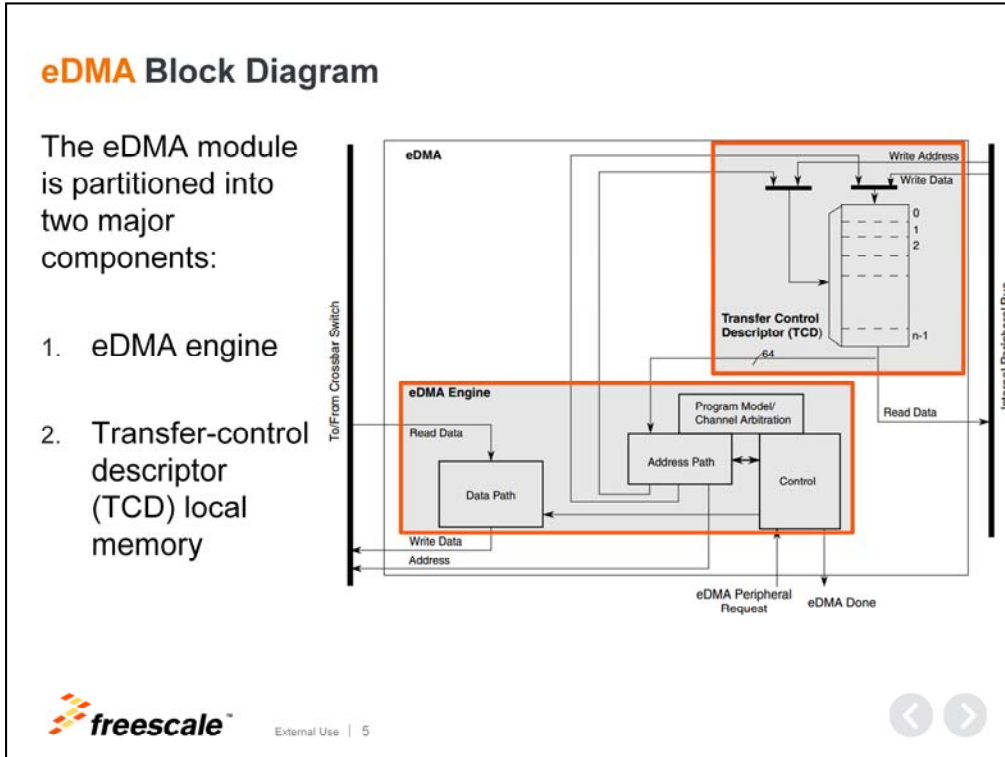
## eDMA and DMAMUX Features

### eDMA Features

- The eDMA controller offers 16 or 32 channels, which are available to perform complex data transfers with minimal CPU intervention. The number of channels is dependent on the Kinetis device. To find out exactly how many channels are available on your MCU, please refer to the reference manual.
- Each channel has its own transfer control descriptor (TCD) that supports two-deep nested transfer operations called minor and major loops, respectively. The TCDs store all the needed information to make the transactions, such as the source and destination addresses and the transfer size. Each TCD needs 32-bytes, which are stored in local memory. A channel can be activated either by software, by another channel through channel linking or by another peripheral through hardware requests.
- Fixed-priority and a round-robin schemes are allowed for channel arbitration. The channels can report through an interrupt request when the half and/or the total of the major loop is completed, which is useful to manage double buffers.

### DMAMUX Features

- Kinetis devices contain one or two multiplexers, depending on the amount of channels. For 16 channel devices, one DMAMUX is available and for 32 channel devices, 2 instances of DMAMUX are available



### eDMA Block Diagram

The eDMA module is partitioned in two major modules: 1) the eDMA engine and 2) the transfer-control descriptor local memory.

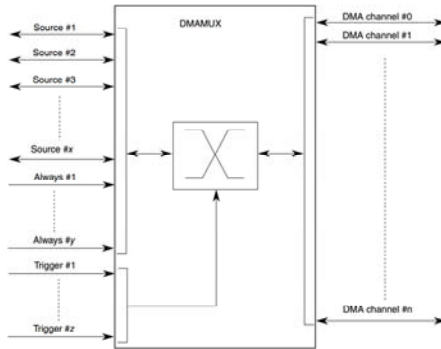
The eDMA engine performs:

- The source and destination address calculations, and
- data movement operations

The transfer-control descriptor local memory contains the transfer-control descriptors for each eDMA channel.

## DMAMUX Block Diagram

The DMAMUX (DMA multiplexer) routes the DMA sources to any of the DMA channels.



Note: Do not set multiple channels with the same DMA source.

## DMAMUX Block Diagram

The DMA multiplexer routes the DMA sources to the DMA channels.

Please note that a DMA source should be configured to use only one channel.

## eDMA Transfer Control Descriptor (TCD)

- The TCD contains all of the information about the data movement
  - Source address, address increment, and transfer size
  - Destination address, address increment, and transfer size
  - Number of bytes to transfer
  - Number of major loops to execute
- Each TCD can follow a two-deep nested loop (major and minor transfer loops).
- TCDs can be processed in a chain so that the eDMA will automatically load the next TCD.
- The eDMA uses a 32-byte transfer control descriptor.



External Use | 7



## eDMA Transfer Control Descriptor (TCD)

The TCD contains all the information about the data movement. This includes the source and destination addresses, the address increment after each transfer and the transfer size. The TCD also includes the number of bytes to transfer and the number of major loops to execute.

Each TCD can be a two-deep nested loop such as major and minor transfer loops,



The TCDs can be processed in a chain so that the eDMA will automatically load the next TCD. This is called channel linking.

Each TCD needs 32-bytes to store all this information.



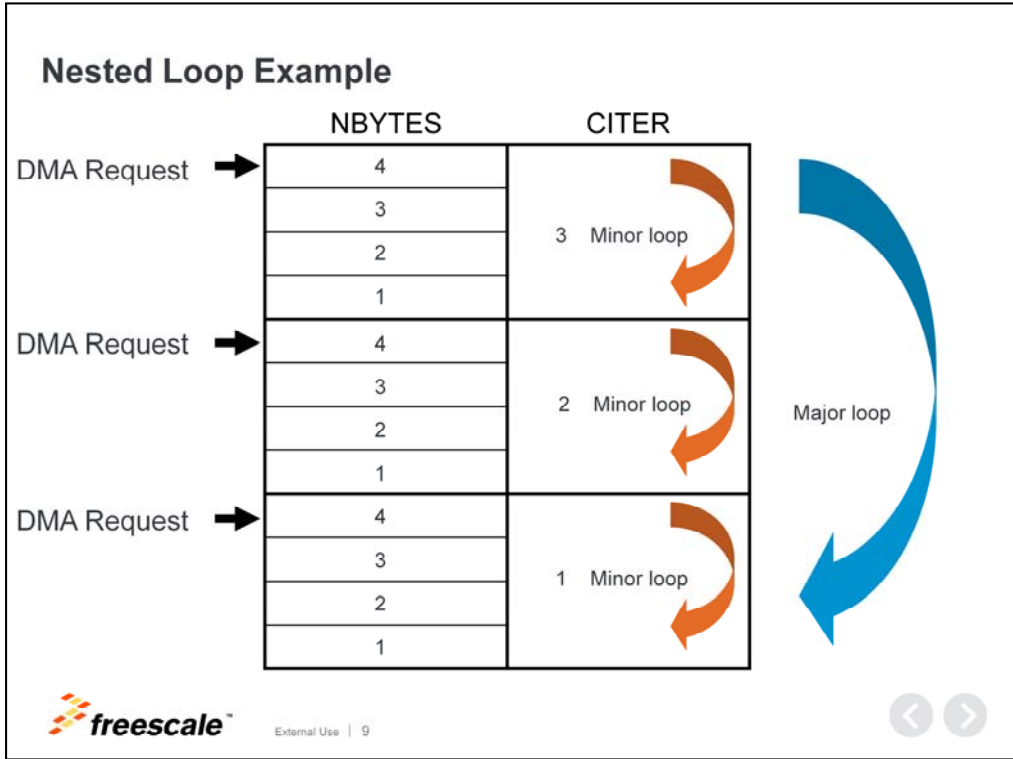
### TCD Memory Map

eDMA Offset	TCDn Register Name	Abbreviation	Width (bits)
0x00	Source Address	TCDn_SADDR	32
0x04	Transfer Attributes	TCDn_ATTR	16
0x06	Signed Source Address Offset	TCDn_SOFF	16
0x08	Minor Byte Count	TCDn_NBYTES	32
0x0C	Last Source Address Adjustment	TCDn_SLAST	32
0x10	Destination Address	TCDn_DADDR	32
0x14	Current Minor Loop Link, Major Loop Count	TCDn_CITER	16
0x16	Signed Destination Address Offset	TCDn_DOFF	16
0x18	Last Destination Address Adjustment/Scatter Gather Address	TDDn_DLAST_SGA	32
0x1C	Beginning Minor Loop Link, Major Loop Count	TCDn_BITER	16
0x1E	Control and Status	TCDn_CSR	16


External Use | 8


## TCD Memory Map

This table shows the memory map of a TCD. It contains all the registers that the eDMA needs to accomplish the transaction. The TCD is responsible for channel activation, address path, data path, and control for source and destination.

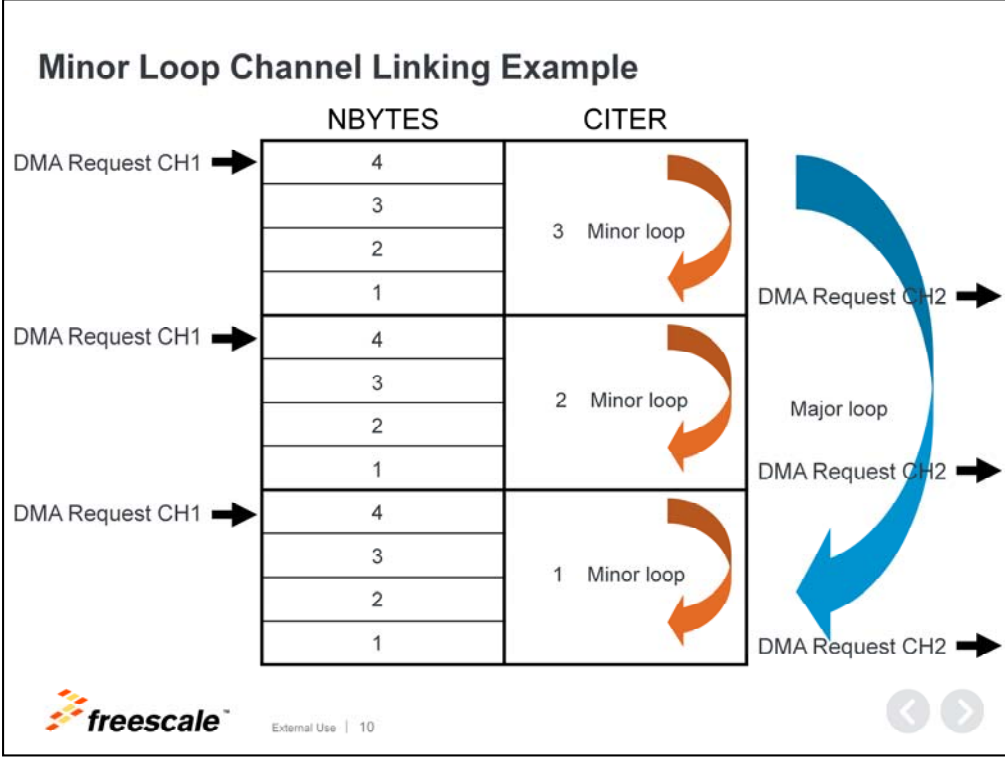


### Nested Loop Example

This example demonstrates two-deep nested loops.

Each DMA request will move the amount of bytes configured in the NBYTES (minor byte count) register. This corresponds to a minor loop. Once this is completed, another DMA request will transfer another minor loop. The CITER (major loop count) register will decrement each time a minor loop is transferred. When the CITER register is equal to zero, that means a major loop has been completed.

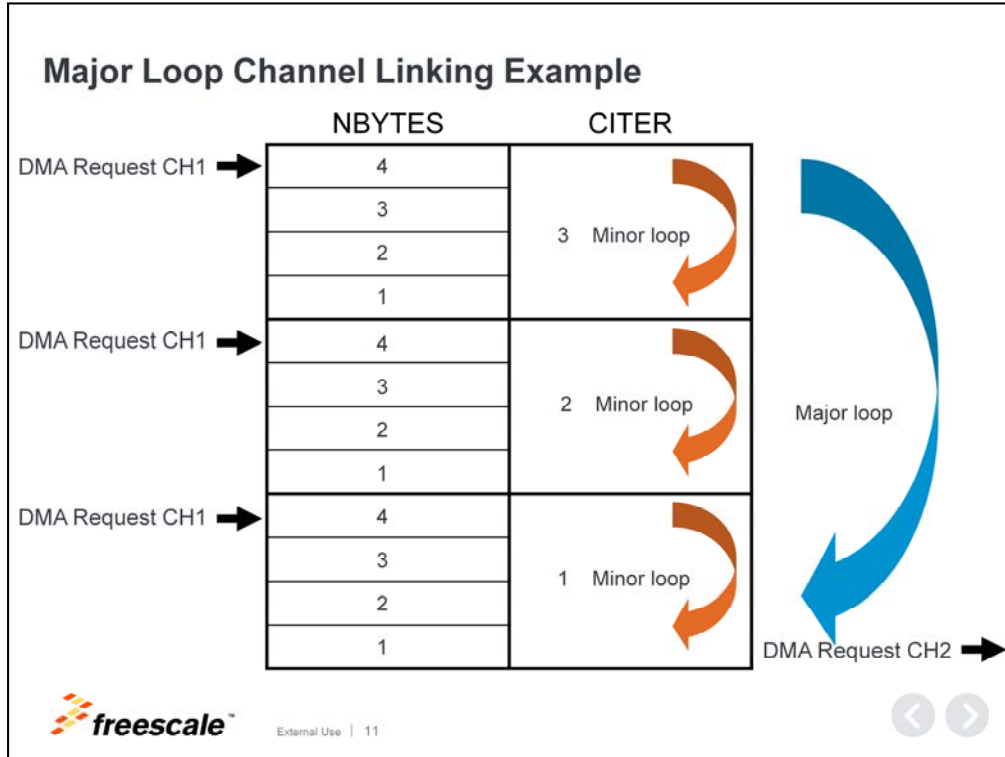
In other words, a minor loop contains NBYTES bytes and a major loop contains CITER minor loops.



### Minor Loop Channel Linking Example

This example demonstrates minor loop channel linking.

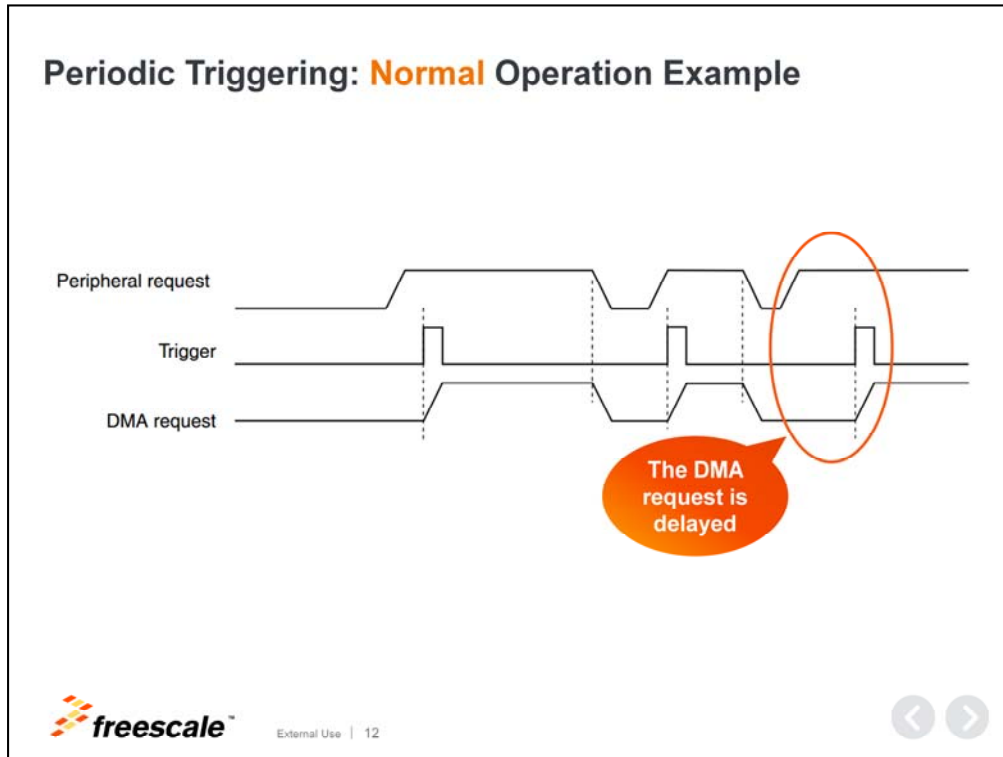
Each DMA request will move the amount of bytes configured in the NBYTES (minor byte count) register like the last example. This corresponds to a minor loop. Once this is completed, a DMA request for the linked channel is generated. Another DMA request will transfer another minor loop and a new DMA request for the linked channel will be generated.



### Major Loop Channel Linking Example

Finally, This example demonstrates major loop channel linking.

Major loop channel linking works exactly like minor loop channel linking, except the DMA request for the linked channel is done once the major loop is completed.

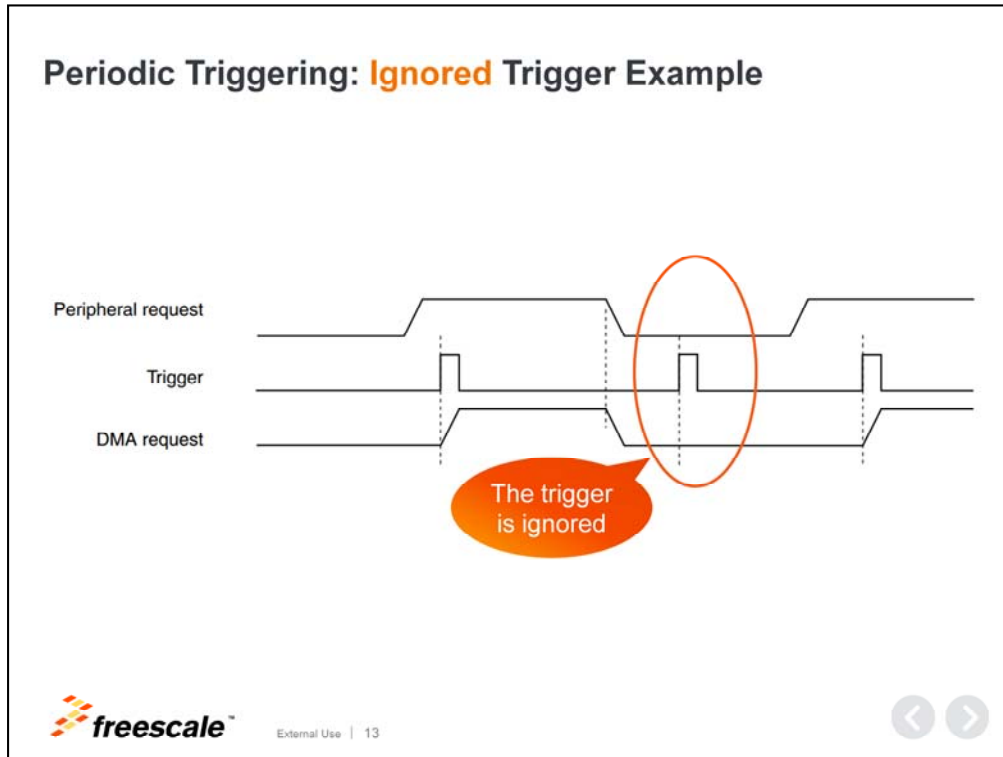


## Periodic Triggering: Normal Operation Example

The DMA periodic trigger capability allows the system to schedule regular DMA transfers, usually on the transmit side of certain peripherals, without the intervention of the processor. This trigger works by gating the request from the peripheral to the DMA until a trigger event has been seen.

There is one DMA request per peripheral request, but the DMA request is delayed until the trigger asserts. This helps ensure a fixed transfer frequency.

This image shows a normal operation example where the trigger is periodic. The DMA request is generated when both the peripheral request and the trigger are present.



### Periodic Triggering: Ignored Trigger Example

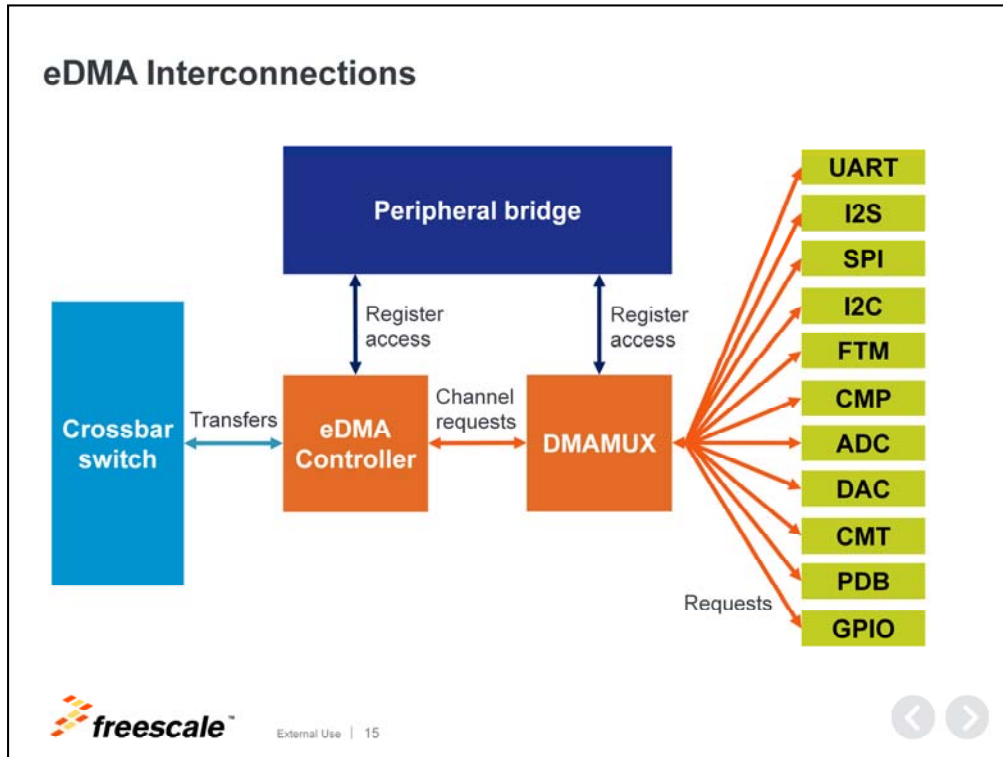
Now let's go over an example where the trigger is ignored.

After the DMA request has been serviced, the peripheral will negate its request, effectively resetting the gating mechanism until the peripheral reasserts its request and the next trigger event is seen.

This means that if a trigger is seen, but the peripheral is not requesting a transfer, then that trigger will be ignored. The peripheral request is not active when the second trigger asserts. The peripheral has to wait until the third trigger before a new DMA request is asserted.

## 2. On-chip Interconnections and Inter-module Dependencies

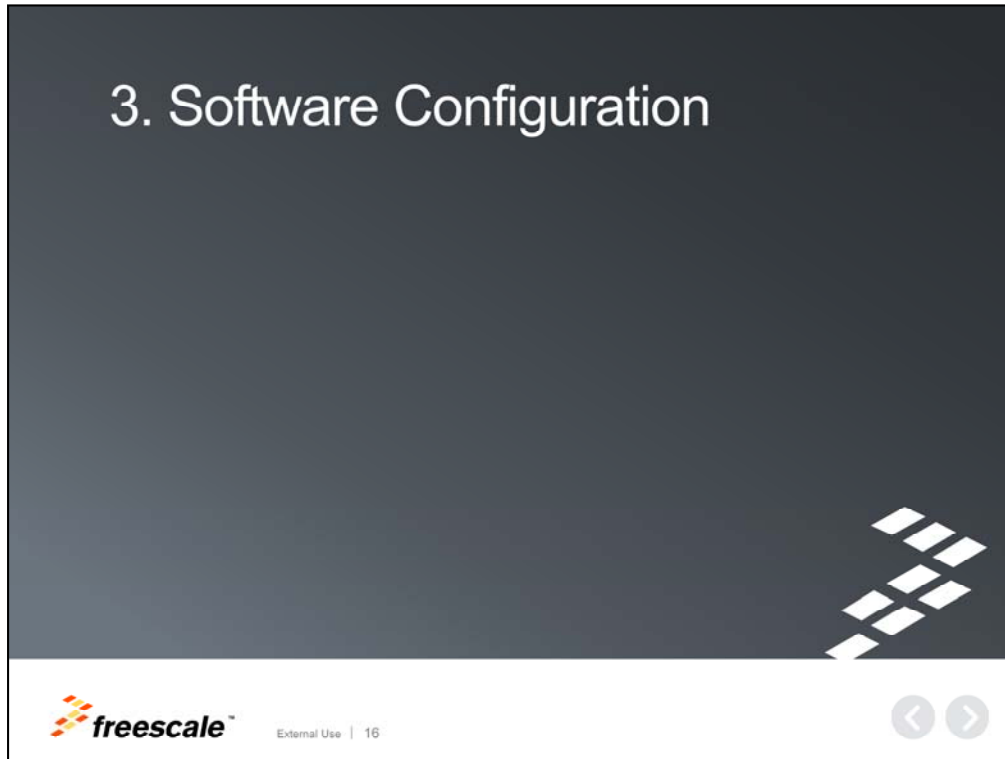
Next, let's discuss on-chip interconnections and inter-module dependencies.



### eDMA Interconnections

This diagram displays how other peripherals make requests to the DMAMUX, which is connected to the eDMA controller through the peripheral bridge. This process makes it possible for transactions to happen through the crossbar switch – mastering the data movement.





Now, let's move on to discuss software configuration.

### Kinetic SDK eDMA Driver Example

```

#define EDMA_CHANNEL            (1)
#define EDMA_TOTAL_BYTES       (8)
#define EDMA_BYTES_PER_SAMPLE (1)
#define EDMA_BYTES_PER_MINOR_LOOP (2)
#define EDMA_TCDS_AMOUNT       (2)
#define EDMA_SAMPLES_AMOUNT    (EDMA_TOTAL_BYTES / EDMA_BYTES_PER_SAMPLE)

uint8_t srcAddr[EDMA_SAMPLES_AMOUNT] = {1,2,3,4,5,6,7,8};
uint8_t dstAddr[EDMA_SAMPLES_AMOUNT] = {0};

void EDMA_IRQ_Handler(void *parameter, edma_chn_status_t status);
edma_callback_t callbackEDMA = EDMA_IRQ_Handler;

// 1. Initialize DMA module.
edma_state_t edmaState;
edma_status_t edmaStatus;
edma_user_config_t edmaUserConfig =
{
    .chnArbitration = kEDMAChnArbitrationRoundrobin,
    .notHaltOnError = true,
};
edmaStatus = EDMA_DRV_Init(&edmaState, &edmaUserConfig);

// 2. Request a DMA channel
uint8_t channel;
edma_chn_state_t edmaChannelState;
channel = EDMA_DRV_RequestChannel(EDMA_CHANNEL, kDmaRequestMuxPortC, &edmaChannelState);

// 3. Configure the TCD.
edma_software_tcd_t stcd[4];


edmaStatus = EDMA_DRV_ConfigLoopTransfer(&edmaChannelState, stcd, kEDMA_MemoryToMemory, (uint32_t)srcAddr, (uint32_t)dstAddr, EDMA_BYTES_PER_SAMPLE, EDMA_BYTES_PER_MINOR_LOOP, EDMA_TOTAL_BYTES, EDMA_TCDS_AMOUNT);

// 4. Register callback function:
edmaStatus = EDMA_DRV_InstallCallback(&edmaChannelState, callbackEDMA, NULL);

// 5. Start the DMA channel: EDMA_DRV_StartChannel.
edmaStatus = EDMA_DRV_StartChannel(&edmaChannelState);

// 6. [OPTION] Stop the DMA channel: EDMA_DRV_StopChannel.
// 7. Free the DMA channel: EDMA_DRV_ReleaseChannel.

```

 External Use | 17

## Kinetic SDK eDMA Driver Example

This is an example of loop transfers in the eDMA using Kinetic SDK .

In this example, we will use channel 1.

We will transfer eight samples of 1-byte length.

Two samples on each request.

We will need two interrupts on each loop, so we configure two TCDs. The interrupts help manage a ping pong buffer.

Step 1, Initialize the DMA module. The channel arbitration will be round-robin. Once the configuration structure is filled, we can call the `EDMA_DRV_Init()` function.

The 2nd step is to request a DMA channel through the `EDMA_DRV_RequestChannel()` function. In this case the request will be generated by GPIO PORT C.

The 3rd step is to configure the TCD. The loop transfer is configured through the `EDMA_DRV_ConfigLoopTransfer()` function.

The callback needs to be installed through the `EDMA_DRV_InstallCallback()` function.

Finally, the channel is started with the `EDMA_DRV_StartChannel()` function. The DMA is waiting for a request.

Additionally, a channel can be stopped through the `EDMA_DRV_StopChannel()` function.

And a channel can be released through the `EDMA_DRV_ReleaseChannel()`

### eDMA Driver Example Results

Expression	Type	Value
srcAddr	uint8_t [8]	0x1fff0004
srcAddr[0]	uint8_t	0x1
srcAddr[1]	uint8_t	0x2
srcAddr[2]	uint8_t	0x3
srcAddr[3]	uint8_t	0x4
srcAddr[4]	uint8_t	0x5
srcAddr[5]	uint8_t	0x6
srcAddr[6]	uint8_t	0x7
srcAddr[7]	uint8_t	0x8
dstAddr	uint8_t [8]	0x1fff00cc
dstAddr[0]	uint8_t	0x0
dstAddr[1]	uint8_t	0x0
dstAddr[2]	uint8_t	0x0
dstAddr[3]	uint8_t	0x0
dstAddr[4]	uint8_t	0x0
dstAddr[5]	uint8_t	0x0
dstAddr[6]	uint8_t	0x0
dstAddr[7]	uint8_t	0x0

- 0 requests

dstAddr	uint8_t [8]	0x1fff00cc
dstAddr[0]	uint8_t	0x1
dstAddr[1]	uint8_t	0x2
dstAddr[2]	uint8_t	0x0
dstAddr[3]	uint8_t	0x0
dstAddr[4]	uint8_t	0x0
dstAddr[5]	uint8_t	0x0
dstAddr[6]	uint8_t	0x0
dstAddr[7]	uint8_t	0x0

- 1 request
- 1 minor loop

dstAddr	uint8_t [8]	0x1fff00cc
dstAddr[0]	uint8_t	0x1
dstAddr[1]	uint8_t	0x2
dstAddr[2]	uint8_t	0x3
dstAddr[3]	uint8_t	0x4
dstAddr[4]	uint8_t	0x5
dstAddr[5]	uint8_t	0x6
dstAddr[6]	uint8_t	0x0
dstAddr[7]	uint8_t	0x0

- 2 requests
- 2 minor loops
- First interrupt

dstAddr	uint8_t [8]	0x1fff00cc
dstAddr[0]	uint8_t	0x1
dstAddr[1]	uint8_t	0x2
dstAddr[2]	uint8_t	0x3
dstAddr[3]	uint8_t	0x4
dstAddr[4]	uint8_t	0x5
dstAddr[5]	uint8_t	0x6
dstAddr[6]	uint8_t	0x7
dstAddr[7]	uint8_t	0x8

- 3 requests
- 3 minor loops

dstAddr	uint8_t [8]	0x1fff00cc
dstAddr[0]	uint8_t	0x1
dstAddr[1]	uint8_t	0x2
dstAddr[2]	uint8_t	0x3
dstAddr[3]	uint8_t	0x4
dstAddr[4]	uint8_t	0x5
dstAddr[5]	uint8_t	0x6
dstAddr[6]	uint8_t	0x7
dstAddr[7]	uint8_t	0x8

- 4 requests
- 4 minor loops
- Second interrupt

External Use | 18

### eDMA Driver Example Results

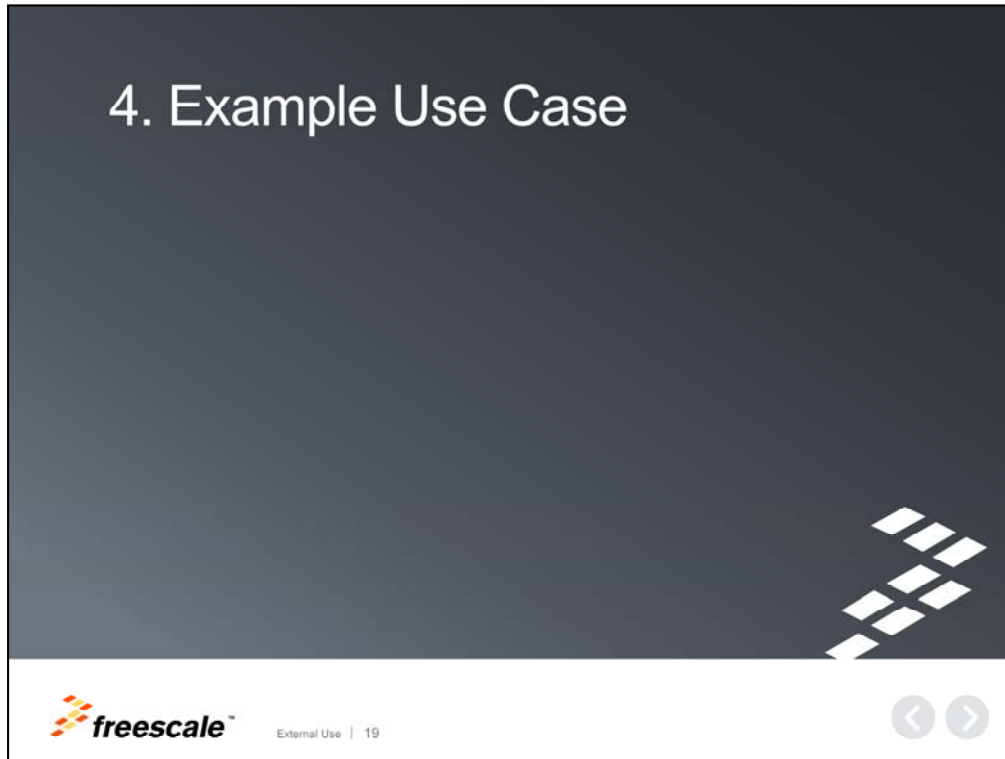
Once the channel is started, the destination buffer is blank.

When the first request is asserted, two bytes are transferred.

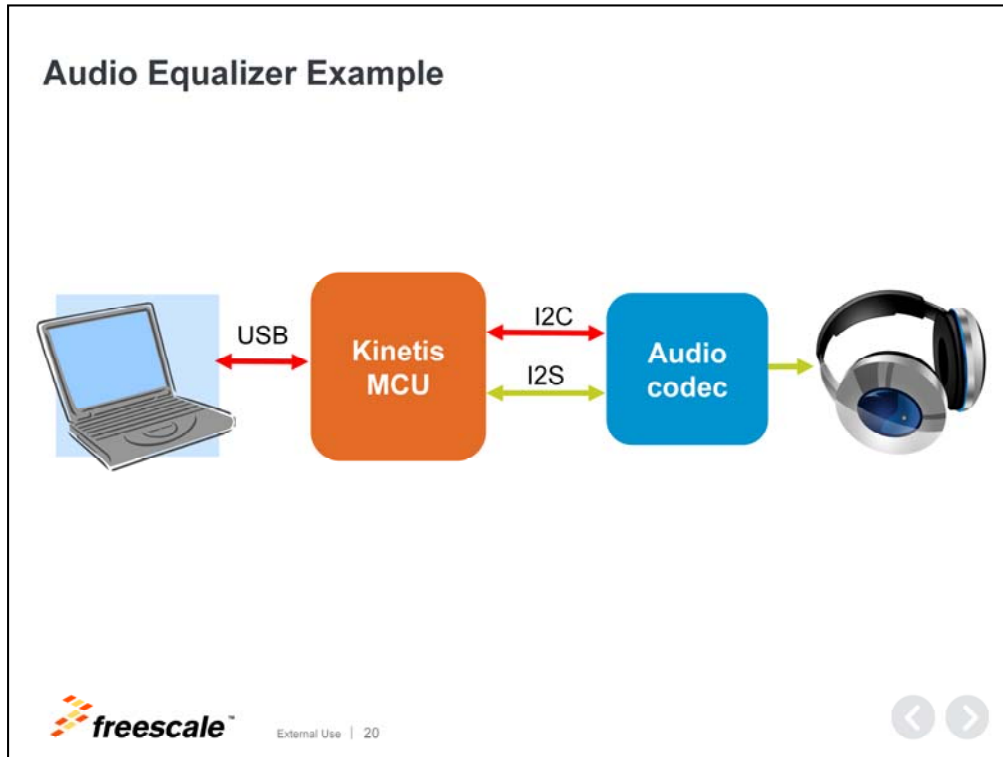
The second request transfers two more bytes, which results in the first of the two interrupts.

The third request copies the other two bytes.

And finally, the fourth request moves the last two bytes – this generates the second interrupt.



Let's review two example use cases.



## Audio Equalizer Example

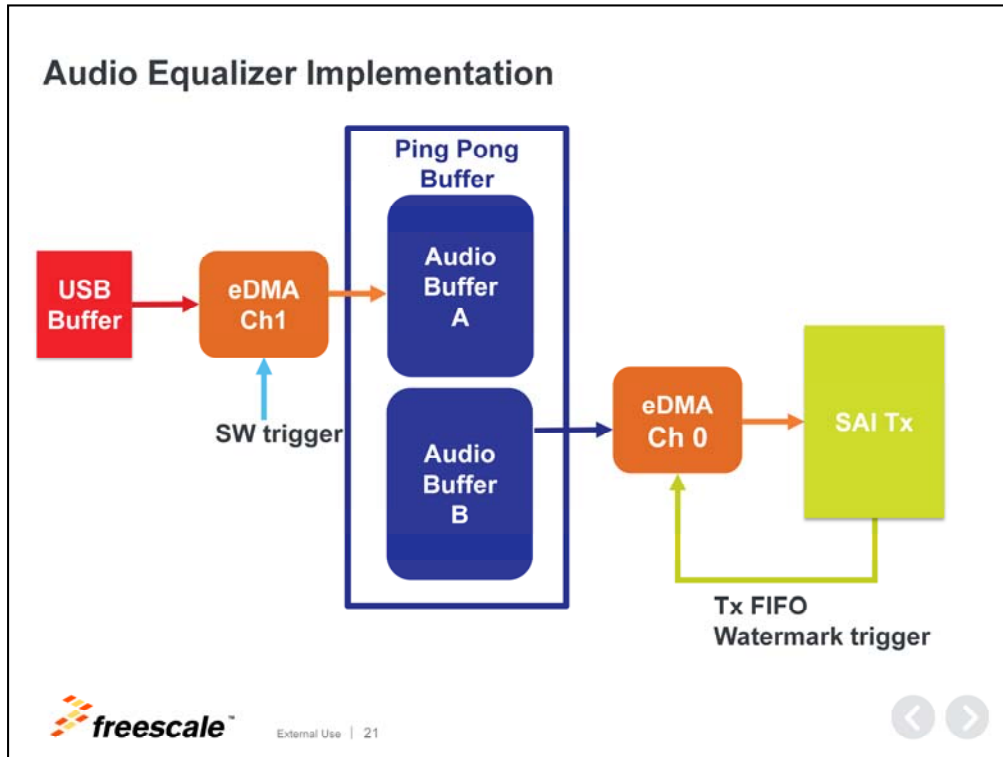
Let's say we want to develop an audio equalizer using Kinetis MCUs.

For this application, we'll need a Kinetis device with USB support to communicate to a PC to acquire the original audio.

An audio codec is required to translate the audio data into sound on your speaker.

The Kinetis MCU will need to configure the audio codec. Let's assume communication between these devices is I2C.

Finally, use I2S to transfer audio data from the Kinetis MCU to the audio codec. The eDMA helps automate the data flow so the CPU will be free to process the audio data.

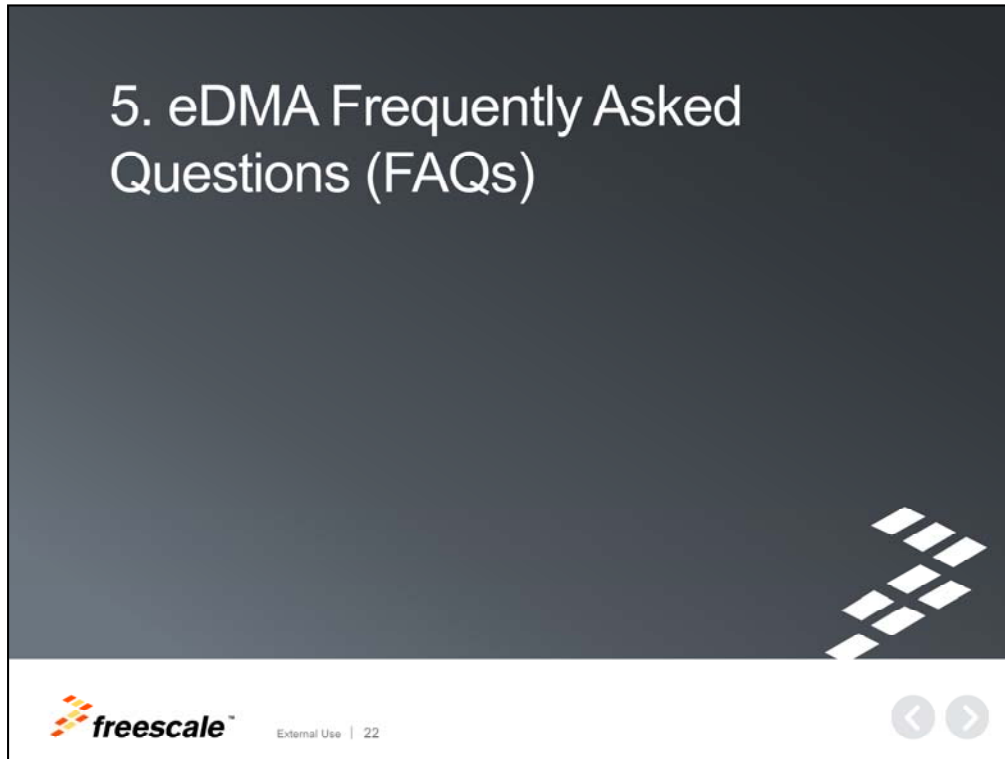


## Audio Equalizer Implementation

The audio data is received from a USB into a buffer. We can use an eDMA channel to take this data and move it to a ping pong buffer dedicated for the audio samples.

A second DMA channel can be used to move the processed data from the ping pong audio buffer to the SAI TX (Transmit) FIFO. This eDMA will be triggered by the SAI TX FIFO watermark signal.

With this use case, the CPU can handle other tasks such the audio filter. There is no need to send audio data to the audio codec, since everything is done in the background with the eDMA.



Finally, let's review some frequently asked questions.

## eDMA FAQs

**Question:** What do eDMA bus cycles look like?

**Answer:** eDMA bus cycles look like any other bus cycle. Slave device architecture (including the FlexBus) don't treat DMA cycles differently than core requests, so the bus timing will be the same as if the core had requested a bus cycle of the same size.

**Question:** Can I assign a single DMA request source to multiple channels? For example, use SPI0 Rx to trigger DMA channels 0 and 1 simultaneously.

**Answer:** No, this can result in unpredictable behavior. In some cases such as when you aren't reading from a FIFO that will update as it is read, you can use channel linking to get a similar effect. If you need two copies of data from a FIFO, then you could use one DMA channel to read from the FIFO and write the data to memory. Then, link to a second channel that makes a copy of the data in memory.



External Use | 23



## eDMA FAQs

**Question:** What do eDMA bus cycles look like?

**Answer:** eDMA bus cycles look like any other bus cycle. Slave device architecture (including the FlexBus) don't treat eDMA cycles differently than core requests, so the bus timing will be the same as if the core had requested a bus cycle of the same size.

**Question:** Can I assign a single eDMA request source to multiple channels? For example, use SPI0 Rx to trigger DMA channels 0 and 1 simultaneously.

**Answer:** No, this can result in unpredictable behavior. In some cases such as when you aren't reading from a FIFO that will update as it is read, you can use channel linking to get a similar effect. If you need two copies of data from a FIFO, then you could use one DMA channel to read from the FIFO and write the data to memory. Then, link to a second channel that makes a copy of the data in memory.



## References

- **Application notes:**

- [AN4560](#) Using DMA to Emulate ADC Flexible Scan Mode
- [AN5083](#) Using DMA for pulse counting on Kinetis
- [AN5121](#) Generating a Fixed Number of PWM Pulses Using TPM

- **Website:** [Freescale.com/Kinetis](http://Freescale.com/Kinetis)

- **Community:** [community.freescale.com/community/Kinetis](http://community.freescale.com/community/Kinetis)

- [DOC-102981](#) Using the DMA module in Kinetis Devices
- [DOC-105019](#) Configuring Kinetis Software Development Kit (KSDK) to measure distance with ultrasonic transducers



External Use | 24



## References

This concludes our presentation on the eDMA module for Kinetis MCUs.

For more information on eDMA, please visit the application notes listed here.

We also invite you to visit us on the web [Freescale.com/Kinetis](http://Freescale.com/Kinetis) and check out our Kinetis community page.



[www.Freescale.com](http://www.Freescale.com)

© 2015 Freescale Semiconductor, Inc. | *External Use*