EFM32
... the world's most energy friendly microcontrollers

# Direct Memory Access

## AN0013 - Application Note

### *Introduction*

This application note demonstrates how to use the Direct Memory Access (DMA) in the EFM32. Several software examples are provided that shows how to use the DMA with the ADC, UART, SPI etc. using the various transfer modes of the DMA. The example projects are configured for the EFM32G890F128, but can easily be ported to other EFM32 devices by changing the project settings.

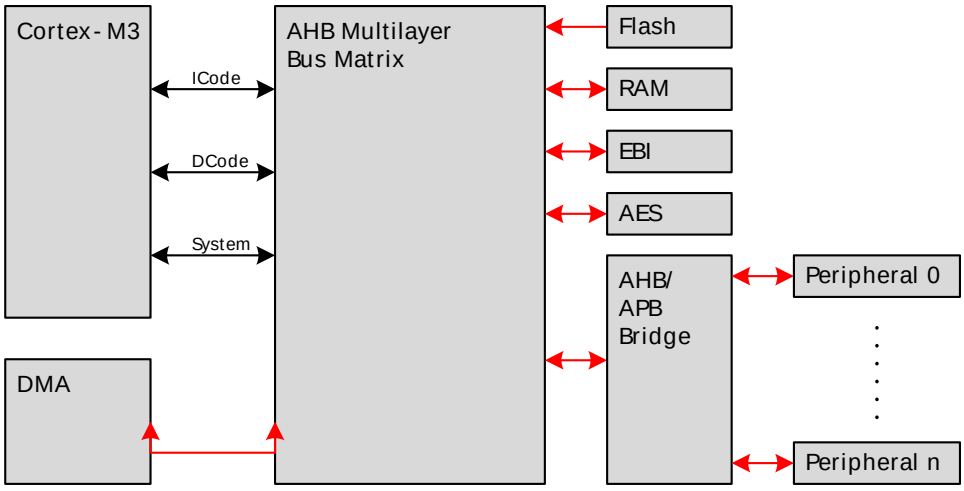
This application note includes:

- This PDF document
- Source files (zip)
  - C-code examples
  - Multiple IDE projects

# 1 Introduction

The DMA is used for data transfer without CPU intervention. Data can be transferred between any readable source and writable destination address within the EFM32 address space and can be initiated either by a peripheral setting a DMA request signal (e.g when a new ADC sample is available) or by the CPU directly. While the DMA is handling the data transfer, the CPU is free to do other work or go to sleep (EM1) in order to save energy. Upon completion the DMA can wake up the CPU by triggering an interrupt. The DMA is connected to the EFM32's bus system as an AMBA AHB master (see Figure 1.1 (p. 2) ) allowing it to access any slave memory region.

***Figure 1.1. The EFM32 bus matrix***




## 1.1 DMA Channels

The EFM32 DMA consists of several channels which may be individually configured with a transfer mode, source and destination memory address. Each channel can also be set to trigger on a DMA request from a specific peripheral. For example, channel 0 may be used to move data received by the I2C to RAM, channel 2 may move conversion data from the ADC to RAM, while channel 3 transfers data from flash to the EBI etc. The number of DMA channels available varies between the different EFM32 families.

## 1.2 General DMA Configuration

The configuration for the DMA transfers is split into 3 main areas:

- **DMA descriptors:** Each DMA channel has two associated channel descriptor structures which are normally located in RAM. These include the source and destination address for the channel as well as information on number of elements to transfer, data size, transfer type etc. When a channel is triggered, the DMA reads the associated descriptor from RAM, which includes the instructions on what actions the DMA should take. When the channel has finished the defined actions, the updated descriptors are written back to the RAM.

- **DMA registers:** Common configurations for the DMA, as well as DMA generated interrupts and trigger sources for the various channels are configured in the DMA registers. The location in RAM of the DMA descriptors are also set here.

- **Registers in trigger peripheral:** The DMA request signals from the peripherals are generated on various events in the peripherals, hence it is important to configure the peripherals correctly to

generate the desired DMA requests. These settings are documented in the chapter for the requesting peripheral in the reference manual.

The configuration of the above settings are described in more detail in the following chapters.

# 2 DMA Channel Configuration

## 2.1 Channel Descriptors

In order for a DMA transaction to take place, several parameters such as source/destination address and transfer length must be specified. The configuration is partly stored in a descriptor normally located in RAM. Depending on the number of channels in the DMA, the start address of the first descriptor must be aligned in memory so that the lowest N+5 bits in the pointer address are zero, where N is the number of DMA channels. Each descriptor consists of 4 configuration words:

- **Source address:** Pointer to *END* address of source memory location. Starting address is decoded by the DMA by using the transfer count configuration.
- **Destination address:** Pointer to *END* address of destination memory location. Starting address is decoded by the DMA by using the transfer count configuration.
- **Configuration:** Data size, increment, transfer count (n_minus_1), arbitration rate (R_power), transfer mode etc.
- **User data:** Used by *emlib* DMA-functions to store state information for each channel, but is not used by the DMA itself.

There are two descriptors per DMA channel, one *primary* and one *alternate*. Normally, only the *primary* descriptor is used, but certain transfer modes (as described below) use both descriptors. Note that even though the DMA descriptors are configured with end address for source and destination, the start address must be used with the *emlib*-functions as these include a conversion to the end address.

### 2.1.1 Source/Destination Address, Size, Length and Increment

The core parameters of the DMA transfer are the source and destination addresses which specifies from which location the data is to be fetched and to where it is to be transferred. The size parameter specifies the size of the transfer unit as either 1, 2 or 4 bytes. Additionally, the number of such unit transfers (n_minus_1 parameter) the DMA is to perform must be set. The increment is set individually for both the source and destination address, and indicates whether the address is to be incremented after transferring each unit. E.g. when transferring RX data from the USARTs RXDATA register, the source address is not to be incremented, as the data is read from the same location each time. The destination address however, must be incremented (e.g. by 1 byte) to gradually fill the receive buffer in RAM as the data elements are received. The transfer size would in this case also be 1 byte.

### 2.1.2 Transfer Mode

The DMA can operate in several different modes as specified in Table 2.1 (p. 5). This configuration is given in the channel's DMA descriptor.

***Table 2.1. DMA transfer modes***

| Mode | Description |
|---|---|
| Basic | This mode is generally used when transferring data to or from a peripheral. The peripheral asserts a DMA trigger each time new data is available or needed, and in response the DMA performs one unit transfer (e.g. transfers one byte to the USART_TXDATA register). Such unit transfers will be repeated until the number of transfers equals the specified transfer length. |
| Auto | This mode is used when both the source and destination location is ready to transfer all the data at once, e.g. when transferring data from flash to RAM. Once the transfer is started, the number of transfers specified by the transfer length is performed without waiting for any further triggers. |
| Ping-Pong | Each channel has one primary and one alternate structure. In pong-pong mode, the DMA first uses the primary descriptor and then switches automatically to the alternate structure when the primary has reached its transfer count. The CPU can then re-initialize the primary descriptor so this can take over again once the alternate descriptor is finished. This mode is useful when it is required that there is an active DMA at all times, and the throughput demands of the application does not allow for pauses when reconfiguring a descriptor in basic mode. E.g. when fetching data from the ADC running at maximum sample rate. |
| Scatter-Gather | In this mode the primary DMA descriptor is used to load new configurations into the alternate DMA descriptor. The DMA will first load the alternate structure and then perform the transfer specified by the alternate structure before loading a new alternate structure which is subsequently executed. In this manner a sequence of different tasks can be specified and performed without any run-time intervention by the CPU. |

For further details on the modes please refer to the DMA section in the reference manual.

### 2.1.3 DMA Access to Descriptors

When a DMA channel is triggered, the DMA first reads the associated descriptor (primary or alternate depending on transfer mode) from RAM. The DMA then performs the configured transfers and then writes the updated descriptor back to RAM. The remaining transfer number (n_minus_1) is the only variable that is changed by the DMA, the rest of the contents of the descriptor are kept constant.

## 2.2 DMA Register Configuration

The following settings are made in the registers in the DMA peripheral memory region.

### 2.2.1 DMA Trigger

When transferring data to or from a peripheral unit, the DMA request signal from the peripheral must be specified for the DMA channel to be used. Make sure that the peripheral is also set up correctly to produce the wanted DMA request signals.

### 2.2.2 DMA Interrupt

After a DMA cycle has completed (remaining transfer number is 0) the DMA channel is disabled and an interrupt request can be generated. If desired the DMA channel can be re-enabled within the DMA interrupt routine. To re-enable a channel the corresponding DMA_CHENS bits must be written and the n_minus_1 parameter in the descriptor must be re-configured to the desired number of elements.

### 2.2.3 DMA Channel Priority

If several DMA channels are used simultaneously, the lower numbered channel has highest priority. To override this order a high priority can be assigned to the most important channels by setting the corresponding bits in DMA_CHPRIS. If a DMA channel is set up to transfer a large number of elements, the R_power attribute in the DMA descriptor can be set to a lower value to allow re-arbitration to a pending higher priority channel during transfer.

# 3 DMA Operation

## 3.1 DMA-CPU Priority

Since the CPU and the different DMA channels operate on the same bus structure, contention is likely to occur. A round robin scheme is used for bus arbitration, so that the CPU and DMA will each have priority in every other cycle. This ensures that neither bus master is starved, but if the CPU is accessing the same AHB bus slave as the DMA, the DMA cycle will be prolonged by interleaved CPU accesses. If the timing for the DMA is tight, the CPU should be put to Sleep (EM1) or forced to access other AHB bus slaves to ensure that no extra cycles are introduced in the DMA transfer procedure. More information on the AHB bus system can be found in the "Memory and Bus System"-chapter in the reference manual for the device.

## 3.2 DMA in Energy Modes

In general the EFM32 must stay in EM1 or EM0 to use the DMA, but some peripherals like LEUARTs and LESENSE can request a DMA transfer while staying in EM2 (not covered in this application note). DMA requests can however be triggered by other peripherals while staying in EM2 or EM3 (peripheral must also be functional in same mode) as long as an interrupt is also enabled to wake the device up. When the interrupt wakes up the device, the DMA request will start to process in parallel with the CPU executing the interrupt routine.

## 3.3 DMA Cycle Timing

Each DMA cycle consist of the following stages (timing is given in Figure 3.1 (p. 6) for a transfer of one word from the ADC to RAM):

1. **Arbitration -** The DMA figures out which of the incoming requests has the highest priority and initiates the transfer.

2. **Read descriptors -** CTRL, Source and Destination pointer words are read from the RAM.

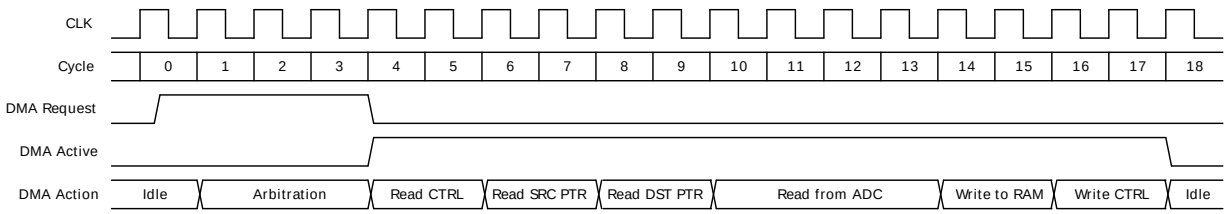
3. **Read from source**

4. **Write to destination**

5. Jump to 3 if n_minus_1>0 and number of transfers in this cycle is lower than the arbitration rate (R_power). Jump to 1 if number of transfers in this cycle equals the arbitration rate. n_minus_1 is decremented after evaluation if it is not already 0.

6. **Write descriptors -** Only CTRL is written as address pointers are never altered by the DMA

***Figure 3.1. DMA cycle moving one 16-bit word from ADC to RAM***



## 3.4 Debug Techniques

Debugging DMA behaviour poses some extra challenges compared to debugging CPU execution since it is not possible to halt or single step the DMA's operations. There are however some techniques that can be used to get a better view into what the DMA is doing. Such techniques include:

- Use memory view or variable watch points in the IDE to check that contents of any buffers in RAM are written correctly
- Use variable watch points for the DMA descriptors (`dmaControlBlock`, in software examples) to check that they are configured as expected.
- See DMA_CHREQSTATUS register to make sure the DMA request is set as expected.
- Use DMA_CHSWREQ to trigger DMA requests from SW to see that DMA responds as expected for each request.
- Use peripheral data underflow/overflow interrupts to raise an alarm if data is not transferred fast enough by DMA.
- Check if DMA_ERR flag is set. This indicates that the DMA is trying to access a memory region that is not mapped to a specific slave or trying to write to a read-only region (like Flash). This is most often caused by improperly set DMA descriptors or source/destination pointers that are incremented too far.

## 3.5 DMA functions in *emlib*

*emlib* includes several functions to easily set up and handle DMA operations. To initialize the DMA, em_dma.h and em_dma.c must be included in your project and the following functions must be run:

1. `DMA_Init` - Set up common DMA configuration registers.
2. `DMA_CfgChannel` - Set up DMA channel configuration registers.
3. `DMA_CfgDescr/DMA_CfgDescrScatterGather` - Configure alternate and primary descriptors in RAM.

Once a DMA channel has been configured, the channel can be activated in the desired mode by using one of these functions:

- `DMA_ActivateBasic` - Activate basic mode for a channel
- `DMA_ActivateAuto` - Activate auto mode for a channel
- `DMA_ActivatePingPong` - Activate ping-pong mode for a channel
- `DMA_ActivateScatterGather` - Activate scatter-gather mode for a channel
- `DMA_RefreshPingPong` - Refresh a primary or alternate descriptor when it has finished in ping-pong mode. `DMA_ActivatePingPong` must be run for the initial activation.

The *emlib* functions also include an interrupt handler for the DMA, which means that as long as these functions are used in your project, you cannot specify your own interrupt handler for the DMA. Instead the `DMA_CfgChannel()`-function allows you to register a call-back function for each DMA channel. When a DMA interrupt is triggered, the *emlib* interrupt handler will call the specified call-back function. The call-back function is usually used for tasks like refreshing descriptors in ping-pong mode and/or signalling to the application that a DMA cycle is finished.

# 4 Software Examples

This application note includes several software example demonstrates how to use the DMA in different modes and in with different peripherals. In the sections below more details about each example-stage are given.

## 4.1 Flash Transfer

The first example illustrates how to use the DMA to transfer data between to memory locations. A set of data located in flash is transferred to RAM using the DMA. In this transfer, the Auto setting is used, which makes the DMA transfer the whole buffer once it is triggered by the CPU. Upon completion the call-back function is called.

## 4.2 ADC Transfer

This example demonstrates how to use the DMA to transfer data from a peripheral to RAM. The DMA channel is set up to use the Basic mode, which makes the DMA transfer one unit each time the peripheral (i.e. the ADC in this example) sets its DMA trigger. When the specified number of units have been transferred, the corresponding call-back function is called. This example also uses a TIMER to trigger the ADC through the Peripheral Reflex System (PRS) at 100 kSamples/s.

## 4.3 ADC Transfer (Ping-Pong)

This example demonstrates how to read a continuous stream of data from the ADC running at full speed. To be able to service the ADC requests fast enough without gaps, ping-pong mode is used with the DMA. When one descriptor is finished, the other one takes over immediately at the next trigger. Meanwhile the CPU can reconfigure the first descriptor without halting the transfers. The DMA is stopped once the descriptors have been re-activated a number of times. As the ADC samples are ready every 13 ADCCLK cycles and one DMA transfer takes at least 17 HFCORECLK cycles, the ADCCLK must be prescaled to half the frequency of the HFPERCLK and HFCORECLK to allow 26 HFCORECLK cycles for the DMA transfer. In this example the HFRCO is running at 21 MHz, giving an ADCCLK of 10.5 MHz. This gives a sample rate of 10.5/13=0.81 Msamples/s. To achieve 1 Msamples/s exactly, an external clock/ crystal at 26 MHz must be used. For more information on ADC settings, please see the ADC specific application note.

## 4.4 SPI Master

This example shows how to implement an SPI master using the DMA to feed the data to transmit from a configurable buffer and transferring the incoming data to a RAM buffer using basic mode in the DMA. As the DMA is mostly useful to transfer larger chunks of data, the CPU will in most cases still be used to send initialization commands etc. to the slave. Although such commands are not included in this example, the user can easily add code to access the USART in this example by following the guidelines in the SPI specific application note.

## 4.5 SPI TX (Ping-Pong)

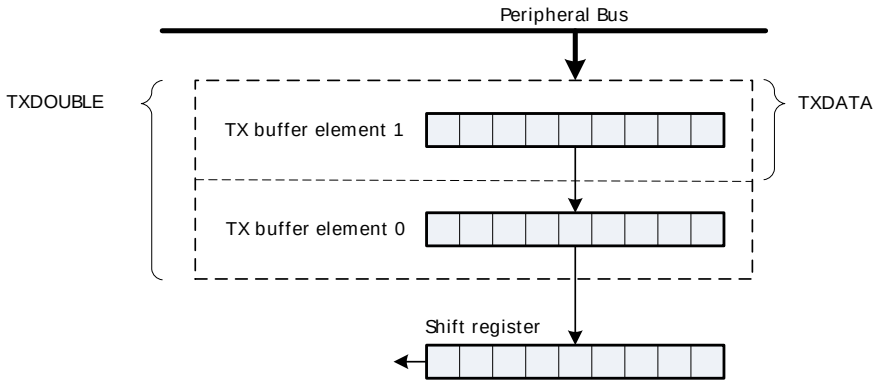
In master SPI mode, the USART in the EFM32 is able to transfer data with a baudrate at up to half the HFPERCLK frequency. This example shows how to send SPI data from a buffer continuously without gaps (given that the CPU does not generate too much bus traffic) using the DMA. Ping-pong mode is used to always have one descriptor armed and ready to transfer data.

### 4.5.1 SPI Buffer Timing

The USART contains 2-level buffers for the SPI frames. As we want to transfer data continuously we cannot allow these buffers to empty completely before there are no more frames to send. Figure 4.1 (p. 9) shows how the buffers are implemented in the USART when using 8-bit frames. The USART has two DMA requests linked to the TX buffer:

- The *TXEMPTY* request is set if both buffer elements and the shift register is empty. The request is cleared once new data is written to the buffers.
- The *TXBL* request is set if both buffer elements are empty. The request is cleared once at least one of the buffer elements are filled.

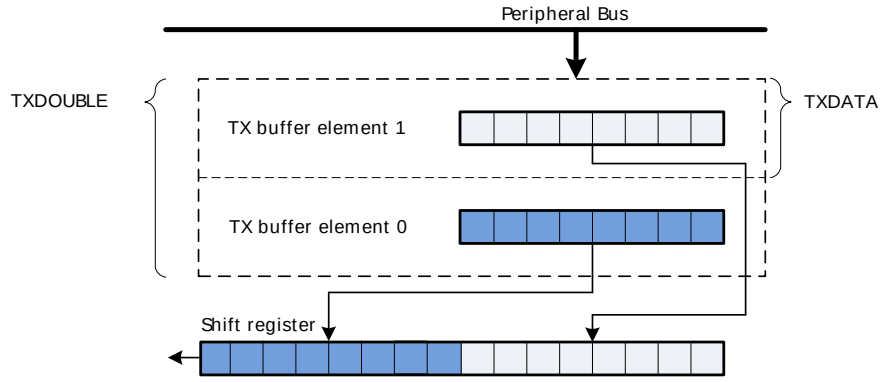
***Figure 4.1. USART buffering with 8-bit frames***




Since we want to transmit data continuously in our example we cannot use TXEMPTY, since there would be a delay from the time this request is sent out to the time the next data is actually filled in the buffer by the DMA. Instead we choose the TXBL request. Then we will be able to fill two frames at a time with the DMA, but we will only have the transmit time of one frame to complete this before the shift register is emptied. As 0.5 bits are sent every cycle, transferring 8 bits at a time would only leave us with 16 cycles for the DMA to for each transfer from the RAM to the USART. As transferring one element (8, 16 or 32 bits) with the DMA takes at least 17 cycles, hence continuous transmit is not possible with 8-bit frames at a SPI clock at half the system clock rate.

To give the DMA enough time to transfer an element before the shift register is empty, we can instead use 16-bit frames. As seen in Figure 4.2 (p. 10), the shift register is then also 16-bits wide, allowing 32 cycles to complete the transfer of each DMA element. The DMA then transfers 16-bit data to the TXDOUBLE register.

***Figure 4.2. USART buffering with 16-bit frames***



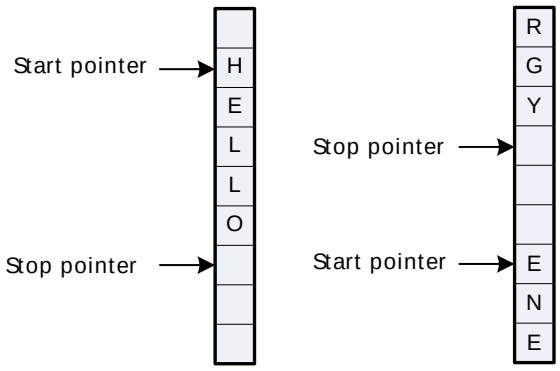

With an SPI clock rate of half the system clock we see that we do not have enough time to both transfer TX data and RX data with the SPI. Gap less TX and RX can however be done at higher clock division rates for the SPI clock, by adding a second descriptor for the RX data as done in the SPI Master example.

## 4.6 UART RX - TX

This example shows how to implement UART receive and transmit using the DMA to transfer incoming and outbound data between RAM buffers and the UART. This implementation also handles varying number of frames in a message. The end of a message is detected when the bus has been inactive (no falling edges) for longer than a configurable time period (set to 3 seconds in example). To detect the timeout, a running TIMER is set up to listen to the RX line and reset itself whenever it detects a falling edge. If no edges reset the counter value it will reach the overflow value after 3 seconds and trigger an interrupt alerting the CPU that a full frame has been received. The TIMER is connected to the RX GPIO pin through the Peripheral Reflex System.

A ring buffer is used to handle the stream of incoming messages the RAM without overwriting old received data before it has been read by the CPU. A start and stop point is used to keep track of which slots in the buffer contains the unread elements (Figure 4.3 (p. 10) ).

***Figure 4.3. UART RX ring buffer***




The CPU stays in Sleep (EM1) until the TIMER interrupt wakes it up when new a message has been received. The new message can then by read by calling `int getUartData(char *buffer, int maxLength);`, which pulls the available number of characters from the ring buffer. In the example, the case of the letters in the received message is then inverted before the message is sent back by calling `void sendUartData(char *buffer, int maxLength);`. This example is made to run on the EFM32G_DK3550 using the RS232 connector to communicate with a terminal program on a computer.

Some DVK-functions are run initially in the code to enable the connection to the RS232 port on the motherboard. The UART is configured to 115200 baud/s with 8 data bits, no parity bits and 1 stop bit.

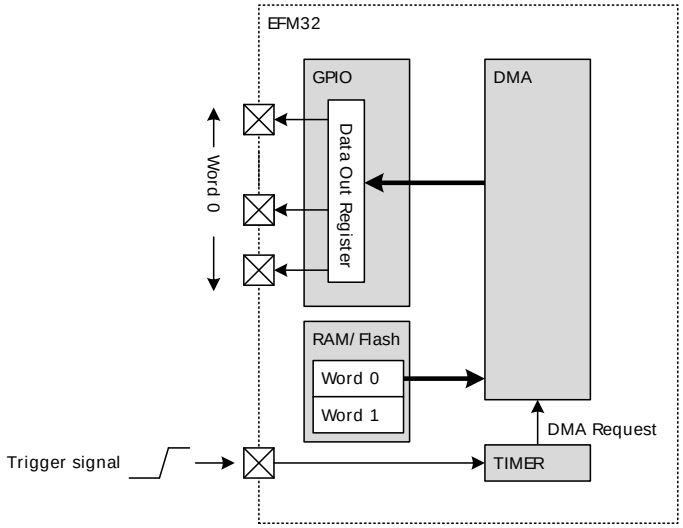
## 4.7 Scatter-Gather Transfer

In this example a sequence of 3 transfers is executed using the Scatter-Gather mode. The primary structure is used to sequentially load the alternate structure with the three transfers to be performed. When the third transfer is completed, the call-back function is called and the example is terminated by entering an infinite while-loop.

## 4.8 GPIO Trigger

This example shows how to trigger a DMA request from a transition on a GPIO pin. The DMA then fetches a new word of data from the RAM for every request and puts this on the Data Out Register (DOUT) of a GPIO port (Figure 4.4 (p. 11)). As the GPIO pins are not able to generate DMA requests themselves, a TIMER is set up to listen to a GPIO pin. The TIMER is started when a falling edge is detected on the pin. The TOP value of the TIMER is set to 0 which will immediately trigger a DMA request once the TIMER is started. Normally a TIMER DMA request is cleared when the DMA reads from the TIMER's count or capture registers. In this case however, we are transferring a word of data from the RAM to the GPIO and the TIMER registers are not accessed by the DMA. To still be able to clear the channel the DMACLRACT bit in TIMERn_CTRL is set. When this bit is set, the DMA request is cleared when the DMA channel that is set up to listen to the TIMER requests, is active.

***Figure 4.4. Data transfer to GPIO triggered by transition on GPIO pin***




## 4.9 I2C Master

This example shows how to perform I2C transfers (read and write) with DMA when acting as an I2C master. The example writes a sequence of bytes and then reads them back from the I2C-connected EEPROM (24AA024) on the EFM32GG-DK3750 kit. The example code assumes that the EFM32 is the only master on the I2C bus.

Note that while most of the code is generic, the 24AA024 EEPROM requires the master to transmit an 8-bit offset value before reading or writing any data. This value is sent by software before starting the DMA transfer and this part of the code may need to be changed when using a different I2C device.

See also AN0011 for more information and examples on I2C.

# 5 Revision History

## 5.1 Revision 2.05

2013-11-11

Added I2C Master Example

## 5.2 Revision 2.04

2013-10-14

New cover layout

Fixed a bug in uart_rx_tx.c where uartGetData() would return the wrong value.

## 5.3 Revision 2.03

2013-05-08

Added software projects for ARM-GCC and Atollic TrueStudio.

Specified UART settings for uart_rx_tx example.

## 5.4 Revision 2.02

2012-11-12

Adapted software projects to new kit-driver and bsp structure.

## 5.5 Revision 2.01

2012-08-15

Fixed bug using INT_Enable() in examples.

Fixed DMA struct alignment bug for devices with 12 DMA channels in examples.

## 5.6 Revision 2.00

2012-07-05

Major updates to description in PDF document.

Added DMA timing and debugging information.

Split DMA examples into separate projects.

Added spi_master, spi_tx_ping_pong, uart_tx_rx and gpio_trigger examples.

## 5.7 Revision 1.04

2012-04-20

Adapted software projects to new peripheral library naming and CMSIS_V3.

## 5.8 Revision 1.03

2012-03-14

Fixed makefile-error for CodeSourcery projects.

## 5.9 Revision 1.02

2011-03-24

Changed some incorrect function descriptions in dmaexample.c .

## 5.10 Revision 1.01

November 16th, 2010.

Changed example folder structure, removed build and src folders.

Added chip-init function.

## 5.11 Revision 1.00

September 20th, 2010.

Initial revision.

# A Disclaimer and Trademarks

## A.1 Disclaimer

*Silicon Laboratories intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Laboratories products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Laboratories reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Laboratories shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products must not be used within any Life Support System without the specific written consent of Silicon Laboratories. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Laboratories products are generally not intended for military applications. Silicon Laboratories products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.*

## A.2 Trademark Information

# B Contact Information

**Silicon Laboratories Inc.**
400 West Cesar Chavez
Austin, TX 78701

Please visit the Silicon Labs Technical Support web page:
http://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.

# Table of Contents

# List of Figures

# List of Tables

silabs.com


ZERO
ARM Cortex-M0+
TINY
ARM Cortex-M3
GECKO
ARM Cortex-M3
LEOPARD
ARM Cortex-M3
GIANT
ARM Cortex-M3
WONDER
ARM Cortex-M4