AN1029: Linked Direct Memory Access (LDMA) Controller

This application note demonstrates how to use the linked direct memory access (LDMA) controller in the EFM32 Gecko Series 1 and EFR32 Wireless Gecko Series 1 devices.

Several software examples are provided that shows how to use the various transfer modes of the LDMA. The example projects are configured for the EFM32 Pearl Gecko, but can easily be ported to other EFR32 Wireless Gecko devices by changing the project settings.

For simplicity, EFM32 Wonder Gecko, Gecko, Giant Gecko, Leopard Gecko, Tiny Gecko, Zero Gecko, and Happy Gecko are a part of the EFM32 Gecko Series 0.

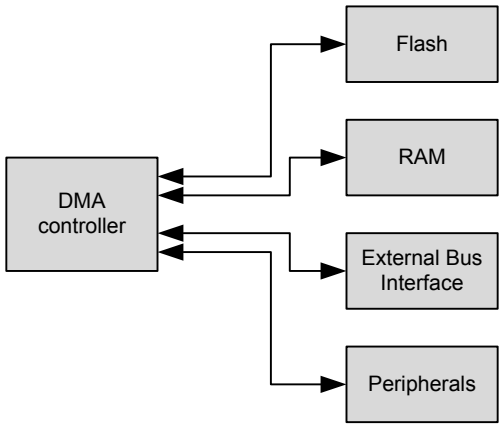
EZR32 Wonder Gecko, Leopard Gecko, and Happy Gecko are a part of the EZR32 Wireless MCU Series 0.

EFM32 Pearl Gecko and Jade Gecko (and future devices) are a part of the EFM32 Gecko Series 1.

EFR32 Blue Gecko, Flex Gecko and Mighty Gecko are a part of the EFR32 Wireless Gecko Series 1.

KEY POINTS

- Comparison between µDMA and LDMA.
- LDMA configuration and operation.
- The DMADRV is a high-level library that easily enables use of the DMA.
- This application note includes:
  - This PDF document
  - Source files
    - Example C-code
    - Multiple IDE projects



1. Introduction

The DMA is used for data transfer without CPU intervention. Data can be transferred between any readable source address and writable destination address within the CPU address space and can be initiated either by a peripheral setting a DMA request signal or by the CPU directly. While the DMA is handling the data transfer, the CPU is free to do other work or stay in low energy modes in order to save energy. Upon completion, the DMA can wake up the CPU by triggering an interrupt DMA channel.

The LDMA of EFM32 Gecko Series 1 and EFR32 Wireless Gecko Series 1 devices consists of several channels which can be individually configured, and the number of DMA channels available may vary between the different product families. Each channel can be set to trigger on a DMA request from a specific peripheral and it can also be triggered directly by software, which is useful for memory-to-memory transfers.

1.1 General LDMA Configuration

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

- LDMA Channel registers and Channel descriptor:

Each LDMA channel has associated channel descriptor(s) 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 LDMA reads the associated channel descriptor from RAM to channel descriptor registers with no CPU intervention, which includes the instructions on what actions the LDMA should take.

- LDMA registers:

Common configurations for the LDMA are configured in the LDMA registers as well as LDMA generated interrupts and trigger sources for the various channels.

- Registers in trigger peripheral:

The DMA request signals from the peripherals are usually generated on various events in the peripherals. Because of this, 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 EFM32 Gecko Series 1 and EFR32 Wireless Gecko Series 1 device reference manuals.

1.2 DMA Comparison

There are two different DMA Controllers, µDMA and LDMA, for the EFM32 Gecko Series 0 and 1, EZR32 Series 0, and EFR32 Wireless Gecko Series 1 devices. See the table below for a comparison between the two DMA controllers.

Table 1.1. DMA Comparison

Item	LDMA	µDMA
Product family	EFM32 Gecko Series 1 and EFR32 Wireless Gecko Series 1	EFM32 Gecko Series 0 and EZR32 Series 0
Data transfer	Memory <-> Peripheral Memory <-> Memory Peripheral <-> Peripheral	Memory <-> Peripheral Memory <-> Memory
Number of DMA transfers	2048 (maximum)	1024 (maximum)
Transfer FIFO	16 x 32 bits (burst reads/writes)	N
Transfer modes: Basic, Ping-Pong, Scatter-gather	Y	Y
Loop transfer	Y	Channels 0 & 1 in EFM32LG, EFM32GG, EFM32WG, EZR32LG and EZR32WG
2D copy	Y	Channel 0 in EFM32LG, EFM32GG, EFM32WG, EZR32LG and EZR32WG
Little-endian/Big-endian conversion	Y	N
Inter-channel and hardware event synchronization	Y (hardware events to pause and restart a DMA sequence)	N
DMA write-immediate function	Y (write a constant anywhere in the memory map)	N
Debug halt	Y	N
PRS trigger	Y	N
Channel descriptor data structure	XFER, SYNC, and WRI descriptors	Primary and Alternate descriptors
Channel descriptor data structure alignment	Word aligned, fixed 16 bytes offset if using relative addressing	Contiguous, word aligned, and proper space alignment

2. LDMA Configuration

The channel descriptors determine what the Linked DMA Controller will do when it receives DMA transfer request. The initial descriptor is written directly to the LDMA's channel registers (see 7.1 Single Direct Register DMA Transfer). If desired, the initial descriptor can link to additional linked descriptors stored in memory (RAM or flash). Alternatively, software may also load the initial descriptor by writing the descriptor address to the LDMA_CHx_LINK register and then setting the corresponding bit in the LDMA_LINKLOAD register (see Figure 2.4 Descriptor List Operation Flow on page 9).

Before enabling a channel, the software must take care to properly configure the channel registers including the link address and any linked descriptors. When a channel is triggered (see Table 3.1 Start a LDMA Transfer on page 12), the LDMA Controller will perform the memory transfers as specified by the descriptors.

2.1 Channel Descriptor Registers

In order for a DMA transaction to take place, several parameters such as source/destination address and transfer length must be specified. Each LDMA channel has descriptor registers to store this configuration. A transfer can be initialized by software writing to the registers or by the LDMA itself copying a descriptor from memory to registers.

- LDMA_CHx_CTRL: Channel Descriptor Control Word Register
- LDMA_CHx_SRC: Channel Descriptor Source Data Address Register
- LDMA_CHx_DST: Channel Descriptor Destination Address Register
- LDMA_CHx_LINK: Channel Descriptor Link Structure Address Register

CHx is from 0 to the maximum DMA channel number of the device (e.g., CH0 to CH7 in the EFM32PG1 Pearl Gecko device).

The contents of the descriptor registers are dynamically updated during the DMA transfer. The contents of descriptors in memory are not edited by the controller.

For further details on channel descriptor registers, refer to the LDMA section in the EFM32 Gecko Series 1 and EFR32 Wireless Gecko Series 1 device reference manuals.

2.1.1 LDMA_CHx_CTRL Register

The corresponding bit fields of the LDMA_CHx_CTRL register are described in the following figure and table.

Offset	Bit Position																													
0x08C	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0x0		0x0		0x0		0	0	0	0	0x0				0	0x000											0		0x0	
Access	R	R	RWH		RWH		RWH		RWH	RWH	RWH	RWH	RWH				RWH	RWH											W1		R	
Name	DSTMODE	SRCMODE	DSTINC		SIZE		SRCINC		IGNORESREQ	DECLOOPCNT	REQMODE	DONEIFSEN	BLOCKSIZE				BYTESWAP	XFERCNT											STRUCTREQ		STRUCTTYPE	

Figure 2.1. LDMA_CHx_CTRL Register

Table 2.1. Bit Fields of the LDMA_CHx_CTRL Register

Bit Field	Description
DSTMODE	Destination addressing mode of the linked descriptor, this bit is read only.
SRCMODE	Source addressing mode of the linked descriptors, this bit is read only.
IGNORESREQ	The LDMA will ignore Single Request (SREQ) and wait for a full Request (REQ) signal when this bit is set (see 7.3 Single Descriptor Looped Transfer).
DECLOOPCNT	Enable loop transfer (see 2.3.3 Loop Counter)
REQMODE	Transfers one BLOCKSIZE or XFERCNT (all units) per transfer request.
DONEIFSEN	Set the interrupt flag when the transfer is done (see 3.4 Interrupts).
SIZE	Data width of one DMA transfer, the LDMA supports byte (1 byte), half-word (2 bytes) and word sized (4 bytes) transfers.
SRCINC/ DSTINC	A DMA transfer is the smallest unit of data (depends on the value of the SIZE field) that can be transferred by the LDMA. The increments is in units of DMA transfers, it can be 1, 2 or 4 unit data size(s). Determines the increment size for source/destination between DMA transfers, the LDMA can pack or unpack data by using a different increment size for source and destination. This field may also be set to NONE which will cause the LDMA to read or write the same location for every DMA transfer. This is useful for accessing peripheral FIFO or data registers.
XFERCNT	Defines how many DMA transfers to perform, the maximum is 2048. The number of bytes transferred by the descriptor will depend on both the transfer count XFERCNT and the SIZE field settings. Total bytes = XFERCNT * SIZE

Bit Field	Description
BLOCKSIZE	Defines the amount of data transferred in one arbitration, the maximum is 1024. The number of DMA transfers that need to be done is specified by the XFERCNT field. When XFERCNT > BLOCKSIZE and is not an integer multiple of BLOCKSIZE then the controller always performs sequences of BLOCKSIZE transfers until XFERCNT < BLOCKSIZE remain to be transferred. The controller performs the remaining XFERCNT transfers at the end of the DMA cycle.
BYTESWAP	Reverses the endianness (little-endian/big-endian conversion) of the incoming source data read into the LDMA's FIFO. Byte swap is only valid for transfer sizes of word and half-word.
STRUCREQ	Trigger a transfer if this bit is set in linked descriptor (see 3.1 Starting a Transfer).
STRUCTTYPE	Only used for linked descriptors, this bit is read only (see 2.2.1 Descriptor Data Structure Type).

2.1.2 LDMA_CHx_SRC Register

This register is a pointer (SRCADDR field) to the address of the next transfer source memory location. The value of this register is unchanged, incremented, or decremented with each source read. The LDMA will update the source address after each transfer.

2.1.3 LDMA_CHx_DST Register

This register is a pointer (DSTADDR field) to the address of the next transfer destination memory location. The value of this register is unchanged, incremented, or decremented with each destination read. The LDMA will update the destination address after each transfer.

2.1.4 LDMA_CHx_LINK Register and Addressing Modes

When a descriptor is finished the LDMA will either halt or load the next linked descriptor depending on the value of the LINK field in the LDMA_CHx_LINK register. If the LINK bit is set, the DMA will load the next linked descriptor. If the next linked descriptor also has this bit set, the DMA will load the next linked descriptor until this bit is 0 in the loaded descriptor.

Offset	Bit Position																													
0x098	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0x00000000																														0	0
Access	RWH																														RWH	R
Name	LINKADDR																														LINK	LINKMODE

Figure 2.2. LDMA_CHx_LINK Register

Note that the linked descriptor must be word aligned in memory. The two least significant bits of the LDMA_CHx_LINK register are used by the LINK and LINKMODE bits. The two least significant bits of the link address (LINKADDR) are always zero.

Relative addressing is most useful for the link address. The initial descriptor will indicate the absolute address of the linked descriptors in memory. The linked descriptors might be an array of structures. In this case the offset between descriptors is constant and is always 16 bytes (four 32-bit words). The LINK address is not incremented or decremented after each transfer. Thus, a relative offset of 0x10 may be used for all linked descriptors.

Table 2.2. Absolute Addressing Mode Versus Relative Addressing Mode

Item	Absolute Addressing (LINKMODE = 0)	Relative Addressing (LINKMODE = 1)
Initial descriptor	Mandatory	LINKMODE bit is ignored
Next action	LINK = 0, DMA stops LINK = 1, DMA loads the next linked descriptor	LINK = 0, DMA stops LINK = 1, DMA loads the next linked descriptor
Linked descriptors are in contiguous memory (an array of channel descriptors)	LINKADDR is the absolute address of next descriptor	Offset between descriptors is constant and is always 16 bytes. A relative offset (LINKADDR) of 16 points to the previous (minus) or next (plus) descriptor. A relative offset (LINKADDR) of multiples of 16 skips one or multiple descriptor(s) from the linked list.

2.2 Channel Descriptor Data Structure

Each channel descriptor consists of four 32-bit words: CTRL, SRC, DST and LINK. These words map directly to the LDMA_CHx_CTRL, LDMA_CHx_SRC, LDMA_CHx_DST and LDMA_CHx_LINK registers.

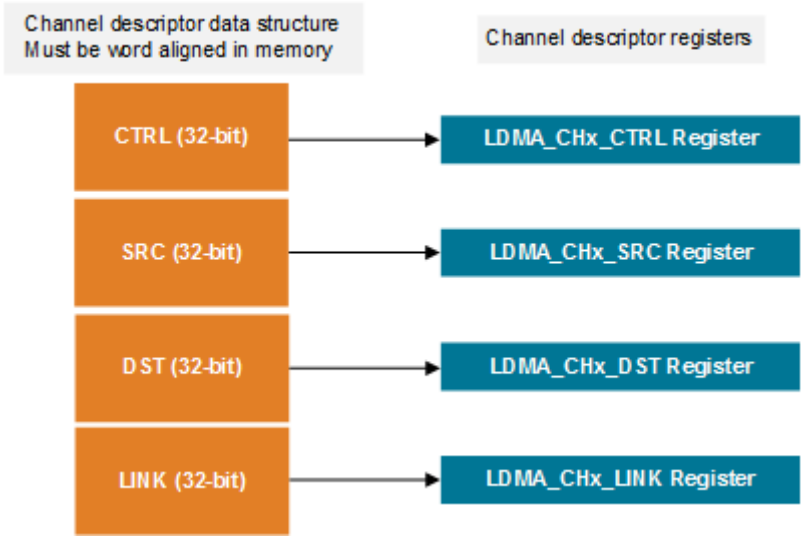


Figure 2.3. Channel Descriptor Data Structure

2.2.1 Descriptor Data Structure Type

There are three different types of descriptor data structures: XFER, SYNC and WRI. The usage of the SRC and DST fields may differ depending on the structure type.

The SYNC descriptors do nothing until a condition is met. The condition is formed by the SYNCTRIG field in the LDMA_SYNC register and the MATCHEN and MATCHVAL fields of the descriptor. When (SYNCTRIG & MATCHEN) == (MATCHVAL & MATCHEN) the next descriptor is loaded. In addition to waiting for the condition a Link descriptor can set (SYNCSET) or clear (SYNCCLR) bits in SYNCTRIG to meet the conditions of another channel and cause it to continue. The MCU also has the ability to set and clear the SYNCTRIG bits from software.

For further details on channel descriptor data structure, refer to the LDMA section in the EFM32 Gecko Series 1 and EFR32 Wireless Gecko Series 1 device reference manuals.

Table 2.3. Different Types of Descriptor Data Structures

Item	XFER	SYNC	WRI
CTRL	Maps to LDMA_CHx_CTRL	DONEIFSEN and STRUCTTYPE field only	DONEIFSEN and STRUCTTYPE field only
SRC	Maps to LDMA_CHx_SRC	SYNCSET and SYNCCLR	IMMVAL
DST	Maps to LDMA_CHx_DST	MATCHVAL and MATCHEN	DSTADDR
LINK	Maps to LDMA_CHx_LINK	Maps to LDMA_CHx_LINK	Maps to LDMA_CHx_LINK
STRUCTTYPE in CTRL	0	1	2
Usage	Defines a typical data transfer Memory <-> Peripheral Memory <-> Memory Peripheral <-> Peripheral	Allows the channel to wait for external stimulus to proceed to the next descriptor Provides stimulus to another channel to indicate that it may continue	Allows a list of descriptors to write a value to a register or memory location DSTADDR = IMMVAL

2.2.2 Descriptor List

The different DMA transfer modes are implemented by the descriptor list in memory. A descriptor list consists of one or more descriptors that are serially executed.

Table 2.4. Types of Descriptor List

Descriptor Type	Usage
Single XFER	Transfers required bytes of data and then stops (see 7.1 Single Direct Register DMA Transfer)
Linked XFER	Transfers required bytes of data and then loads the next linked descriptor (see 7.2 Descriptor Linked List)
Loop XFER	Transfers required bytes of data and then performs loop control (see 7.3 Single Descriptor Looped Transfer and 7.4 Descriptor List with Looping)
SYNC	Handles synchronization of the list with other entities (see 7.5 Simple Inter-Channel Synchronization)
WRI	Writes a value to a location in memory (see 7.5 Simple Inter-Channel Synchronization)

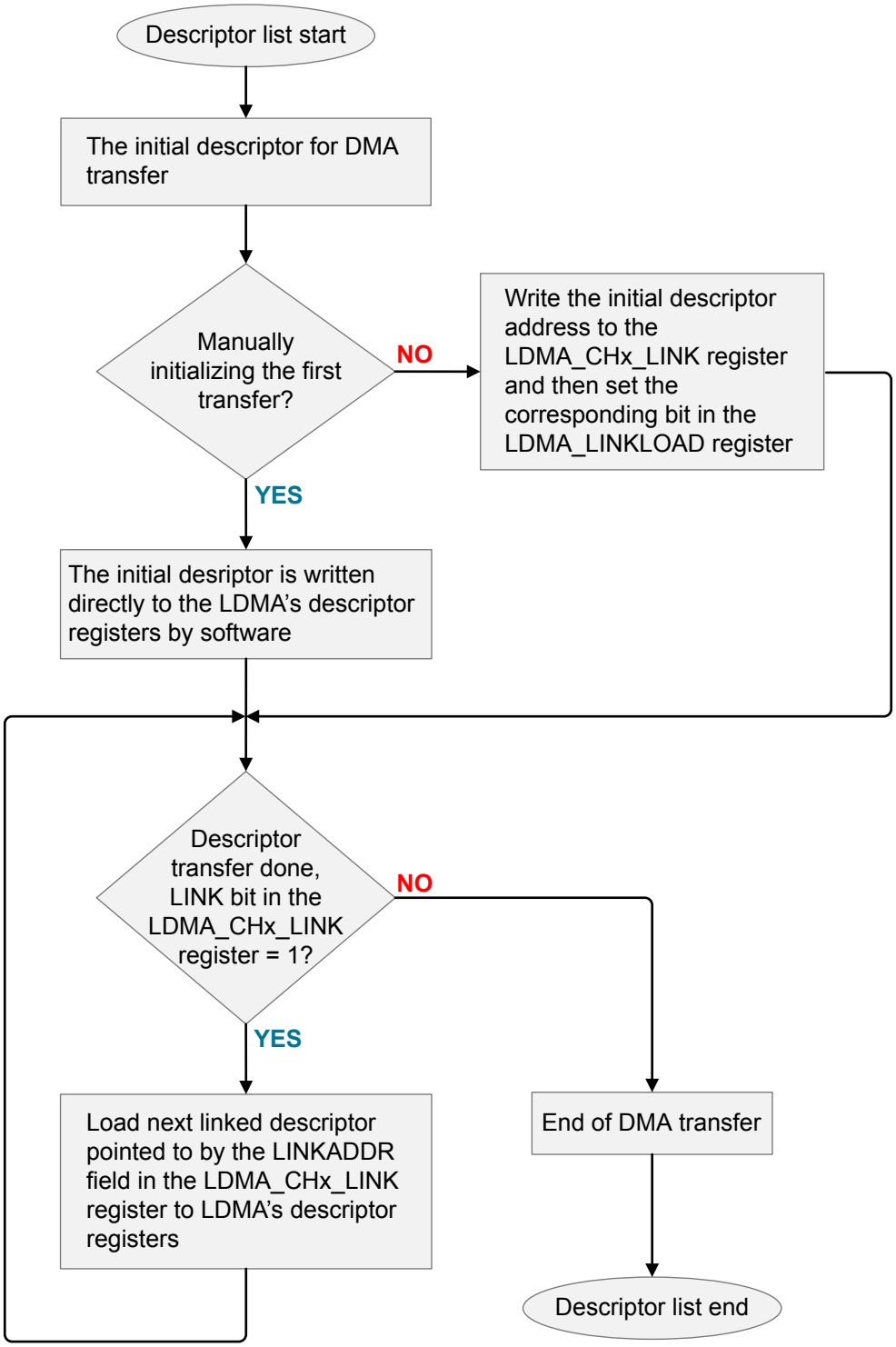


Figure 2.4. Descriptor List Operation Flow

2.3 Channel Configuration

Each DMA channel has associated configuration and loop counter registers for controlling the direction of address increment, arbitration slots, and descriptor looping.

- LDMA_CHx_CFG: Channel Configuration Register
- LDMA_CHx_LOOP: Channel Loop Counter Register (maximum value is 255)

CHx is from 0 to the maximum DMA channel number of the device (e.g., CH0 to CH7 in the EFM32PG1 Pearl Gecko device).

2.3.1 Address Increment/Decrement

The SRCINCSIGN and DSTINCSIGN bits in the LDMA_CHx_CFG register of each channel control whether the source and destination addresses increment or decrement after each DMA transfer.

Table 2.5. Data Flipping with LDMA

DSTINCSIGN	SRCINCSIGN	Destination Address	Source Address	Transfer
0	0	Increment	Increment	Copy data from the top
0	1	Increment	Decrement	Flip the data (tail to head)
1	0	Decrement	Increment	Flip the data (head to tail)
1	1	Decrement	Decrement	Copy data from the bottom

The SRCINCSIGN and DSTINCSIGN bits apply to all descriptors used by that channel. Firmware should take care to set the starting source and/or destination address to the highest data address when decrementing.

2.3.2 Arbitration

The LDMA Controller supports both fixed priority and round robin arbitration. The number of fixed and round robin channels is programmable using the NUMFIXED field in the LDMA_CTRL register. For round robin channels, the number of arbitration slots requested for each channel is programmable using the ARBSLOT field in the LDMA_CHx_CFG register. Using this scheme, it is possible to ensure that timing-critical transfers are serviced on time.

For further details on arbitration slots, refer to the LDMA section in the EFM32 Gecko Series 1 and EFR32 Wireless Gecko Series 1 device reference manuals.

Table 2.6. Arbitration Priority

Item	Fixed Priority	Round Robin Priority
Priority level	Priority (channel 0 is the highest) decreases as the channel number increases. When the LDMA controller is idle or when a transfer completes, the highest priority channel with an active request is granted the next transfer.	Each active requesting channel is served in the order of priority (channel number). A late arriving request on a higher priority channel will not get serviced until the next round. Fixed priority channels always take priority over round robin channels.
Advantage	Guarantees smallest latency for the highest priority requesters.	Minimizes the risk of starving low-priority, latency-tolerant requesters.
Drawback	The possibility of starvation for lowest priority requesters.	Higher risk of starving low-latency requesters.
NUMFIXED=0	No channel in fixed priority.	All channels are round robin priority.
NUMFIXED=n	Channels 0 through (n-1) are fixed priority.	Channels n through maximum are round robin priority.
NUMFIXED= maximum-1	All channels are fixed priority (default setting after reset).	No channel in round robin priority.

2.3.3 Loop Counter

Each channel has a LDMA_CHx_LOOP register that includes a loop counter field (LOOPCNT). To use looping, software should initialize the loop counter with the desired number of repetitions before enabling the transfer. A descriptor with the DECLOOPCNT bit (field in the LDMA_CHx_CTRL register) set to TRUE will repeat the loop and decrement the loop counter until LOOPCNT = 0.

Table 2.7. Loop Transfer

Descriptor Setting	LINKADDR Field and LINK Bit	Action
LOOPCNT > 0 DECLOOPCNT = 1	LINKADDR in the LDMA_CHx_LINK register points to itself	Looping of single descriptor (See 7.3 Single Descriptor Looped Transfer)
LOOPCNT > 0 DECLOOPCNT = 1	LINKADDR in the LDMA_CHx_LINK register points to another descriptor	Load the descriptor pointed to by LINKADDR, enables looping of multiple descriptors (See 7.4 Descriptor List with Looping)
LOOPCNT reaches 0	LINK bit in the LDMA_CHx_LINK register is clear	DMA transfer stops
LOOPCNT reaches 0	LINK bit in the LDMA_CHx_LINK register is set LINKADDR in the LDMA_CHx_LINK register points to another descriptor	Continue execution after looping, load the descriptor pointed to by LINKADDR (See 7.4 Descriptor List with Looping)

Note that because there is only one LOOPCNT per channel, software intervention is required to update the LOOPCNT if a sequence of transfers contains multiple loops. It is also possible to use a write-immediate DMA data transfer (WRI) to update the LDMA_CHx_LOOP register.

2.4 Channel Select Configuration

The channel select block determines which peripheral request signal connects to each LDMA channel.

- LDMA_CHx_REQSEL: Channel Peripheral Request Select Register

CHx is from 0 to the maximum DMA channel number of the device (e.g., CH0 to CH7 in the EFM32PG1 Pearl Gecko device).

This configuration is done by firmware through the SOURCESEL and SIGSEL fields of the LDMA_CHx_REQSEL register. SOURCESEL selects the peripheral and SIGSEL picks which DMA request signals to use from the selected peripheral. Make sure that the peripheral is also set up correctly to produce the desired DMA request signals.

3. LDMA Operation

3.1 Starting a Transfer

A LDMA transfer may be started by firmware, a peripheral request, or a descriptor load as described in the table below.

Table 3.1. Start a LDMA Transfer

DMA Trigger	Setting
Firmware	The SOURCESEL in the LDMA_CHx_REQSEL register of desired channel should set to NONE to prevent unintentional triggering. Set the bit for the desired channel in the LDMA_SWREQ register.
Peripheral request	Configure the peripheral source and signal as described in 2.4 Channel Select Configuration.
Descriptor load	The LDMA can be configured to begin a transfer immediately after a new descriptor is loaded by setting the STRUCTREQ field of the LDMA_CHx_CTRL register.

3.2 Managing Transfer Errors

Firmware should clear the ERROR bit in the LDMA_IF register and enable error interrupts by setting the ERROR bit in the LDMA_IEN register before initiating a DMA transfer.

If the ERROR bit of the LDMA_IF register is set in the LDMA interrupt handler, firmware should then read the CHERROR field in the LDMA_STATUS register to determine the errant channel. The interrupt handler should reset this channel and clear the ERROR bit in the LDMA_IF register before returning.

3.3 Interaction with the EMU

In general, the EFM32 Gecko Series 1 and EFR32 Wireless Gecko Series 1 devices must stay in EM0 or EM1 to use the LDMA, but some peripherals like LEUART and ADC can request a DMA transfer while staying in EM2 (See *"AN0017 Low Energy UART"* for details).

DMA requests can however be triggered by other peripherals while staying in EM2 or EM3 (the peripheral must also be functional in the 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.

3.4 Interrupts

The LDMA_IF Interrupt flag register contains one DONE bit for each channel and one combined ERROR bit. When enabled, these interrupts are available as interrupts to the Cortex-M core and are combined into one LDMA interrupt vector. If the interrupt for the LDMA is enabled in the ARM Cortex-M core, an interrupt will be made if one or more of the interrupt flags in LDMA_IF and their corresponding bits in LDMA_IEN are set.

Interrupts may optionally be signaled to the CPU's interrupt controller at the end of any DMA transfer or at the completion of a descriptor if the DONEIFSEN bit in the LDMA_CHx_CTRL register is set.

Table 3.2. LDMA Interrupt Registers

Condition	LDMA_IF Register	LDMA_CHDONE Register
DONEIFSEN bit in the LDMA_CHx_CTRL register is set to 1	Corresponding channel DONE bit is set when EACH channel descriptor finishes execution	Corresponding channel CHDONE bit is set when the final channel descriptor finishes execution (entire transfer is done)
DONEIFSEN bit in the LDMA_CHx_CTRL register is cleared to 0	Corresponding channel DONE bit is set when the final channel descriptor finishes execution (entire transfer is done)	Corresponding channel CHDONE bit is set when the final channel descriptor finishes execution (entire transfer is done)

The corresponding CHDONE bit in the LDMA_CHDONE register can be cleared by LDMA interrupt service routine or re-enabling the corresponding LDMA channel.

3.5 Debugging

For a peripheral request DMA transfer, the LDMA will halt during a debug halt if a bit for a channel in the LDMA_DBGHALT register is set. Then the contents of the LDMA registers, channel descriptors, and any buffers in RAM can be checked by the register view and variable watch points in the IDE. Otherwise, during debug halt the LDMA will continue to run and complete the entire transfer.

The peripheral data underflow or overflow interrupts raise an alarm if data is not transferred fast enough by the LDMA.

4. LDMA Initializers and Functions in emlib

The emlib includes certain initializers (in `em_ldma.h`) and functions (in `em_ldma.c`) to easily setup and handle the LDMA operations.

4.1 LDMA Initializers

The pre-defined LDMA initializers are the basic framework for firmware to initialize and configure the LDMA for simple and complex data transfers.

4.1.1 LDMA Initialization

The initializer `LDMA_INIT_DEFAULT` for LDMA initialization is based on the defined structure `LDMA_Init_t`. It configures all LDMA channels in fixed priority arbitration, disables the PRS SYNCTRIG CLEAR and SET and programs the LDMA interrupt priority to 3.

```
typedef struct
{
 uint8_t ldmaInitCtrlNumFixed; /**< Arbitration mode separator.*/
 uint8_t ldmaInitCtrlSyncPrsClrEn; /**< PRS Synctrig clear enable. */
 uint8_t ldmaInitCtrlSyncPrsSetEn; /**< PRS Synctrig set enable. */
 uint8_t ldmaInitIrqPriority; /**< LDMA IRQ priority (0..7). */
} LDMA_Init_t;
```

Table 4.1. Initializer for LDMA Initialization

LDMA_Init_t Structure Member	Related Register - Bit Field	LDMA_INIT_DEFAULT Initializer
ldmaInitCtrlNumFixed	LDMA_CTRL – NUMFIXED	Maximum - 1 (All channels in fixed priority arbitration)
ldmaInitCtrlSyncPrsClrEn	LDMA_CTRL – SYNCPRSCLREN	0 (Disable)
ldmaInitCtrlSyncPrsSetEn	LDMA_CTRL – SYNCPRSSETEN	0 (Disable)
ldmaInitIrqPriority	SCB and NVIC registers in Cortex-M	Priority 3 (0 is the highest priority)

4.1.2 LDMA Transfer Configuration

The initializers for LDMA transfer configuration in the table below are based on the defined structure `LDMA_TransferCfg_t`.

```
typedef struct
{
 uint32_t ldmaReqSel; /**< Selects DMA trigger source. */
 uint8_t ldmaCtrlSyncPrsClrOff; /**< PRS Synctrig clear enables to clear. */
 uint8_t ldmaCtrlSyncPrsClrOn; /**< PRS Synctrig clear enables to set. */
 uint8_t ldmaCtrlSyncPrsSetOff; /**< PRS Synctrig set enables to clear. */
 uint8_t ldmaCtrlSyncPrsSetOn; /**< PRS Synctrig set enables to set. */
 bool ldmaReqDis; /**< Mask the PRS trigger input. */
 bool ldmaDbgHalt; /**< Dis. DMA trig when cpu is halted. */
 uint8_t ldmaCfgArbSlots; /**< Arbitration slot number. */
 uint8_t ldmaCfgSrcIncSign; /**< Source addr. increment sign. */
 uint8_t ldmaCfgDstIncSign; /**< Dest. addr. increment sign. */
 uint8_t ldmaLoopCnt; /**< Counter for looped transfers. */
} LDMA_TransferCfg_t;
```

These initializers configure all aspects of a LDMA transfer and the corresponding emlib function will use these macros to initialize the LDMA_CTRL, LDMA_DBGHALT, LDMA_REQDIS, LDMA_CHx_REQSEL, LDMA_CHx_CFG and LDMA_CHx_LOOP registers.

Table 4.2. LDMA_TransferCfg_t Structure Member

LDMA_TransferCft_t Structure Member	Related Register - Bit Field	Usage
ldmaReqSel	LDMA_CHx_REQSEL – SOURCESEL LDMA_CHx_REQSEL – SIGSEL	Selects input source and signal to DMA channel
ldmaCtrlSyncPrsClrOff	LDMA_CTRL – SYNCPRSCLREN	Disables (if = 1) the corresponding PRS input to clear the respective bit in the SYNCTRIG field of the LMDA_SYNC register
ldmaCtrlSyncPrsClrOn	LDMA_CTRL – SYNCPRSCLREN	Enables (if = 1) the corresponding PRS input to clear the respective bit in the SYNCTRIG field of the LMDA_SYNC register
ldmaCtrlSyncPrsSetOff	LDMA_CTRL – SYNCPRSSETEN	Disables (if = 1) the corresponding PRS input to set the respective bit in the SYNCTRIG field of the LMDA_SYNC register
ldmaCtrlSyncPrsSetOn	LDMA_CTRL – SYNCPRSSETEN	Enables (if = 1) the corresponding PRS input to set the respective bit in the SYNCTRIG field of the LMDA_SYNC register
ldmaReqDis	LDMA_REQDIS - REQDIS	Disables (if TRUE) peripheral requests for the corresponding channel
ldmaDbgHalt	LDMA_DBGHALT - DBGHALT	Masks (if TRUE) the corresponding DMA channel's request when debugging and the MCU is halted
ldmaCfgArbSlots	LDMA_CHx_CFG – ARBSLOTS	Uses to select the number of slots in the round robin queue
ldmaCfgSrcIncSign	LDMA_CHx_CFG – SRCINCSIGN	Destination address increment sign
ldmaCfgDstIncSign	LDMA_CHx_CFG – DSTINCSIGN	Source address increment sign
ldmaLoopCnt	LDMA_CHx_LOOP - LOOPCNT	Specifies the number of iterations when using looping descriptors

Table 4.3. Initializers for LDMA Transfer Configuration

Initializer	Usage
LDMA_TRANSFER_CFG_MEMORY ()	Generic DMA transfer configuration for memory to memory transfers
LDMA_TRANSFER_CFG_MEMORY_LOOP (loopCnt)	Generic DMA transfer configuration for looped memory to memory transfers
LDMA_TRANSFER_CFG_PERIPHERAL (signal)	Generic DMA transfer configuration for memory to a peripheral or a peripheral to memory transfers
LDMA_TRANSFER_CFG_PERIPHERAL_LOOP (signal, loopCnt)	Generic DMA transfer configuration for looped memory to a peripheral or a looped peripheral to memory transfers

Table 4.4. Parameters for LDMA Transfer Configuration Initializers

Parameter	Usage
loopCnt	Counter for looped transfer
signal	LDMA trigger source (defined in `LDMA_PeripheralSignal_t` enumeration)

4.1.3 LDMA Transfer Descriptor

The initializers for LDMA transfer descriptor (XFER, SYNC and WRI) in the table below are based on the defined union `LDMA_Descriptor_t`.

```
typedef union
{
 /**
 * TRANSFER DMA descriptor, this is the only descriptor type which can be
 * used to start a DMA transfer.
 */
 struct
 {
 uint32_t structType : 2; /**< Set to 0 to select XFER descriptor type. */
 uint32_t reserved0 : 1;
 uint32_t structReq : 1; /**< DMA transfer trigger during LINKLOAD. */
 uint32_t xferCnt : 11; /**< Transfer count minus one. */
 uint32_t byteSwap : 1; /**< Enable byte swapping transfers. */
 uint32_t blockSize : 4; /**< Number of unit transfers per arb. cycle. */
 uint32_t doneIfs : 1; /**< Generate interrupt when done. */
 uint32_t reqMode : 1; /**< Block or cycle transfer selector. */
 uint32_t decLoopCnt : 1; /**< Enable looped transfers. */
 uint32_t ignoreSrec : 1; /**< Ignore single requests. */
 uint32_t srcInc : 2; /**< Source address increment unit size. */
 uint32_t size : 2; /**< DMA transfer unit size. */
 uint32_t dstInc : 2; /**< Destination address increment unit size. */
 uint32_t srcAddrMode: 1; /**< Source addressing mode. */
 uint32_t dstAddrMode: 1; /**< Destination addressing mode. */
 uint32_t srcAddr; /**< DMA source address. */
 uint32_t dstAddr; /**< DMA destination address. */
 uint32_t linkMode : 1; /**< Select absolute or relative link address.*/
 uint32_t link : 1; /**< Enable LINKLOAD when transfer is done. */
 int32_t linkAddr : 30; /**< Address of next (linked) descriptor. */
 } xfer;

 /** SYNCHRONIZE DMA descriptor, used for intra channel transfer
 * syncronization.
 */
 struct
 {
 uint32_t structType : 2; /**< Set to 1 to select SYNC descriptor type. */
 uint32_t reserved0 : 1;
 uint32_t structReq : 1; /**< DMA transfer trigger during LINKLOAD. */
 uint32_t xferCnt : 11; /**< Transfer count minus one. */
 uint32_t byteSwap : 1; /**< Enable byte swapping transfers. */
 uint32_t blockSize : 4; /**< Number of unit transfers per arb. cycle. */
 uint32_t doneIfs : 1; /**< Generate interrupt when done. */
 uint32_t reqMode : 1; /**< Block or cycle transfer selector. */
 uint32_t decLoopCnt : 1; /**< Enable looped transfers. */
 uint32_t ignoreSrec : 1; /**< Ignore single requests. */
 uint32_t srcInc : 2; /**< Source address increment unit size. */
 uint32_t size : 2; /**< DMA transfer unit size. */
 uint32_t dstInc : 2; /**< Destination address increment unit size. */
 uint32_t srcAddrMode: 1; /**< Source addressing mode. */
 uint32_t dstAddrMode: 1; /**< Destination addressing mode. */
 uint32_t syncSet : 8; /**< Set bits in LDMA_CTRL.SYNCTRIG register. */
 uint32_t syncClr : 8; /**< Clear bits in LDMA_CTRL.SYNCTRIG register*/
 uint32_t reserved3 : 16;
 uint32_t matchVal : 8; /**< Sync trig match value. */
 uint32_t matchEn : 8; /**< Sync trig match enable. */
 uint32_t reserved4 : 16;
 uint32_t linkMode : 1; /**< Select absolute or relative link address.*/
 uint32_t link : 1; /**< Enable LINKLOAD when transfer is done. */
 int32_t linkAddr : 30; /**< Address of next (linked) descriptor. */
 } sync;

 /** WRITE DMA descriptor, used for write immediate operations. */
 struct
 {
 uint32_t structType : 2; /**< Set to 2 to select WRITE descriptor type.*/
 uint32_t reserved0 : 1;
 uint32_t structReq : 1; /**< DMA transfer trigger during LINKLOAD. */
 uint32_t xferCnt : 11; /**< Transfer count minus one. */
```

```
 uint32_t byteSwap : 1; /**< Enable byte swapping transfers. */
 uint32_t blockSize : 4; /**< Number of unit transfers per arb. cycle. */
 uint32_t doneIfs : 1; /**< Generate interrupt when done. */
 uint32_t reqMode : 1; /**< Block or cycle transfer selector. */
 uint32_t decLoopCnt : 1; /**< Enable looped transfers. */
 uint32_t ignoreSrec : 1; /**< Ignore single requests. */
 uint32_t srcInc : 2; /**< Source address increment unit size. */
 uint32_t size : 2; /**< DMA transfer unit size. */
 uint32_t dstInc : 2; /**< Destination address increment unit size. */
 uint32_t srcAddrMode: 1; /**< Source addressing mode. */
 uint32_t dstAddrMode: 1; /**< Destination addressing mode. */
 uint32_t immVal; /**< Data to be written at dstAddr. */
 uint32_t dstAddr; /**< DMA write destination address. */
 uint32_t linkMode : 1; /**< Select absolute or relative link address.*/
 uint32_t link : 1; /**< Enable LINKLOAD when transfer is done. */
 int32_t linkAddr : 30; /**< Address of next (linked) descriptor. */
 } wri;
} LDMA_Descriptor_t;
```

These initializers configure all aspects of a LDMA transfer descriptor and the corresponding emlib function will use these macros to initialize the LDMA_CHx_CTRL, LDMA_CHx_SRC, LDMA_CHx_DST and LDMA_CHx_LINK registers.

The transfer descriptor initializers are provided for the most common single and linked transfer types. Due to the flexibility of the LDMA peripheral, only a small subset of all possible initializers are defined. New initializers can be defined when needed.

Table 4.5. Initializers for LDMA Transfer Descriptor

Initializer	Usage
LDMA_DESCRIPTOR_SINGLE_M2M_BYTE (src, dest, count)	Initializer for single memory to memory byte transfer
LDMA_DESCRIPTOR_SINGLE_M2M_HALF (src, dest, count)	Initializer for single memory to memory half-word transfer
LDMA_DESCRIPTOR_SINGLE_M2M_WORD (src, dest, count)	Initializer for single memory to memory word transfer
LDMA_DESCRIPTOR_LINKABS_M2M_BYTE (src, dest, count)	Initializer for linked (absolute address) memory to memory byte transfer
LDMA_DESCRIPTOR_LINKABS_M2M_HALF (src, dest, count)	Initializer for linked (absolute address) memory to memory half-word transfer
LDMA_DESCRIPTOR_LINKABS_M2M_WORD (src, dest, count)	Initializer for linked (absolute address) memory to memory word transfer
LDMA_DESCRIPTOR_LINKREL_M2M_BYTE (src, dest, count, linkjmp)	Initializer for linked (relative address) memory to memory byte transfer
LDMA_DESCRIPTOR_LINKREL_M2M_HALF (src, dest, count, linkjmp)	Initializer for linked (relative address) memory to memory half-word transfer
LDMA_DESCRIPTOR_LINKREL_M2M_WORD (src, dest, count, linkjmp)	Initializer for linked (relative address) memory to memory word transfer
LDMA_DESCRIPTOR_SINGLE_P2M_BYTE (src, dest, count)	Initializer for single byte transfers from a peripheral to memory
LDMA_DESCRIPTOR_LINKREL_P2M_BYTE (src, dest, count, linkjmp)	Initializer for linked (relative address) byte transfers from a peripheral to memory
LDMA_DESCRIPTOR_SINGLE_M2P_BYTE (src, dest, count)	Initializer for single byte transfers from memory to a peripheral
LDMA_DESCRIPTOR_LINKREL_M2P_BYTE (src, dest, count, linkjmp)	Initializer for linked (relative address) byte transfers from memory to a peripheral
LDMA_DESCRIPTOR_SINGLE_WRITE (value, address)	Initializer for single Immediate WRITE transfer

Initializer	Usage
LDMA_DESCRIPTOR_LINKABS_WRITE (value, address)	Initializer for linked (absolute address) Immediate WRITE transfer
LDMA_DESCRIPTOR_LINKREL_WRITE (value, address, linkjmp)	Initializer for linked (relative address) Immediate WRITE transfer
LDMA_DESCRIPTOR_SINGLE_SYNC (set, clr, matchValue, matchEnable)	Initializer for single SYNC transfer
LDMA_DESCRIPTOR_LINKABS_SYNC (set, clr, matchValue, matchEnable)	Initializer for linked (absolute address) SYNC transfer
LDMA_DESCRIPTOR_LINKREL_SYNC (set, clr, matchValue, matchEnable, linkjmp)	initializer for linked (relative address) SYNC transfer

For linked transfer initializers with absolute address mode (XXXX_LINKABS_XXXX), the link address must be set at runtime, it is `linkAddr` member in the `LDMA_Descriptor_t` union.

Table 4.6. Parameters for LDMA Transfer Descriptor Initializers

Parameter	Usage
src	Source data address (for XFER, memory or peripheral)
dest	Destination data address (for XFER, memory or peripheral)
count	Number of byte/half-word/word to transfer (for XFER)
linkjmp (relative address)	Address of descriptor to link to expressed as a signed number of descriptors from current descriptor. For example: 1 = one descriptor forward in memory 0 = current descriptor -1 = one descriptor back in memory
value	Immediate value to write (for WRI)
address	Write address (for WRI)
set	Synchronization pattern bits to set (for SYNC)
clr	Synchronization pattern bits to clear (for SYNC)
matchValue	Synchronization pattern to match (for SYNC)
matchEnable	Synchronization pattern bits to enable for match (for SYNC)

4.2 LDMA Functions

To initiate the LDMA transfer, the emlib functions in below table are required.

Table 4.7. The emlib Functions for LDMA Transfer

Function	Usage
`LDMA_Init (LDMA_Init_t *init)`	Must have been executed once to use the LDMA controller, normally during system initialization. The LDMA configuration is controlled by the contents of `LDMA_Init_t` structure parameters. It will clear the LDMA_CHEN, LDMA_DBGHALT, and LDMA_REQDIS registers and the LDMA ERROR interrupt is enabled.
`LDMA_StartTransfer (int ch, LDMA_TransferCfg_t *transfer, LDMA_Descriptor_t *descriptor)`	LDMA transfers are initiated by a call to this function, the transfer properties are controlled by the contents of `LDMA_TransferCfg_t` structure and `LDMA_Descriptor_t` union parameters. The `LDMA_Descriptor_t` union parameter may be a pointer to an array of channel descriptors, the descriptors in the array should be linked together as needed.

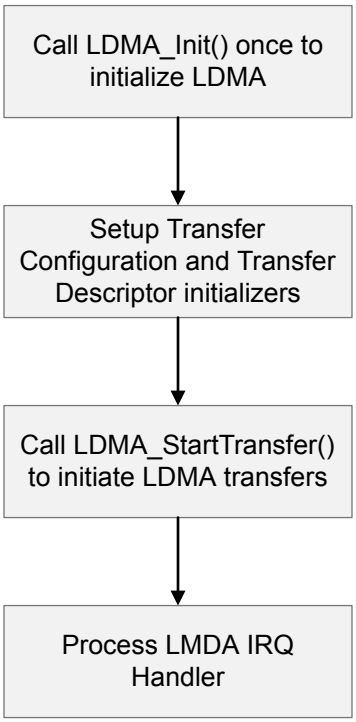


Figure 4.1. LDMA Program Flow

The LDMA emlib module does not implement the LDMA interrupt handler. A template for a limited function LDMA interrupt service routine is included in the `em_ldma.c` and can be activated by defining the symbol `LDMA_IRQ_HANDLER_TEMPLATE` in the IDE. The user can also use this template to tailor make an interrupt handler for the LDMA when needed.

4.3 Examples of LDMA Usage

- A simple memory to memory transfer:

```
/* A single transfer of 4 half words. */
const LDMA_TransferCfg_t memTransfer = LDMA_TRANSFER_CFG_MEMORY();
const LDMA_Descriptor_t xfer = LDMA_DESCRIPTOR_SINGLE_M2M_HALF( src, dest, 4 );

LDMA_Init_t init = LDMA_INIT_DEFAULT;
LDMA_Init( &init );
LDMA_StartTransfer( 0, (void*)&memTransfer, (void*)&xfer );
```

- A list of three memory to memory transfers:

```
/* A transfer of 4 half words which links to another transfer of 4 half words, */
/* which again links to a third transfer of 4 half words. */
const LDMA_TransferCfg_t memTransfer = LDMA_TRANSFER_CFG_MEMORY();
const LDMA_Descriptor_t xfer[] =
{
LDMA_DESCRIPTOR_LINKREL_M2M_HALF( src, dest, 4, 1 ),
LDMA_DESCRIPTOR_LINKREL_M2M_HALF( src + 2, dest + 5, 4, 1 ),
LDMA_DESCRIPTOR_SINGLE_M2M_HALF ( src + 4, dest + 10, 4 )
};

LDMA_Init_t init = LDMA_INIT_DEFAULT;
LDMA_Init( &init );
LDMA_StartTransfer( 0, (void*)&memTransfer, (void*)&xfer );
```

- Peripheral (USART) to memory transfer:

```
/* Transfer 4 characters from USART1. */
const LDMA_TransferCfg_t periTransferRx =
LDMA_TRANSFER_CFG_PERIPHERAL( ldmaPeripheralSignal_USART1_RXDATAV );
const LDMA_Descriptor_t xfer =
LDMA_DESCRIPTOR_SINGLE_P2M_BYTE( &USART1->RXDATA, /* Peripheral address */
                                          dest, /* Destination (SRAM) */
                                           4 ); /* Number of bytes */

LDMA_Init_t init = LDMA_INIT_DEFAULT;
LDMA_Init( &init );
LDMA_StartTransfer( 0, (void*)&periTransferRx, (void*)&xfer );
```

5. PRS on LDMA

Up to two independent DMA requests (PRSREQ0 and PRSREQ1) can be generated by the PRS.

The PRS signals triggering the DMA requests are selected with the SOURCESEL (= 0x1 for PRS) and SIGNAL (= 0x0 for PRSREQ0 or = 0x1 for PRSREQ1) fields in LDMA_CHx_REQSEL register.

The PRS channels for DMA requests are configured in the PRS_DMAREQ0 and PRS_DMAREQ1 registers (see 7.3 Single Descriptor Looped Transfer).

Table 5.1. DMA Request on PRS

Register	Bit Field	DMA Request
LDMA_CHx_REQSEL	SOURCESEL	0x1 (PRS)
	SIGNAL	0x0 (PRSREQ0) or 0x1 (PRSREQ1)
PRS_DMAREQ0	PRSSEL	Selects PRS channel (0 to maximum) for DMA request 0 from the PRS
PRS_DMAREQ1	PRSSEL	Selects PRS channel (0 to maximum) for DMA request 1 from the PRS

The signals from the PRS producers can be used to set and clear the respective bits in the SYNCTRIG field of the LDMA_SYNC register.

The SYNC descriptor allows the LDMA channel to wait for some external stimulus from PRS before continuing on to the next descriptor.

Table 5.2. PRS for SYNCTRIG Set and Clear

Register	Bit Field	SYNCTRIG Set and Clear
LDMA_CTRL	SYNCPRSSETEN	Bit 0 = 1 to enable PRS channel 0 to set SYNCTRIG bit 0 … Bit 7 = 1 to enable PRS channel 7 to set SYNCTRIG bit 7
LDMA_CTRL	SYNCPRSCLREN	Bit 0 = 1 to enable PRS channel 0 to clear SYNCTRIG bit 0 … Bit 7 = 1 to enable PRS channel 7 to clear SYNCTRIG bit 7

6. The DMADRV

The EMDRV (EnergyAware Driver) is a set of function specific high-performance drivers for EFM32 Gecko Series 0 and 1, EZR32 Series 0, and EFR32 Wireless Gecko Series 1 devices' on-chip peripherals.

The DMADRV, one of the EMDRV modules, makes it possible to write code using DMA which will work regardless of the type of DMA controller on the underlying Microcontroller or Wireless SoC. It will also make it possible to use DMA in several modules without the modules knowing about each other.

The below DMA transfer modes are currently supported by DMADRV.

- DMA basic transfer from memory to a peripheral.
- DMA basic transfer from a peripheral to memory.
- DMA ping-pong transfer from memory to a peripheral.
- DMA ping-pong transfer from a peripheral to memory.

Refer to the [**Software Documentation**] tile in Simplicity Studio for details on how to configure the DMADRV.

7. Software Examples

This application note includes software examples demonstrating how to use the LDMA in different transfer modes. The examples in sections 7.1 Single Direct Register DMA Transfer to 7.7 Ping-Pong are based on examples in the EFM32 Pearl Gecko Family Reference Manual sections 7.4.1 to 7.4.7.

All examples are run on the EFM32 Pearl Gecko Starter Kit (SLSTK3401A). The board controller on the starter kit provides a virtual COM port (CDC) interface when connected to a computer. The on board EFM32PG1B200F256GM48 can connect to this serial port interface and communicate directly (baudrate 115200-8-N-1) with the host computer terminal program (e.g., Tera Term).

The examples in sections 7.1 Single Direct Register DMA Transfer to 7.7 Ping-Pong are grouped into one project (`ldma_example_pg` or `SLSTK3401A_ldma_example`) and each example is selected by the main menu in the host computer terminal program as below.

```
LDMA Examples
Press 1 for Single Direct Register DMA Transfer Example
Press 2 for Descriptor Linked List Example
Press 3 for Single Descriptor Looped Transfer Example
Press 4 for Descriptor List with Looping Example
Press 5 for Simple Inter-Channel Synchronization Example
Press 6 for 2D Copy Example
Press 7 for Ping-Pong Example
Press ? to print this menu
```

This project does not use the LDMA interrupt handler template in `em_ldma.c`. The LDMA interrupt service routine in `main_ldma_example.c` is used for error checking, clear the interrupt flag and no callback function is invoked inside the handler.

More details about examples 7.1 Single Direct Register DMA Transfer to 7.7 Ping-Pong can be found in the EFM32 Pearl Gecko Family Reference Manual section 7.4.

The example in section 7.8 DMADRV (project `dmadrv_example_pg/dmadrv_example_gg` or `SLSTK3401A_dmadrv_example/STK3700_dmadrv_example`) demonstrates how to use DMADRV to write DMA controller independent software.

This software package can be found on the Silicon Labs website (www.silabs.com/32bit-appnotes) or within Simplicity Studio using [**Application Notes**].

7.1 Single Direct Register DMA Transfer

This example illustrates how to use the LDMA to transfer 127 contiguous half words (254 bytes) between two memory locations in RAM. The software does not use a memory-based descriptor list and writes directly to the LDMA channel descriptor registers.

Instead of using LDMA_SWREQ register to start the LDMA operation, the transfer is triggered by the Peripheral Reflex System (PRS through DMAREQ0) from rising edge of pushbutton 1 (BTN1) on starter kit. This setup is achieved by calling the `gpioPrsSetup()` function in `main_ldma_example.c`.

Press 1 in the main menu to display the following output on the host computer.

```
Single Direct Register DMA Transfer Example
Source buffer
 0: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 16: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 32: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
 64: 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
 80: 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96: 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
 112: 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126

Destination buffer before LDMA transfer
 0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 16: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 32: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 48: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 64: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 80: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 96: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 112: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Press and release pushbutton BTN1 to start memory transfer
Destination buffer after LDMA transfer
 0: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
 16: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 32: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
 64: 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
 80: 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96: 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
 112: 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
```

7.2 Descriptor Linked List

This example demonstrates how to use the LDMA to transfer linked-list data from RAM to a peripheral. The firmware uses a linked list of descriptors to transfer four strings from RAM to the USART.

Press 2 in the main menu to display the following output on the host computer.

```
Descriptor Linked List Example
String_1111
String_2222
String_3333
String_4444
```

7.3 Single Descriptor Looped Transfer

This example demonstrates how to use the LDMA to transfer data from a peripheral (ADC) to RAM. The LDMA channel is setup to use full request signal (REQ instead of SREQ), which makes the LDMA transfer the whole ADC FIFO (four 32-bit words) each time the ADC sets its DMA request (FIFO is full). This example also uses a CRYOTIMER to trigger the ADC through the Peripheral Reflex System (PRS) at 1024 Hz.

The absolute addressing is used in the first descriptor source and destination addresses to initialize the transfer. Since the destination address is incremented after each transfer, the final address will point to one unit past the last transfer. Thus the relative addressing is used in the second descriptor (for single loop transfer) destination address and an offset of zero will give the next sequential data address.

Press 3 in the main menu to display the following output on the host computer.

```
Single Descriptor Looped Transfer Example (Press pushbutton BTN0 to ground the ADC input)
ADC[0]: 3.2709
ADC[1]: 3.2726
ADC[2]: 3.2734
ADC[3]: 3.2734
ADC[4]: 3.2734
ADC[5]: 3.2742
ADC[6]: 3.2734
ADC[7]: 3.2734
ADC[8]: 3.2742
ADC[9]: 3.2750
ADC[10]: 3.2742
ADC[11]: 3.2742
ADC[12]: 3.2750
ADC[13]: 3.2742
ADC[14]: 3.2742
ADC[15]: 3.2734
```

7.4 Descriptor List with Looping

This example demonstrates how to use the LDMA to transfer multiple descriptors with looping. The software uses a linked list of descriptors to transfer two strings from RAM to USART three times, then jump to a different linked descriptor to transfer one string after looping.

Press 4 in the main menu and display the following output on the host computer.

```
Descriptor List with Looping Example
String_1111
String_2222
String_1111
String_2222
String_1111
String_2222
String_3333
```

7.5 Simple Inter-Channel Synchronization

This example demonstrates how to use the LDMA synchronization descriptors to pause and restart a DMA sequence.

The descriptors for inter-channel synchronization are listed as below.

- Descriptor A is a XFER structure type to transfer a string (“Press any key to toggle LED0”) from RAM to USART.
- Descriptor B is a SYNC structure type to pause channel 0 and wait on SYNCTRIG[7] to be set.
- Descriptor Y is a XFER structure type to wait for a key press from USART.
- Descriptor Z is a SYNC structure type to set SYNCTRIG[7] to restart channel 0.
- Descriptor C is a WRI structure type to toggle LED0 on starter kit when receiving trigger from descriptor Z.

The SYNCTRIG bits in the LDMA_SYNC register can be set and cleared by SYNC descriptor, PRS signal (SYNCPRSCLREN and SYNCPRSSETEN fields in the LDMA_CTRL register), or software.

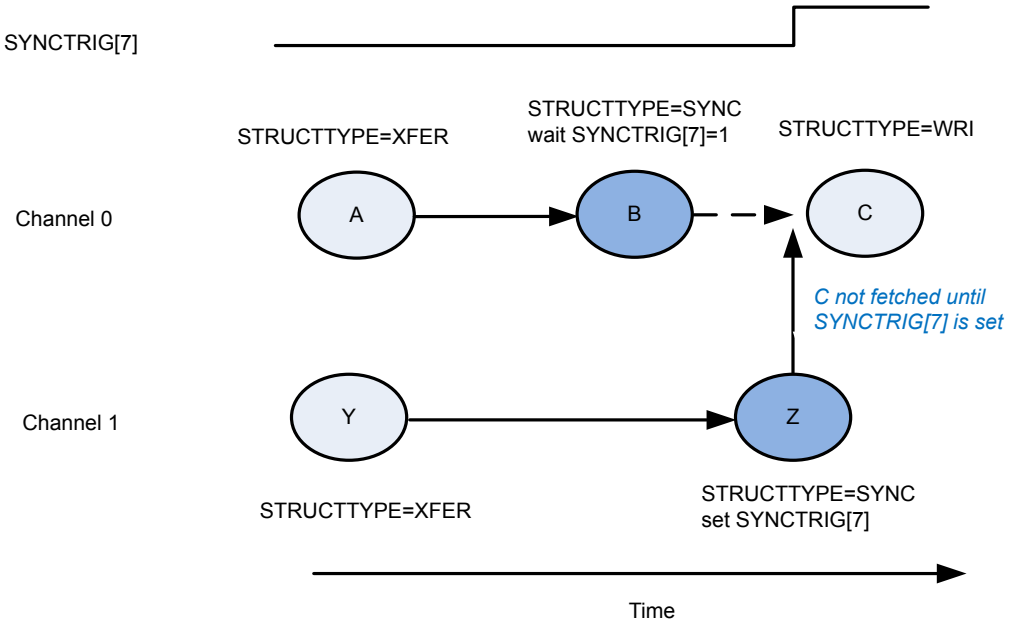


Figure 7.1. Simple Inter-Channel Synchronization Example

Press 5 in the main menu to output the following on the host computer.

```
Simple Inter-Channel Synchronization Example
Press any key to toggle LED0
Trigger received
```

7.6 2D Copy

This example demonstrates how to use the LDMA descriptor list with looping to perform 2D copy. The first descriptor will use absolute addressing mode and the source and destination addresses should point to the desired target addresses. The second descriptor for looping will use relative addressing and the source and destination addresses are set to the desired offset.

When using relative addressing with the source or destination address registers, the LDMA adds the relative offset to the current contents of the respective address register. Since the source and destination addresses are normally incremented after each transfer, the final address will point to one unit past the last transfer. Thus, an offset of zero will give the next sequential data address.

The parameters for 2D copy are listed as below.

- Source buffer size: 64 x 64 bytes
- Destination buffer size: 64 x 64 bytes
- Transfer width: 16 bytes
- Transfer height: 16 bytes
- Source start address: The [16][16] element of source buffer
- Destination start address: The [48][32] element of destination buffer

Press 6 in the main menu to output the following on the host computer.

```
2D Copy Example
2D Rectangle Source buffer
[16][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[17][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[18][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[19][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[20][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[21][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[22][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[23][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[24][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[25][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[26][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[27][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[28][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[29][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[30][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[31][16]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

2D Rectangle Destination buffer before LDMA transfer
[48][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[49][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[50][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[51][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[52][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[53][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[54][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[55][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[56][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[57][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[58][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[59][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[60][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[61][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[62][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[63][32]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

2D Rectangle Destination buffer after LDMA transfer
[48][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[49][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[50][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[51][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[52][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[53][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[54][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[55][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[56][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[57][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[58][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[59][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[60][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

```
[61][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[62][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
[63][32]: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

7.7 Ping-Pong

This example demonstrates how to use the LDMA to transmit ping-pong buffers. This requires two descriptors with LINKADDR field in the LINK word points to the other descriptor.

The LDMA will transmit the first or second buffer data while software is filling the second or first buffer. The DONEIFSEN bit in each descriptor should be set to generate an interrupt on the completion of each descriptor for software to fill the buffer.

This example is not an infinite ping-pong transmitter. The operation stops after three ping-pong transfers by setting LINK field in the LINK word of the two descriptors to zero.

Press 7 in the main menu to output the following on the host computer.

```
Ping-Pong Example
1111111
2222222
3333333
4444444
5555555
6666666
```

7.8 DMADRV

In the DMADRV example (`main_dmadrv_example.c`), the DMA transfer is triggered by the real time counter every 3000 ms (using RTCDRV). The ping-pong example in section 7.7 Ping-Pong is replicated by using the DMADRV API. The DMADRV has its own LDMA interrupt handler and the callback function is called when the DMA transfer is completed.

The Real Time Counter peripheral (RTC or RTCC) and Direct Memory Access controller (µDMA or LDMA) actually employed by the RTCDRV and DMADRV are transparent to the user. Thus the same source code can run on EFM32 Gecko Series 0 and 1, EZR32 Series 0, and EFR32 Wireless Gecko Series 1 devices

The below output will display on the host computer when the program is running.

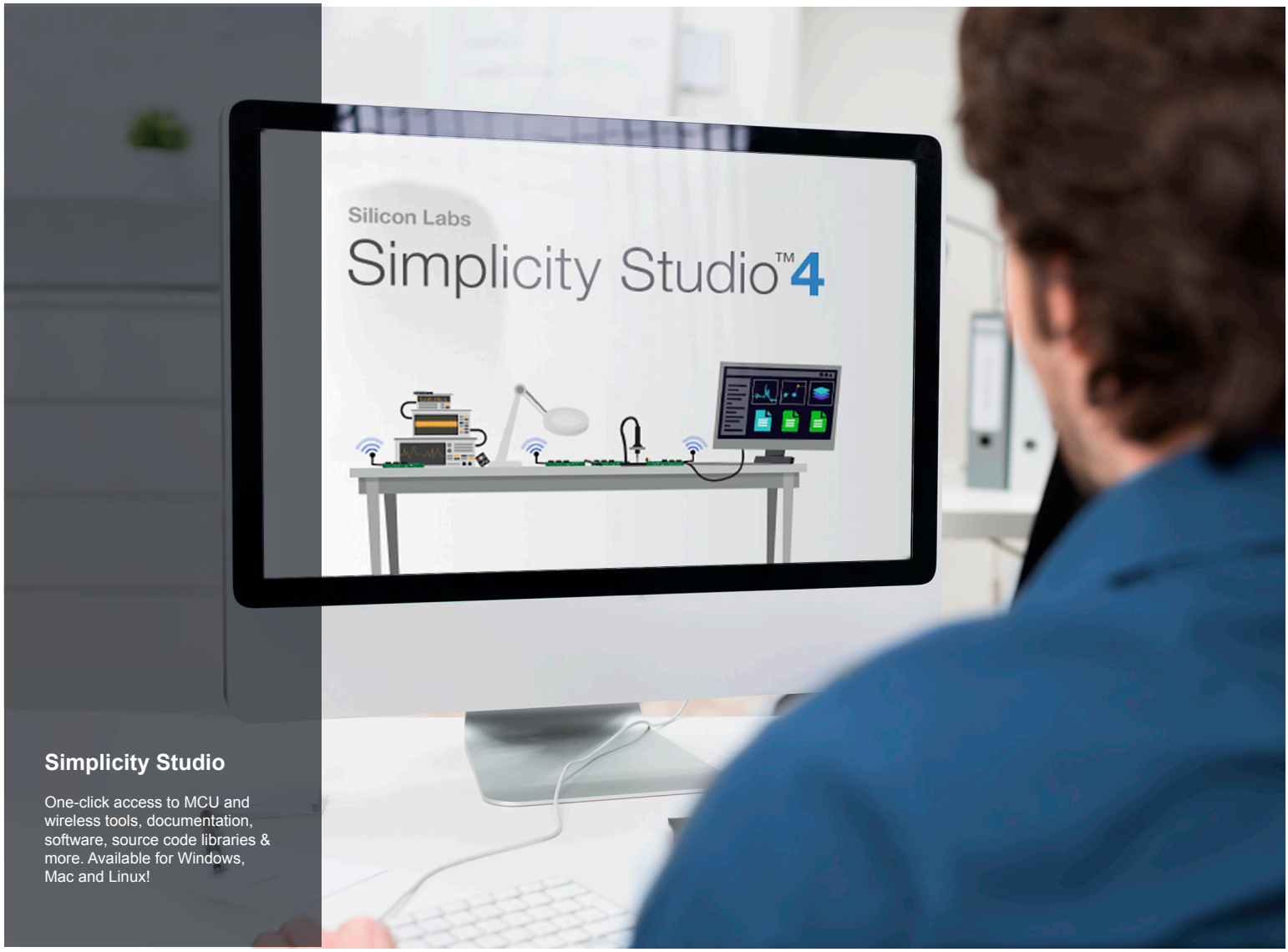
```
1111111
2222222
3333333
4444444
5555555
6666666
1111111
2222222
3333333
4444444
5555555
6666666
```

8. Revision History

Revision 0.1

August 18, 2016

Initial release.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS®

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

http://www.silabs.com