LATTICE SEMICONDUCTOR

# LatticeMico DMA Controller

The LatticeMico DMA controller is a direct memory access controller that provides a master read port, a master write port, and one slave port to control data transmission.

## Version

This document describes the 3.3 version of the LatticeMico DMA controller.

## Features

The LatticeMico DMA controller includes the following features:

- WISHBONE B.3 interface
  - WISHBONE Slave Interface to configure the DMA
  - Two WISHBONE Master Interfaces: one to perform reads and the second to perform writes
- WISHBONE data bus width of the WISHBONE slave and master interfaces can be individually configured to be 8 or 32 bits wide.
- An interrupt port (INT_O) to indicate completion of a DMA transfer (successful or not)
- Constant or incrementing address for the source and destination in a DMA transfer. Addresses can be configured to increment by 1, 2, or 4 bytes.
- Classic or Burst WISHBONE cycles. Number of transfers in a burst can be configured to be 4, 8, 16, 32, or 64.
- WISHBONE ERR support (DMA transfer is terminated) and ability to indicate an unsuccessful completion via status register

Copyright © June 2012 Lattice Semiconductor Corporation.

- WISHBONE RTY support
  - DMA controller will retry current WISHBONE transfer after a timeout.
  - Timeout can be configured to any value between 1 and 255 WISHBONE clock cycles.

For additional details about the WISHBONE bus, refer to the *LatticeMico8 Processor Reference Manual* or the *LatticeMico32 Processor Reference Manual*.

# Functional Description

The LatticeMico DMA controller provides a high performance memory-to-memory data transfer engine for use in LatticeMico system platforms. The DMA controller is simultaneously a WISHBONE master and a WISHBONE slave. The slave side of the DMA controller is the controlled by the LatticeMico8 microcontroller or the LatticeMico32 microprocessor. It's primary role is to set up the parameters of a new transfer, monitor existing transfers, and abort them if necessary. The master side of the DMA controller performs the memory-to-memory transfers. It consists of independent read and write I/O ports that can access memories via WISHBONE.

DMA transfers are initialized and initiated by the LatticeMico8 microcontroller or the LatticeMico32 microprocessor. The processor sets the following parameters of the transfer: source address, destination address, number of bytes to be transferred, and number of bytes per WISHBONE transaction. Once these parameters have been set, the processor then requests the DMA engine to begin the transfer. The master side of the DMA engine starts the transfer as soon as it has access to the WISHBONE bus. Figure 1 and Figure 2 show the logical flow of data from source address to destination address for different arbitration schemes. DMA transfers proceed unassisted until a pre-programmed number of bytes have been transferred. There are two special events that require special handling by the DMA engine:

- WISHBONE ERR: It is possible that the source or destination memory may terminate a WISHBONE read/write initiated by the DMA engine with an error signal. The DMA engine monitors for an error and terminates the entire transfer immediately. The error event is registered within the Status Register.
- WISHBONE RTY: It is possible that the source or destination memory may terminate a WISHBONE read/write initiated by the DMA engine with a retry signal indicating that the current WISHBONE transfer needs to be restarted. The DMA engine monitors for a retry and terminates the current WISHBONE transaction immediately, waits for a pre-programmed delay, and restarts the terminated WISHBONE transaction.

**Figure 1: DMA Usage with Shared-Bus Arbitration in LatticeMico System**

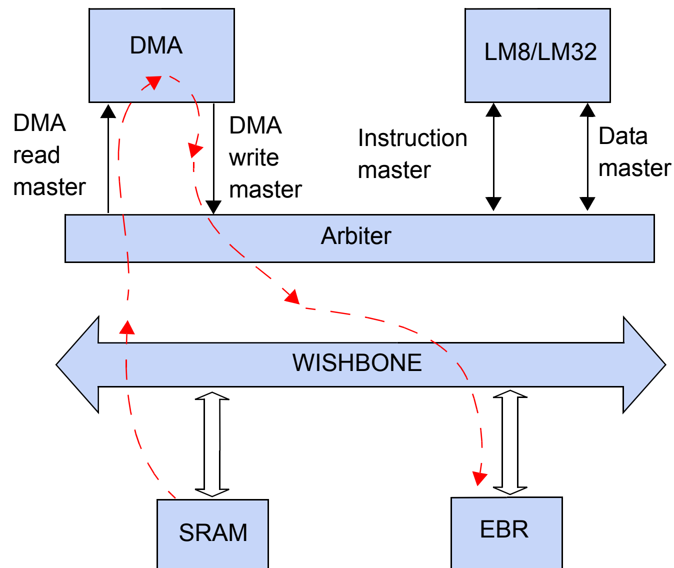
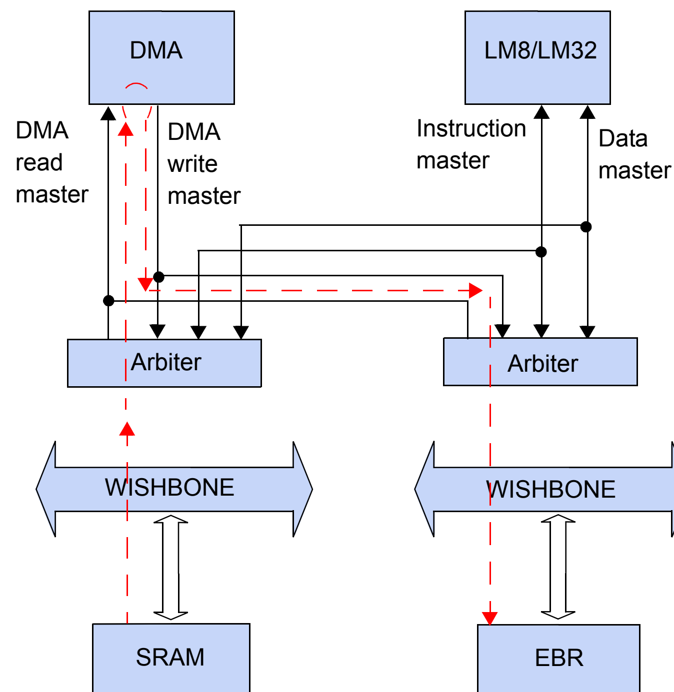



**Figure 2: DMA Usage with Slave-Side Arbitration in LatticeMico System**



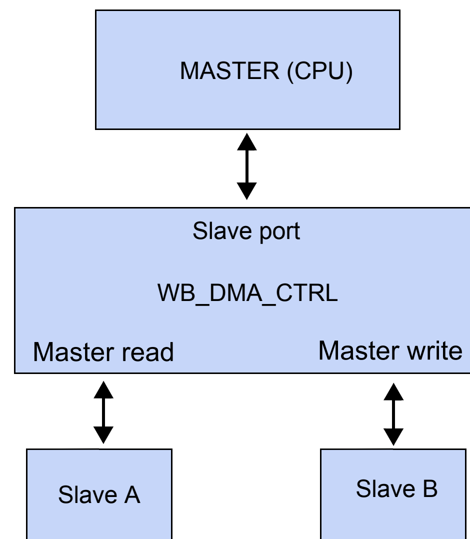

The microprocessor can use one of the following two methods for determining when a DMA transfer has finished:

- The first is to periodically poll the controller's status register and monitor the DMA_BUSY bit. As long as the DMA_BUSY bit is 1, the DMA controller is in the process of transferring data.
- The second method is to configure the DMA controller to assert an interrupt to the microprocessor. When the DMA controller completes the preprogrammed number of memory transfers or ERR_I WISHBONE ERR is asserted, the controller asserts the interrupt signal.

Regardless of the method used to detect the termination of the DMA cycle, the microprocessor must perform the necessary clean-up to make the DMA controller ready to accept the next transfer request.

The function of the DMA controller is illustrated in Figure 3.

**Figure 3: DMA Controller Usage**




# DMA Transfer Flow

A typical DMA transfer includes the following steps:

1. The CPU first reads the DMA controller internal registers to ensure that the DMA controller is idle.
2. The CPU sets up the DMA parameters by writing to the internal registers through the WISHBONE slave port.
3. The CPU enables the DMA controller. The DMA controller starts the transfer. The controller reads data through the read master port and writes the data out through the write master port.
4. The number of transfers that the DMA performs depends on the transfer mode. For a fixed-length transfer, the number is determined by the length configuration set by the CPU.
5. After the transfer is complete, the DMA controller generates an interrupt request to the CPU, when the interrupt enable bit in the status register is asserted. The CPU must read the status register to clear this interrupt

request; otherwise, this interrupt request is maintained until the next interrupt request is generated. If interrupt generation is masked, it is up to the firmware to poll the DMA_BUSY bit in the status register to detect the end of a DMA transfer.

### Supported Features:

1. Constant or incrementing address for the source and destination in a DMA transfer. Addresses can be configured to increment by 1, 2, or 4 bytes.
2. Classic or Burst WISHBONE cycles. Number of transfers in a burst can be configured to be 4, 8, 16, 32, or 64.

### Limitations:

1. WISHBONE BTE is fixed at 00.
2. WISHBONE LOCK is not supported.
3. WISHBONE data bus width can be configured to 8 or 32 bits only. The 32-bit data bus can accommodate 1, 2, or 4 byte transfers. The 8-bit data bus can only accommodate a 1 byte transfer.
4. Transfer addresses must be aligned to the transfer size boundary. A 4 byte transfer means that the least-significant 2 bits of the address must be zero.

## Configuration

The following sections describe the graphical user interface (UI) parameters and the I/O ports that you can use to configure and operate the LatticeMico DMA controller.

# UI Parameters

Table 1 shows the UI parameters available for configuring the LatticeMico DMA controller through the Mico System Builder (MSB) interface.

**Table 1: DMA Controller UI Parameters**

| Dialog Box Option | Description | Allowable Values | Default Value |
|---|---|---|---|
| Instance Name | Specifies the name of the DMA controller instance. | Alphanumeric and underscores | dma |
| Base Address | Specifies the base address for accessing the internal registers. The minimum boundary alignment is 0X80. | 0X80000000–0XFFFFFFFF<br>If other components are included in the platform, the range of allowable values will vary. | 0X80000000 |
| FIFO Implementation | Determines whether the FIFO is implemented as an EBR or a LUT. | EBR or LUT | EBR |
| **Settings** | | | |
| Retry Timeout | Specifies the number of WISHBONE clock cycles that the DMA controller must wait after the source or destination asserts the WISHBONE RTY before retrying the same WISHBONE cycle. | 1 – 255 | 16 |
| **WISHBONE Configuration** | | | |
| Control Port Data Bus Width | Configures the control port's WISHBONE data bus to be 8 or 32 bits wide. | 8, 32 | 32 |
| Read/Write Port Data Bus Width | Configures the read and write WISHBONE master port data buses to be 8 or 32 bits wide. | 8, 32 | 32 |

# I/O Ports

Table 2 describes the input and output ports of the LatticeMico DMA controller.

**Table 2: DMA Controller I/O Ports**

| I/O Port | Active | Direction | Initial State | Description |
|---|---|---|---|---|
| **Master Read Port** | | | | |
| MA_ADR_O | High | O | 0 | Master A address bus |
| MA_WE_O | High | O | 0 | Master A write enable signal |
| MA_SEL_O | High | O | 0 | Master A select signal |
| MA_STB_O | High | O | 0 | Master A strobe signal |

**Table 2: DMA Controller I/O Ports (Continued)**

| I/O Port | Active | Direction | Initial State | Description |
|---|---|---|---|---|
| MA_CYC_O | High | O | 0 | Master A cycle signal |
| MA_LOCK_O | High | O | 0 | Master A lock signal |
| MA_CTI_O | High | O | 0 | Master A CTI signal |
| MA_BTE_O | High | O | 0 | Master A BTE signal |
| MA_DAT_I | High | I | X | Master A input data bus |
| MA_DAT_O | High | O | 0 | Master A output data bus |
| MA_ACK_I | High | I | X | Acknowledged signal from the slave device |
| MA_ERR_I | High | I | X | Error signal from the slave device, indicating abnormal termination |
| MA_RTY_I | High | I | X | Retry signal from the slave device, requesting the write master to generate the write cycles again |
| **Master Write Port** | | | | |
| MB_ADR_O | High | O | 0 | Master B address bus |
| MB_DAT_O | High | O | 0 | Master B data bus |
| MB_WE_O | High | O | 0 | Master B write enable signal |
| MB_SEL_O | High | O | 0 | Master B select signal |
| MB_STB_O | High | O | 0 | Master B strobe signal |
| MB_CYC_O | High | O | 0 | Master B cycle signal |
| MB_LOCK_O | High | O | 0 | Master B lock signal |
| MB_CTI_O | High | O | 0 | Master B CTI signal |
| MB_BTE_O | High | O | 0 | Master B BTE signal |
| MB_DAT_I | High | I | X | Master B input data bus |
| MB_ACK_I | High | I | X | Acknowledged signal from the slave device |
| MB_ERR_I | High | I | X | Error signal from the slave device, indicating abnormal termination |
| MB_RTY_I | High | I | X | Retry signal from the slave device, requesting the write master to generate the read cycles again |
| **Slave Port** | | | | |
| S_ADR_I | High | I | X | Slave address bus |
| S_DAT_I | High | I | X | Slave data input bus |
| S_WE_I | High | I | X | Slave write enable signal |
| S_STB_I | High | I | X | Slave strobe signal |
| S_CYC_I | High | I | X | Slave cycle signal |

**Table 2: DMA Controller I/O Ports (Continued)**

| I/O Port | Active | Direction | Initial State | Description |
|---|---|---|---|---|
| S_SEL_I | High | I | X | Slave select signal |
| S_LOCK_I | High | I | X | Slave lock signal |
| S_CTI_I | High | I | X | Slave CTI signal |
| S_BTE_I | High | I | X | Slave BTE signal |
| S_DAT_O | High | O | 0 | Output data bus |
| S_ACK_O | High | O | 0 | Acknowledge to master device |
| S_ERR_O | High | O | 0 | Slave error signal |
| S_RTY_O | High | O | 0 | Slave retry signal |
| S_INT_O | High | O | 0 | Interrupt to master (CPU) |
| **Clock and Reset Signal** | | | | |
| CLK_I | High | I | X | Input clock signal |
| RST_I | High | I | X | Reset signal (active high) |

# Register Definitions

The LatticeMico DMA controller includes the registers shown in Table 3.

**Table 3: Register Map**

| Register Name | Offset | 31-24 | 23-16 | 15-8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SA | 0x00 | LSB | | | MSB | | | | | | | |
| DA | 0x04 | LSB | | | MSB | | | | | | | |
| LR | 0x08 | LSB | | | MSB | | | | | | | |
| CR | 0x0C | Reserved | | | BurstMode | Burst Size | | | Increment | | D_CON | S_CON |
| SR | 0x10 | Reserved | | | | | | | Start | Status | IE | DMA_BUSY |

Table 4 through Table 8 provide details about each register of the LatticeMico DMA controller.

**Table 4: SA Register**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| Source address | 31: 0 | Read/write | Source address for the DMA transfer. Most-significant byte of address is in bits 7:0 and least-significant byte of address is in bits 31:24. |

**Table 5: DA Register**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| Destination address | 31: 0 | Read/write | Destination address for the DMA transfer. Most-significant byte of address is in bits 7:0 and least-significant byte of address is in bits 31:24. |

**Table 6: Length Register Bit Definition**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| Length of Transfer | 31:0 | Read/write | Length of the DMA transfer in bytes. Most-significant byte of address is in bits 7:0 and least-significant byte of address is in bits 31:24. |

**Table 7: Control Register Bit Definition**

| Bit Name | Bit | Default | Access Mode | Description |
|---|---|---|---|---|
| S_CON | 0 | 0 | Read/write | The source address register is held constant when S_CON is 1. Otherwise, it increments according to the values in the INC register. |
| D_CON | 1 | 0 | Read/write | The destination address register is held constant when D_CON is 1. Otherwise, it increments according to the values in the INC register. |
| INC | 3-2 | 0 | Read/write | Increment the SA/DA/LR registers:<br>00 = increment SA/DA by 1, decrement LR by 1<br>01 = increment SA/DA by 2, decrement LR by 2<br>10 = increment SA/DA by 4, decrement LR by 4<br>11 = increment SA/DA by 4, decrement LR by 4 |
| Burst Size | 6-4 | 000 | Read/write | 000 = burst length = INC * 4<br>001 = burst length = INC * 8<br>010= burst length = INC * 16<br>011 = burst length = INC * 32<br>100 = burst length = INC * 64 |
| Burst Mode Enable | 7 | 0 | Read/write | 1 = Enable burst mode<br>0 = Disable burst mode |
| Reserved | 31-8 | 0 | Read/write | Reserved |

**Table 8: Status Register Bit Definition**

| Bit Name | Bit | Access Mode | Description |
|---|---|---|---|
| DMA_BUSY | 0 | Read | A 1 in this bit indicates that the DMA controller is busy. |
| IE (interrupt enable) | 1 | Read/write | When set by the CPU, this bit causes the DMA controller to generate an interrupt on completion of the transfer. |
| Status bit | 2 | Read | This bit is read-only by the CPU. It indicates whether the last transfer finished successfully. A 0 indicates that the last transfer was successful. A 1 indicates that the last transfer ended in a bus error. |
| Start bit | 3 | Write | Writing a 1 in this bit position causes a transfer to start. This bit should be written only when the DMA controller is not busy. This bit is always 0 when read by the CPU. |
| Reserved | 31:4 | Read/write | Reserved |

**Note**

Reset value is 0 for all.

# Timing Diagrams

The timing diagrams featured in Figure 4 through Figure 10 show the timing of the DMA controller's WISHBONE and external signals.

Figure 4 shows how the DMA controller's slave port updates the data in the internal register.

**Figure 4: DMA Controller Write Internal Register**

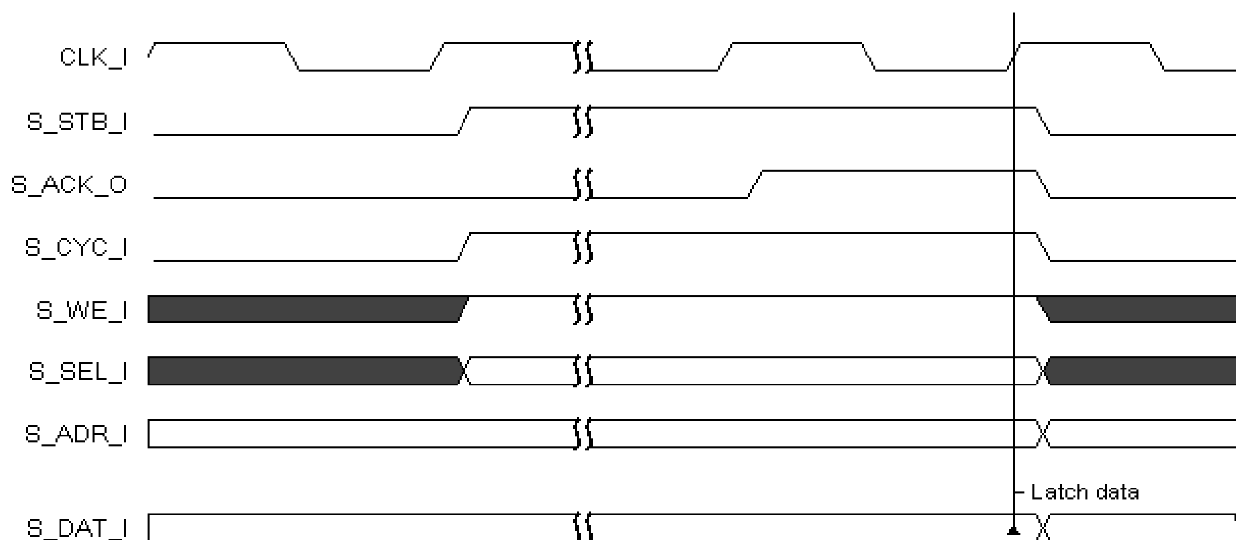


Figure 5 shows how the DMA controller's slave port reads the data in the internal register.

**Figure 5: DMA Controller Read Internal Register**

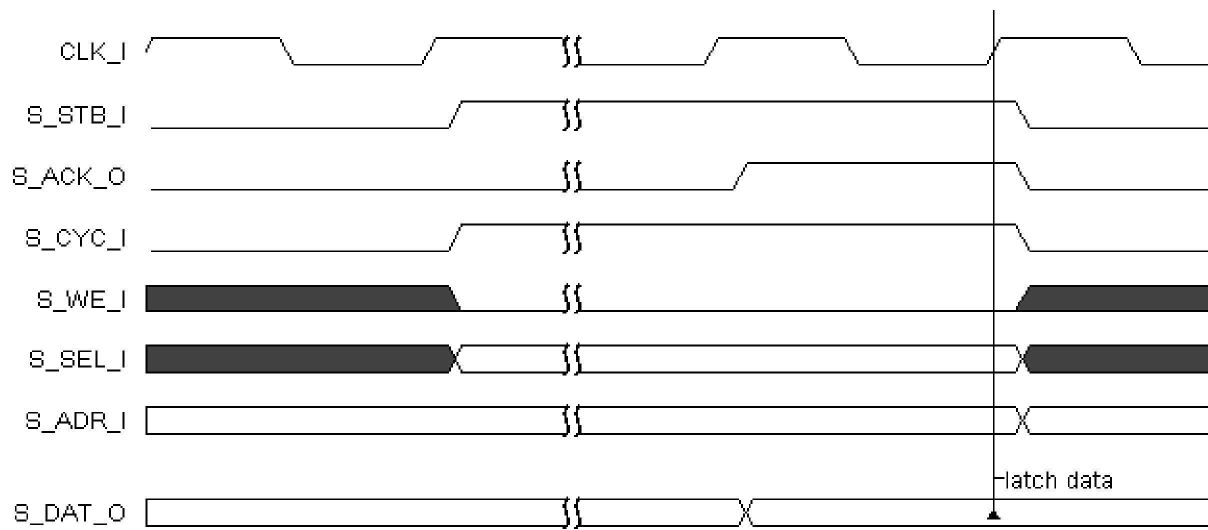


Figure 6 shows how the DMA controller's master ports read the data from the source address.

**Figure 6: DMA Controller Reading Data from Source Address**

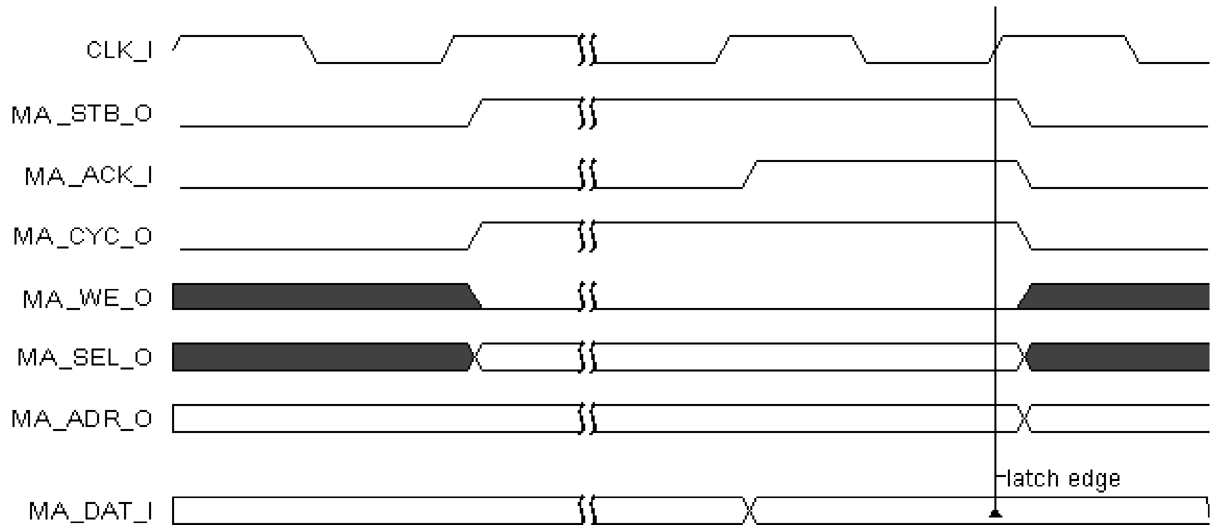



Figure 7 shows how the DMA controller's master ports write the data to the destination address.

**Figure 7: DMA Controller Writing Data to Destination Address**

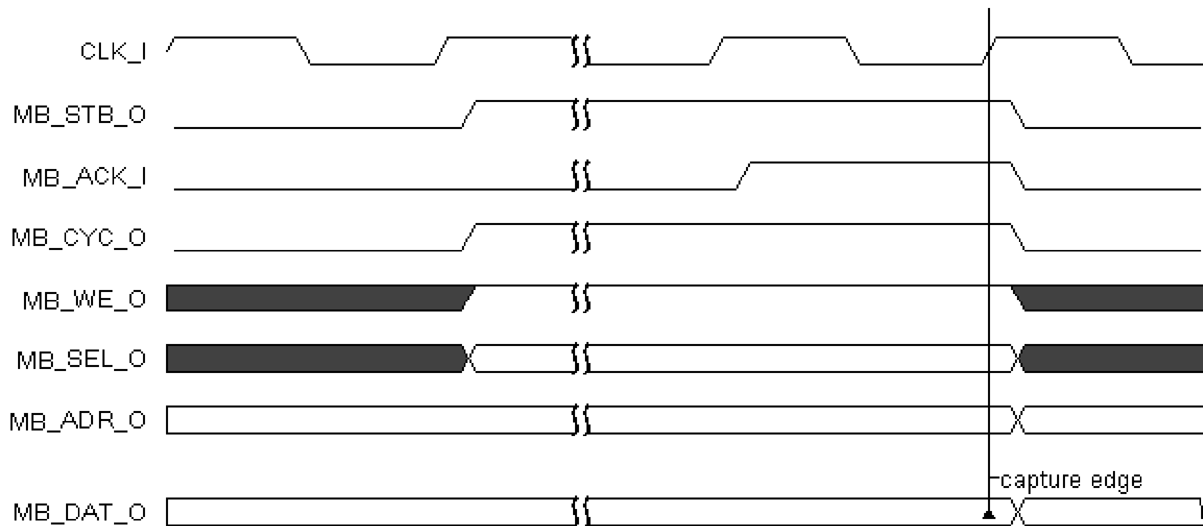


Figure 8 shows how the DMA controller's master ports copy data from one slave device to another.

**Figure 8: DMA Controller Reading Data from Slave A Device and Writing Data to Slave B Device**

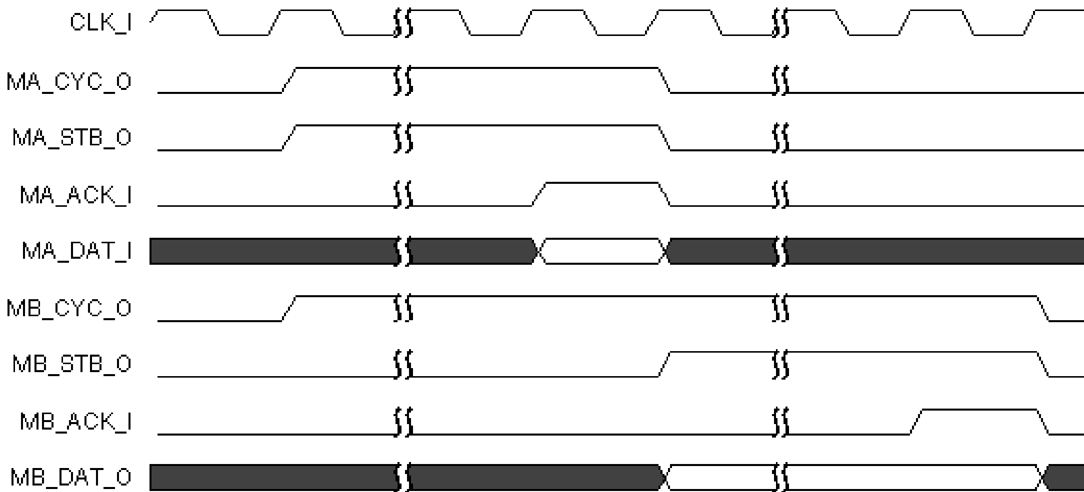



Figure 9 shows the port names and timing diagram of the DMA memory controller in single data transfer mode.

**Figure 9: DMA Controller Timing Diagram for Single Data Transfer**

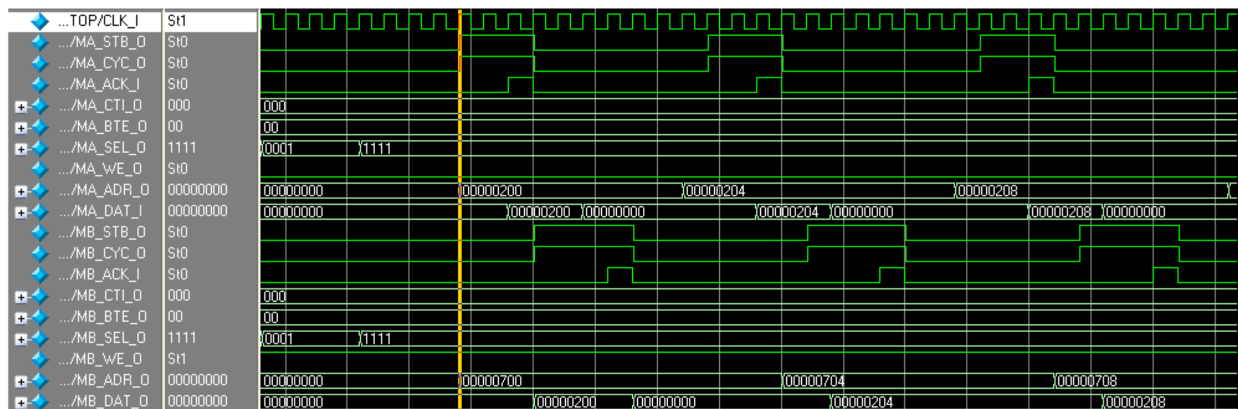
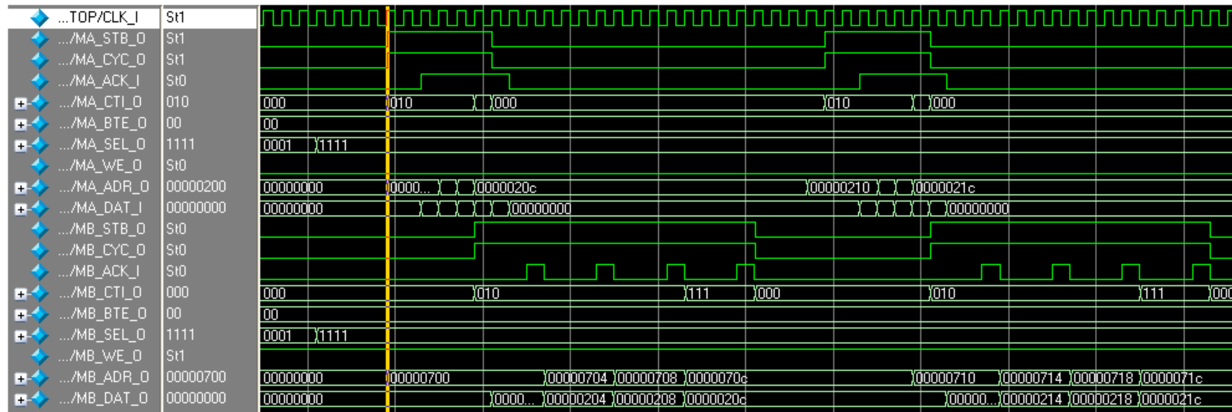


Figure 10 shows the port names and timing diagram of the DMA memory controller in burst data transfer mode.

**Figure 10: DMA Controller Timing Diagram for Burst Data Transfer**




# EBR Resource Utilization

The number of EBRs in the LatticeMico DMA depends on the FIFO implementation.

- If the FIFO is implemented as an EBR, the DMA has one EBR.
- If the FIFO is implemented as a LUT, the DMA has no EBRs.

# Usage Model

This section describes the software usage model for the LatticeMico DMA controller. The DMA device drivers provide a simple and easy-to-use interface to interact with the physical DMA controller.

The device driver presents the DMA as a WISHBONE slave peripheral which can be configured via the memory map shown in Table 3 on page 8. The DMA can be configured through LatticeMico System Builder (MSB) to operate in an interrupt-driven mode or in a polled mode. In the polled mode, the hardware does not raise an interrupt to indicate whether a transfer has successfully completed or failed. Therefore, it is the responsibility of the application layer to keep track of the transfer. In the interrupt-driven mode, the hardware raises an interrupt when the transfer has either completed or failed. Therefore, the application layer does not need to continuously monitor the DMA controller. The interrupt-driven mode is more suitable to an asynchronous mode of operation.

# LatticeMico32 Microprocessor Software Support

This section describes the LatticeMico32 microprocessor software support provided for the LatticeMico DMA controller. It first describes the basic DMA device-driver interface and then describes the DMA lookup service.

**Note**

The supporting routines are meant for use in a single-threaded environment. If you use them in a multi-tasking environment, you must provide re-entrance protections.

## Register Map Structure

The structure in Figure 11 depicts the register map layout for the DMA controller. The elements are self-explanatory and are based on the register map shown in Table 3 on page 8. This structure, which is defined in the MicoDMA.h header file, enables you to directly access the DMA registers, if desired. It is also used internally by the device driver for manipulating the DMA registers.

**Figure 11: DMA Controller Write Internal Register**

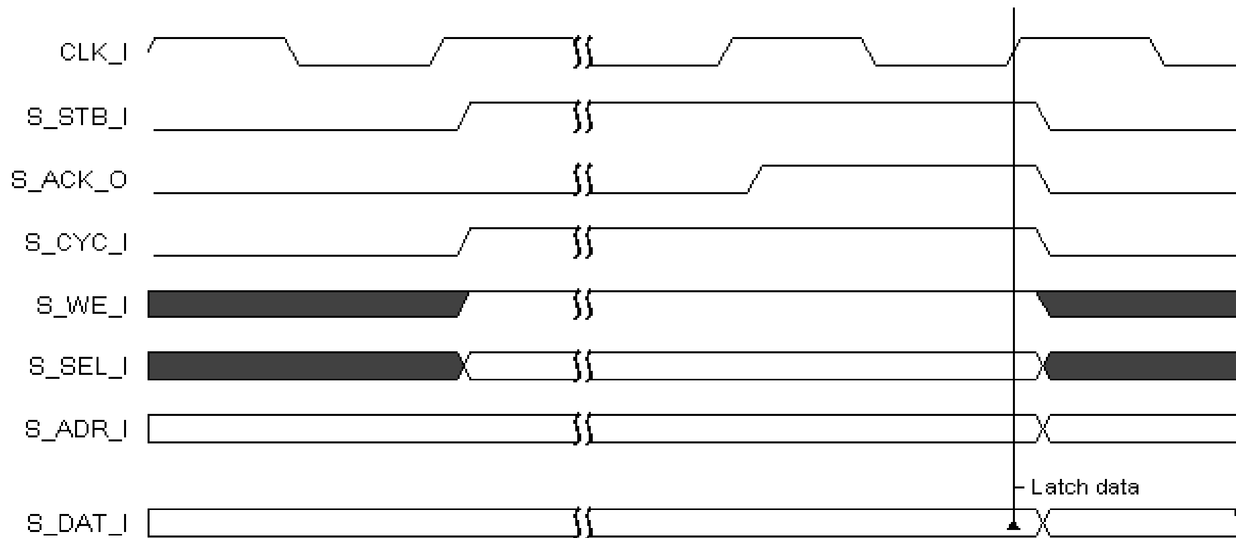


**Figure 12: DMA Controller Register Map Structure**

```
typedef struct st_MicoDMA{
   /* address to read data from */
   volatile unsigned int sAddr;

   /* address to write data to */
   volatile unsigned int dAddr;

   /* dma length */
   volatile unsigned int len;

   /* control register */
   volatile unsigned int control;

   /* status register */
   volatile unsigned char status;
}MicoDMA_t;
```

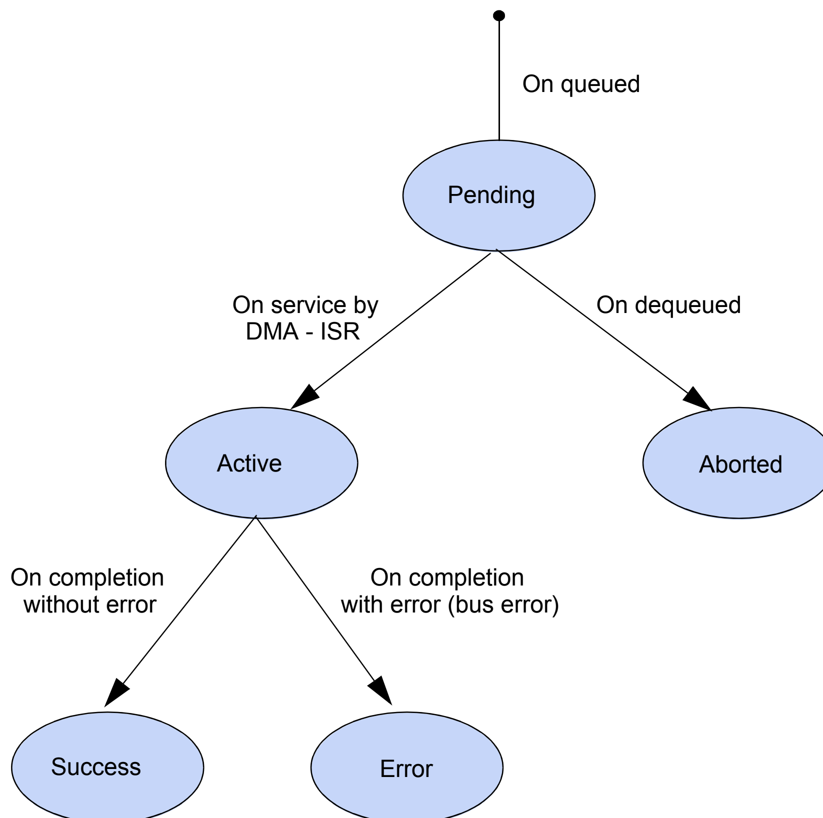
# Device Driver

The DMA device driver directly interacts with the DMA instance. This section describes the type definitions, and structures of the DMA device driver.

The LatticeMico DMA controller has two master ports, each of which can be connected to any address region. The programmer must set the DMA parameters, including source and destination addresses, that are valid for a given DMA instance. The DMA device driver implementation model requires that you queue a DMA request. The parameters defining the required DMA transaction are provided through a structure known as the DMA descriptor, which is described in "DMA Descriptor Structure" on page 19. The DMA device driver relies on DMA descriptors for initiating a DMA transaction. The DMA device driver manages the state of each queued DMA descriptor as it services them on a first-queued, first-serviced basis. The DMA device driver implementation enables you to queue multiple DMA descriptors for multiple DMA transfer requests. Such multiple requests are sequentially serviced by the DMA device driver. The DMA device driver relies on the DMA interrupt requests to identify the completion of a programmed DMA request. When a DMA request is completed, the DMA device driver notifies you through a

callback routine that you provided when you queued the request. The state diagram for a DMA descriptor is shown in Figure 13.

**Figure 13: DMA Descriptor State Chart**




A descriptor is set to the pending state when it is queued, indicating that it is added to the list of pending DMA descriptors for the associated DMA device. The DMA interrupt service routine (ISR) sets up the DMA parameters for performing a DMA transaction that is associated with a given descriptor. This ISR always services in a first-in, first-out fashion and marks the DMA descriptor that it is servicing as ACTIVE. When the DMA transaction is completed, the ISR is invoked. It checks the completion status of the DMA device. If the DMA device indicates a successful completion, it marks the DMA descriptor just serviced with SUCCESS; if the DMA device indicates error, the ISR marks the descriptor with ERROR. Whether the completion is successful or not, the ISR calls the callback routine registered for the descriptor to allow you to free up the descriptor, if desired, or perform any other operation. After completion, the DMA descriptor that was just serviced is removed from the list of descriptors. If a DMA descriptor is marked PENDING and you issue a request to de-queue the descriptor, the state of the descriptor is marked as ABORTED and is removed from the list of pending DMA descriptors.

## Type Definitions

This section explains the type definitions for the DMA device context structure, descriptor structure, and completion callback prototype.

**DMA Device Context Structure** The DMA device context structure, shown in Figure 14, contains the DMA component instance-specific information and is dynamically generated in the DDStructs.h header file. This information is largely filled in by the MSB managed build process, which extracts the DMA component-specific information from the platform definition file. The members should not be manipulated directly, because this structure is used exclusively by the device driver.

**Figure 14: DMA Device Context Structure**

```
typedef struct st_MicoDMACtx_t {
const char* name;
unsigned int base;
unsigned int irq;
unsigned int maxLength;
DeviceReg_t lookupReg;
unsigned int flags;
void * pCurr;
void * pHead;
void * prev;
void * next;
} MicoDMACtx_t;
```

Table 9 describes the parameters of the DMA device context structure shown in Figure 14.

**Table 9: DMA Device Context Structure Parameters**

| Parameter | Data Type | Description |
|---|---|---|
| name | const char * | Instance-specific component name entered in MSB |
| base | unsigned int | MSB-assigned base address |
| irq | unsigned int | MSB-assigned interrupt line |
| maxLength | unsigned int | Maximum length of any DMA transaction |
| lookupReg | DeviceReg_t | Used by the device driver to register the DMA controller component instance with the LatticeMico32 lookup service. Refer to the *LatticeMico32 Software Developer User Guide* for a description of the DeviceReg_t data type. |
| flags | unsigned int | Used internally by the device driver |
| pCurr | void * | Used internally for tracking active DMA descriptors |
| pHead | void * | Used internally for tracking queued DMA descriptors |

**Table 9: DMA Device Context Structure Parameters (Continued)**

| Parameter | Data Type | Description |
|---|---|---|
| prev | void * | Used internally by the lookup service for tracking multiple registered DMA instances |
| next | void * | Used internally by the lookup service for tracking multiple registered DMA instances |

**Note**

You may need to access theDMA device registers directly, but some of these registers are write-only. Implementing shadow registers in RAM can be an effective way to replace this missing capability.

**DMA Descriptor Structure** The DMA descriptor structure in Figure 15 contains information for a single DMA operation. It is key to all device-driver-implemented DMA functions, as described in “Functions” on page 21.

**Figure 15: DMA Descriptor Structure**

```
typedef struct st_DMADesc_t DMADesc_t;
struct st_DMADesc_t{
   /* address to read data from */
   unsigned int sAddr;

   /* address to write data to */
   unsigned int dAddr;

   /* Length of transfer.
    * NOTE: This is NOT length in bytes; it is the
    * number of data units to transfer. So for a 32-byte
    * transfer, this value must be set to 32; for thirty-two
    * 32-bit transfers, this value must still be set to 32.
    */
   unsigned int length;

   /* DMA transfer qualifier */
   unsigned int type;

   /* User-provided private data */
   void *priv;

   /* descriptor state */
   unsigned int state;

   /* used internally for chaining descriptors */
   DMACallback_t onCompletion;
   DMADesc_t *prev;
   DMADesc_t *next;
};
```

The type and description of each parameter in the DMA descriptor structure are shown in Table 10.

**Table 10: DMA Descriptor Structure**

| Parameter | Data Type | Description |
|---|---|---|
| sAddr | unsigned int | Starting address from which to read data for a DMA transaction |
| dAddr | unsigned int | Starting address to write data for a DMA transaction |
| length | unsigned int | Specifies the number of reads and writes to perform. This value must be set to the number of units to transfer. For a 32 8-bit transfer, this parameter must be set to a value of 32. For a 32 32-bit transfer, this value must still be set to 32 but the type parameter must be set to indicate a 32-bit transfer.<br>**Note**: The device driver code changes the length value before writing it to the DMA controller length register.<br>The DMA controller expects a length value of 128 in order to perform 32 32-bit transfers. The DMA controller expects a length value of 64 to perform 32 16-bit transfers.<br>The device-driver code performs the necessary arithmetic to make sure the correct number of transactions are performed. |
| type | DMAType_t | Can be an OR of the following enumerated type values:<br>▶ DMA_CONSTANT_SRC_ADDR<br>▶ DMA_CONSTANT_DST_ADDR<br>▶ DMA_16BIT_TRANSFER<br>▶ DMA_32BIT_TRANSFER<br>▶ DMA_BURST_SIZE_4<br>▶ DMA_BURST_SIZE_8<br>▶ DMA_BURST_SIZE_16<br>▶ DMA_BURST_SIZE_32<br>▶ DMA_BURST_SIZE_64<br>▶ DMA_BURST_ENABLE (Legacy. Replaced by BURST_SIZE_*X*)<br>▶ DMA_BURST_SIZE (Legacy. Replaced by BURST_SIZE_*X*) |
| state | unsigned int | Used internally to indicate the state of the descriptor. It is one of the following values once it is queued:<br>▶ MICODMA_STATE_SUCCESS<br>▶ MICODMA_STATE_PENDING<br>▶ MICODMA_STATE_ACTIVE<br>▶ MICODMA_STATE_ERROR<br>▶ MICODMA_STATE_ABORTED |
| onCompletion | DMACallback_t | Callback routine, if non-zero, to be invoked as part of the DMA ISR to indicate completion (success or failure) of this descriptor |
| prev | DMADesc_t * | Used internally for maintaining a list of queued descriptors |
| next | DMADexc_t * | Used internally for maintaining a list of queued descriptors |
| priv | void * | Your private data pointer |

**DMA Completion Callback Prototype** This prototype is the expected completion callback routine prototype. You can provide a valid pointer to a function like the callback routine that is invoked when a DMA descriptor transaction is completed.

```
typedef void(*DMACallback_t)(DMADesc_t *desc, unsigned int
status);
```

This callback takes two parameters:

- Pointer to the DMA descriptor
- Status of the DMA descriptor, which denotes the state of the DMA descriptor, as given in the “DMA Descriptor Structure” on page 19.

## Functions

This section describes the implemented device-driver-specific functions.

### MicoDMAInit Function

```
void MicoDMAInit(MicoDMACtx_t *ctx);
```

This function initializes a LatticeMico DMA instance according to the passed DMA context structure. This initialization function is responsible for stopping the DMA (not currently supported in RTL) and initializing members of the passed DMA context.

As part of the managed build process, the LatticeDDInit function calls this initialization routine for each DMA instance in the platform.

Table 11 describes the parameter in the MicoDMAInit function syntax.

**Table 11: MicoDMAInit Function Parameter**

| Parameter | Description |
|---|---|
| MicoDMACtx_t * | Pointer to the DMA context representing a valid DMA instance |

### MicoDMAQueueRequest Function

```
unsigned int MicoDMAQueueRequest(MicoDMACtx_t *ctx, DMADDesc_t
*desc, DMACallback_t callback);
```

This function queues a DMA descriptor to the end of the list of the queued descriptors. If it is the first descriptor being queued, this function initiates a DMA transaction for this descriptor.

Set the following parameters of the DMA descriptor before calling this function:

- sAddr – Specifies the starting address of the read location
- dAddr – Specifies the starting address of the write location
- length **–** Specifies the number of transactions to perform

- type – Specifies the type of transaction to perform. It must be an OR of the listed values, if the transaction involves a constant source address (read) or a constant destination address (write), or if it involves 16-bit or 32-bit transfers. If DMA_16BIT_TRANSFER or DMA_32BIT_TRANSFER is not part of the type member, the DMA transactions will be 8-bit transfers. The type of transaction can be one of the following:
  - DMA_CONSTANT_SRC_ADDR – For transferring from a single source address
  - DMA_CONSTANT_DST_ADDR – For transferring to a constant destination address
  - DMA_16BIT_TRANSFER – For transferring 16-bit data between 16-bit locations
  - DMA_32BIT_TRANSFER – For transferring 32-bit data between 32-bit locations
  - DMA_BURST_SIZE_4 – Indicates that the burst size is four times the transfer size.
  - DMA_BURST_SIZE_8 – Indicates that the burst size is eight times the transfer size.
  - DMA_BURST_SIZE_16 – Indicates that the burst size is sixteen times the transfer size.
  - DMA_BURST_SIZE_32 – Indicates that the burst size is thirty-two times the transfer size.
  - DMA_BUSRT_SIZE_64 – Indicates that the burst size is sixty-four times the transfer size.
  - DMA_BURST_SIZE – For selecting burst transfer sizes. This setting is valid only when DMA_BURST_ENABLE is also set. When it is set, the burst size is eight times the transfer size. When it is not set and burst is enabled, the burst size is four times the transfer size. The transfer length is required to be a multiple of the burst size. This parameter is for legacy code.
  - DMA_BURST_ENABLE – For enabling burst-mode transfers. This parameter is for legacy code.

If none of the qualifiers are needed, the “type” parameter must be set to 0.

**Note**

The provided DMA descriptor must not be modified or removed from memory until either the DMA transaction associated with this descriptor has been completed or until it has been successfully dequeued.

Table 12 describes the parameters in the MicoDMAQueueRequest function syntax.

**Table 12: MicoDMAQueueRequest Function Parameters**

| Parameter | Description | Notes |
|---|---|---|
| MicoDMACtx_t * | Pointer to a DMA context | |
| DMADesc_t * | Pointer to a valid DMA descriptor | Must be preserved until the DMA operation is completed and its callback routine is invoked or until it is successfully dequeued. |
| DMACallback_t | Callback to be invoked when a DMA transaction associated with this descriptor is completed | Can be 0 if no callback is desired. |

Table 13 shows the values returned by the MicoDMAQueueRequest function.

**Table 13: Values Returned by MicoDMAQueueRequest Function**

| Return Value | Description |
|---|---|
| 0 | Successfully queued the descriptor |
| MICODMA_ERR_INVALID_ POINTERS | Returned if either the DMA context pointer is null or the descriptor pointer is null |
| MICODMA_ERR_DESC_ LEN_ERR | Returned if the requested length of the transaction exceeds the maximum allowed by DMA or if the requested length is 0 |

### MicoDMADequeueRequest Function

```
unsigned int MicoDMADequeueRequest(MicoDMACtx_t *ctx, DMADesc_t
*desc, unsigned int callback);
```

This function dequeues the provided descriptor from the list of queued descriptors. Only those descriptors that are pending can be dequeued.

Table 14 describes the parameters in the syntax of the MicoDMADequeueRequest function syntax.

**Table 14: MicoDMADequeueRequest Function Parameters**

| Parameter | Description | Notes |
|---|---|---|
| MicoDMACtx_t * | Pointer to a DMA context representing a valid DMA instance | |
| DMADesc_t * | Pointer to a valid DMA descriptor | This descriptor must be preserved until the DMA operation is completed and its callback routine is invoked or until it is successfully dequeued. |
| callback | Indicates whether the associated callback routine with this descriptor should be called, if successfully dequeued | If this value is non-zero, the associated callback, if registered, will be called with the status argument set to MICODMA_STATE_ ABORTED. |

Table 15 shows the values returned by the MicoDMADequeueRequest function.

**Table 15: Values Returned by MicoDMADequeueRequest Function**

| Return Value | Description |
|---|---|
| 0 | Successfully dequeued the descriptor |
| MICODMA_ERR_DESCRIPTOR_ NOT_PENDING | Descriptor was not in pending state and therefore not removed, if still queued. |
| MICODMA_ERR_INVALID_POINTERS | Returned to indicate that the `ctx` or `desc` pointers are null or that the descriptor link-list parameters seem invalid |

### MicoDMAGetState Function

```
unsigned int MicoDMAGetState(DMADesc_t *desc);
```

This function retrieves the state of a queued descriptor and takes the pointer to the descriptor as an argument.

The returned value indicates the state of the descriptor and is one of the following values:

- MICODMA_STATE_SUCCESS
- MICODMA_STATE_PENDING
- MICODMA_STATE_ACTIVE
- MICODMA_STATE_ERROR
- MICODMA_STATE_ABORTED

### MicoDMAPause Function

```
unsigned int MicoDMAPause(MicoDMACtx_t *ctx);
```

This function pauses the processing of the DMA descriptors queued for the given DMA instance. Any active DMA transaction is allowed to finish.

Table 16 describes the parameter in the MicoDMAPause function syntax.

**Table 16: MicoDMAPause Function Parameter**

| Parameter | Description |
|---|---|
| MicoDMACtx_t * | Pointer to a DMA context representing a valid DMA instance |

Table 17 shows the values returned by the MicoDMAPause function.

**Table 17: Values Returned by MicoDMAPause Function**

| Return Value | Description |
|---|---|
| 0 | Successfully paused the DMA transactions |
| MICODMA_ERR_INVALID_POINTERS | Returned to indicate that the `ctx` value is 0 |

### MicoDMAResume Function

```
unsigned int MicoDMAResume(MicoDMACtx_t *ctx);
```

This function resumes processing of any queued DMA descriptors associated with the provided DMA instance.

Table 18 describes the parameter in the MicoDMAResume function syntax.

**Table 18: MicoDMAResume Function Parameter**

| Parameter | Description |
|---|---|
| MicoDMACtx_t * | Pointer to a DMA context representing a valid DMA instance |

Table 19 shows the values returned by the MicoDMAResume function.

**Table 19: Values Returned by MicoDMAResume Function**

| Return Value | Description |
|---|---|
| 0 | Successfully resumed the DMA transactions |
| MICODMA_ERR_INVALID_POINTERS | Returned to indicate that the `ctx` value is 0 |

### Services

The DMA device driver registers DMA instances with the LatticeMico32 lookup service, using their instance names for device names and “DMADevice” as the device type.

For information on the LatticeMico32 lookup service, refer to the *LatticeMico32 Software Developer User's Guide*.

## Software Usage Example

The default managed build process initializes the DMA instance by invoking the DMA initialization function described in “Functions” on page 21. As a result, the DMA instance becomes available to the LatticeMico32 lookup service.

In the following example, it is assumed that the platform contains a DMA controller named “dma.”

**Figure 16: DMA Controller Sofware Example**

```
#include <stdio.h>
#include "LookupServices.h"
#include "MicoDMA.h"
/*
 * This function is a DMA completion callback and is
 * invoked within an interrupt service routine.
 * Therefore, it must be quick and short.
 */
void OnDMAComplete(DMADesc_t *desc, unsigned int status)
{
   /*
    * NOTE: It is already known at this point how the DMA
    * transaction ended in the "status" field.
    */
   /* access the private data */
   volatile unsigned int *p_iDone = desc->priv;

   /*signal the DMA is done */
   *p_iDone = 1;
   return;
}
```

**Figure 16: DMA Controller Sofware Example (Continued)**

```
int main(void)
   volatile unsigned int iDone = 0;
   static DMADesc_t dmaDesc;
   /* Static ensures that this remains valid for the duration
    * of the program */
   /* Fetch dma device by name */
   MicoDMACtx_t *dma = (MicoDMACtx_t *)MicoGetDevice("dma");

   /* Prepare the DMA descriptor. We want to transfer:
    *       256 32-bit words from
    *       0x00002000 (valid memory location) to
    *       0x00004000 (valid memory location)
    */


   dmaDesc.sAddr = 0x00002000;       /* SOURCE ADDRESS      */
   dmaDesc.dAddr = 0x00004000;        /* DESTINATION ADDRESS */
   dmaDesc.length = 256;              /* 256 reads/writes    */
   dmaDesc.type = DMA_32BIT_TRANSFER; /* 32-bit transfers    */
   dmaDesc.priv = (void *) &iDone;    /* my private data     */

   /* Queue this DMA descriptor and provide a callback for
    * completion notification */
   if {MicoDMAQueRequest (dma, &dmaDesc, OnDMAComplete) == 0;){
      printf ("successfully queued DMA request\n");
   }else{
      printf("failed to queue DMA request\n");
   }
   /* wait for DMA transaction to be completed */
   while(iDone == 0);
   /*
    * Print status of DMA transaction (we knew it in the
    * callback already, but as an example, still query the
    * status of the DMA descriptor.)
    */

   switch (MicoDMAGetState(&dmaDesc)){
      case MICODMA_STATE_ERROR:{
         /* Since DMA is complete, it can complete with error
          * if an address is incorrect */
         printf("dma completed with error \n");
         break;
      }

      case MICODMA_STATE_SUCCESS:{
         /* Since DMA is complete, it can complete successfully
          * if the addresses are okay */
         printf("successfully completed DMA  \n");
         break;
      }
```

**Figure 16: DMA Controller Sofware Example (Continued)**

```
      case MICODMA_STATE_PENDING:{
         /*
          * We queued a single request and waited for it to
          * be completed, and so the state will not
          * be this.
          */
         printf("dma pending (cannot happen in this sample
         code) \n");
         break;
      }

      case MICODMA_STATE_ACTIVE:{
         /*
          * We queued a single request and waited for it to be
          * completed, and so the state will not be this.
          */
         printf("dma active (cannot happen in this sample code)
         \n");
         break;
      }

      case MICODMA_STATE_ABORTED:{
         /*

         * We queued a single request and waited for it to be
          * completed, and so the state will not be this
          */
         printf("dma aborted (cannot happen in this sample
         code) \n");
         break;
      }
      default:{
         printf("unknown state\n");
         break;
      }
   }
   return(0);
}
```

# Accessing DMA Controller without Device Drivers

The device driver functions and macros hide the DMA controller's implementation from the software developer by providing a software translation layer between the developer's application and the actual hardware-specific details. It is, nevertheless, possible to directly access the DMA controller without using Lattice-provided drivers. You can do this by accessing (reading from or writing to) the registers defined in Table 3 on page 8. The orientation of data is very important, because LatticeMico32 is a big-endian microprocessor. Therefore the software developer is advised to read the following information to understand the impact of endianness when the microprocessor interacts with the component's registers. In a big-endian

architecture, the most-significant byte of a multi-byte object is stored at the lowest address, and the least-significant byte of that object is stored at the highest address.

Assume that you have a design that contains the DMA controller and that it is assigned a base address of 0x80000000. Now let's consider that one wants to write to the "SA" register at an offset of 0x0 from the base address. The least-significant byte of the address is in bits 31-24 (byte address 3), while the most-significant byte of the address is in bits 7-0 (byte address 0). Lets assume that the source address of the DMA transfer is 0x10203040 and that we want to update the SA register with this address. There are two ways in C to write to this register, depending on whether one is performing a byte ("unsigned char" or "signed char") write or whether one is performing a word ("unsigned int" or "signed int") write.

Figure 17 shows sample code that uses a byte write.

**Figure 17: Correct access to SA register using byte write**

```
unsigned char byte0 = 0x10;
unsigned char byte1 = 0x20;
unsigned char byte2 = 0x30;
unsigned char byte3 = 0x40;
*(volatile unsigned char *)0x80000000 = byte0;
*(volatile unsigned char *)0x80000001 = byte1;
*(volatile unsigned char *)0x80000002 = byte2;
*(volatile unsigned char *)0x80000003 = byte3;
```

Figure 18 shows sample code that uses a word write. It will write the value 0x10203040 to the SA register. This is correct.

**Figure 18: Correct access to SA register using word write**

```
unsigned int address = 0x10203040;
*(volatile unsigned int *)0x80000000 = address;
```

On the other hand, the sample code in Figure 19 will produce incorrect behavior, because a value of 0x40302010 will be written to the SA register.

**Figure 19: Incorrect access to SA register using word write**

```
unsigned int address = 0x40302010;
*(volatile unsigned int *)0x80000000 = address;
```

# LatticeMico8 Microcontroller Software Support

This section describes the LatticeMico8 microcontroller software support provided for the LatticeMico DMA controller. It describes the basic DMA device-driver interface.

## Device Driver

The DMA device driver directly interacts with the DMA instance. This section describes the type definitions, functions and macros of the DMA device driver.

### Type Definitions

This section explains the DMA device context structure shown in Figure 20. It contains the DMA component instance-specific information and is generated dynamically in the DDStructs.h header file. This information is largely filled in by the MSB managed build process, which extracts the DMA component-

specific information from the platform definition file. The members should not be manipulated directly, because this structure is used exclusively by the device driver.

**Figure 20: DMA Device Context Structure**

```
typedef struct st_MicoDMACtx_t {
const char* name;
unsigned int base;
unsigned int irq;
} MicoDMACtx_t;
```

Table 20 describes the parameters of the DMA device context structure shown in Figure 20.

**Table 20: DMA Device Context Structure Parameters**

| Parameter | Data Type | Description |
|---|---|---|
| name | const char * | Instance-specific component name entered in MSB |
| base | unsigned int | MSB-assigned base address |
| irq | unsigned int | MSB-assigned interrupt line |

## Functions

This section describes the implemented device driver-specific functions.

### MicoDMAInit Function

```
void MicoDMAInit(MicoDMACtx_t *ctx);
```

This function initializes a LatticeMico DMA instance according to the passed DMA context structure. This initialization function is responsible for stopping the DMA (not currently supported in RTL) and initializing members of the passed DMA context. As part of the managed build process, the LatticeDDInit function calls this initialization routine for each DMA instance in the platform.

Table 21 describes the parameter in the MicoDMAInit function syntax.

**Table 21: MicoDMAInit Function Parameter**

| Parameter | Description |
|---|---|
| MicoDMACtx_t * | Pointer to the DMA context representing a valid DMA instance. |

### MicoDMATransfer Function

This function configures the LatticeMico DMA instance for a new transfer, initiates the transfer, and then optionally monitors the transfer for successful or unsuccessful completion.

Table 22 describes the parameter in the MicoDMATransfer function syntax.

**Table 22: MicoDMATransfer Function Parameters**

| Parameter | Description |
|---|---|
| MicoDMACtx_t * | Pointer to the DMA context representing a valid DMA instance. |
| unsigned long srcAddress | Source memory address of current DMA transfer. |
| unsigned long dstAddress | Destination memory address of current DMA transfer. |
| unsigned long totalBytes | Total number of bytes in current DMA transfer. |
| unsigned char burst | Indicates whether DMA engine should use WISHBONE Burst cycles to perform current DMA transfer or WISHBONE Classic cycles. 1 indicates Burst and 0 indicates Classic. Burst transfers are faster but lock up use of WISHBONE bus by the DMA engine for longer durations than Classic transfers. |
| unsigned char burstSize | Indicates the number of transfers performed in each WISHBONE Burst cycle. The legal values are 1, 4, 8, 16, 32, and 64. The larger the value, the faster the transfer (at the expense of locking up the WISHBONE bus for longer durations). A value of 1 is the same as a WISHBONE Classic cycle. |
| unsigned char srcConstant | Indicates whether the read address increments or remains constant during the entire transfer. 1 indicates constant address transfer. 0 indicates linear incrementing address transfer in which the read address is incremented by a constant value for each subsequent WISHBONE transfer. |
| unsigned char dstConstant | Indicates whether the write address increments or remains constant during the entire transfer. 1 indicates constant address transfer. 0 indicates linear incrementing address transfer in which the write address is incremented by a constant value for each subsequent WISHBONE transfer. |
| unsigned char incValue | Indicates the constant value by which the address is incremented. The legal values are 1, 2, or 4. If the WISHBONE data bus width is 8 bits, then the function will always set the increment value to 1 byte regardless of the value of this variable. |
| unsigned char interrupt | Indicates whether the DMA engine should be configured to raise an interrupt after the transfer is complete. 1 indicates interrupts are to be enabled. 0 indicates that interrupts are to be disabled. When interrupts are disabled, the function monitors the Status register to check if the transfer has completed and then exits. |

### MicoDMAISR Function

```
void MicoDMAISR(MicoDMACtx_t *ctx);
```

This function implements the interrupt handler for the DMA controller. When the DMA controller is configured to raise an interrupt on the completion of the DMA transfer, this function is invoked by LatticeMico8. This function is a sample implementation. The software developer can customize it to their own requirements by:

- Defining the __MICODMA_USER_IRQ_HANDLER__ preprocessor macro which disables the sample implementation, and
- Implementing MicoDMAISR function in the user's code.

Table 23 describes the parameter in the MicoDMAISR function syntax.

**Table 23: MicoDMAISR Function Parameters**

| Parameter | Description |
|---|---|
| MicoDMACtx_t * | Pointer to the DMA context representing a valid DMA instance. |

## C Preprocessor Macro Definitions

This section describes the 'function-like' macro definitions that are defined in the LatticeMico8 software driver for the DMA controller to access the component's Register Map shown in Table 3 on page 8 and perform certain operations. All 'function-like' macro definitions take input parameters that are used in performing the operations encoded within the macro. Table 24 describes these macros and can be found in C header file 'MicoDMA.h'. The macros have either one or two input arguments. The first input argument "X" is always the base address of the DMA controller, as assigned by MSB. The

second input argument "Y" is optional, depending on the macro's function. Table 24 also shows how each macro can be used by the software developer in his application code.

**Table 24: C Preprocessor Macros**

| Macro Name | Second Argument to Macro | Description |
|---|---|---|
| MICO_DMA_RD_SRCADDR | The current contents of the SA register in Table 3 on page 8. | This macro reads the contents of the SA register.<br>Usage:<br>If DMA controller's base address is 0x80001000,<br>unsigned long srcAddress;<br>MICO_DMA_RD_SRCADDR (0x80001000, srcAddress); |
| MICO_DMA_WR_SRCADDR | The 32-bit address of the source memory from which the DMA engine will start reading. | This macro updates the SA register with a new address of the source memory from where the DMA engine will read.<br>Usage:<br>If DMA controller's base address is 0x80001000 and source memory's address is 0x10000000 then,<br>MICO_DMA_WR_SRCADDR (0x80001000, 0x10000000); |
| MICO_DMA_RD_DSTADDR | The current contents of the DA register in Table 3 on page 8. | This macro reads the contents of the DA register.<br>Usage:<br>If DMA controller's base address is 0x80001000,<br>unsigned long dstAddress;<br>MICO_DMA_RD_DSTADDR (0x80001000, dstAddress); |
| MICO_DMA_WR_DSTADDR | The 32-bit address of the destination memory to which the DMA engine will start writing. | This macro updates the DA register with a new address of the destination memory to which the DMA engine will write.<br>Usage:<br>If DMA controller's base address is 0x80001000 and destination memory's address is 0x10000000 then,<br>MICO_DMA_WR_DSTADDR (0x80001000, 0x10000000); |
| MICO_DMA_RD_XFERLEN | The current contents of the LR register in Table 3 on page 8. | This macro reads the contents of the LR register.<br>Usage:<br>If DMA controller's base address is 0x80001000,<br>unsigned long totalBytes;<br>MICO_DMA_RD_XFERLEN (0x80001000, totalBytes); |
| MICO_DMA_WR_XFERLEN | The 32-bit value indicating the number of bytes that will be transferred. | This macro updates the LR register with the total number of bytes that need to be transferred in the current transfer.<br>Usage:<br>If DMA controller's base address is 0x80001000 and the number of bytes to transfer is 256 then,<br>MICO_DMA_WR_XFERLEN (0x80001000, 256); |

**Table 24: C Preprocessor Macros (Continued)**

| Macro Name | Second Argument to Macro | Description |
|---|---|---|
| MICO_DMA_RD_CONTROL | The current contents of the CR register in Table 3 on page 8. | This macro reads the contents of the CR register.<br>Usage:<br>If DMA controller's base address is 0x80001000,<br>unsigned char control_reg;<br>MICO_DMA_RD_CR (0x80001000, control_reg); |
| MICO_DMA_WR_CONTROL | The value to be written to the CR register in Table 3 on page 8. | This macro updates the CR register with type of transfer that is performed by the DMA engine. Refer to Table 3 for details on how to set up CR register for different types of transfers.<br>Usage:<br>If DMA controller's base address is 0x80001000 and the value to write to CR register is 0x80 then,<br>MICO_DMA_WR_CR (0x80001000, 0x80); |
| MICO_DMA_RD_STATUS | The current contents of the SR register in Table 3 on page 8. | This macro reads the contents of the SR register.<br>Usage:<br>If DMA controller's base address is 0x80001000,<br>unsigned char status_reg;<br>MICO_DMA_RD_SR (0x80001000, status_reg); |
| MICO_DMA_WR_STATUS | The value to be written to the SR register in Table 3 on page 8. | This macro updates the SR register. Refer to Table 3 for details on how to set up SR register.<br>Usage:<br>If DMA controller's base address is 0x80001000 and the value to write to SR register is 0x08 then,<br>MICO_DMA_WR_SR (0x80001000, 0x08); |
| MICO_DMA_BUSY | - | This macro checks if the DMA engine is busy with a transfer.<br>Usage:<br>If DMA controller's base address is 0x80001000,<br>unsigned char busy;<br>busy = MICO_DMA_BUSY (0x80001000); |
| MICO_DMA_START | - | This macro initiates a new transfer in the polling mode (i.e., interrupts are disabled).<br>Usage:<br>If DMA controller's base address is 0x80001000,<br>MICO_DMA_START (0x80001000); |

## Software Usage Example

The example in Figure 21 assumes that the platform contains a DMA controller named “dma.” The DMA controller is configured to raise an interrupt when a transfer is completed and the software developer is implementing his own interrupt handler.

**Figure 21: DMA Controller Sofware Example**

```
#include "DDStructs.h"
#include "MicoDMA.h"

unsigned char done;

/*
 * This function is called when the DMA controller issues an
interrupt. The
 * interrupt is issued when the DMA completes the transfer or
when there is
 * an error in the transfer. Since the interrupt handler is
implemented in
 * user code, the __MICODMA_USER_IRQ_HANDLER__ preprocessor
macro must be
 * enabled while compiling the software project.
 */
void MicoDMAISR (MicoDMACtx_t *ctx)
{
  /* signal the DMA is done */
  done = 0x1;
  return;
}

int main (void)
{
  /* Fetch DMA instance named 'dma' */
  MicoDMACtx_t *dma = &wb_dma_ctrl_dma;

  /* Reset done flag */
  done = 0x0;

  /* Initiate DMA transfer */
  MicoDMATransfer(dma,
      0x10000000, // source address
      0x20000000, // destination address
      1024,       // number of bytes to transfer
      1,          // use WISHBONE burst cycles for transfer
      4,          // 4 transactions per WISHBONE burst cycle
      0,          // source address linearly increments
      0,          // destination address linearly increments
      1,          // addresses increment by 1 byte
      1);         // raise interrupt on completion of transfer


  /* Check if transfer is done */
  do {
    if (done == 0x1)
      break;
  } while (1);

  return 0;
}


---
```

## Revision History

| Component Version | Description |
|---|---|
| 1.0 | Initial release. |
| 3.0 (7.0 SP2) | Because the read and write channel worked in parallel, the write channel started writing data to the slave as soon as the FIFO is not empty.<br>Increased burst size to support bigger bursts from a current value of 4 and 8 to 16 and 32, respectively. DMA now supports four burst sizes: 4, 8, 16, and 32. The Burst Size field of the control register was increased to 2 bits.<br>A glitch was removed on the S_ACK_O signal. |
| 3.1 (8.0) | DMA Engine upgraded to comply with Rule 3.100 of Wishbone Specifications, which deal with byte alignment for transfers that are less than the width of Wishbone data bus. |
| 3.2 (8.1 SP1) | The data busses on the three WISHBONE interfaces can be configured to be 8 or 32 bits. Support added for handling WISHBONE RTY (retry) for burst transfers. Support added for handling WISHBONE ERR (error). Register map updated to support 8-bit and 32-bit WISHBONE data bus. |
| 3.3 | Added software support for LatticeMico8. |
| 3.3 | Updated document with new corporate logo. |

.

## Trademarks

Lattice Semiconductor Corporation, L Lattice Semiconductor Corporation (logo), L (stylized), L (design), Lattice (design), LSC, CleanClock, Custom Movile Device, DiePlus, E2CMOS, Extreme Performance, FlashBAK, FlexiClock, flexiFLASH, flexiMAC, flexiPCS, FreedomChip, GAL, GDX, Generic Array Logic, HDL Explorer, iCE Dice, iCE40, iCE65, iCEcable, iCEchip, iCEcube, iCEcube2, iCEman, iCEprog, iCEsab, iCEsocket, IPexpress, ISP, ispATE, ispClock, ispDOWNLOAD, ispGAL, ispGDS, ispGDX, ispGDX2, ispGDXV, ispGENERATOR, ispJTAG, ispLEVER, ispLeverCORE, ispLSI, ispMACH, ispPAC, ispTRACY, ispTURBO, ispVIRTUAL MACHINE, ispVM, ispXP, ispXPGA, ispXPLD, Lattice Diamond, LatticeCORE, LatticeEC, LatticeECP, LatticeECP-DSP, LatticeECP2, LatticeECP2M, LatticeECP3, LatticeECP4, LatticeMico, LatticeMico8, LatticeMico32, LatticeSC, LatticeSCM, LatticeXP, LatticeXP2, MACH, MachXO, MachXO2, MACO, mobileFPGA, ORCA, PAC, PAC-Designer, PAL, Performance Analyst, Platform Manager, ProcessorPM, PURESPEED, Reveal, SiliconBlue, Silicon Forest, Speedlocked, Speed Locking, SuperBIG, SuperCOOL, SuperFAST, SuperWIDE, sysCLOCK, sysCONFIG, sysDSP, sysHSI, sysI/O, sysMEM, The Simple Machine for Complex Design, TraceID, TransFR, UltraMOS, and specific product designations are either registered trademarks or trademarks of Lattice Semiconductor Corporation or its subsidiaries in the United States and/or other countries. ISP, Bringing the Best Together, and More of the Best are service marks of Lattice Semiconductor Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.