

LatticeMico32 DMA Controller

The LatticeMico32 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 3.0 version (formerly the 7.0 SP2 version) of the LatticeMico32 DMA controller.

Features

The LatticeMico32 DMA controller includes the following features:

- ◆ WISHBONE B.3 interface
- ◆ Parameterized data bus width (8-, 16-, or 32-bit)
- ◆ Memory-to-memory transfer (contains addressable peripheral)
- ◆ A slave port that adds an interrupt port (INT_O) to the master port (CPU):
 - ◆ Fixed- and variable-length transfers
 - ◆ Static or dynamic address assertion
- ◆ Parameterized burst support (4, 8, 16, or 32)

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

Functional Description

The LatticeMico32 DMA controller provides a high-performance memory-to-memory data transfer engine for use in LatticeMico32 System platforms. The DMA controller is simultaneously a WISHBONE master and a WISHBONE slave device. The slave side of the component is connected and controlled by the LatticeMico32 microprocessor. The master side provides independent read and write I/O ports that can access memory and peripheral devices connected to the WISHBONE bus.

DMA transfers are initialized and started by the LatticeMico32 microprocessor. The LatticeMico32 microprocessor sets the start address, destination address, byte count, and then requests the DMA engine to begin memory-to-memory transactions. As a WISHBONE master device, the DMA controller begins reading and writing memory values as soon as it has access to the WISHBONE bus. The LatticeMico32 microprocessor and the DMA controller access the WISHBONE bus through an arbiter component. The arbiter assigns each master device a priority for access to the WISHBONE bus. The lower the priority number assigned, the higher the priority given to the master device for access to the WISHBONE bus. Figure 1 is an example of two master devices accessing the WISHBONE bus through the arbiter.

Figure 1: DMA Usage with Shared-Bus Arbitration in LatticeMico32 System

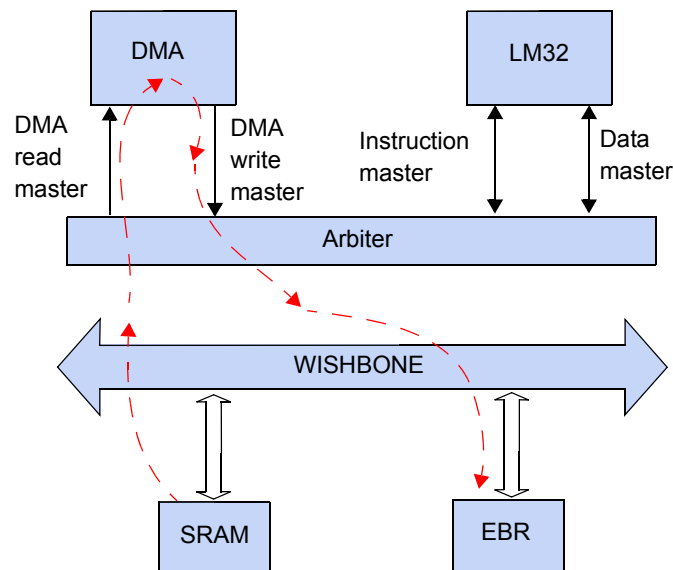
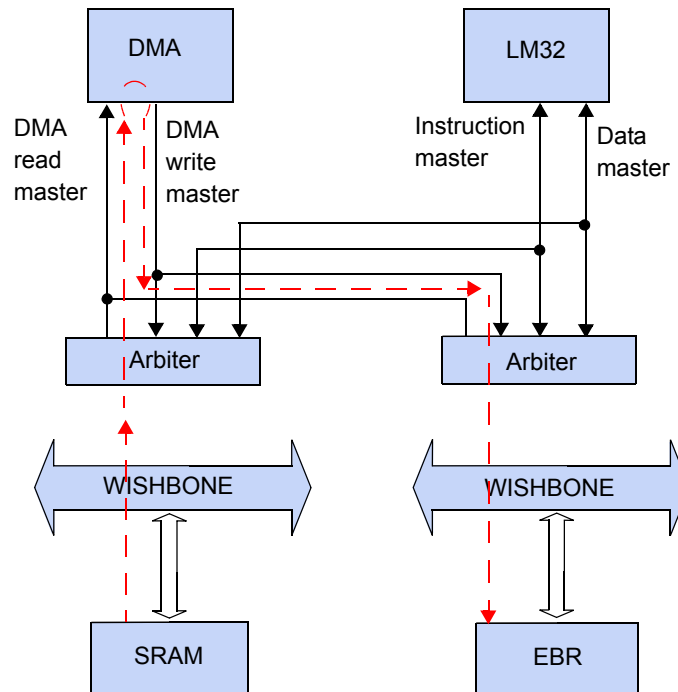


Figure 2 is an example of two master devices accessing the WISHBONE bus through two arbiters in a slave-side arbitration scheme.

Figure 2: DMA Usage with Slave-Side Arbitration in LatticeMico32 System



DMA transfers proceed unassisted until stopped. The DMA unit stops transferring data when a preprogrammed number of data bytes have been transferred, or an external device requests that the DMA controller terminate transactions. Transactions are terminated externally by either the RTY_I or the ERR_I inputs. RTY_I can terminate variable-length DMA transfers, and ERR_I can terminate both fixed- and variable-length transfers. If the ERR_I input is used to terminate the DMA transactions, the status bit in the status register is asserted to indicate that the transfer aborted because of an error. The LatticeMico32 microprocessor cannot terminate a DMA transaction currently in progress.

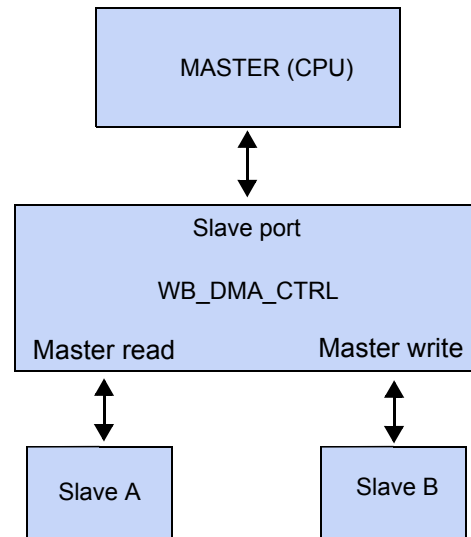
The LatticeMico32 microprocessor can use one of two methods for determining when a DMA transfer has finished:

- ◆ 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 RTY_I or ERR_I is asserted, the controller asserts the interrupt signal. This is the preferred mode of operation and is directly supported by the LatticeMico32 DMA controller device-driver source code.

Regardless of the method used to detect the termination of the DMA cycle, the LatticeMico32 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. For a variable-length transfer, the controller continues to read from the read address until the peripheral component asserts either `RTY_I` or `ERR_I`. A variable-length transfer is defined when the CPU sets the length register to zero and writes a 1 to the start bit in the status register.
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 LatticeMico32 firmware to poll the DMA_BUSY bit in the status register to detect the end of a DMA transfer..

Note

When the last interrupt request has not been cleared and a DMA transaction is in progress, the interrupt request can be cleared by reading the status register.

When DMA_BUSY is 1, the status bit in the status register reports the completion state of the previous DMA transfer.

The DMA controller supports burst-mode transfers.

The DMA controller does not assert the WISHBONE LOCK_O signal. Other master components can gain access to the bus at the conclusion of any read/write cycle initiated by the DMA controller.

The source address boundary and destination address boundary must comply with the data width. For example, if the data bus width is 16-bit, the least significant bit of the address must always be zero.

The cache line wrap feature is not supported; BTE_O is fixed at 00.

The slave port does not have RTY_O and ERR_O signals.

The slave port only accepts 32-bit memory transactions. The DMA controller does not respond to the SEL_O controls from the Mico32.

The DMA controller does not allow data to be transferred across data buses of differing widths. Only the transactions shown in Table 1 are permitted:

Table 1: Data Transfers Permitted by the DMA Controller

RD Data Bus Width	WR Data Bus Width
8	8
16	16
32	32

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 LatticeMico32 DMA controller.

UI Parameters

Table 2 shows the UI parameters available for configuring the LatticeMico32 DMA controller through the Mico System Builder (MSB) interface.

Table 2: 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.	0X80000000–0XFFFFFFF 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			
Length Width	Specifies the number of bits in the length register. The length register holds a count value that determines the number of DMA transactions to be performed. The default value permits up to 65535 (0xFFFF) memory transactions to be performed.	1 – 32	16

I/O Ports

Table 3 describes the input and output ports of the LatticeMico32 DMA controller.

Table 3: 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 3: DMA Controller I/O Ports

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 3: DMA Controller I/O Ports

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 LatticeMico32 DMA controller includes the registers shown in Table 4.

Table 4: Register Map

Register Name	Offset	31-7	6	5-4	3	2	1	0
SA	0x00							
DA	0x04							
LR	0x08							
CR	0x0c	Reserved	Burst mode	Burst size	Increment		D_CON	S_CON
SR	0x10	Reserved				Status	IE	DMA_BUSY

Table 5 through Table 9 provide details about each register of the LatticeMico32 DMA controller.

Table 5: SA Register

Register Name	Bit	Access Mode	Description
Source address	31: 0	Read/write	Source address for the DMA transfer

Table 6: DA Register

Register Name	Bit	Access Mode	Description
Destination address	31: 0	Read/write	Destination address for the DMA transfer

Table 7: Length Register

Register Name	Bit	Access Mode	Description
Length register	LENGTH_WIDTH-1:0	Read/write	Data length counted by byte unit. Width is selectable on synthesis.

Note

The length register contains the number of bus cycles to perform. Any non-zero value is defined as a fixed-length transfer. Variable-length transfers are executed when the length register is written to zero. For example, the length field should be set to 32 for eight 32-bit transfers, sixteen 16-bit transfers, and thirty-two 8-bit transfers. For a variable-length transfer, where the register data value is 0, the RTY_I from the peripheral device indicates the end of variable-length transmission. This register is decremented as the transfer progresses. The CPU can use the decremented count to determine the number of bytes that the DMA controller transferred.

When burst mode is enabled, if the length is less than the burst size, a minimum of the burst size will be transferred.

Table 8: Control Register Bit Definition

Register 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: 00 = increment SA/DA by 1, decrement LR by 1 01 = increment SA/DA by 2, decrement LR by 2 10 = increment SA/DA by 4, decrement LR by 4 11 = increment SA/DA by 4, decrement LR by 4
Burst Size	5-4	00	Read/write	00 = burst length = INC * 4 01 = burst length = INC * 8 10 = burst length = INC * 16 11 = burst length = INC * 32
Burst Mode Enable	6	0	Read/write	1 = Enable burst mode 0 = Disable burst mode
Reserved	31:7	0	Read/write	Reserved

Table 9: Status Register Bit Definition

Register 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.

The structure in Figure 4 depicts the register map layout for the DMA controller. The elements are self-explanatory and are based on the register map shown in Table 4. 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 4: 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 int status;
}MicoDMA_t;
```

Timing Diagrams

The timing diagrams featured in Figure 5 through Figure 11 show the timing of the DMA controller's WISHBONE and external signals.

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

Figure 5: DMA Controller Write Internal Register

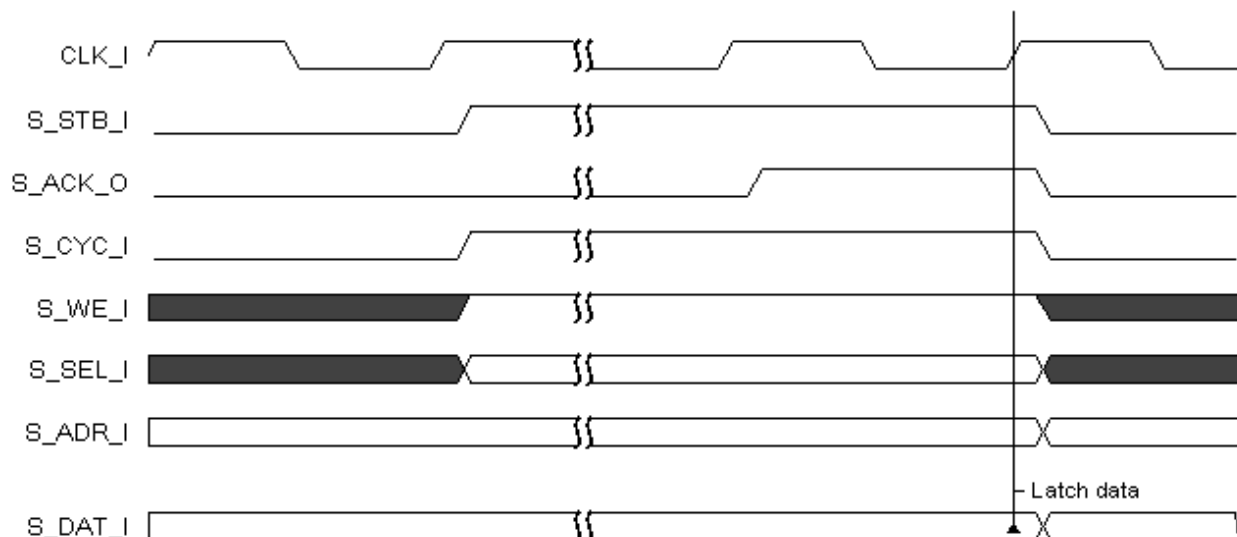


Figure 6 shows how the DMA controller's slave port reads the data in the internal register.

Figure 6: DMA Controller Read Internal Register

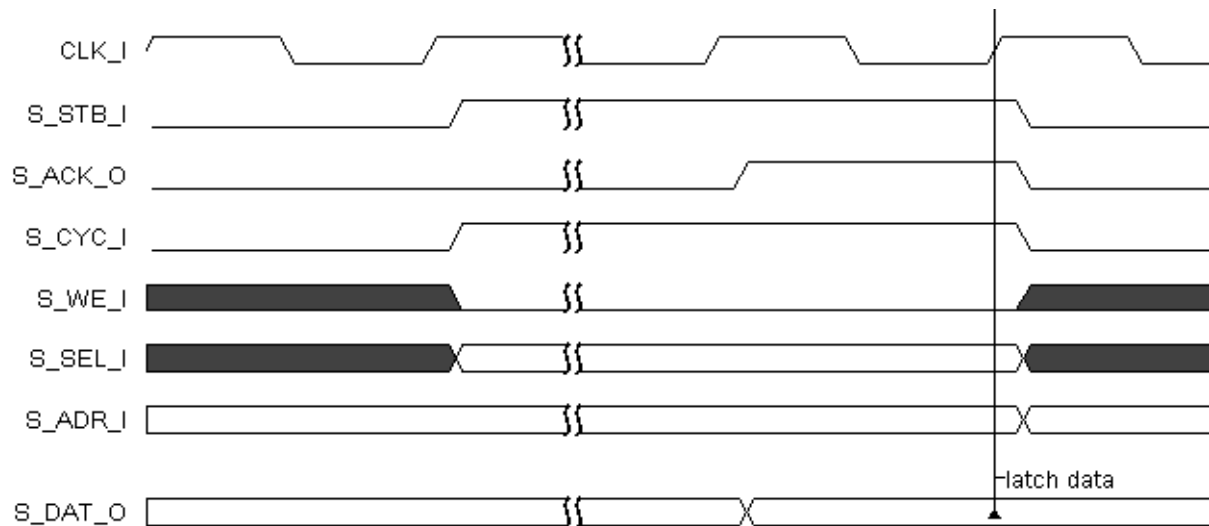


Figure 7 shows how the DMA controller's master ports read the data from the source address.

Figure 7: DMA Controller Reading Data from Source Address



Figure 8 shows how the DMA controller's master ports write the data to the destination address.

Figure 8: DMA Controller Writing Data to Destination Address

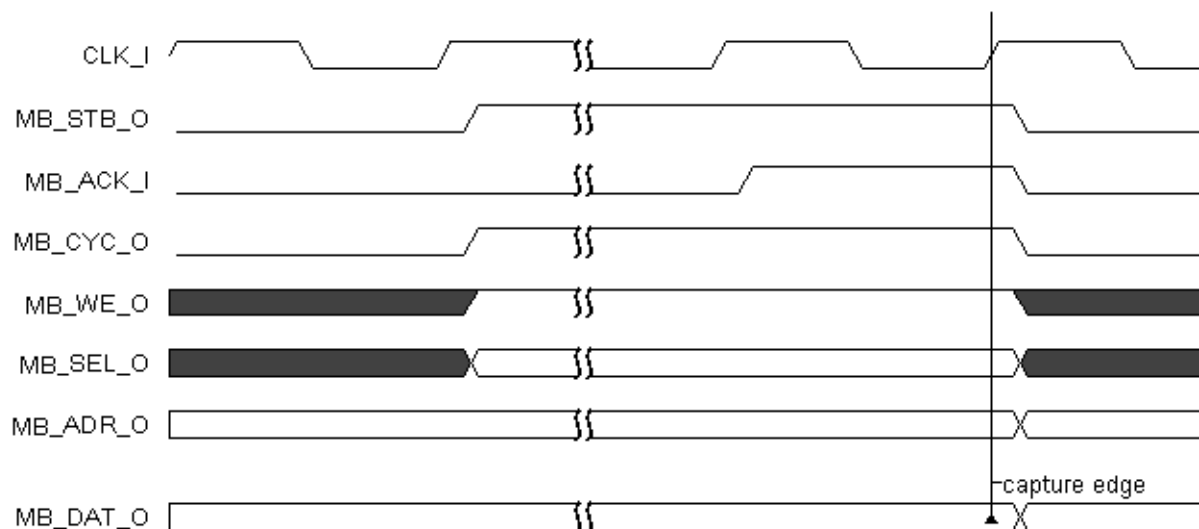


Figure 9 shows how the DMA controller’s master ports copy data from one slave device to another.

Figure 9: DMA Controller Reading Data from Slave A Device and Writing Data to Slave B Device

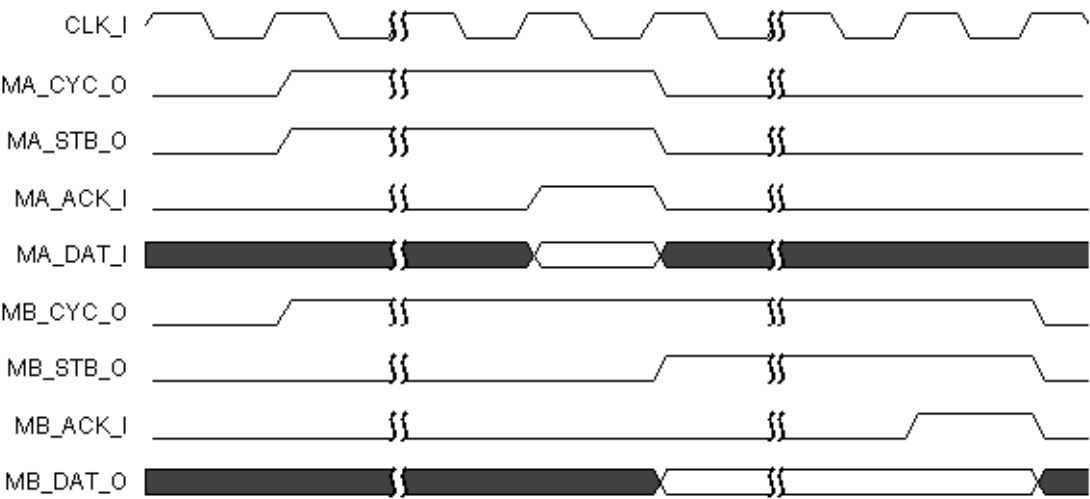


Figure 10 shows the port names and timing diagram of the DMA memory controller in single data transfer mode.

Figure 10: DMA Controller Timing Diagram for Single Data Transfer

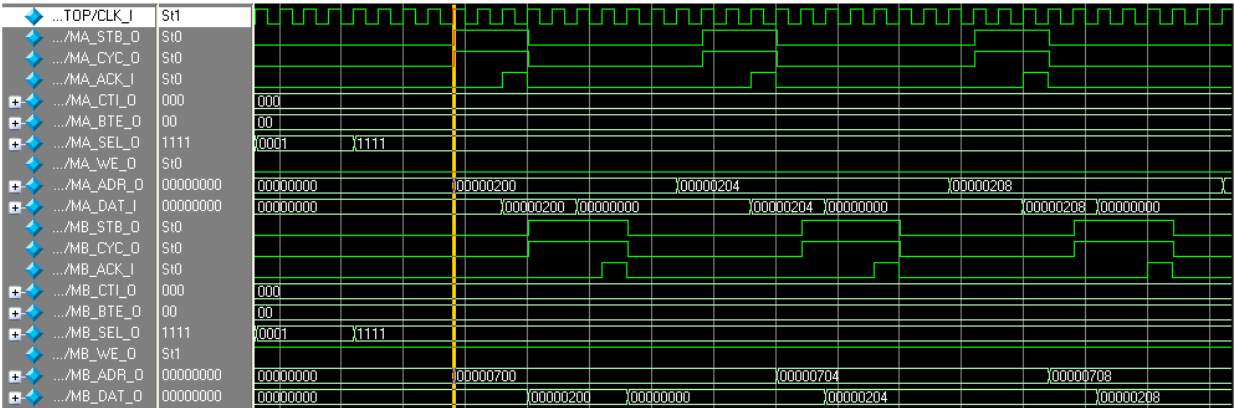
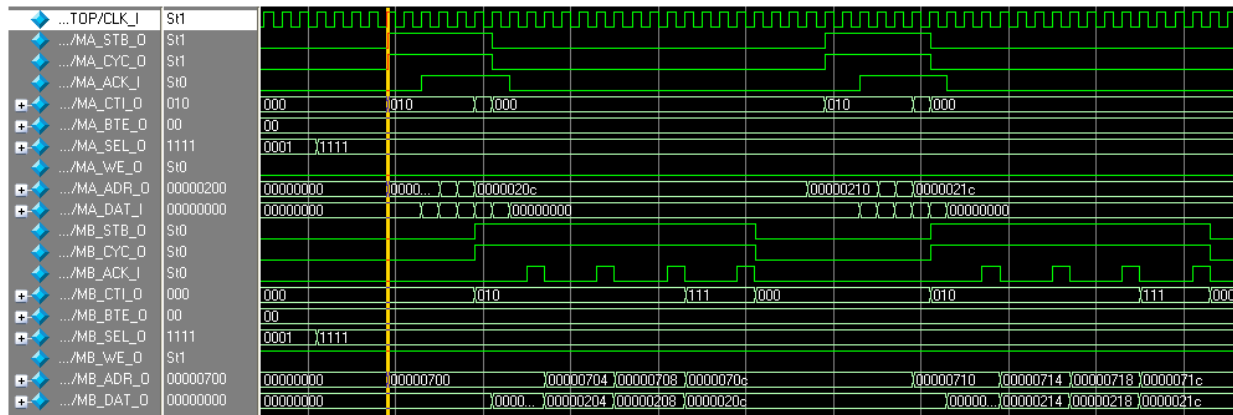


Figure 11 shows the port names and timing diagram of the DMA memory controller in burst data transfer mode.

Figure 11: DMA Controller Timing Diagram for Burst Data Transfer



EBR Resource Utilization

The number of EBRs in the LatticeMico32 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.

Software Support

This section describes the software support provided for the LatticeMico32 DMA controller.

The support 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.

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.

Usage Model

The LatticeMico32 DMA controller has two master ports, each of which can be connected to any address region.

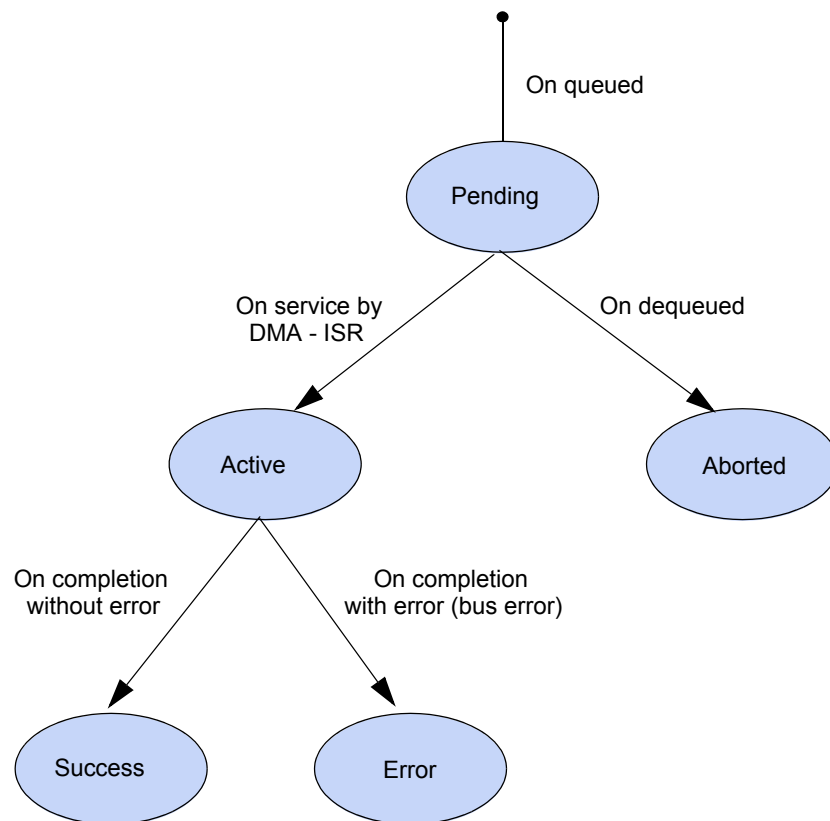
The programmer must set the DMA parameters, including source and destination addresses, that are valid for a given DMA instance.

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 20. 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-served 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 12.

Figure 12: 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 dequeue the descriptor, the state of the descriptor is marked as ABORTED and is removed from the list of pending DMA descriptors.

Arbitration Schemes

When you generate a platform in MSB, you can select the bus arbitration scheme.

Shared-Bus Arbitration If the system is generated with a shared-bus arbiter, all master ports use the same bus to access the slave ports, even though different master ports try to access different slave ports. For example, the DMA might want to access an on-chip memory instance, but the CPU might want to access an SRAM instance. Although the master ports attempt to access different slave ports, they go through the shared bus arbiter, which allocates access on a highest-priority-first basis. However, if a lower-priority master port is granted access to the bus before a higher-priority master port requests the bus, the lower-priority master port is allowed to retain access to the bus until the WISHBONE transaction is complete. Once the arbiter detects this transaction, it releases the bus from the lower-priority master port and gives it to the higher-priority master port.

As an example, assume that a platform's LatticeMico32 microprocessor is configured to have its data and instruction master ports as the highest-priority master ports. The DMA master ports are configured to have priorities lower than the processor master ports. The DMA controller acknowledges a transaction at the end of a read cycle and at the end of a write cycle. If the DMA controller reads from a device such as a UART receive port, it does not yield the bus until the UART provides data or the UART generates a retry or an error signal. Until the DMA controller gives up the shared bus, the microprocessor has no access to the bus.

As this example demonstrates, the shared-bus arbitration must be taken into account when you incorporate a DMA device or multi-master devices into the platform.

Slave-Side Arbitration In the slave-side arbitration scheme, each multi-master slave has its own arbiter. Arbitration between different masters occurs at the slave side. This scheme eliminates the drawback of shared-bus arbitration—that is, a single master having access to the bus, even though multiple masters want to access different slaves—but it uses more area. Figure 2 on page 3 demonstrates how multiple arbiters are instantiated in the platform.

The difference between the slave-side fixed and slave-side round-robin arbitration schemes is how the arbiter grants the masters access to the bus.

In the slave-side fixed scheme, the master having the highest priority has more access to the bus. In the slave-side round-robin arbitration scheme, once a master finishes its transfer, the next master obtains access to the bus in round-robin fashion if that master requests access to the bus.

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 13, 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 13: 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 10 describes the parameters of the DMA device context structure shown in Figure 13.

Table 10: 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 <i>Software Developer User's Guide</i> for a description of the DeviceReg_t data type.
flags	unsigned int	Used internally by the device driver

Table 10: DMA Device Context Structure Parameters

Parameter	Data Type	Description
pCurr	void *	Used internally for tracking active DMA descriptors
pHead	void *	Used internally for tracking queued DMA descriptors
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 the DMA 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 14 contains information for a single DMA operation. It is key to all device-driver-implemented DMA functions, as described in “Functions” on page 22.

Figure 14: 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 11.

Table 11: 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

Table 11: DMA Descriptor Structure (Continued)

Parameter	Data Type	Description
length	unsigned int	<p>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.</p> <p>Note: The device driver code changes the length value before writing it to the DMA controller length register.</p> <p>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.</p> <p>The device-driver code performs the necessary arithmetic to make sure the correct number of transactions are performed.</p>
type	DMAType_t	<p>Can be an OR of the following enumerated type values:</p> <ul style="list-style-type: none"> ◆ DMA_CONSTANT_SRC_ADDR ◆ DMA_CONSTANT_DST_ADDR ◆ DMA_16BIT_TRANSFER ◆ DMA_32BIT_TRANSFER ◆ DMA_BURST_SIZE_4 ◆ DMA_BURST_SIZE_8 ◆ DMA_BURST_SIZE_16 ◆ DMA_BURST_SIZE_32 ◆ DMA_BURST_ENABLE (Legacy. Replaced by BURST_SIZE_X) ◆ DMA_BURST_SIZE (Legacy. Replaced by BURST_SIZE_X)
state	unsigned int	<p>Used internally to indicate the state of the descriptor. It is one of the following values once it is queued:</p> <ul style="list-style-type: none"> ◆ MICODMA_STATE_SUCCESS ◆ MICODMA_STATE_PENDING ◆ MICODMA_STATE_ACTIVE ◆ MICODMA_STATE_ERROR ◆ MICODMA_STATE_ABORTED

Table 11: DMA Descriptor Structure (Continued)

Parameter	Data Type	Description
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 20.

Functions

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

MicoDMAInit Function

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

This function initializes a LatticeMico32 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 12 describes the parameter in the MicoDMAInit function syntax.

Table 12: 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, DMADesc_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_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 13 describes the parameters in the MicoDMAQueueRequest function syntax.

Table 13: 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 14 shows the values returned by the MicoDMAQueueRequest function.

Table 14: 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 15 describes the parameters in the syntax of the MicoDMADequeueRequest function syntax.

Table 15: 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 16 shows the values returned by the MicoDMADequeueRequest function.

Table 16: 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 <code>ctx</code> or <code>desc</code> 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 17 describes the parameter in the MicoDMAPause function syntax.

Table 17: MicoDMAPause Function Parameter

Parameter	Description
MicoDMActx_t *	Pointer to a DMA context representing a valid DMA instance

Table 18 shows the values returned by the MicoDMAPause function.

Table 18: 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 19 describes the parameter in the MicoDMAResume function syntax.

Table 19: MicoDMAResume Function Parameter

Parameter	Description
MicoDMActx_t *	Pointer to a DMA context representing a valid DMA instance

Table 20 shows the values returned by the MicoDMAResume function.

Table 20: 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 22. 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 15: DMA Controller Software 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;
}

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 */
}
```

Figure 15: DMA Controller Software Example (Continued)

```

/* Queue this DMA descriptor and provide a callback for
 * completion notification */
if (MicoDMAQueueRequest (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;
    }

    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;
    }
}

```

Figure 15: DMA Controller Software Example (Continued)

```
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);
}
```
