MITE RLPM: MITE PCI/PXI Interface and DMA Controller

Chapter 1: General Description

The MITE ASIC is a PCI Bus Master. It is the ASIC that connects the PCI or PXI bus to the rest of the National Instruments PCI or PXI board. The MITE does three things at the register programming level:

1. Register access interface
2. Interrupt interface
3. DMA interface

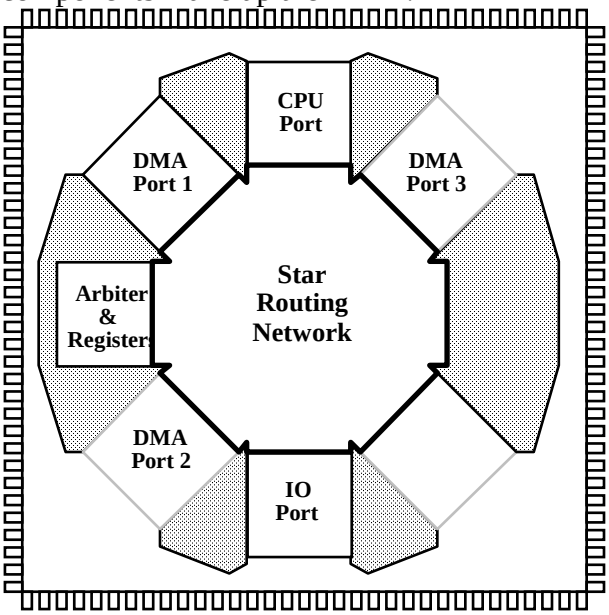
There have been several versions of the MITE including the original MITE, the mMITE-3, mMITE-1 and dustMITE. All versions have similar register maps. The differences between the versions will be noted in this manual in Chapter 3: Register Map and Descriptions.

The MITE is used on a wide variety of National Instruments PCI and PXI boards including GPIB, data acquisition, dynamic signal acquisition, and modular instrumentation. This manual may be useful to programmers using any of these boards.

Chapter 2: Theory of Operation

Functional Overview

The following major components make up the MITE:



CPU Port

The CPU Port connects the MITE to the PCI bus. The CPU port can burst transfer across the PCI bus during DMA operations. A large bank of local registers configure the operating modes of the MITE interface. The MITE has independent transfer paths from the CPU port to local registers and peripherals on the IO bus. These paths guarantee that

cycles to IO port will not affect the ability of the driver to access local resources on the MITE.

DMA Port 1..3

The original MITE, mMITE-3, and dustMITE have three DMA channels 1,2, and 3 which each operate independent from each other, while the mMITE-1 has only one DMA channel. Each DMA channel is capable of transferring data either direction between the rest of the board and system memory. Each DMA channel has its own set of control registers which are described in Chapter 3: Register Map and Descriptions. Each channel contains a FIFO and independent processes for filling and emptying the FIFO. This allows the buses involved in the transfer to operate independently for maximum performance. Data is transferred simultaneously between the ports. The DMAC supports multiple modes of operation including Normal, Continue, Ring Buffer, and Link Chaining.

IO Port

The IO Port connects the MITE to the other chips on the board. It allows additional local registers and peripheral chips such as the DAQ-STC, DAQ-DIO, TNT4882 or TIO to be interfaced to the MITE without connecting to the CPU interface. The IO port also provides a burst mode which allows fast transfer rates to IO devices or memory. In addition the IO port interface supports hardware DMA requests and acknowledges.

Registers

The registers on the MITE configure the other components.

MITE Window Initialization

Window initialization involves the CPU port and the IO port of the MITE. After a power on or reset, PCI BAR0 is enabled, which allows access to the MITE's registers. To enable BAR1 on the card and access registers on other chips, the IO window must be initialized. The IO Device Window Base Size Register must be written with the physical address of BAR1, and told to enable the window. This is discussed further in Chapter 4: Programming.

Once the IO window is initialized, register accesses to BAR1 are redirected by the MITE to chips located on the IO port, such as the DAQ-STC.

MITE Interrupts

The MITE can generate interrupts or forward interrupts from chips attached to its IO port. MITE interrupts can come from internal events, DMA events or events from other chips on the board. Each of these interrupt sources can be masked individually or all sources can be masked together.

MITE DMA

Each MITE DMA channel can transfer either direction between the IO port and the CPU port. DMA channels are configured independently, and can burst across the PCI bus allowing high throughput for applications like data acquisition.

Chapter 3: Register Map and Descriptions

Register Listing by Offset

Register Name	Register Offset
Interrupts	
Local CPU Interrupt Mask 1	0x08
Local CPU Interrupt Status 1	0x0C
Local CPU Interrupt Mask 2	0x10
Local CPU Interrupt Status 2	0x14
Miscellaneous	
IO Device Window Base Size Register	0xC0
Chip Signature Register	0x460
DMA Channel 1	
Channel Operation	0x500
Channel Control	0x504
Transfer Count	0x508
Memory Configuration	0x50C
Memory Address	0x510
Device Configuration	0x514
Device Address	0x518
Base Address	0x528
Base Count	0x52C
Channel Status	0x53C
FIFO Count	0x540
DMA Channel 2	
Channel Operation	0x600
Channel Control	0x604
Transfer Count	0x608
Memory Configuration	0x60C
Memory Address	0x610
Device Configuration	0x614
Device Address	0x618
Base Address	0x628
Base Count	0x62C
Channel Status	0x63C
FIFO Count	0x640
DMA Channel 3	
Channel Operation	0x700
Channel Control	0x704
Transfer Count	0x708
Memory Configuration	0x70C
Memory Address	0x710
Device Configuration	0x714

Device Address	0x718
Base Address	0x728
Base Count	0x72C
Channel Status	0x73C
FIFO Count	0x740

Register Descriptions

Interrupt Registers

Local CPU Interrupt Mask 1 (LCIMR1)

Offset 08 (hex) Read/Write

OOW IE	X	X	X	X	X	IO(1) IE	IO(0) IE
X	X	X	X	X	X	IOW1 IE	IOW0 IE
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X

All bits in LCIMR1 are cleared by MITERST*.

OOW IE- Out of Window Interrupt Enable. Setting this bit enables a interrupt to the CPU bus if a CPU access did not address any window and IOOW in LCR1 is clear. The interrupt clears when OOW in LCISR1 is cleared.

IO(1:0) IE - IO Bus Interrupt Enables. Setting the bit corresponding to a particular IO interrupt enables that interrupt to the CPU port.

IOW1 IE - IO Window 1 Interrupt Enable. Setting this bit will enable a CPU interrupt when an access through IO window 1 generates an interrupt (see IOWCR1).

IOW0 IE - IO Window 0 Interrupt Enable. Setting this bit will enable a CPU interrupt when an access through IO Device Window generates an interrupt (see IODWCR).

X - These bits should always be written with a 0 and will always read back as X.

Local CPU Interrupt Status 1 (LCISR1)

Offset 0C (hex) Read

OOW	X	X	X	X	X	IO(1)	IO(0)
X	X	X	X	X	X	IOW1	IOW0
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X

All bits in LCISR1 are cleared by MITERST*.

OOW - Out of Window Interrupt . This bit is set if a CPU access did not address any window. The bit is cleared on a read. This bit will never be set if IOOW in LCR1 is set.

IO(1:0) - IO Bus Interrupts. The bit corresponding to the IO interrupt is set when the appropriate IOBINT* is asserted.

IOW1 - IO Window 1 Interrupt. This bit is set when an IO window 1 interrupt is asserted.

IOW0 - IO Window 0 Interrupt. This bit is set when an IO Device Window interrupt is asserted.

X - These bits should always be written with a 0 and will always read back as X.

Local CPU Interrupt Mask 2 (LCIMR2)

Offset 10 (hex) Read/Write

Set CPU INT IE	Clr CPU INT IE	X	X	X	X	X	X
X	X	Set DMA3 IE	Clr DMA3 IE	Set DMA2 IE	Clr DMA2 IE	Set DMA1 IE	Clr DMA1 IE
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X

Set CPUINT IE - Set CPU Interrupt Enable. Writing a one to this bit will enable the CPU interrupt pin. This bit will read as 1 if CPU interrupt is enabled and 0 if CPU interrupt is disabled.

Clear CPUINT IE -Clear CPU Interrupt Enable. Writing a one to this bit will disable the CPU interrupt pin. This bit will read as 1 if CPU interrupt is disabled and 0 if CPU interrupt is enabled.

Set DMA3 IE -Set DMA Controller 3 Interrupt Enable. Writing a 1 to this bit enables DMA channel 3 to interrupt the CPU port. This bit will read as 1 if DMA3 interrupt is enabled and 0 if DMA3 interrupt is disabled.

Clear DMA3 IE -Clear DMA Controller 3 Interrupt Enable. Writing a 1 to this bit disables DMA channel 3 to interrupt the CPU port. This bit will read as 1 if DMA3 interrupt is disabled and 0 if DMA3 interrupt is enabled.

Set DMA2 IE -Set DMA Controller 2 Interrupt Enable. Writing a 1 to this bit enables DMA channel 2 to interrupt the CPU port. This bit will read as 1 if DMA2 interrupt is enabled and 0 if DMA2 interrupt is disabled.

Clear DMA2 IE -Clear DMA Controller 2 Interrupt Enable. Writing a 1 to this bit disables DMA channel 2 to interrupt the CPU port. This bit will read as 1 if DMA2 interrupt is disabled and 0 if DMA2 interrupt is enabled.

Set DMA1 IE -Set DMA Controller 1 Interrupt Enable. Writing a 1 to this bit enables DMA channel 1 to interrupt the CPU port. This bit will read as 1 if DMA1 interrupt is enabled and 0 if DMA1 interrupt is disabled.

Clear DMA1 IE -Clear DMA Controller 1 Interrupt Enable. Writing a 1 to this bit disables DMA channel 1 to interrupt the CPU port. This bit will read as 1 if DMA1 interrupt is disabled and 0 if DMA1 interrupt is enabled.

X - These bits should always be written with a 0 and will always read back as X.

Local CPU Interrupt Status 2 (LCISR2)

Offset 14 (hex) Read

CPUINT	X	X	X	X	X	X	X
X	X	DMA3	X	DMA2	X	DMA1	X
Pending	X	X	X	X	X	X	X
X	X	X	X	X	miniMITE Src[2]	miniMITE Src[1]	miniMITE Src[0]

CPUINT - CPU Interrupt. This bit is set if the miniMITE is currently asserting a CPU interrupt (i.e. INTA* is asserted).

DMA3 - DMA Controller 3 Interrupt . This bit is set when an interrupt from DMA channel 3 is true.

DMA2 - DMA Controller 2 Interrupt . This bit is set when an interrupt from DMA channel 2 is true.

DMA1 - DMA Controller 1 Interrupt . This bit is set when an interrupt from DMA channel 1 is true.

Pending - Pending. This bit is set when there is a Window, DMA, or external IO bus interrupt pending.

miniMITESrc(2:0) miniMITE Interrupt Source. These bits represent the highest priority IRQ pending:

miniMITESrc(2:0)	IRQ
101	Window IRQ
100	IOBINT(1)
010	DMA IRQ
001	IOBINT(0)
000	none

X - These bits will always read back as X.

Miscellaneous Registers

IO Device Window Base Size Register (IODWBSR)

Offset C0 (hex) Read/Write

BA31	BA30	BA29	BA28	BA27	BA26	BA25	BA24
BA23	BA22	BA21	BA20	BA19	BA18	BA17	BA16
BA15	BA14	BA13	BA12	X	X	X	X
WENAB	X	X	X	X	X	X	X

The IODWBSR determines the location of the IO Device window in the CPU address space. This window is used to access the DEVICE* address space and BOOTROM* address space on the IO port.

BA[31:13] - Base Address. The base address of the 8k Device/BootROM window in CPU address space. The number of bits compared is determined by the GSIZE bits in LCR1 and the WSIZE bit. These bits are only valid when WENAB=1 and WSIZE=0. These bits are cleared by MITERST*.

BA[12] - Device Base Address. This address bit along with BA[31:13] points to the base address of the 4k window for the Device Region. This bit is used when the window is enabled to the small size (WENAB = 1 and WSIZE =0) and the CA[31:13] matches BA[31:13]. If all of these conditions are met and CA[12] matches BA[12], the CPU access will map to the IO bus with DEVICE* asserted. If the previous conditions are met, but CA[12] does not match BA[12], the CPU access will map to the IO bus with BOOTROM* asserted. These bits are cleared by MITERST*.

WENAB - Window Enable. If this bit is set, the Window is enabled. If this bit is clear the window is disabled.

X - These bits should always be written with a 0 and will always read back as X.

Chip Signature Register (CSIGR)

Offset 460 (hex) from CPU port Read Only

IOWINS(2)	IOWINS(1)	IOWINS(0)	WINS(4)	WINS(3)	WINS(2)	WINS1)	WINS(0)
X	WPDEP(2)	WPDEP(1)	WPDEP(0)	DMAC(3)	DMAC(2)	DMAC(1)	DMAC(0)
0	0	IMODE(1)	IMODE(0)	0	0	MMODE1	MMODE0

TYPE3	TYPE2	TYPE1	TYPE0	VERS3	VERS2	VERS1	VERS0

The original MITE signature is 0x2832--01. miniMITE signature is 0x201-3211

IOWINS(2:0) - Number of I/O Windows. This bit field contains the number of I/O windows the MITE has.

WPDEP[2:0] - Write Post FIFO Depth. These bits reveal the depth of the Write Post FIFOs at the IO ports. The following table illustrates the depths associated with each pattern:

WPDEP[2:0]	Depth in Words
000	0
001	1
010	2
011	4
100	8
101	16
110	32
111	64

DMAC(3:0) - Number of DMA Channels. This bit field contains the number of DMA Channels the MITE contains.

IMODE[1:0] - CPU Port Interface Mode. IMODE is 11 indicating PCI mode.

MMODE[1:0] - MITE Mode. MMODE is 01 indicating miniMITE.

TYPE[3:0] - Type Register. This field will diferentiate between different types of MITE derivatives. The original MITE is 0. The miniMITE is 1.

VERS[3:0] - Version Register. The version register. Revision A is 0001.

DMA Channel Registers

Channel Operation Register (CHOR)

Offset 0 (hex) + 100*(DMA Channel#-1) Read/Write

DMA RESET	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0

CLR DONE	CLRRB	0	FRESET	ABORT	STOP	CONT	START

DMA RESET - Reset DMA Channel Bit. Writing a one to this bit will reset the DMA Channel unconditionally. The bit is automatically cleared.

CLR DONE - Clear Done Status Bit. Writing a one to this bit will clear the DONE status bit. The DONE status bit is also automatically cleared when a new operation is started.

CLRRB - Clear Ring Buffer Complete Bit. Writing a one to this bit will clear the CONTS/RB bit in the Channel Status Register when in Ring Buffer Mode. The bit is automatically cleared.

FRESET - FIFO Reset. Writing a one to this bit clears the FIFO. The bit is automatically cleared.

ABORT - Abort DMA Operation. When this bit is written with a one, the current DMA will stop after the completion of any transfer started before the bit was set. Any bytes in the FIFO will be lost. This bit clears when START is set.

STOP - Stop DMA. When this bit is written with a one, the current DMA is stopped after the FIFO has been allowed to empty. This bit clears when START is set.

CONT - Continue Operation. This bit is used when continue mode is selected. Setting this bit indicates the Base registers have been loaded for a continue operation. At the end of the current transfer, the Base registers are copied into the transfer registers and the CONT bit is cleared.

START - Start DMA Operation. A DMA transfer is started by writing this bit with a one after programming the appropriate address, count, configuration, and control registers.

Channel Control Register (CHCR)

Offset 4 (hex) + 100*(DMA Channel#-1) Read/Write

SET DMA IE	CLR DMA IE	0	0	0	0	SET DONE IE	CLR DONE IE
0	0	0	0	0	0	SET CONT/RB IE	CLR CONT/RB IE
0	BURSTEN	0	0	0	0	0	0
0	0	0	0	DIR	XMODE2	XMODE1	XMODE0

SET DMA IE - Set DMA Interrupt Enable. Setting this bit will enable the DMAC interrupt.

CLR DMA IE - Clear DMA Interrupt Enable. Setting this bit will disable the DMAC interrupt.

SET DONE IE - Set Done Interrupt Enable. Setting this bit will enable interrupts on the assertion of DONE.

CLR DONE IE - Clear Done Interrupt Enable. Setting this bit will disable interrupts on the assertion of DONE.

SET CONT/RB IE - Set Continue/Ring Buffer Interrupt. Setting this bit will enable interrupts when the CONT bit is cleared in Continue Mode. In Ring Buffer Mode it will enable interrupts on the completion of a Ring Buffer transfer.

CLR CONT/RB IE - Clear Continue/Ring Buffer Interrupt. Setting this bit will disable interrupts when the CONT bit is cleared in Continue Mode. In Ring Buffer Mode it will disable interrupts on the completion of a Ring Buffer transfer.

DIR - Transfer Direction.
0 - Memory to Device
1 - Device to Memory

XMODE[2:0] - Transfer Mode Select. Determines the mode of operation for the DMAC Channel.
000 - Normal
001 - Continue
010 - Ring Buffer

Channel Status Register (CHSR)

Offset 3C (hex) + 100*(DMA Channel#-1) Read Only

INT	X	X	X	X	X	DONE	X
MRDY	X	DRDY	X	X	X	CONTS/ RB	X
ERROR	SABORT	X	STOPS	OPERR1	OPERR0	XFERR	X
DRQB	DRQA	X	X	MERR1	MERR0	DERR1	DERR0

INT - Interrupt. The DMAC is interrupting because the interrupt condition is true and enabled.

DONE - DMA Done. This bit clears when the DMA is started and sets when the transfer is completed normally or by an error.

MRDY - Memory Ready. The MRDY and DRDY flags are used when using programmed I/O for transfers. The MRDY bit is set when the memory state machine is requesting to transfer. The MRDY bit takes into account the settings for the trip points and FIFO disable modes. When doing PIO, the MRDY is the signal to read or write the FIFO depending on the transfer direction.

DRDY - Device Ready. This bit is functionally identical to the MRDY bit except that it monitors the device state machine.

CONTS/RB - Continue/Ring Buffer Status. This bit is set by the channel operation register(CHOR) and clears when the BAR and BCR are transferred to the MAR and TCR to start the continue operation. In Ring Buffer Mode this bit is set when a Ring Buffer Transfer has completed, this bit is cleared by writting the CLRRB bit in the CHOR.

ERROR- Error Occurred. Indicates the transfer completed due to an error. The other bits indicate the type of error.

SABORT - Software Abort. The ABORT bit was set in the CHOR.

STOPS - Stopped Status. The STOP bit of the CHOR was set. Note: STOPS is not status that the DMAC has stopped but that the STOP bit was written. To get status that the DMAC has STOPPED enable the DONE IE, then write STOP in the CHOR to get an interrupt that the DMAC has actually stopped.

OPERR[1:0] - Operation Error Code. An illegal FIFO operation such as reading an empty FIFO or writing a full FIFO occurred. OPERR is just a status bit, it will not stop the DMAC from finishing a transfer.
-00 No Error
-01 FIFO Error
-10 Reserved
-11 Reserved

XFERR - Transfer Error. One or more of the transfer processes terminated with an error. Refer to the LERR,MERR, and DERR bit to determine the type.

DRQB - DRQB status. The state of the external DRQ1 signal for channels 0 and 1. It reflects the status of DRQ3 for channels 3 and 4.

DRQA - DRQA status. The state of the external DRQ0 signal for channels 0 and 1. It reflects the status of DRQ2 for channels 3 and 4.

MERR[1:0] - Memory Transfer Error . These bits indicate the type of error which stopped the memory transfer process.
00 - No Error
01 - Bus Error
10 - Retry Limit Exceeded
11 - Other Transfer Error (Parity for MXI)

DERR[1:0] - Device Transfer Error . These bits indicate the type of error which stopped the device transfer process.
00 - No Error
01 - Bus Error
10 - Retry Limit Exceeded
11 - Other Transfer Error (Parity for MXI)

Counter Register (BCR,TCR)

Offset 2C,8 (hex) + 100*(DMA Channel#-1) Read/Write

C31	C30	C29	C28	C27	C26	C25	C24
C23	C22	C21	C20	C19	C18	C17	C16
C15	C14	C13	C12	C11	C10	C9	C8
C7	C6	C5	C4	C3	C2	C1	C0

TCR [31:0] - Transfer Count Register. The TCR represents the number of bytes remaining to be loaded into the FIFO. It is programmed with the length of the transfer in bytes, and is decremented by 1,2, or 4 as data enters the FIFO.

BCR[31:0] - Base Count Register. The contents of the BCR are loaded into the TCR when chaining or continue mode is used. When in continue mode, the user programs the BCR with the next transfer count before setting the continue bit

FIFO Count Register (FCR)

Offset 40 (hex) + 100*(DMA Channel#-1) Read Only

X	X	X	X	X	X	X	X
ECR7	ECR6	ECR5	ECR4	ECR3	ECR2	ECR1	ECR0
X	X	X	X	X	X	X	X
FCR7	FCR6	FCR5	FCR4	FCR3	FCR2	FCR1	FCR0

FCR [7:0] - The FCR indicates the number of bytes remaining in the FIFO. A transfer is complete when both the TCR and FCR reach zero.

ECR[7:0] - The ECR bits indicate the number of empty locations (bytes) in the FIFO.

Memory Configuration Register (MCR)

Offset C (hex) + 100*(DMA Channel#-1) Read/Write

0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0
0	0	0	0	0	1	PSIZE1	PSIZE0
0	0	0	0	0	0	0	0

PSIZE[1:0] - Port Transfer Size. The size of the port for the transfer. The actual transfer size may be smaller when aligning addresses and draining the FIFO.
01 - 8 Bit 10 - 16 Bit 11 - 32 Bit

Device Configuration Register (DCR)

Offset 14 (hex) + 100*(DMA Channel#-1) Read/Write

0	0	0	0	0	0	0	0
0	0	0	0	0	REQS2	REQS1	REQS0
0	X	0	0	0	1	PSIZE1	PSIZE0
0	1	X	X	X	0	X	0

REQS[2:0] - Request Source. This selects the type of DMA request for the process.
000 - Internal Maximum Rate
010 - Disable Process. Programmed I/O to the FIFO
100 - Hardware Request 1, DRQ1 line from the IO bus
101 - Hardware Request 2, DRQ2 line from the IO bus
110 - Hardware Request 3, DRQ3 line from the IO bus
111 - Hardware Request 4, DRQ4 line from the IO bus

PSIZE[1:0] - Port Transfer Size. The size of the port for the transfer. The actual transfer size may be smaller when aligning addresses and draining the FIFO.
01 - 8 Bit 10 - 16 Bit 11 - 32 Bit

Address Register (BAR,MAR,DAR)

Offset 20,28,10,18,30 (hex) + 100*(DMA Channel#-1) Read/Write

A31	A30	A29	A28	A27	A26	A25	A24
A23	A22	A21	A20	A19	A18	A17	A16
A15	A14	A13	A12	A11	A10	A09	A08
A07	A06	A05	A04	A03	A02	A01	A00

BAR[31:0] - Base Address Register. The contents of the BAR are loaded into the MAR when chaining or continue mode is used. When in continue mode, the user programs the BAR with the next memory address before setting the continue bit

MAR[31:0] - Memory Address Register. The MAR is loaded with the address to be used by the memory process. It is modified by each successful transfer according to the ASEQ bits of the configuration register. In the case of a transfer that received an error response, the contents would be the address which generated the error. If the address is not aligned to the programmed size boundary, the DMAC will do smaller transfer until alignment occurs.

DAR[31:0] - Device Address Register. The DAR is loaded with the address to be used by the device process. It is modified by each successful transfer according to the ASEQ bits of the configuration register. In the case of a transfer that received an error response, the contents would be the address which generated the error. If the address is not aligned to the programmed size boundary, the DMAC will do smaller transfer until alignment occurs.

Chapter 4: Programming

This chapter contains programming instructions for controlling the MITE. These examples read and write to registers on the MITE to enable IO to the rest of the PCI board, to use the MITE to detect and mask interrupts, and to perform DMA transfers between the rest of the PCI/PXI board and system memory.

Initializing the IO Window

This example shows how to enable an IO window on the MITE, which allows register accesses to other chips on the PCI/PXI board. After power up or system reset, the MITE

is accessible through BAR0 of the PCI board. However the rest of the board must be enabled, before it can be accessed in the BAR1 address space.

1) Get the physical address of BAR1 on the PCI card.

    physicalBar1 = get_physical_address_of_PCI_Board(BAR1)

2) Write address and enable to IO Device Window Base Size Register

    Bar0.write32(0xC0, (physicalBar1 & 0xffffff00L) | 0x80);

3) Accesses to BAR1 are now enabled

Interrupts

This example shows how to mask interrupts from the other chips on the PCI/PXI board. Masking interrupts from other chips allows a developer to create an interrupt service routine (ISR) that can work with a wide variety of National Instruments boards.

1) Initialize the IO Window. You will need to write to the other chips on the PCI board to configure them to generate interrupts.

2) Register an ISR with the operating system.

    a. The ISR can detect an interrupt from the PCI/PXI board by reading the CPUINT bit in the Local CPU Interrupt Status 2 register

    If (mite->localCpuIntStatus2.readCpuInt() == 1)
        PCI board is generating an interrupt
    Else
        PCI board is not generating an interrupt

    b. The ISR can stop the interrupt by writing to the Clr CPU Int IE bit in the Local CPU Interrupt Mask 2 register. Once a board specific interrupt handler has identified the cause of the interrupt and the rest of the board is no longer asserting an interrupt to the MITE, the board specific interrupt handler should re-enable MITE interrupts as described in step 3.

    mite->localCpuIntMask2.writeClrCpuIntIe(1)

3) Enable Mite Interrupts by writing to the Local CPU Interrupt Mask 1 register.

    mite->localCpuIntMask2.writeSetCpuIntIe(1)

Normal Mode DMA

    //Normal mode transfer

```
mite->ChannelControl.setXMode(0);
mite->ChannelControl.setBurstEnable(1);
mite->ChannelControl.setDir(1);
mite->ChannelControl.flush();

mite->MemoryConfig.setRegister(0x00E00400);
mite->MemoryConfig.setPortSize(2); //16 bits
mite->MemoryConfig.flush();

mite->DeviceConfig.setRegister(0x00000440);
mite->DeviceConfig.setReqSource(4+drq);
mite->DeviceConfig.setPortSize(2); //16 bits
mite->DeviceConfig.flush();

mite->MemoryAddress.writeRegister(buf.physAddr);
mite->TransferCount.writeRegister(buf.size);

//address of 16 bit fifo on the MITE
//Actually, the DAR is ignored.  The FIFO
//is only connected by the DRQ line and bus
//we just use this register for counting
mite->DeviceAddress.writeRegister(0);

mite->ChannelOperation.writeStart(1);
```

Ring Buffer Mode DMA

```
//Ring Buffer mode transfer
mite->ChannelControl.setXMode(2);
mite->ChannelControl.setBurstEnable(1);
mite->ChannelControl.setDir(1);
mite->ChannelControl.flush();

mite->MemoryConfig.setRegister(0x00E00400);
mite->MemoryConfig.setPortSize(2); //16 bits
mite->MemoryConfig.flush();

mite->DeviceConfig.setRegister(0x00000440);
mite->DeviceConfig.setPortSize(2); //16 bits
mite->DeviceConfig.setReqSource(4+drq);
mite->DeviceConfig.flush();

mite->MemoryAddress.writeRegister(buffer.physAddr);
mite->TransferCount.writeRegister(buffer.size);

//The Base Address and Count are loaded into MemoryAddress
//and TransferCount every time the buffer fills completely
```

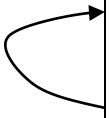
```
mite->BaseAddress.writeRegister(buffer.physAddr);
mite->BaseCount.writeRegister(buffer.size);



//address of 16 bit fifo on the MITE
//Actually, the DAR is ignored.  The FIFO
//is only connected by the DRQ line and bus
//we just use this register for counting
mite->DeviceAddress.writeRegister(0);

mite->TransferCount.writeRegister(buf.size);

mite->ChannelOperation.writeStart(1);
```

Appendix A: Overview of DMA

DMA Transfers Data

There are three ways to transfer data between a DAQ board and an application. They are polled IO, Interrupts, and Direct Memory Access.

Polled IO is the simplest. This is direct register reads and writes. To read data from a FIFO on a DAQ board, a polling IO program would read a status register to see if data is available in the FIFO. Once the status register indicates data is present, the program would read one sample from the FIFO, then wait for the status register to indicate more data. This is the method used in most of the register level programming examples released by National Instruments.



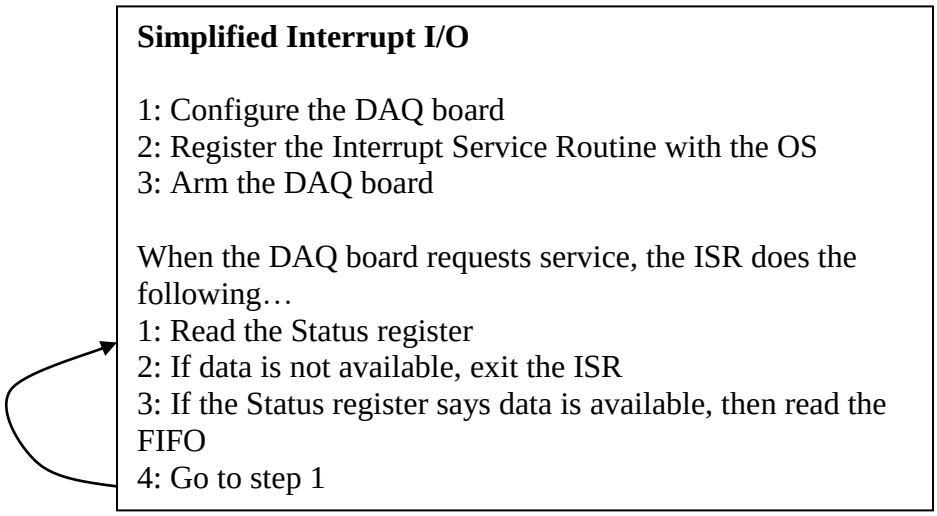
Simplified Polled I/O

1: Configure the DAQ board
2: Arm the DAQ board
3: Read a Status register
4: If the Status register says data is available, then read the FIFO
5: Go to step 3

Interrupt based IO is similar. National Instruments DAQ boards can be configured to request service, usually when a FIFO is almost full (for input) or almost empty (for output). When a DAQ board requests service, it generates an interrupt. Then, the operating system calls an interrupt handler which reads the FIFO (or writes to the FIFO, for output) until the DAQ board no longer needs service.

Interrupts are faster than polled IO because the software only polls the DAQ board after the DAQ board has requested service.

Operating systems usually put severe restrictions on the interrupt handler, or ISR, such as restricting memory allocation and limiting the functions which can be called by the ISR. Also, the ISR must have a way to communicate with the rest of the program to indicate how much of the application's data buffer is full, if the data transfer is finished, and if an error occurred.



DMA transfers are much faster. National Instruments PCI and PXI cards Bus Master the PCI/PXI bus and can transfer data directly into the application's memory. The PCI/PXI bus is designed to make DMA transfers as fast as possible, sometimes 30 times faster than either polled or interrupt based transfers. National Instruments PCI/PXI DAQ boards have the mMITE ASIC which has either one or three DMA channels, depending on the DAQ board. If the application tells the mMITE where to transfer data in system memory, the mMITE will automatically read the FIFO and write the data to the specified buffer. The application just checks to see when the transfer finishes. Instead of reading a status register to detect if the DMA transfer has finished, the mMITE can generate an interrupt at the end of the transfer.

Simplified DMA I/O

1: Configure the DAQ board
2: Allocate memory for the DMA buffer
3: Configure the DMA controller (the MITE)
4: Arm the DMA controller (the MITE)
5: Arm the DAQ board
6: Wait until the DAQ transfer has finished
7: Free the DMA buffer, only after the DMA transfer is finished

Memory Issues

The DMA controller usually accesses the same memory with different addresses than the application software. This is because most operating systems use virtual memory. In these operating systems, applications use virtual addresses instead of directly accessing physical memory. The operating system translates virtual addresses into physical addresses during run time. This allows the OS to move blocks of virtual addresses around in physical memory and sometimes to cache the block to a swap file.

Memory for a DMA buffer needs to:

1) Be Contiguous—Contiguous virtual memory addresses often are not contiguous in physical memory. The DMA controller will access memory sequentially, which means that the buffer needs to be contiguous in physical memory.
2) Be Pagelocked—The OS should not move the buffer in physical memory. Since the DMA controller references memory only by its physical address, the OS should not move the virtual memory to a new physical address. Further, the OS should not cache the buffer to a swap file. If either of these happen, the DMA controller would continue to write to the same physical address. This would overwrite whatever the OS moved into that location, crashing the system.
3) Have a Physical Address—The application needs a way to get the physical address of the buffer.

Most operating systems, such as Windows 95,98,ME,NT,2000,XP, QNX, and Linux use virtual memory. Some operating systems do not use virtual memory. These include MS-DOS, VxWorks, and Pharlap. If your operating system does not use virtual memory, then every buffer probably meets the requirements for a DMA buffer.

Using DMA Transfers

1: Configure the DAQ board

This will be very similar to the configuration used for interrupt or polled IO transfers. You will probably need to tell the other ASICs on the DAQ board which DMA Request line (DRQ) to use.

2: Allocate memory for the DMA buffer

Allocate a physically contiguous buffer. Pagelock it. Then get both its physical address and virtual address. Be sure to check for an error. If the buffer can not be allocated, then MITE DMA will likely crash the whole system. For this simple mode of DMA (Normal Mode), the DMA buffer should be large enough to hold all the data transferred.

buf = dmaBufferAlloc(BufferSize,error)

3: Configure the DMA controller (the mMITE)

Configure the mMITE for a DMA input or output transfer. The DMA channel will need to use the same DRQ as the FIFO it is reading from or to.

dmaConfigMiteBlockInput(buffer, DRQ line, mMITE channel)

4: Arm the DMA controller (the mMITE)

This is usually done in dmaConfigMiteBlockInput( ). Even though the mMITE's DMA channel has been started, no data will be transferred until the FIFO is ready.

5: Arm the DAQ board

Arm the DAQ board the same way it is armed for polled IO transfers or interrupt transfers. Since the DMA channel has already been configured and started, data can be transferred without any additional software intervention.

6: Wait until the DAQ transfer has finished

The software can do other processing while the DMA transfer is handled completely in hardware. The software can poll the mMITE to see how many bytes have been transferred and read those bytes from the buffer as they become available.

Bytes = dmaReadBytesInput(DMA Channel)

7: Free the DMA buffer, only after the DMA transfer is finished

Once the data acquisition finishes, stop the mMITE DMA channel which was transferring its data. Reset the mMITE DMA channel. Then free the DMA buffer in system memory. Do not free the DMA buffer until the mMITE has stopped its transfer.

```
dmaResetMite(DMA Channel)
dmaBufferFree(DMA Buffer)
```

DMA and Memory RLP Functions

These functions are the API used in the RLP example programs for National Instruments DAQ boards. They should function on all NI PCI and PXI DAQ boards on DMA channel 0. Some boards also have two additional DMA channels: 1 and 2.

Memory API

dmaBuffer dmaBufferAlloc(int size)

Allocate a buffer of size bytes. This buffer will be the destination for the DMA transfer.

void dmaBufferFree(dmaBuffer buf)

Free the buffer and clean up everything that was done in dmaBufferAlloc().

DMA API

void dmaConfigMiteBlockInput(tMITE *mite, dmaBuffer buf, int drq, int MiteChan)

Configure the MITE to begin a DMA transfer into the buffer provided. Even though the MITE is ready to transfer immediately after this function, no data will be transferred until the DAQ board's FIFOs are ready. The DRQ line specified here must match the DRQ line that the DAQ board has been configured for.

For a Normal mode input DMA transfer:

```
mite->ChannelControl.setXMode(0);
mite->ChannelControl.setBurstEnable(1);
mite->ChannelControl.setDir(1);
mite->ChannelControl.flush();

mite->MemoryConfig.setRegister(0x00E00400);
mite->MemoryConfig.setPortSize(2); //16 bits
mite->MemoryConfig.flush();
```

```
mite->DeviceConfig.setRegister(0x00000440);
mite->DeviceConfig.setReqSource(4+drq);
mite->DeviceConfig.setPortSize(2); //16 bits
mite->DeviceConfig.flush();

mite->MemoryAddress.writeRegister(buf.physAddr);
mite->TransferCount.writeRegister(buf.size);
mite->DeviceAddress.writeRegister(0);
mite->ChannelOperation.writeStart(1);
```

int dmaReadBytesInput(tMITE *mite, int MiteChan)

Return the number of bytes transferred from the DAQ board into system memory.

```
bytesRead = mite->DeviceAddress.readRegister();
bytesRead = bytesRead - mite->FifoCount.readFifoCR();
```

int dmaReadBytesOutput(tMITE *mite, int MiteChan)

Return the number of bytes transferred from system memory into the DAQ board's FIFOs.

```
bytesWritten = mite->DeviceAddress.readRegister();
bytesWritten = bytesWritten + mite->FifoCount.readFifoCR();
```

void dmaResetMite(tMITE *mite, int MiteChan)

Reset a MITE DMA channel. This stops the current transfer and prepares the MITE DMA channel for another dmaConfigMiteBlockInput( ).

```
mite->ChannelOperation.setRegister(0);
mite->ChannelOperation.writeDmaReset(1);

mite->ChannelControl.writeRegister(0);
mite->BaseCount.writeRegister(0);
mite->TransferCount.writeRegister(0);
mite->MemoryConfig.writeRegister(0);
mite->DeviceConfig.writeRegister(0);
mite->BaseAddress.writeRegister(0);
mite->MemoryAddress.writeRegister(0);
mite->DeviceAddress.writeRegister(0);
```

Platform Specific Functions

National Instruments register level programming examples use only two additional platform specific functions to enable DMA on a board. These two functions provide a generic operating system interface to request memory for DMA buffers. You may need to redefine these functions to make them work on your operating system.

dmaBuffer dmaBufferAlloc(int size)

dmaBufferAlloc() allocates a suitable buffer for DMA. It returns a dmaBuffer structure which contains information about the DMA buffer. Size is the request size of the buffer in bytes.

void dmaBufferFree(dmaBuffer buf)

dmaBufferFree() frees a buffer requested by dmaBufferAlloc().

dmaBuffer

```
struct dmaBuffer {

        unsigned long physAddr; //physical address of the buffer
        void * virtAddr;        //virtual address of the buffer
        int index;              //index in buffer used when software updates the buffer
        int size;
        enum {block, scatterGather, invalid} type; //type of buffer
        void * extra;                   //extra data can be stored here
};
```

dmaBuffer is a structure which contains information about a DMA buffer. physAddr is the physical address of the memory buffer. The MITE DMA controller will write to this memory location. virtAddr is a virtual memory address for the same buffer. The application will need to read the data from, or write to the buffer using this virtual address. type describes the type of buffer. Ring Buffer mode and Normal mode DMA use a block buffer, while Scatter Gather modes use a scatter gather buffer. At this time, only Ring Buffer and Normal modes are used in NI register level programming examples.