NATIONAL INSTRUMENTS®
The Software is the Instrument®

Release Notes

NI-VXI Software Enhancements for the VXI/VME-AT4000 Series Kit

Thank you for purchasing the NI-VXI bus interface software from National Instruments. These release notes describe new features of the NI-VXI bus interface software along with enhancements of existing features.

New Features and Terminology

There are four major areas in which new functionality has been added to NI-VXI to exploit features in MXI-2 and the MITE custom ASIC, a proprietary chip used in all MXI-2 boards. These features are as follows:

- Window mapping
- DMA
- Shared memory
- Remote controllers

Window Mapping

The MITE architecture allows much more flexibility in low-level mapping of VXI address spaces. In particular, the CPU-MXI interface of the MITE has dynamically resizable and relocatable windows from CPU to VXI space. The NI-VXI low-level functions have new extensions that reflect this feature. Refer to Chapter 6, *Low-Level VXIbus Access Functions*, in the *NI-VXI Software Reference Manual for C* for more information.

As in earlier versions of NI-VXI, `MapVXIAddress()` checks whether a sharable window already maps to the desired address space and location. If so, it returns a pointer to that window. If the desired space is not already mapped, `MapVXIAddress()` sets up a new MITE window to the VXI address and returns a pointer to the new window.

The success of this address allocation depends on the availability of three factors:

- MITE windows
- CPU address space
- Memory for allocating data structures for the map

MITE Windows

The NI-VXI driver for the AT-MXI-2 allows one CPU window for user application.

CPU Address Space

The AT-MXI-2 can be placed at any address in the upper memory region of the first 1 MB of address space (A000 to FFFF). The operating system or computer architecture may limit which addresses can be assigned to the AT-MXI-2.

Memory for Allocating Data Structures

Memory is needed to set up the necessary page tables, and a large number of segment descriptors may be needed to map the CPU address space.

The `MapVXIAddressSize()` function is the standard mechanism for specifying how large a window the driver should map on a call to `MapVXIAddress()`. The default size of a mapped window is 64 KB.

DMA

The MITE has two DMA channels, which NI-VXI uses to improve the throughput of block transfers to and from the VXI system. The DMA channels can use various high-speed bus protocols, such as the following:

- MXI block
- MXI synchronous

The DMA channels can transfer data between a VXI device and local memory, or between VXI devices. The DMA channel can handle contiguous or noncontiguous local memory. If it is handling noncontiguous memory, it can perform scatter-gather operations on the noncontiguous memory.

The `VXImove()` function automatically uses appropriate bus protocols and transfer types to efficiently perform the data transfer specified in the function. You can also use some extra configuration options and function parameters to instruct the NI-VXI software to use DMA channels for particular types of operations and to designate what protocols the channel should use. See Chapter 7, *High-Level VXIbus Access Functions*, in the *NI-VXI Software Reference Manual for C* for complete descriptions of `VXImove()` and other high-level functions. However, previously written NI-VXI code will use the DMA capabilities without modification.

Shared Memory

You can share a portion of your system RAM in VXI space, as on previous drivers. In addition, MITE-based boards have SIMM slots for installing onboard RAM. On an AT-MXI-2, this RAM can be shared in a VXI address space. Furthermore, you can divide the VXI space assigned to your device into two halves, sharing the onboard RAM in one half and system RAM in the other. The configuration options, such as byte-ordering, can be different for each half of the region of VXI space you are sharing. For more information, review the options of the AT-MXI-2 Logical Address Configuration Editor in Chapter 6, *NI-VXI Configuration Utility*, of *Getting Started with Your VXI/VME-AT4000 Series and the NI-VXI Software.*

Remote Controllers

Each VXI-MXI-2 has a full MITE chip set onboard. Although there is no CPU connected to it, this MITE chip set gives the VXI-MXI-2 many of the same features as the AT-MXI-2, including onboard memory, DMA channels, TIC-based triggering, and interrupt mapping. This means that a MXI-2 system has at least one full featured bus controller sitting in each VXI chassis, controlled by and communicating with the local controller via the VXIbus and the MXIbus. These MITE-based bus controllers are called *remote controllers*.

In previous versions of NI-VXI, the CPU-MXI-based controller was called an *external controller*, and VXI-MXI boards on the same MXIbus as the external controller were called *extending controllers*. The extending controllers and the external controller together made up one *extended controller*. The presence of the full featured remote controllers has led to a slight change in this model. The CPU-MXI is still an external controller, but instead of having extending controllers, each parent-side VXI-MXI-2 is a separate remote controller. Other VXI-MXI-2 boards are treated simply as extenders that may have additional memory present, but may not perform bus access, trigger, or interrupt services. This definition assigns one remote controller to each chassis, resolving any resource arbitration issues for the system.

Any NI-VXI function that previously accepted a `controller` parameter—whether extended, external, embedded, or local—will now accept a remote controller as well. There is a new option in VXIedit to determine which controller is associated with the value of -1 in the `controller` parameter of an NI-VXI function. This new option indicates whether the -1 value is referring to the AT-MXI-2 (the local controller) or the first remote VXI/VME-MXI-2 controller. For more details, refer to the *Default Controller (LA -1)* section in Chapter 6, *NI-VXI Configuration Utility*, of your getting started manual.

You can also use VXIedit to select which VXI interrupt line the remote controller uses to report remote events to the local controller. This is called the *system IRQ line*. You need to map the system IRQ line back to the local controller to receive remote controller interrupts. This mapping is performed automatically in the parent-side VXI-MXI-2 controllers, but not in other mainframe extenders. You can map interrupts by using the `MapVXIint()` function, which is described in Chapter 13, *VXIbus Extender Functions*, in the *NI-VXI Software Reference Manual for C*, or by using the Interrupt Configuration Editor in VXIedit.

Enhancements to the NI-VXI Software

The NI-VXI software for AT-MXI-2 makes use of many new features available in MXI-2 and on the MITE ASIC. You have two options for controlling these features: through new configuration options in VXIedit, or by extensions to the NI-VXI API. Refer to Chapter 6, *NI-VXI Configuration Utility*, and Chapter 7, *Using the NI-VXI Software*, in your getting started manual for more information on the new configuration options.

NI-VXI API Extensions

The following paragraphs discuss the additional options beyond what was documented in the *NI-VXI Software Reference Manual for C*.

Compatibility

NI-VXI applications that follow the guidelines in the *NI-VXI Software Reference Manual for C* will work with NI-VXI for the AT-MXI-2 with a few exceptions. These exceptions deal primarily with low-level VXIbus access functions.

Applications that use `VXIpeek()` and `VXIpoke()` may need to be recompiled. The `VXIpeek()` and `VXIpoke()` macros in other hardware platforms accessed locations in I/O space to check for retries on the MXIbus. AT-MXI-2 registers are no longer located in I/O space, and the AT-MXI-2 automatically handles these retry situations, so the macros have changed. Previous releases of NI-VXI provided a flag, `MXI1AND2_COMPATIBLE`. Applications compiled with this flag use AT-MXI-2 binary-compatible `VXIpeek()` and `VXIpoke()` macros and will not require recompilation.

Applications that set the context bits directly for use in `SetContext()` may not be compatible with the new format for context. The AT-MXI-2 allows more flexible window mapping, so extra bits have been added to this field to reflect these new features. In general, we recommend that you do not manipulate the context bits.

System Configuration Functions

The *mainframe extender/controller information* field of the `DevInfo` structure (field 17) defines a new bit. Bit 13 reports if the extender is a remote controller. This bit is now significant in any function that uses the *mainframe extender/controller information* field (`GetDevInfo()`, `SetDevInfo()`, `GetDevInfoShort()`, `SetDevInfoShort()`, and `FindDevLA()`). NI-VXI automatically sets bit 13. Changing the value of this bit in a program, even via the documented API calls, is not recommended.

If a remote controller is sharing its onboard RAM, you can find the size, space, and offset of that RAM in the VXI system by calling `GetDevInfo()` for the remote controller's logical address.

Low-Level VXIbus Access Functions

Do not make any assumptions about the size and features of a window returned from `MapVXIAddress()`. Use `GetWindowRange()` to determine the size of a window.

The 32-bit value returned from `GetContext()` and passed to `SetContext()` has a new format.

MapVXIAddressSize

`MapVXIAddressSize()` is the interface for requesting a particular window size for all successive calls to `MapVXIAddress()`.

Syntax: `ret = MapVXIAddressSize(size)`

Action: Sets the default size for the window returned from `MapVXIAddress()`.

Remarks: Input Parameter:

`size`	`uint32`	Size in bytes of window to be mapped

Output Parameters:

none

Return Value:

`ret`	`int16`	Return Status 0 = Successful

Example:

```
/* Set the size of future mappings to 1 MB */
ret = MapVXIAddressSize(0x100000);
```

High-Level VXIbus Access Functions

As previously mentioned, `VXImove()` has new bits defined in `srcParm` and `destParm` for using features of the MITE DMA channel.

- Bit 11 — Use programmed I/O (PIO) instead of DMA. DMA is used by default and setting this bit forces NI-VXI to use PIO instead of DMA. In general, DMA is more efficient than PIO. However, when virtual memory is involved, a DMA channel needs to lock down memory before it can be used (locking is done by the NI-VXI DMA manager automatically). Virtual memory can be enabled under Windows, but is not available under DOS. Because of the amount of time required for buffer locking, it may be more efficient to transfer buffers by PIO.
- Bit 12 — No increment. Do not increment the address. This is used when the source or destination is a FIFO buffer and is valid for any address space, whether local, A16, A24, or A32.
- Bit 13 — Disable MXI block mode. MXI block mode is enabled by default in `VXImove()` because it makes MXI transfers more efficient, and MXI block mode is supported by the VXI/VME-MXI-2. However, if you are communicating with a MXI device that does not support MXI block mode, you need to disable this mode by setting this bit.
- Bit 14 — Disable MXI synchronous mode. MXI synchronous mode is enabled by default in `VXImove()` because it makes MXI transfers more efficient, and MXI synchronous mode is supported by the VXI/VME-MXI-2. However, if you are communicating with a MXI device that does not support MXI synchronous mode, you need to disable this mode by setting this bit.

For best performance, keep the following in mind when using `VXImove()`.

- Make sure your buffers are 16-bit aligned.
- Transfer 16-bit data whenever possible.
- `VXImove()` must lock the user buffer in memory on virtual memory systems, so locking the buffer yourself will optimize `VXImove()`.
- `VXImove()` must build a scatter gather list for the user buffer on paged memory systems, so using a contiguous buffer will optimize `VXImove()`.

`VXImemAlloc()` will return 32-bit aligned, page-locked, contiguous buffers, which work efficiently with `VXImove()`.

Local Resource Access Functions

`VXImemAlloc()` does not allocate onboard RAM on the AT-MXI-2; it only allocates system RAM on the motherboard. If you want to access onboard RAM on the AT-MXI-2, access it as if it were VXI memory—that is, by using high-level or low-level VXIbus access functions. You can use `GetDevInfo()` on the AT-MXI-2 device to determine the VXI address space and VXI address of this onboard RAM.

VXI Interrupt Functions

Every function that takes a controller's logical address as an argument can now accept the local controller or a remote controller's logical address as well.

VXI Trigger Functions

Every function that takes a controller's logical address as an argument can now accept the local controller or a remote controller's logical address as well.

TIC functionality is now available on the local controller and on every remote controller.

System Interrupt Handler Functions

Every function that takes a controller's logical address as an argument can now accept the local controller or a remote controller's logical address as well.