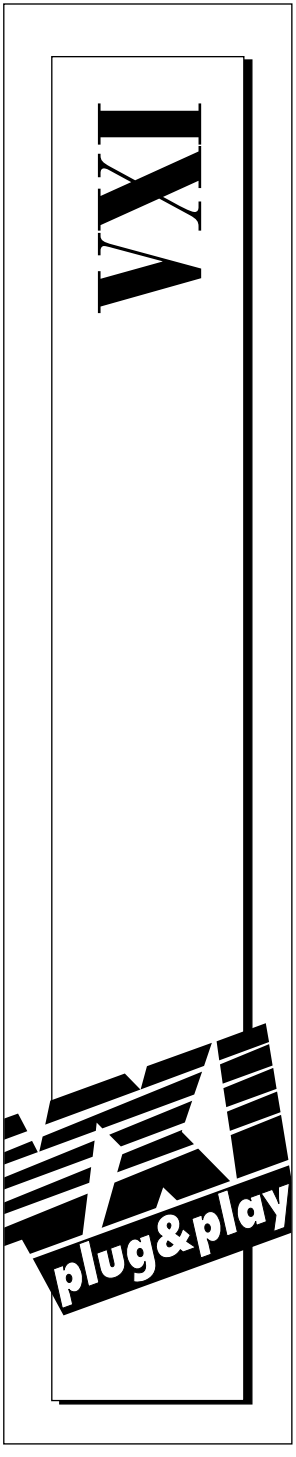


NI-VXI™ User Manual

July 1996 Edition
Part Number 371702A-01

Internet Support

GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabVIEW: lv.support@natinst.com
LabWindows: lw.support@natinst.com
HiQ: hiq.support@natinst.com
VISA: visa.support@natinst.com
Lookout: lookout.support@natinst.com

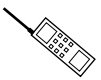
E-mail: info@natinst.com
FTP Site: ftp.natinst.com
Web Address: http://www.natinst.com

Bulletin Board Support

BBS United States: (512) 794-5422 or (800) 327-3077
BBS United Kingdom: 01635 551422
BBS France: 1 48 65 15 59

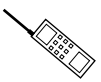
FaxBack Support

(512) 418-1111



Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678



International Offices

Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 90 527 2321, France 1 48 14 24 24, Germany 089 741 31 30,
Hong Kong 2645 3186, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456,
Mexico 95 800 010 0793, Netherlands 0348 433466, Norway 32 84 84 00,
Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70,
Switzerland 056 200 51 51, Taiwan 02 377 1200, U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway Austin, TX 78730-5039 Tel: (512) 794-0100

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

LabVIEW®, NI-488.2™, NI-VISA™, NI-VXI™, and VXIpc™ are trademarks of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Table of Contents

About This Manual

Chapter 1
Overview of NI-VXI

Chapter 2
Introduction to the NI-VXI Functions

Chapter 3
Software Overview

Figures

Tables

About This Manual

This manual describes in detail the features of the NI-VXI software and the VXI/VME function calls in the C/C++ and BASIC languages.

Organization of This Manual

The *NI-VXI User Manual* for C/C++ and BASIC is organized as follows:

- Chapter 1, *Overview of NI-VXI*, introduces you to the concepts of VXI (VME eXtensions for Instrumentation), VME, MXI (Multisystem eXtension Interface), and their relationship to the NI-VXI application programmer's interface (API).
- Chapter 2, *Introduction to the NI-VXI Functions*, introduces you to the NI-VXI functions and their capabilities. Additional discussion is provided for each function's parameters and includes descriptions of the application development environment. This chapter concludes with an overview on using the NI-VXI application programming interface.
- Chapter 3, *Software Overview*, describes the C/C++ and BASIC usage of VXI and VME functions and briefly describes each function. Functions are listed alphabetically in each functional group.
- Appendix A, *Function Classification Reference*, contains two tables you can use as a quick reference. Table A-1, *Function Listing by Group*, lists the NI-VXI functions by their group association. This arrangement can help you determine easily which functions are available within each group. Table A-2, *Function Listing by Name*, lists each function alphabetically. You can refer to this table if you don't remember the group association of a particular function. Both tables use checkmarks to denote whether a VXI function also applies to VME and also whether it is associated with C/C++ and/or BASIC.

- Appendix B, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, and metric prefixes.
- The *Index* contains an alphabetical list of key terms and topics used in this manual, including the page where each one can be found.

Conventions Used in This Manual

The following conventions are used in this manual:

bold	Bold text denotes parameters, menus, menu items, dialog box buttons or options, or error messages.
bold italic	Bold italic text denotes a note, caution, or warning.
`bold monospace`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of example code that are different from the other examples.
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
`monospace`	Text in this font denotes the names of all VXI function calls, source code, sections of code, function syntax, console responses, variable names, and syntax examples.

In this manual numbers are decimal unless noted as follows:

- Binary numbers are indicated by a -b suffix (for example, 11010101b).
- Octal numbers are indicated by an -o suffix (for example, 325o).
- Hexadecimal numbers are indicated by an -h suffix (for example, D5h).
- ASCII character and string values are indicated by double quotation marks (for example, "This is a string").
- Long values are indicated by an *-L* suffix (for example, 0x1111L).

Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the *Glossary*.

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- *IEEE Standard for a Versatile Backplane Bus: VMEbus*, ANSI/IEEE Standard 1014-1987
- *Multisystem Extension Interface Bus Specification*, Version 2.0
- *VXI-1, VXIbus System Specification*, Revision 1.4, VXIbus Consortium
- *VXI-6, VXIbus Mainframe Extender Specification*, Revision 1.0, VXIbus Consortium

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix B, *Customer Communication*, at the end of this manual.

Chapter 1

Overview of NI-VXI

This chapter introduces you to the concepts of VXI (VME eXtensions for Instrumentation), VME, MXI (Multisystem eXtension Interface), and their relationship to the NI-VXI application programmer's interface (API).

Comprehensive functions for programming the VXIbus/VMEbus are included with the NI-VXI software. They are available for a variety of controller platforms and operating systems. Among the compatible platforms are the National Instruments line of embedded controllers and external computers that have a MXIbus interface.

☞ **Note:** ***The following chapter discusses features unique to VXI as well as common VXI/VME features. VME users can skip to the section entitled*** *Interrupts and Asynchronous Events.*

VXIbus Overview

Concepts of the VXIbus specification include the VXI device, message-based devices, the World Serial Protocol, the Commander/Servant hierarchy, and hardware interrupts and asynchronous events.

VXI Devices

A VXI device has a unique *logical address*, which serves as a means of referencing the device in the VXI system. This logical address is analogous to a GPIB device address. VXI uses an 8-bit logical address, allowing for up to 256 VXI devices in a VXI system.

Each VXI device must have a specific set of registers, called *configuration registers* (Figure 1-1) .These registers are located in the upper 16 KB of the 64 KB A16 VME address space. The logical address of a VXI device determines the location of the device's configuration registers in the 16 KB area reserved by VXI.

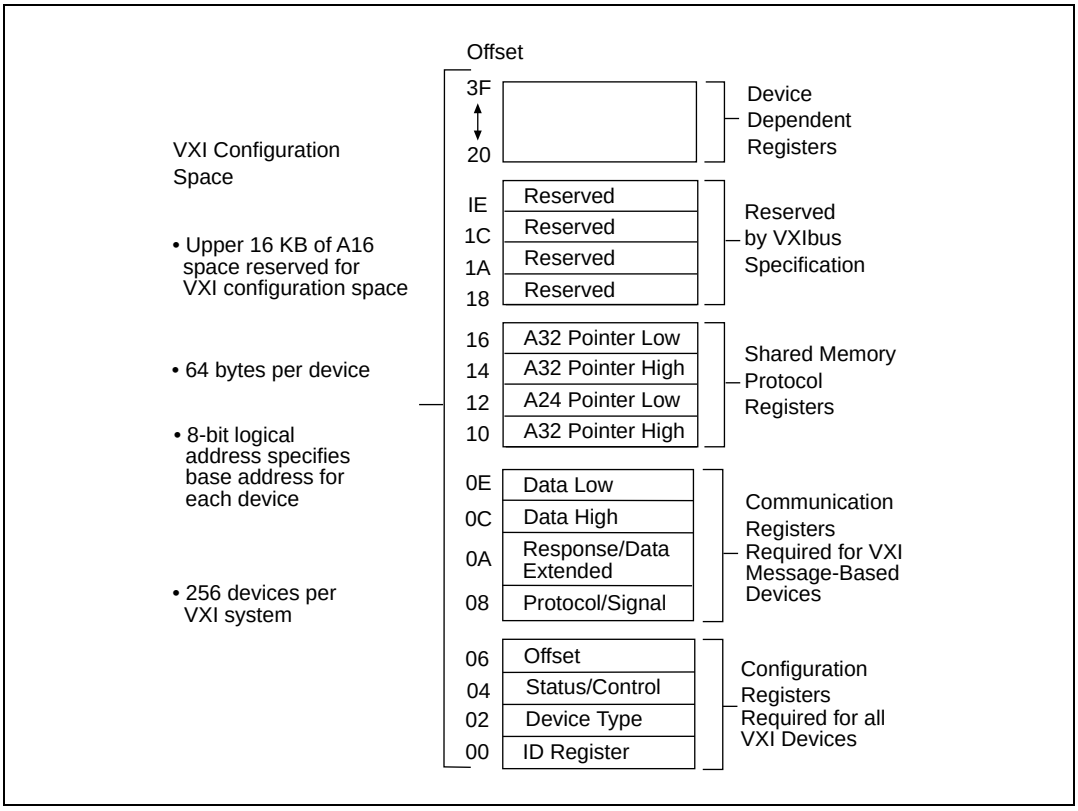


Figure 1-1. VXI Configuration Registers

Register-Based Devices

Through the use of the VXI configuration registers, which are required for all VXI devices, the system can identify each VXI device, its type, model and manufacturer, address space, and memory requirements. VXIbus devices with only this minimum level of capability are called *register-based* devices. With this common set of configuration registers, the centralized *Resource Manager* (RM), a software module, can perform automatic system configuration when the system is initialized.

Message-Based Devices

In addition to register-based devices, the VXIbus specification also defines *message-based* devices, which are required to have *communication registers* in addition to configuration registers. All message-based VXIbus devices, regardless of the manufacturer, can communicate at a minimum level using the VXI-specified *Word Serial Protocol.* In addition, you can establish higher-performance communication channels, such as shared-memory channels, to take advantage of the VXIbus bandwidth capabilities.

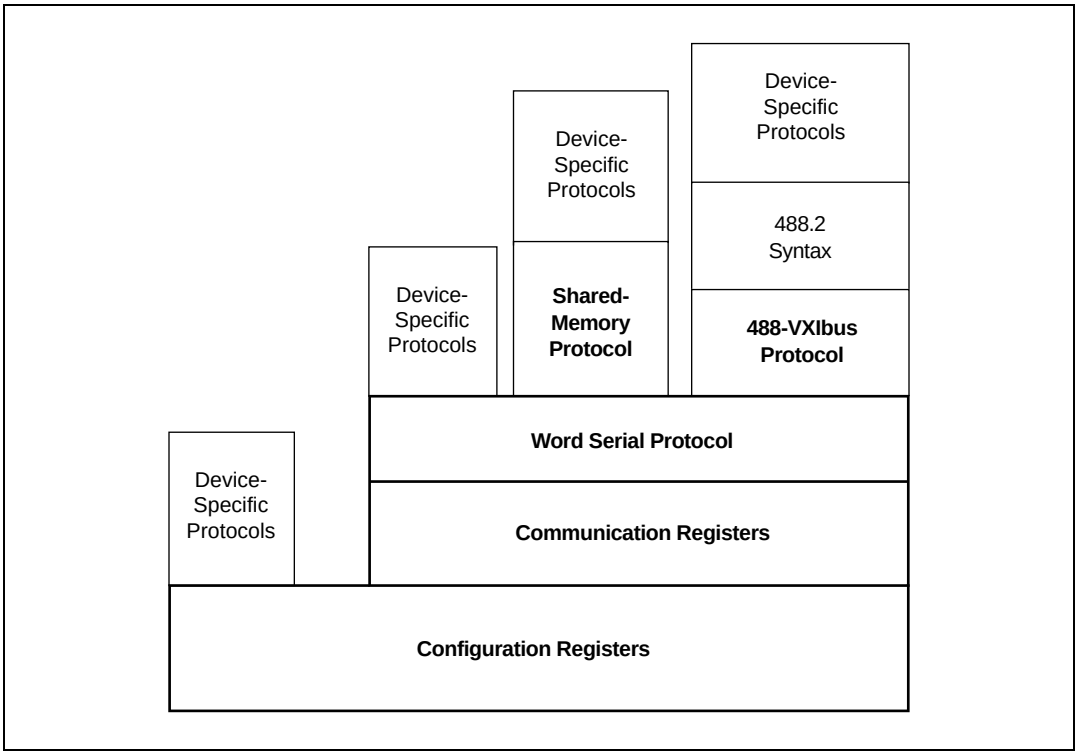


Figure 1-2. VXI Software Protocols

Word Serial Protocol

The VXIbus Word Serial Protocol is a standardized message-passing protocol. This protocol is functionally very similar to the IEEE 488 protocol, which transfers data messages to and from devices one byte (or word) at a time. Thus, VXI message-based devices communicate in

a fashion very similar to IEEE 488 instruments. In general, message-based devices typically contain some level of local intelligence that uses or requires a high level of communication. In addition, the Word Serial Protocol has messages for configuring message-based devices and system resources.

All VXI message-based devices are required to use the Word Serial Protocol and communicate in a standard way. The protocol is called *word serial*, because if you want to communicate with a message-based device, you do so by writing and reading 16-bit *words* one at a time to and from the Data In (write Data Low) and Data Out (read Data Low) hardware registers located on the device itself. Word serial communication is paced by bits in the device's response register that indicate whether the Data In register is empty and whether the Data Out register is full. This operation is very similar to the operation of a Universal Asynchronous Receiver Transmitter (UART) on a serial port.

Commander/Servant Hierarchies

The VXIbus specification defines a Commander/Servant communication protocol you can use to construct hierarchical systems using conceptual layers of VXI devices. The resulting structure is like a tree. A *Commander* is any device in the hierarchy with one or more associated lower-level devices, or *Servants*. A Servant is any device in the subtree of a Commander. A device can be both a Commander and a Servant in a multiple-level hierarchy.

A Commander has exclusive control of its immediate Servants' (one or more) communication and configuration registers. Any VXI module has one and only one Commander. Commanders use the Word Serial Protocol to communicate with Servants through the Servants' communication registers. Servants communicate with their Commander, responding to the Word Serial commands and queries from their Commander. Servants can also communicate asynchronous status and events to their Commander through hardware interrupts, or by writing specific messages directly to their Commander's Signal register.

Interrupts and Asynchronous Events

Servants can communicate asynchronous status and events to their Commander through hardware interrupts or by writing specific messages (signals) directly to their Commander's hardware Signal

register. Devices that do *not* have bus master capability always transmit such information via interrupts, whereas devices that *do* have bus master capability can either use interrupts or send signals. Some devices can receive only signals, some only interrupts, while some others can receive both signals and interrupts.

The VXIbus specification defines Word Serial commands so that a Commander can understand the capabilities of its Servants and configure them to generate interrupts or signals in a particular way. For example, a Commander can instruct its Servants to use a particular interrupt line, to send signals rather than generate interrupts, or configure the reporting of only certain status or error conditions.

Although the Word Serial Protocol is reserved for Commander/Servant communications, you can establish peer-to-peer communication between two VXI/VME devices through a specified shared-memory protocol or simply by writing specific messages directly to the device's Signal register, in addition to the VXI/VME interrupt lines.

MXIbus Overview

The MXIbus is a high-performance communication link that interconnects devices with a cabled communication link for very high-speed communication between physically separate devices. The emergence of the VXIbus inspired MXI. National Instruments, a member of the VXIbus Consortium and the VITA organization, recognized that VXI requires a new generation of connectivity for the instrumentation systems. Additionally, National Instruments realized that the same technology could be used also for the VMEbus, which is the foundation technology under VXI. National Instruments developed the MXIbus specification over a period of two years and announced it in April 1989 as an open industry standard.

MXI-2 Overview

MXI-2 is the second generation of the National Instruments MXIbus product line. The MXIbus is a general-purpose, 32-bit, multimaster system bus on a cable. MXI-2 expands the number of signals on a standard MXI cable by including VXI triggers, all VXI/VME interrupts, CLK10, and all of the utility bus signals (SYSFAIL*, SYSRESET*, and ACFAIL*).

Because MXI-2 incorporates all of these new signals into a single connector, the triggers, interrupts, and utility signals can be extended not only to other mainframes but also to the local CPU in all MXI-2 products using a single cable. Thus, MXI-2 lets CPU interface boards such as the PCI-MXI-2 perform as though they were plugged directly into the VXI/VME backplane.

In addition, MXI-2 boosts data throughput performance past previous-generation MXIbus products by defining new high-performance protocols. MXI-2 is a superset of MXI. However, MXI-2 defines synchronous MXI block data transfers which surpass previous block data throughput benchmarks. The new synchronous MXI block protocol increases MXI-2 throughput to a maximum of 33 MB/s between two MXI-2 devices. All National Instruments MXI-2 boards are capable of initiating and responding to synchronous MXI block cycles.

Chapter 2

Introduction to the NI-VXI Functions

This chapter introduces you to the NI-VXI functions and their capabilities. Additional discussion is provided for each function's parameters and includes descriptions of the application development environment. This chapter concludes with an overview on using the NI-VXI application programming interface.

The NI-VXI functions are a set of C/C++ and BASIC language functions you can use to perform operations with a VXI/VME system. The NI-VXI C/C++ language interface is consistent across hardware platforms and operating systems.

Function Groups

The NI-VXI functions are divided into several groups. All of them apply to VXI, but some groups are not applicable to VME.

VXI/VME Function Groups

The following NI-VXI function groups apply to both VXI and VME.

- *System Configuration Functions*—The system configuration functions provide functionality to initialize the NI-VXI software. In addition, the system configuration functions can retrieve or modify information about devices in your VXI/VME system.
- *High-Level VXIbus Access Functions*—Similar to the low-level VXI/VMEbus access functions, the high-level VXI/VMEbus access functions give you direct access to the VXI/VMEbus address spaces. You can use these functions to read, write, and move blocks of data between any of the VXI/VMEbus address spaces. You can specify the main VXI/VMEbus privilege mode or byte order. The functions trap and report bus errors. When the execution speed is not a critical issue, the high-level VXI/VMEbus access functions provide an easy-to-use interface.

- *Low-Level VXIbus Access Functions*—Low-level VXI/VMEbus access functions are the fastest access method for directly reading from or writing to any of the VXI/VMEbus address spaces. You can use these functions to obtain a pointer that is directly mapped to a particular VXI/VMEbus address. Then you use the pointer with the low-level VXI/VMEbus access functions to read from or write to the VXI/VMEbus address space. When using these functions in your application, you need to consider certain programming constraints and error conditions such as bus errors (BERR*).
- *Local Resource Access Functions*—Local resource access functions let you access miscellaneous local resources such as the local CPU VXI register set, Slot 0 MODID operations (when the local device is configured for Slot 0 operation), and the local CPU VXI Shared RAM. These functions are useful for shared memory type communication, for the non-Resource Manager operation (when the local CPU is not the Resource Manager), and for debugging purposes.
- *VXI Signal Functions*—VXI signals are a method for VXI bus masters to interrupt another device. You can route VXI signals to a handler or queue them on a global signal queue. You can use these functions to specify the signal routing, install handlers, manipulate the global signal queue, and wait for a particular signal value (or set of values) to be received.

☞ **Note:** ***Although signals are defined in the VXI specification, VME customers may still use the signal register available on any VXI/VME/MXI hardware. This register provides a simple notification mechanism that can be used by any bus-master.***

- *VXI/VME Interrupt Functions*—By default, interrupts are processed as VXI signals (either with a handler or by queuing on the global signal queue). The VXI/VME interrupt functions can specify the processing method and install interrupt service routines. In addition, the VXI/VME interrupt functions can assert specified VXI/VME interrupt lines with a specified status/ID value.
- *System Interrupt Handler Functions*—The system interrupt handler functions let you install handlers for the various system interrupt conditions. These conditions include Sysfail, ACfail, bus error, and soft reset interrupts.
- *VXI/VMEbus Extender Functions*—The VXI/VMEbus extender functions can dynamically configure multiple-mainframe mappings

of the VXI/VME interrupt lines, VXI TTL triggers, VXI ECL triggers, and utility bus signals. The National Instruments Resource Manager configures the mainframe extenders with settings based on user-modifiable configuration files.

VXI-Only Function Groups

The following NI-VXI function groups do not apply to VME.

- *Commander Word Serial Protocol Functions*—Word Serial is a form of communication between VXI message-based devices. The Commander Word Serial functions give you the necessary capabilities to communicate with a message-based Servant device using the Word Serial, Longword Serial, or Extended Longword Serial protocols. These capabilities include the sending of commands and queries and the reading and writing of buffers.
- *Servant Word Serial Protocol Functions*—Servant Word Serial functions allow you to communicate with the message-based Commander of the local CPU (the device on which the NI-VXI interface resides) using the Word Serial, Longword Serial, or Extended Longword Serial protocols. These capabilities include command/query handling and buffer reads/writes.
- *VXI Trigger Functions*—The VXI trigger functions let you source and accept any of the VXIbus trigger protocols. The actual capabilities available depend on the specific hardware platform. The VXI trigger functions can install handlers for various trigger interrupt conditions.

Calling Syntax

The interface is the same regardless of the development environment or the operating system used. Great care has been taken to accommodate all types of operating systems with the same functional interface (C/C++ source-level compatible), whether it is non-multitasking (for example, MS-DOS), cooperative multitasking (such as Microsoft Windows 3.*x* or Macintosh OS), multitasking (for example, UNIX, Wndows 95, or Windows NT), or real-time (such as LynxOS or VxWorks). The NI-VXI interface includes most of the mutual exclusion necessary for a multitasking environment. Each individual platform has been optimized within the boundaries of the particular hardware and operating system environment.

LabWindows/CVI

You can use the functions described in this manual with LabWindows/CVI. LabWindows/CVI is an integrated development environment for building instrumentation applications using the ANSI C programming language. You can use LabWindows/CVI with Microsoft Windows on PC-compatible computers or with Solaris on Sun SPARCstations. The source code you develop is portable across either platform.

National Instruments offers VXI/VME development systems for these two platforms that link the NI-VXI driver software into LabWindows/CVI to control VXI instruments from either embedded VXI/VME controllers or external computers equipped with a MXI interface. All of the NI-VXI functions described in this manual are completely compatible with LabWindows/CVI.

Type Definitions

The following data types are used for all parameters in the NI-VXI functions and in the actual NI-VXI library function definitions. NI-VXI uses this list of parameter types as an independent method for specifying data type sizes among the various operating systems and target CPUs of the NI-VXI software interface.

C/C++ Example:

```
typedef   char            INT8;   /* 8-bit signed integer    */
typedef   unsigned char   UINT8;  /* 8-bit unsigned integer  */
typedef   short           INT16;  /* 16-bit signed integer   */
typedef   unsigned short  UINT16; /* 16-bit unsigned integer */
typedef   long            INT32;  /* 32-bit signed integer   */
typedef   unsigned long   UINT32; /* 32-bit unsigned integer */
```

Input Versus Output Parameters

Because all C/C++ function calls pass function parameters by value (not by reference), you must specify the address of the parameter when the parameter is an output parameter. The C/C++ “&” operator accomplishes this task.

For example:

```
ret = VXIinReg (la, reg, &value);
```

Because `value` is an output parameter, `&value` is used when calling the function instead of `value`. The input parameters are `la` and `reg`.

Return Values and System Errors

All NI-VXI functions return a status indicating success or failure. The return code of 0x8000 is reserved as a return status value for any function to signify that a system error occurred during the function call except for the commander word serial operations. This error is specific to the operating system on which the NI-VXI interface is running.

Multiple Mainframe Support

The NI-VXI functions described in this manual support multiple mainframes both in external CPU configurations and embedded CPU configurations. The Startup Resource Manager supports one or more mainframe extenders and configures a single- or multiple-mainframe VXI/VME system. Refer to the *VXIbus Mainframe Extender Specification*, Revision 1.3 or later, for more details on multiple mainframe systems.

If you have a multiple-mainframe VXI/VME system, please continue with the following sections. If you have a single-mainframe system, you can skip to the *Using NI-VXI* section later in this chapter.

Controllers

A *controller* is a device that is capable of controlling other devices. A desktop computer with a MXI interface board, an embedded computer in a VXI/VME chassis, a VXI-MXI, and a VME-MXI may all be controllers depending on the configuration of the system.

There are several types of controllers that may exist in a VXI/VME system; embedded, external, and remote.

- *Embedded controller*—A computer plugged directly into the VXI/VME backplane. An example is the National Instruments VXIpc-850. All of the required VXI/VME interface capabilities are built directly onto the computer itself. An embedded computer has direct access to the VXI/VMEbus backplane in which it is installed.
- *Remote controller*—A device in the VXI/VME system that has the capability to control the VXI/VMEbus, but has no intelligent CPU installed. An example is the VXI-MXI-2. In NI-VXI, the parent-side VXI-MXI-2 (that is, the VXI-MXI-2 with a MXI-2

cable connected towards the root frame) in the frame acts as a remote controller. An embedded or external controller may use a remote controller to control the remote mainframe.

- *External controller*—A desktop computer or workstation connected to the VXI/VME system via a MXI interface board. An example is a standard personal computer with a PCI-MXI-2 installed.

In general, a multiple mainframe VXI/VME system will have one of the following controller configurations:

- An embedded controller in one frame that is connected to other frames via mainframe extenders using MXI-2. VXI-MXI-2 or VME-MXI-2 boards in the other frames can also be used as remote controllers. See Figure 2-1.

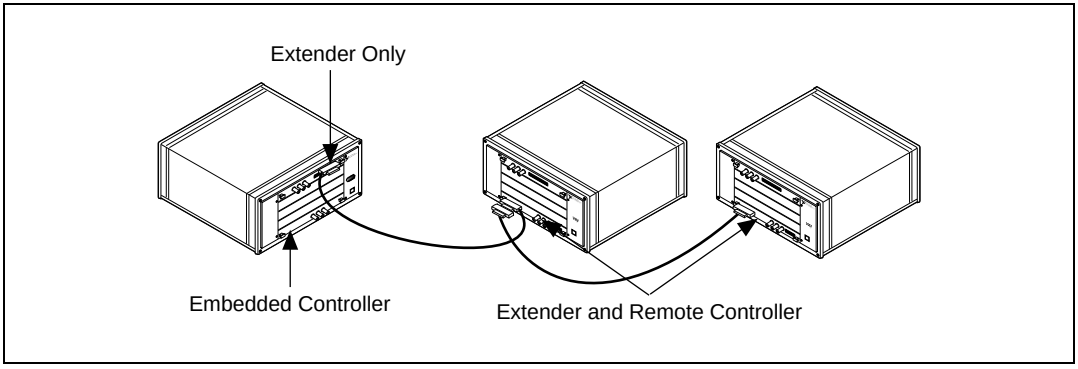


Figure 2-1. An Embedded Controller Connected to Other Frames via Mainframe Extenders Using MXI-2

- An external controller connected using MXI-2 to a number of remote controllers, each in a separate frame. The external controller can use the remote controllers for control of the VXI/VME system, or it can use its own controller capabilities. See Figure 2-2.

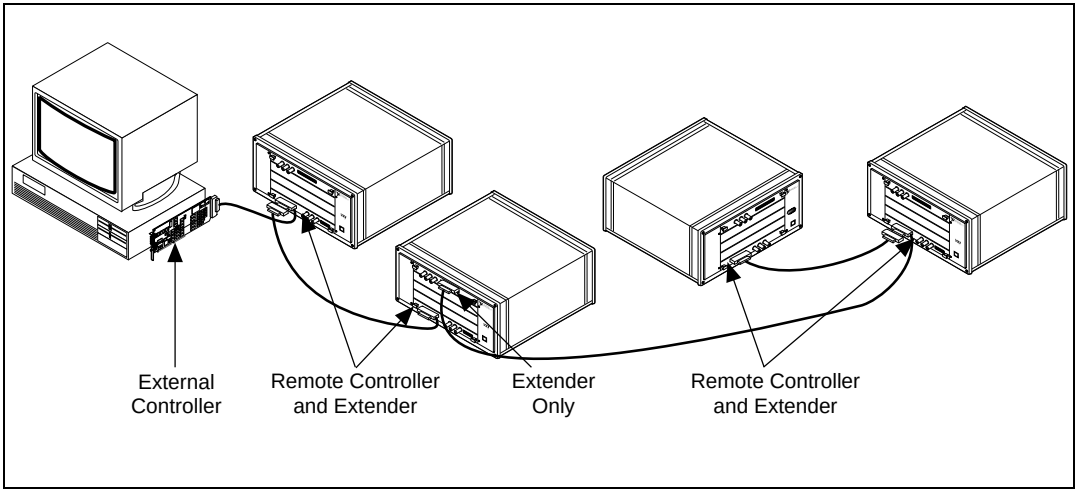


Figure 2-2. An External Controller Connected Using MXI-2 to a Number of Remote Controllers

The extender and controller Parameters

In NI-VXI, some functions require a parameter named **extender** or **controller**. Since some extenders act as controllers, there is often confusion concerning what logical addresses should be passed to these functions.

The **extender** parameter is the logical address of a mainframe extender on which the function should be performed. Usually, functions with an **extender** parameter involve the mapping of interrupt lines or trigger lines into or out of a frame.

The **controller** parameter is the logical address of an embedded, external, extending, or remote controller. Usually, functions with a **controller** parameter involve sourcing or sensing particular interrupts or triggers in a frame. According to the definitions of the different types of controllers, the only valid logical addresses for the **controller** parameter are:

- The external or embedded controller on which the program is running
- A parent-side VXI-MXI-2 or VME-MXI-2 in a frame

Most functions that take a **controller** parameter will allow you to pass (-1) as the logical address. This selects the default controller for the system. Notice that the default controller is determined by the following factors:

- If the program is running on an embedded controller, the default controller is the embedded controller.
- If the program is running on an external controller, you will be able to configure whether the default controller is the external controller or the remote controller with the lowest numbered logical address. With this behavior, if you write a program on an embedded controller referring to the controller as logical address-1, you will be able to swap the embedded controller configuration with an external controller configuration without changing your source code.

Notice that -1 is never a valid value for the **extender** parameter. In addition, the logical addresses of embedded and external controllers also are never valid values for the **extender** parameter. The **extender** parameter refers only to devices that can map interrupt lines, trigger lines, or other signals into or out of a frame.

Using NI-VXI

This section presents a general overview of the more commonly used class of functions available in NI-VXI. Additional information summarizes how you can use the functions to perform certain tasks and further describes the general structure of NI-VXI programming.

Header Files

Although `nivxi.h` is the only header file you need to include in your program for NI-VXI, the software distribution actually includes several additional header files along with `nivxi.h`. Some of these files have type definitions and macros that can make using NI-VXI easier, and make the code more portable across different platforms. The three main files of interest are `datasize.h`, `busacc.h`, and `devinfo.h`.

The datasize.h File

The `datasize.h` file defines the integer types for use in your program. For example, INT16 is defined as a 16-bit signed integer, and UINT32 is defined as a 32-bit unsigned integer. Using these types benefits you by letting you apply specific type sizes across platforms. Using undefined types can cause problems when porting your application between platforms. For example, an int in DOS is a 16-bit number but a 32-bit number in Solaris or LabWindows/CVI.

In addition to the integers, `datasize.h` defines several types for other uses such as interrupt handlers. For example, `NIVXI_HVXIINT` is an interrupt handler type. Merely defining a variable with this type is sufficient to create the function prototype necessary for your interrupt handler. Also, different platforms require different flags for use with interrupt handlers. To simplify this problem, `datasize.h` defines `NIVXI_HQUAL` and `NIVXI_HSPEC`, which are used in the handler definition and take care of the platform dependencies. See the *Interrupts and Signals* section later in this chapter and your `read me` file for more information. In addition, refer to Chapter 3, *Software Overview* for specific information.

The busacc.h File

The busacc.h file defines constants and macros for use with the high/low-level and slave memory access functions (see the *Master Memory Access* and *Slave Memory Access* sections later in this chapter). To make the code more readable, busacc.h defines such elements as memory space, privilege mode, and byte order as constants, and it defines macros to combine these constants into the necessary access parameters. Examine the header file for more information on the available macros and constants. You can see these tools in use by reviewing the example programs on memory accesses that appear later in this chapter and also the example programs included with your software.

The devinfo.h File

The devinfo.h file contains a data type that is used with the GetDevInfo()function described in the *System Configuration Functions* section in Chapter 3, *Software Overview*. The purpose of this function is to return various information about the system. GetDevInfo()can return the information either a piece at a time, or in one large data structure. The header file devinfo.h contains the type UserLAEntry, which defines the data structure that the function uses. Refer to the header file for the exact definition of the data structure.

The Beginning and End of an NI-VXI Program

All NI-VXI programs must call InitVXIlibrary() to initialize the driver before using any other functions. You must call CloseVXIlibrary() before exiting from your program to free resources associated with NI-VXI. The first function creates the internal structure needed to make the NI-VXI interface operational. When InitVXIlibrary() completes its initialization procedures, other functions can access information obtained by RESMAN, the VXIbus Resource Manager, as well as use other NI-VXI features such as interrupt handlers and windows for memory access. The second function destroys this structure and frees the associated memory. All programs using NI-VXI must call InitVXIlibrary() before any other NI-VXI function. In addition, your program should include a call to CloseVXIlibrary() before exiting.

An important note about these two functions is that the internal structure maintains a record of the number of calls to `InitVXIlibrary()` and `CloseVXIlibrary()`. Although `InitVXIlibrary()` needs to be called only once, the structure of your program may cause the function to be called multiple times. A successful call to `InitVXIlibrary()` returns either a zero or a one. A zero indicates that the structure has been created, and a one indicates that the structure was created by an earlier call so no action was taken (other than incrementing the count of the number of `InitVXIlibrary()` calls).

When `CloseVXIlibrary()` returns a successful code, it also returns either a zero or a one. A zero indicates that the structure has been successfully destroyed, and a one indicates that there are still outstanding calls to `InitVXIlibrary()` that must be closed before the structure is destroyed. The outcome of all of this is that when exiting a program, you should call `CloseVXIlibrary()` *the same number of times* that you have called `InitVXIlibrary()`.

Caution: ***In environments where all applications share NI-VXI, and hence the internal structure (such as Microsoft Windows), it can be dangerous for any one application to call*** `CloseVXIlibrary()` ***until it returns zero because this can close out the structure from under another application. It is vital to keep track of the number of times you have called*** `InitVXIlibrary()`.

System Configuration Tools

The *System Configuration Functions* section of Chapter 3, *Software Overview*, describes functions that a program can use to access information about the system. This is obtained either through configuration information or from information obtained by `RESMAN`. Armed with these functions, a program can be more flexible to changes within the system.

Note: ***The examples in this manual do not check for either warnings or errors in most of the functions' return codes. This step is omitted only to simplify the example programs. We strongly recommend that you include error checking in your own programs.***

For example, all VXI devices have at least one logical address by which they can be accessed. However, it is simple to change the logical address of most devices. Therefore, any program that uses a constant as a logical address of a particular device can fail if that device is reassigned to a different logical address. Programmers can use the NI-VXI function `FindDevLA()` to input information about the device—such as the manufacturer ID and model code—and receive the device's current logical address.

Consider the case of wanting to locate a device with manufacturer's code *ABCh* and model number *123h*. You could use the following code to determine the logical address.

C/C++ Example:

```
main() {
      INT16 ret, la;

      ret = InitVXIlibrary();

      /* -1 and empty quotes are used for don't cares */
      ret = FindDevLA("", 0xABC, 0x123, -1, -1, -1, -1, &la);
      if (ret < 0)
            printf("No such device found.\n");
      else
            printf("The logical address is %d\n", la);

      ret = CloseVXIlibrary();
}
```

In a similar fashion, the function `GetDevInfo()` can return a wide assortment of information on a device, such as the manufacturer name and number, the base and size of A24/32 memory, and the protocols that the device supports. This information can be returned in either a piecemeal fashion or in one large data structure. Notice that this data structure is a user-defined type, `UserLAEntry`, which is defined in the `devinfo.h` header file.

- For VME devices, this information cannot be determined by the VXIbus Resource manager. However, you can enter this information into the Non-VXI Device Editor in `VXIedit` or `VXItedit`. This will allow you to use these functions to retrieve information about the devices at run-time.

Word Serial Communication

When communicating with a message-based devices (MBD) in VXI, the protocol for string passing is known as *Word Serial*. The term is derived from the fact that all commands are 16 bits in length (word length), and that strings are sent serially, or one byte at a time. VXI also accommodates Long Word Serial (32-bit commands), and Extended Long Word Serial (48-bit commands). However, the VXIbus specification revision 1.4 states that only Word Serial commands have been defined.

Word Serial Protocol is based on a Commander writing 16-bit commands to a Servant register (See the *Commander Word Serial Protocol Functions* in Chapter 3, *Software Overview*, for more information on the protocol). The VXIbus specification has defined several commands, such as *Byte Available*, *Byte Request*, and *Clear*. The bit patterns for Word Serial commands have been laid out in the VXIbus specification, and your application can send these commands to a Servant via the `WScmd()` function. However, because string communication is the most common use for Word Serial Protocol, the functions `WSwrt()` and `WSrd()` use the Word Serial commands *Byte Available* (for sending a byte to a servant) and *Byte Request* (for retrieving a byte from a Servant) repetitively to send or receive strings as defined by the Word Serial Protocol. In addition, other common commands such as *Clear* have been encapsulated in their own functions, such as `WSclr()`.

Chapter 3, *Software Overview* describes all NI-VXI functions pertaining to message-based communication for the Commander. However, there are times when you want the controller to operate as a Word Serial Servant. NI-VXI allows for the controller to accept Word Serial commands from a Commander. This chapter also describes a different set of functions that a Servant uses for message-based communication with its Commander.

For example, `WSSrd()` (Word Serial Servant Read) sets up the controller to accept the *Byte Request* commands from a controller and respond with the string specified in the function. In a similar fashion, the `WSSwrt()` function programs the controller to accept *Byte Available* commands. National Instruments strongly recommends that if you want to program the controller as a Servant, you should aim to become familiar with the Word Serial Protocol in detail, and implement as much of the protocol as possible to simplify the debugging and operation of the program.

Master Memory Access

You can access VXIbus memory directly through the NI-VXI high-level and low-level VXIbus access functions, within the capabilities of the controller. The main difference between the high-level and low-level access functions is in the amount of encapsulation given by NI-VXI.

The high-level VXIbus access functions include functions such as `VXIin()` and `VXImove()` that you can use to access memory in the VXI system without dealing with such details as memory-mapping windows, status checking, and recovering from bus timeouts. Although these functions tend to have more overhead associated with them than the low-level functions, they are much simpler to use and typically require less debugging. We recommend that beginner programmers in VXI rely on the high-level functions until they are familiar with VXI memory accesses.

You can use the low-level VXI/VMEbus access functions if you want to access VXI/VME memory with as little overhead as possible. Although you now have to perform such actions as bus error handling and mapping—which are handled automatically by the high-level functions—you can experience a performance gain if you optimize for the particular accesses you are performing. Consider the following sample code, which performs a memory access using the low-level functions. Notice that there is no bus error handler installed by the program (See the *Interrupts and Signals* section). Instead, the program uses the NI-VXI default bus error handler. This handler automatically increments the `BusErrorRecv` global variable.

C/C++ Example:

```
#include <nivxi.h>  /* BusErrorRecv defined in nivxi.h */
#include <stdio.h>

main() {
       INT16 ret, la;
       UINT16 *addrptr, svalue;
       UINT32 addr, window1;
       INT32  timeout;
       UINT32 addrptr1;

       /* Start all programs with this function */
       ret = InitVXIlibrary();
       BusErrorRecv = 0;               /* Reset global variable */

       /* The following code maps the A16 space with the Access Only */
       /* access in order to access the A16 space directly.  */
       addr = 0xc000L;  /* Map upper 16 KB of the A16 space */
```

```
        timeout = 2000L; /* 2 seconds */

        /* Notice the use of the macros for defining the access */
        /* parameters.  These can be found in the NI-VXI header files */
        addrptr1 = (UINT32) MapVXIAddress(AccessP_Space(A16_SPACE) |
                                  AccessP_Priv(NonPriv_DATA) |
                                  AccessP_BO(MOTOROLA_ORDER) |
                                  AccessP_Owner(0),
                                  addr, timeout, &window1, &ret);
        if (ret >= 0)  /** MapVXIAddress call is successful **/
        {
              /* The following code reads the ID register of a device */
              /* at logical address 10.  */
              la = 10;
              addrptr = (UINT16 *)((UINT32) addrptr1 + 64 * la);
              VXIpeek(addrptr,2, &svalue));

              if (BusErrorRecv)
                    printf("Bus Error has occurred.\n");
              else
                    printf("Value read was %hd.\n", svalue));

              ret = UnMapVXIAddress(window1);
        } else
              printf("Unable to access window.\n");

        /* Close library when done */
        ret = CloseVXIlibrary();
}
```

Notice that the return variable for the `MapVXIAddress()` function is a pointer. While you can dereference this pointer directly on some platforms, we recommend that you use the `VXIpeek()` and `VXIpoke()` macros and functions in NI-VXI instead.

You can define `BINARY_COMPATIBLE` when compiling your program to force NI-VXI to use a version of `VXIpeek()` and `VXIpoke()` macros that will work on all embedded and MXI platforms. In addition, you can use the functions, rather than the macros, to ensure future compatibility. To force the compiler to use the functions, put the function name in parentheses, for example,

```
(VXIpoke) (addrptr, 2, 0);
```

instead of

```
VXIpoke (addrptr, 2, 0);
```

☞ **Note:** ***On modern, 32-bit operating systems running on high-performance processors (such as Microsoft Windows NT on a Pentium or Solaris 2 on a SPARC), we have found no performance gained by using macros instead of functions. For this reason, we strongly recommend that you use functions on these platforms to allow your program to be more portable across future platforms.***

Slave Memory Access

It is possible to share local resources such as RAM with the VXI/VMEbus. You can accomplish this functionality by setting the appropriate fields in the `VXIedit` or `VXItedit` NI-VXI resource editor utility to instruct the controller to respond to bus accesses as a slave. What address space is used is dependent on the settings in `VXIedit` or `VXItedit`. However, the actual VXI/VMEbus memory addresses are assigned by `RESMAN` and should be read by the program through the `GetDevInfo()` function.

Keep in mind that when the controller shares its resources, it may not allocate them from the local system first. For example, if you instruct the system to share 1 MB of RAM, the controller will map VXI/VME addresses (as defined by `RESMAN`) to 1 MB of local memory. However, the controller may not have prevented the local system from also using this space. For example, on a IBM compatible PC, the first 1 MB of address space contains not only user RAM, but also the interrupt vector table, video memory, BIOS, and so on. Therefore, it is important that you first use `VXImemAlloc()` to reserve a portion of the shared memory, and then communicate this address to the remote master that will be accessing the slave memory. For example, assume that the following code will run on a controller that has shared 1 MB of local RAM.

C/C++ Example:

```
main() {
     INT16 ret;
     UINT32 *useraddr, vxiaddr;
     void *bufaddr;

     /* Initialize and allocate 4 KB of memory */
     ret = InitVXIlibrary();
     ret = VXImemAlloc(4096, &useraddr, &vxiaddr);

     /* Put code here to communicate vxiaddr */
     /* returned by VXImemAlloc */
```

```
        /* At this point, the remote master can perform */
        /* I/O on the shared, allocated space.  In addition, */
        /* the program can use the local address to perform */
        /* I/O on the same space, such as reading back a block */
        /* of data */
        bufaddr = malloc (4096);
        ret = VXImemCopy (useraddr, bufaddr, 4096, 0);

        /* Return memory to local system */
        ret = VXImemFree(useraddr);
        ret = CloseVXIlibrary();
}
```

Interrupts and Signals

In NI-VXI, you can set up your controller to function as both an interrupt handler and an interrupter. You can also have your controller respond to writes to its signal register. Signaling another device requires the high-level or low-level VXI/VMEbus access functions, as discussed earlier. In addition, NI-VXI lets you configure both interrupts and signals to be handled either through callback handlers or through the signal queue. See the *VXI Signal Functions* section in Chapter 3, *Software Overview*, for more details about the signal queue, but for now you can look upon it as a FIFO (first-in, first-out) queue that you can access via the signal queue management functions, such as `SignalDeq()`. Both the signal queue and the callback handler will provide the status/ID obtained from the interrupt acknowledge or from the signal register. You can use this value to determine which device generated the interrupt/signal as well as the cause of the event. See the *VXI Interrupt Functions* section in Chapter 3, *Software Overview*, for more information.

Handling either signals or interrupts through the signal queue is very straightforward. You can use the `RouteVXIint()` and `RouteSignal()` functions to specify that the events should be handled by the signal queue. After you have enabled the event handler through either the `EnableSignalInt()` or the `EnableVXItoSignalInt()` call, the event is placed on the queue when it occurs. You can use the `SignalDeq()` or `WaitForSignal()` functions to retrieve the event from the queue.

☞ **Note:** `RESMAN` ***allocates interrupt lines to VXI devices that support the programmable interrupt command. Devices should use only those interrupt lines allocated to them. Again, you can use*** `GetDevInfo()` ***to determine what interrupt lines have been allocated to the controller.***

Alternatively, you can choose to handle either signals or interrupts with a callback handler. You can use `RouteSignal()` to specify that the events should be handled by the callback handlers rather than the signal queue. After you have enabled the callback handler through either the `EnableSignalInt()` or the `EnableVXIint()` call, the callback function will be invoked when the event occurs. Installing and using callback handlers is very simple with NI-VXI because all of the operating system interaction is handled for you. The following section of code gives an example of using an callback handler.

C/C++ Example:

```
#define VXI_INT_LEVEL 1  /* this sample only interested in level 1 */

/* NIVXI_HVXIINT is a type defined for VXI/VME interrupt callback handlers */
NIVXI_HVXIINT *OldVXIintHandler; /* pointer to save the old handler */
NIVXI_HVXIINT UserVXIintHandler; /* function declr for new handler */

main () {
     INT16 ret, controller;

     /* Always begin by initializing the NI-VXI library */
     ret = InitVXIlibrary ();
     controller = -1;

     /* Get address of the old handler */
     OldVXIintHandler = GetVXIintHandler (VXI_INT_LEVEL);

     /* Set callback handler to new user-defined procedure */
     ret = DisableVXIint (controller, 1<<(VXI_INT_LEVEL-1));
     ret = SetVXIintHandler (1<<(VXI_INT_LEVEL-1), UserVXIintHandler);
     ret = EnableVXIint (controller, 1<<(VXI_INT_LEVEL-1));

     /**/
     /* user code */
     /**/

     /* Restore callback handler to what it was before we changed it */
     ret = DisableVXIint (controller, 1<<(VXI_INT_LEVEL-1));
     SetVXIintHandler (1<<(VXI_INT_LEVEL-1), OldVXIintHandler);
     ret = EnableVXIint (controller, 1<<(VXI_INT_LEVEL-1));

     /* Always close the NI-VXI library before exiting */
     CloseVXIlibrary ();
}

/* The NIVXI_HQUAL and NIVXI_HSPEC should bracket */
/* every interrupt handler as shown below.  */
NIVXI_HQUAL void NIVXI_HSPEC UserVXIintHandler (INT16 controller,
                                        UINT16 level, UINT32 statusID)
{
     /* user code for processing statusID */
}
```

☞ **Note:** ***Although NI-VXI simplifies the installation and use of callback handlers, it cannot affect how the system handles interrupts. The programmer must follow programming guidelines set by the chosen operating system. Some of these guidelines could include using only reentrant functions, adhering to timing restrictions, and on Macintosh computers, regaining access to global variables.***

Triggers

The addition of trigger lines to the VMEbus is one improvement the VXIbus has over VME in the field of instrumentation. To take advantage of this feature, NI-VXI has a wide selection of functions you can use to set up your controller to both source and acknowledge trigger lines. The TIC is a National Instruments ASIC (Application Specific Integrated Circuit) that gives you the capability to map trigger lines to trigger lines as well as to external lines, use special counter/timers, and monitor multiple trigger lines simultaneously.

Chapter 3

Software Overview

This chapter describes the C/C++ and BASIC usage of VXI and VME functions and briefly describes each function. Functions are listed alphabetically in each functional group.

System Configuration Functions

The VXI system configuration functions copy all of the Resource Manager (RM) table information into data structures at startup so that you can find device names or logical addresses by specifying certain attributes of the device for identification purposes.

Initializing and closing the NI-VXI software interface, and getting information about devices in the system are among the most important aspects of the NI-VXI software. All applications need to use the system configuration functions at one level or another. When the NI-VXI RM runs, it logs the system configuration information in the RM table file, `resman.tbl`. The `InitVXIlibrary` function reads the information from `resman.tbl` into data structures accessible from the `GetDevInfo` and `SetDevInfo` functions. From this point on, you can retrieve any device-related information from the entry in the table. In most cases you do not need to modify resource manager information. However, you can use `SetDevInfo` functions to modify the information in the table. In this manner, both the application and the driver functions have direct access to all the necessary VXI/VME system information. Your application must call the `CloseVXIlibrary` function upon exit to free all data structures and disable interrupts.

The following paragraphs describe the system configuration functions. The descriptions are presented at a functional level describing the operation of each function.

CloseVXIlibrary ()

`CloseVXIlibrary` is the application termination routine, which must be included at the end (or abort) of any application. `CloseVXIlibrary` disables interrupts and frees dynamic memory allocated for the internal RM table and other structures. You must include a call to `CloseVXIlibrary` at the termination of your application (for whatever reason) to free all data structures allocated by `InitVXIlibrary` and disable interrupts. Failure to call `CloseVXIlibrary` when terminating your application can cause unpredictable and undesirable results. If your application can be aborted from some operating system abort routine (such as a *break* key or a process kill signal), be certain to install an abort/close routine to call `CloseVXIlibrary`.

CreateDevInfo (la)

`CreateDevInfo` creates a new entry in the dynamic NI-VXI RM table for the specified logical address. It installs default `NULL` values into the entry. You must use one of the `SetDevInfo` functions after this point to change any of the device information as needed. This operation is not needed for VME devices since it is recommended that you use the Non-VXI Device Editor in the `VXIedit` or `VXItedit` NI-VXI resource editor utility. At the startup of your application, `InitVXIlibrary` completely initializes the RM table to how the RM configured the VXI system. No initial changes/creations are necessary for VXI devices. You can use `CreateDevInfo` to add non-VXI devices or pseudo-devices (future expansion).

FindDevLA (namepat, manid, modelcode, devclass, slot, mainframe, cmdrla, la)

`FindDevLA` scans the RM table information for a device with the specified attributes and returns its VXI logical address. You can use any combination of attributes to specify a device. A -1 (negative one) or `""` specifies to ignore the corresponding field in the attribute comparison. After finding the VXI logical address, you can use one of the `GetDevInfo` functions to get any information about the specified device.

GetDevInfo (la, field, fieldvalue)

`GetDevInfo` returns information about the specified device from the NI-VXI RM table. The **field** parameter specifies the attribute of the information to retrieve. Possible **fields** include the device name, Commander's logical address, mainframe number, slot, manufacturer ID number, model code, model name, device class, VXI address space/base/size allocated, VXI interrupt lines/handlers allocated, protocols supported, and so on. A **field** value of zero (0) specifies to return a structure containing all possible information about the specified device.

GetDevInfoLong (la, field, longvalue)

`GetDevInfoLong` returns information about the specified device from the NI-VXI RM table. The **field** parameter specifies the attribute of the information to retrieve. `GetDevInfoLong` is a function layered on top of `GetDevInfo` for languages (such as BASIC) that cannot typecast the **fieldvalues** of `GetDevInfo`. `GetDevInfoLong` returns only the **fields** of `GetDevInfo` that are *32-bit integers*. Possible **fields** include the VXI address base and size allocated to the device by the RM.

GetDevInfoShort (la, field, shortvalue)

`GetDevInfoShort` returns information about the specified device from the NI-VXI RM table. The **field** parameter specifies the attribute of the information to retrieve. `GetDevInfoShort` is a function layered on top of `GetDevInfo` for languages (such as BASIC) that cannot typecast the **fieldvalues** of `GetDevInfo`. `GetDevInfoShort` returns only the **fields** of `GetDevInfo` that are *16-bit integers*. Possible **fields** include the Commander's logical address, mainframe number, slot, manufacturer ID number, manufacturer name, model code, device class, VXI address space allocated, VXI interrupt lines/handlers allocated, protocols supported, and so on.

GetDevInfoStr (la, field, stringvalue)

`GetDevInfoStr` returns information about the specified device from the NI-VXI RM table. The **field** parameter specifies the attribute of the information to retrieve. `GetDevInfoStr` is a function layered on top of `GetDevInfo` for languages (such as BASIC) that cannot typecast the **fieldvalues** of `GetDevInfo`. `GetDevInfoStr` returns only the **fields** of `GetDevInfo` that are *character strings*. Possible **fields** include the device name, manufacturer name, and model name.

InitVXIlibrary ()

`InitVXIlibrary` is the NI-VXI initialization routine. An application must call `InitVXIlibrary` at application startup. `InitVXIlibrary` performs all necessary installation and initialization procedures to make the NI-VXI interface functional. This includes copying all of the RM device information into the data structures in the NI-VXI library. This function configures all hardware interrupt sources (but leaves them disabled) and installs the corresponding default handlers. It also creates and initializes any other data structures required internally by the NI-VXI interface. When your application completes (or is aborted), it must call `CloseVXIlibrary` to free data structures and disable all of the interrupt sources.

SetDevInfo (la, field, fieldvalue)

`SetDevInfo` changes information about the specified device in the NI-VXI RM table. The **field** parameter specifies the attribute of the information to change. Possible **fields** include the device name, Commander's logical address, mainframe number, slot, manufacturer ID number, manufacturer name, model code, model name, device class, VXI address space/base/size allocated, VXI interrupt lines/handlers allocated, protocols supported, and so on. A **field** value of zero (0) specifies to change the specified entry with the supplied structure containing all possible information about the specified device. You should use this function only in very special situations, because it updates information in the NI-VXI interface and can affect execution. At the startup of your application, `InitVXIlibrary` completely initializes the RM table according to how the RM configured the VXI system. No initial changes are necessary for VXI devices.

SetDevInfoLong (la, field, longvalue)

`SetDevInfoLong` changes information about the specified device in the NI-VXI RM table. The **field** parameter specifies the attribute of the information to change. `SetDevInfoLong` is a function layered on top of `SetDevInfo` for languages (such as BASIC) that cannot typecast the **fieldvalues** of `SetDevInfo`. `SetDevInfoLong` returns only the **fields** of `SetDevInfo` that are *32-bit integers*. Possible **fields** include the VXI address base and size allocated to the device by the RM. You should use this function only in very special situations, because it updates information in the NI-VXI interface and can affect execution. At the startup of your application, `InitVXIlibrary` completely initializes the RM table to how the RM configured the VXI system. No initial changes are necessary for VXI devices.

SetDevInfoShort (la, field, shortvalue)

`SetDevInfoShort` changes information about the specified device in the NI-VXI RM table. The **field** parameter specifies the attribute of the information to change. `SetDevInfoShort` is a function layered on top of `SetDevInfo` for languages (such as BASIC) that cannot typecast the **fieldvalues** of `SetDevInfo`. `SetDevInfoShort` changes only the **fields** of `SetDevInfo` that are *16-bit integers*. Possible **fields** include the Commander's logical address, mainframe number, slot, manufacturer ID number, model code, device class, VXI address space allocated, VXI interrupt lines/handlers allocated, protocols supported, and so on. You should use this function only in very special situations, because it updates information in the NI-VXI interface and can affect execution. At the startup of your application, `InitVXIlibrary` completely initializes the RM table to how the RM configured the VXI system. No initial changes are necessary for VXI devices.

SetDevInfoStr (la, field, stringvalue)

`SetDevInfoStr` changes information about the specified device in the NI-VXI RM table. The **field** parameter specifies the attribute of the information to change. `SetDevInfoStr` is a function layered on top of `SetDevInfo` for languages (such as BASIC) that cannot typecast the **fieldvalues** of `SetDevInfo`. `SetDevInfoStr` returns only the **fields** of `SetDevInfo` that are *character strings*. Possible **fields** include the device name, manufacturer name, and model name. You should use this function only in very special situations, because it updates information in the NI-VXI interface and can affect execution. At the startup of your application, `InitVXIlibrary` completely initializes the RM table to how the RM configured the VXI system. No initial changes are necessary for VXI devices.

Commander Word Serial Protocol Functions

Word Serial communication is the minimal mode of communication between VXI message-based devices within the VXI Commander/Servant hierarchy. The Commander Word Serial functions let the local CPU (the CPU on which the NI-VXI interface resides) perform VXI message-based Commander Word Serial communication with its Servants. The four basic types of Commander Word Serial transfers are as follows:

- Command sending
- Query sending

- Buffer writes
- Buffer reads

The Word Serial Protocol is a 16-bit transfer protocol between a Commander and its Servants. The Commander polls specific bits in the Servant's VXI Response register to determine when it can write a command, when it can read a response from the Data Low register, and when a Word Serial protocol error occurs.

Before a Commander can send a Word Serial command to a Servant, it must first poll the Write Ready (WR) bit until it is asserted (set to 1). The Commander can then write the command to the servant's Data Low register. If the Commander is sending a query, it first sends the query in the same manner as sending a command, but then continues by polling the Read Ready (RR) bit until it is asserted. It then reads the response from the servant's Data Low register.

A buffer write involves sending a series of *Byte Available* (BAV) Word Serial commands to the Servant, with the additional constraint that the Data In Ready (DIR) bit as well as the WR bit be asserted before sending the *Byte Available*. The lower 8 bits (bits 0 to 7) of the 16-bit command contain a single byte of data (bit 8 is the END bit). Therefore, one *Byte Available* is sent for each data byte in the buffer written.

A buffer read involves sending a series of *Byte Request* (BREQ) Word Serial queries to the Servant, with the additional constraint that the Data Out Ready (DOR) bit as well as the WR bit must be asserted before sending the *Byte Request*. The lower 8 bits (bits 0 to 7) of the 16-bit response contain a single byte of data (bit 8 is the END bit). Therefore, one *Byte Request* is sent for each data byte in the buffer read.

In addition to the WR, RR, DIR, and DOR bits that get polled during various Word Serial transfers, the functions also check the ERR* bit. The ERR* bit indicates when a Word Serial Protocol error occurs. The Word Serial Protocol errors are Unsupported Command, Multiple Query Error (MQE), DIR Violation, DOR Violation, RR Violation, or WR Violation. After the Servant asserts the ERR* bit, the application can determine the actual error that occurred by sending a *Read Protocol Error* query to the Servant. The NI-VXI Word Serial functions query the Servant automatically and return the appropriate error codes to the caller, at which time the Servant deasserts the ERR* bit.

In addition to the four basic types of Word Serial transfers, there are two special cases: the Word Serial *Clear* and *Trigger* commands. The Word Serial *Clear* command must ignore the ERR* bit. One of the functions of the *Clear* command is to clear a pending protocol error condition. If the ERR* bit was polled during the transfer, the *Clear* would not succeed. The Word Serial *Trigger* command requires polling the DIR bit as well as the WR bit (similar to the buffer write) before writing the *Trigger* to the Data Low register. The VXIbus specification requires polling the DIR bit for the Word Serial *Trigger* to keep the write and trigger model consistent with IEEE 488.2.

The Longword Serial and Extended Longword Serial Protocols are similar to the Word Serial Protocol, but involve 32-bit and 48-bit command transfers, respectively, instead of the 16-bit transfers of the Word Serial Protocol. The VXIbus specification, however, provides no common command usages for these protocols. The commands are either VXI Reserved or User-Defined. The NI-VXI interface gives you the ability to send any one of these commands.

Programming Considerations

The Commander Word Serial functions provide a flexible, easy-to-use interface. Depending upon the hardware and software platforms involved in your system, however, certain issues need to be taken into account. In particular, the behavior of these functions will vary when called from different processes depending on how your operating system performs multitasking.

Interrupt Service Routine Support

If portability between operating systems is essential, the Word Serial Protocol functions should not be called from an interrupt service routine. Only for operating systems in which the user-installed handlers are run at process level (most UNIX, OS/2, and Windows 95/NT systems) is it possible to initiate Word Serial operations from a user-installed handler. The Commander Word Serial functions require operating system support provided only at the application (process) level of execution. Calling these functions from the CPU interrupt level will have undetermined results.

The `WSabort` function is the only exception to this restriction. `WSabort` is used to abort various Word Serial transfers in progress and will usually be called from an interrupt service routine (although it is not limited to interrupt service routines). The most common example of calling this function from an interrupt service routine is with the handling of *Unrecognized Command* events from a device. When an *Unrecognized Command* event is received by the NI-VXI interrupt or Signal interrupt handler, `WSabort` must be called to abort the current Word Serial command transfer in progress that caused the generation of the *Unrecognized Command* event.

Single-Tasking Operating System Support

The Word Serial Protocol functions have no asynchronous or multiple call support for a non-multitasking operating system. Because the Word Serial Protocol functions are polled I/O functions that do not return to the caller until the entire operation is complete, only one call can be pending for the application-level code. No Word Serial Protocol functions, other than `WSabort`, can be called at interrupt service routine time. If a Word Serial operation is underway and an interrupt service routine invokes another Word Serial operation, the polling mechanism may become inconsistent with the state of the Servant's communication registers. This could result in invalid data being transferred, protocol errors occurring, or a timeout. The `WSabort` function is used to asynchronously abort Word Serial operations in progress and can be used at interrupt service routine time.

Cooperative Multitasking Support

NI-VXI supports multiple processes under cooperative multitasking operating systems. The behavior is the same as in single-tasking operating systems, described above.

Multitasking Support (Preemptive Operating System)

The Word Serial Protocol functions have extensive mutual exclusion support when running on a preemptive multitasking operating system. A two-level mutual exclusion algorithm is used to allow read, write, and trigger calls to be made at the same time. Command transfers will automatically suspend read, write, or trigger calls in progress.

Figure 3-1 gives a precise description of this two-level mutual exclusion algorithm. Notice that this mutual exclusion is on a per logical address basis. Any number of logical addresses can have Word Serial transfers in progress without conflict. If the application is to be compatible with IEEE 488.2, the application must perform trigger and write calls in sequential order.

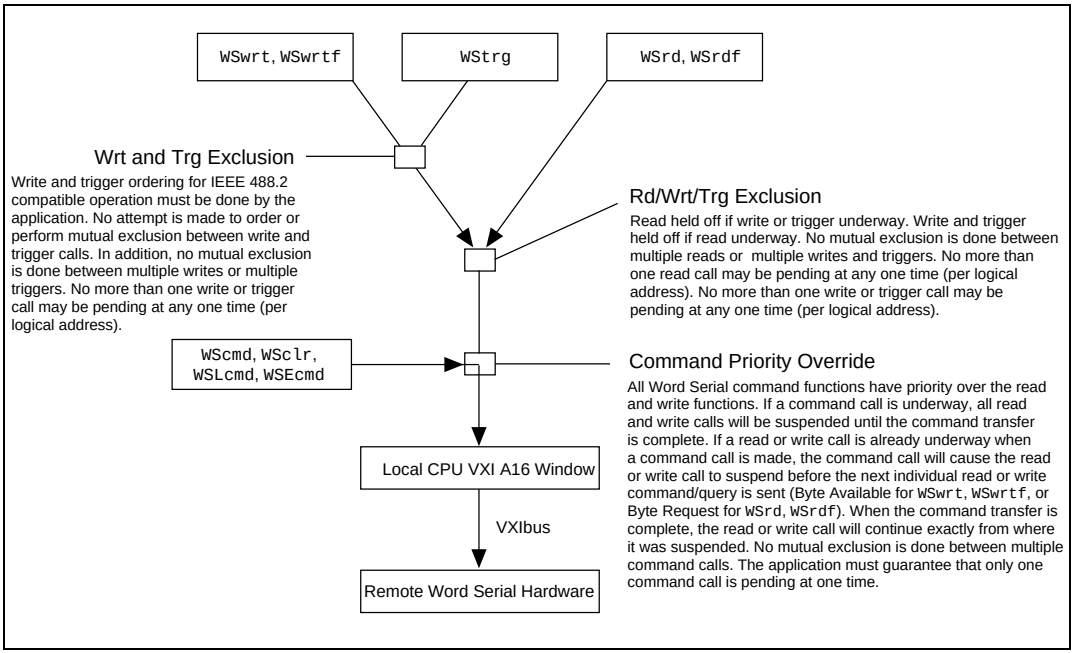


Figure 3-1. Preemptive Word Serial Mutual Exclusion (Per Logical Address)

The Commander Word Serial functions are fully reentrant and preemptive on a per logical address basis. Any number of logical addresses can have Commander Word Serial functions in progress without conflict.

Because Commander Word Serial is a protocol involving extensive polling, support has been added for a *round-robin* effect of Commander Word Serial function calls. If a particular logical address does not respond within a set number of polls to a particular Word Serial command or query, the process is suspended and another process (possibly with a different Commander Word Serial call in progress) can continue to execute. The amount of time for which the process is suspended is dependent upon the operating system. When the original process is resumed, the polling will continue. The polling will continue

until the transfer is complete or a timeout occurs. This support also keeps a word serial device which is not responding from “hanging” on the local CPU.

The following paragraphs describe the Commander Word Serial, Longword Serial, and Extended Longword Serial Protocol functions. The descriptions are grouped by functionality and are presented at a functional level describing the operation of each of the functions.

WSabort (la, abortop)

`WSabort` aborts the Commander Word Serial operation(s) in progress with a particular device. This function does not perform any Word Serial transfers. Instead, it aborts any Word Serial operation already in progress. The **abortop** parameter specifies the type of abort to perform. The `ForcedAbort` operation aborts read, write, and trigger operations with the specified device. The `UnSupCom` operation performs an Unsupported Command abort of the current Word Serial, Longword Serial, or Extended Longword Serial command in progress. The `UnSupCom` operation is called when an *Unrecognized Command* Event is received by `DefaultSignalHandler`.

WSclr (la)

`WSclr` sends the Word Serial *Clear* command to a message-based Servant. The *Clear* command clears any pending protocol error on the receiving device. The ERR* bit is ignored during the transfer. The WR bit is polled until asserted after the *Clear* command is sent to verify that the command executed properly.

WScmd (la, cmd, respflag, response)

`WScmd` sends a Word Serial command or query to a message-based Servant. It polls the WR bit before sending the command, and polls the RR bit before reading the response (if applicable) from the Data Low register. `WScmd` polls the WR bit after either sending the command (for a command) or reading the response (for a query), to guarantee that no protocol errors occurred during the transfer. Under the VXIbus specification, the ERR* bit can be asserted at any time prior to reasserting the WR bit. Do not use this function to send the Word Serial commands *Byte Available* (BAV), *Byte Request* (BREQ), *Trigger*, or *Clear*. All of these Word Serial commands require different Response register polling techniques.

WSEcmd (la, cmdExt, cmd, respflag, response)

`WSEcmd` sends an Extended Word Serial command or query to a message-based Servant. It polls the WR bit before sending the 48-bit command. `WSEcmd` sends the command by writing the Data Extended register first with the upper 16 bits of the command (**cmdExt**), followed by the Data High register with the middle 16 bits of the command (upper 16 bits of **cmd**), and concluding with the Data Low register with the lower 16 bits of the command (lower 16 bits of **cmd**). It then polls the RR bit before reading the 32-bit response from the Data Low and Data High registers (there are no 48-bit responses for Extended Longword Serial). `WSEcmd` polls the WR bit after either sending the command (for a command) or reading the response (for a query), to guarantee that no protocol errors occurred during the transfer.

WSgetTmo (actualtimo)

`WSgetTmo` retrieves the current timeout period for all of the Commander Word Serial Protocol functions. It retrieves the current timeout value in milliseconds to the nearest resolution of the host CPU.

WSLcmd (la, cmd, respflag, response)

`WSLcmd` sends a Longword Serial command or query to a message-based Servant. It polls the WR bit before sending the command. `WSLcmd` sends the command by writing the Data High register first with the upper 16 bits of the 32-bit command, and then writing the Data Low register with the lower 16 bits of the 32-bit command. It then polls the RR bit before reading the 32-bit response from the Data Low and Data High registers. `WSLcmd` polls the WR bit after either sending the command (for a command) or reading the response (for a query), to guarantee that no protocol errors occurred during the transfer.

WSLresp (la, response)

`WSLresp` retrieves a response to a previously sent Longword Serial Protocol query from a VXI message-based Servant.

☞ **Note:** ***This function is intended only for debugging purposes.***

Normally, you would use the WSLcmd function to send Longword Serial queries with the response automatically read (specified with **respflag**). In cases when you need to inspect the Longword Serial transfer at a lower level, however, you can break up the query sending and query response retrieval by using the WSLcmd function to send the query as a command, and using the WSLresp function to retrieve the response. WSLresp polls the RR bit before reading the response from the Data High and Data Low registers to form the 32-bit response. After reading the response, it polls the Response register until the WR bit is asserted to guarantee that no protocol errors occurred during the transfer.

WSrd (la, buf, count, modevalue, retcount)

WSrd is the word serial buffer read function. WSrd reads a specified number of bytes from a Servant device into a local memory buffer, using the VXIbus Byte Transfer Protocol. The process involves sending a series of *Byte Request* (BREQ) Word Serial queries and reading the responses. Each response contains a data byte in the lower 8 bits and the END bit in bit 8. Before sending the BREQ command, WSrd polls both Response register bits—Data Out Ready (DOR) and Write Ready (WR). It polls the Response register Read Ready (RR) bit before reading the response from the Data Low register. The read terminates when it receives a maximum number of bytes or if it encounters an END bit, a carriage return (CR), a line feed (LF), or a user-specified termination character.

WSrdf (la, filename, count, modevalue, retcount)

WSrdf is an extension of the WSrd function. WSrdf reads a specified number of bytes from a Servant device into the specified file, using the VXIbus Byte Transfer Protocol. The process involves calling the function WSrd (possibly many times) to read in a block of data and writing the data to the specified file. The read terminates when it receives a maximum number of bytes or if it encounters an END bit, a carriage return (CR), a line feed (LF), or a user-specified termination character.

WSresp (la, response)

WSresp retrieves a response to a previously sent Word Serial Protocol query from a VXI message-based Servant.

☞ **Note:** ***This function is intended only for debugging purposes.***

Normally, you would use the WScmd function to send Word Serial queries with the response automatically read (specified with **respflag**). In cases when you need to inspect the Word Serial transfer at a lower level, however, you can break up the query sending and query response retrieval by using the WScmd function to send the query as a command and using the WSresp function to retrieve the response. During the interim period between sending the WScmd and WSresp functions, you can check register values and other hardware conditions. WSresp polls the RR bit before reading the response from the Data Low register. After reading the response, it polls the Response register until the WR bit is asserted.

WSsetTmo (timo, actualtimo)

WSsetTmo sets the timeout period for all of the Commander Word Serial Protocol functions. It sets the timeout value in milliseconds to the nearest resolution of the host CPU. When a timeout occurs during a Commander Word Serial Protocol function, the function terminates with a corresponding error code.

WStrg (la)

WStrg sends the Word Serial *Trigger* command to a message-based Servant. Before sending the *Trigger* command (by writing to the Data Low register), WStrg polls both Response register bits—Data In Ready (DIR) and Write Ready (WR)—until asserted. You cannot use the WScmd function to send the Word Serial *Trigger* command (WScmd polls only for WR before sending the command). WStrg polls the WR bit until asserted again after sending the *Trigger* command to guarantee that no protocol errors occurred during the transfer.

WSwrt (la, buf, count, modevalue, retcount)

This function is the buffer write function. WSwrt writes a specified number of bytes from a memory buffer to a message-based Servant using the VXIbus Byte Transfer Protocol. The process involves sending a series of *Byte Available* (BAV) Word Serial commands with a single byte in the lower 8 bits of the command. Before sending the BAV command, WSwrt polls both Response register bits—Data In Ready (DIR) and Write Ready (WR)—until asserted. The **modevalue** parameter in the call specifies whether to send BAV only or BAV with END for the last byte of the transfer.

WSwrtf (la, filename, count, modevalue, retcount)

`WSwrtf` is an extension of the `WSwrt` function. `WSwrtf` writes a specified number of bytes from the specified file to a message-based Servant using the VXIbus Byte Transfer Protocol. The process involves calling the `WSwrt` function (possibly many times) to write out a block of data read from the specified file. The **modevalue** parameter in the call specifies whether to send BAV only or BAV with END for the last byte of the transfer.

Servant Word Serial Protocol Functions

Word Serial communication is the minimal mode of communication between VXI message-based devices within the VXI Commander/Servant hierarchy. The local CPU (the CPU on which the NI-VXI functions are running) uses the Servant Word Serial functions to perform VXI message-based Servant Word Serial communication with its Commander. These functions are needed only in the case where the local CPU is not a top-level Commander (probably not the Resource Manager), such as in a multiple CPU situation. In a multiple CPU situation, the local CPU must allow the Resource Manager device to configure the local CPU and can optionally implement some basic message-transfer Word Serial communication with its Commander. The four basic types of Servant Word Serial functions are as follows:

- Receiving commands
- Receiving and responding to queries
- Responding to requests to send buffers
- Receiving buffers

The Word Serial Protocol is a 16-bit transfer protocol between a Commander and its Servants. The Commander polls specific bits in the Servant's VXI Response register to determine when it can write a command or read a response from the Data Low register. It also determines when a Word Serial protocol error occurs. Before a Commander can send a Word Serial command to a Servant, it must first poll the Write Ready (WR) bit until it is asserted (set to 1). The Commander can then write the command to the Data Low register. If the Commander is sending a query, it first sends the query in the same manner as sending a command, but then continues by polling the Read Ready (RR) bit until it is asserted. It then reads the response from the Data Low register.

A buffer write is a series of *Byte Available* Word Serial commands sent to the Servant, with the additional constraint that the Data In Ready (DIR) bit as well as the WR bit must be asserted before sending the *Byte Available* command. The lower 8 bits (bits 0 to 7) of the 16-bit command contain a single byte of data (bit 8 is the END bit). Therefore, one *Byte Available* is sent for each data byte in the buffer written.

A buffer read is a series of *Byte Request* Word Serial queries sent to the Servant, with the additional constraint that the Data Out Ready (DOR) bit as well as the WR bit must be asserted before sending the *Byte Request*. The lower 8 bits (bits 0 to 7) of the 16-bit response contain a single byte of data (bit 8 is the END bit). Therefore, one *Byte Request* is sent for each data byte in the buffer read.

In addition to polling the WR, RR, DIR, and DOR bits during various Word Serial transfers, the functions also check the ERR* bit. The ERR* bit indicates when a Word Serial Protocol error occurs. The Word Serial Protocol errors are: Unsupported Command, Multiple Query Error (MQE), DIR Violation, DOR Violation, RR Violation, or WR Violation. The Servant Word Serial Protocol functions let the local CPU generate any of the Word Serial Protocol errors and respond to the *Read Protocol Error* Word Serial query with the corresponding protocol error. The functions automatically handle asserting and deasserting of the ERR* bit.

The Longword Serial and Extended Longword Serial Protocols are similar to the Word Serial Protocol, but involve 32-bit and 48-bit command transfers, respectively, instead of the 16-bit transfers of the Word Serial Protocol. The VXI specification, however, provides no common command usages for these protocols. The commands are either VXI Reserved or User-Defined. The NI-VXI interface gives you the ability to receive and process any one of these commands.

Programming Considerations

Most of the Servant Word Serial functions require an interrupt handler. The word serial commands must be parsed (and responded to) within the appropriate interrupt handler. Word Serial commands *Byte Available* (BAV) and *Byte Request* (BREQ) are handled as a special case for reads and writes. For reads and writes, a user-supplied handler is notified only that the transfer is complete and not for each byte processed. Asserting and unasserting of all Response register bits

(DIR, DOR, WR, RR, and ERR*) are done automatically within the functions as required.

Figure 3-2 provides a graphical overview of the Servant Word Serial functions.

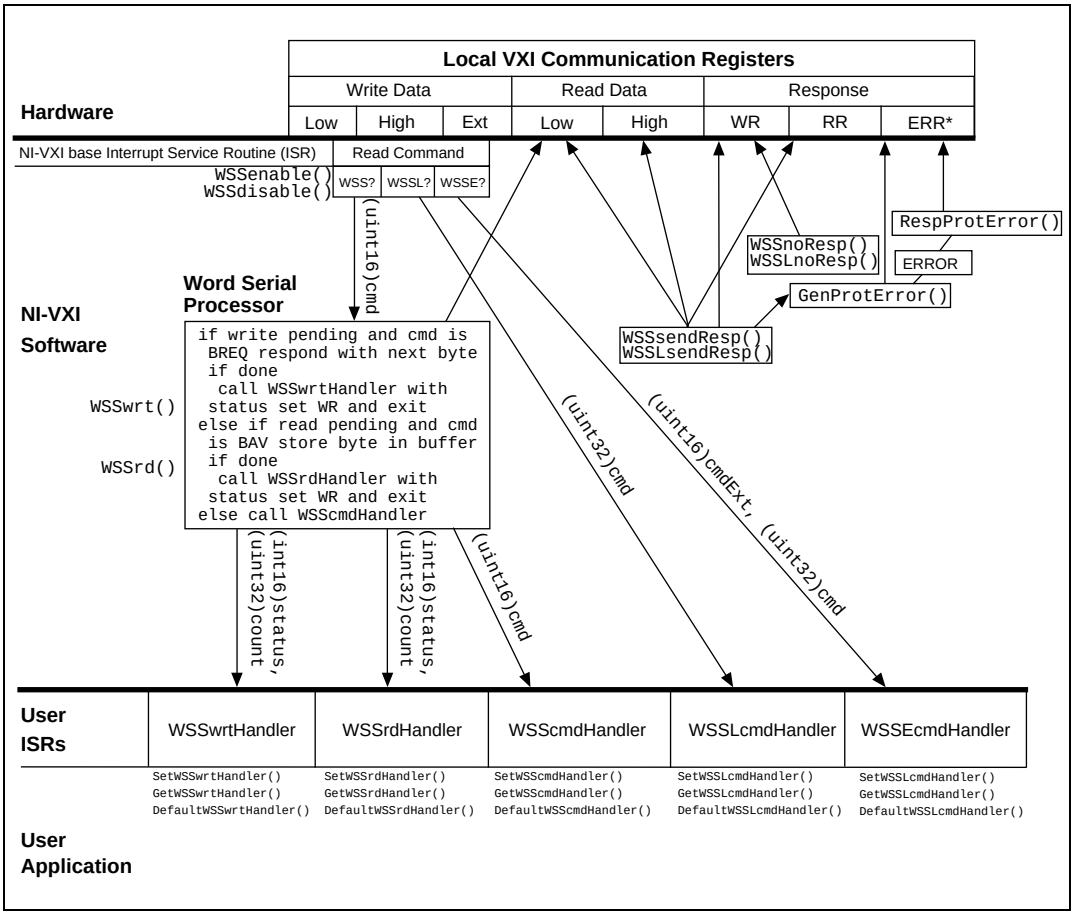


Figure 3-2. NI-VXI Servant Word Serial Model

The following paragraphs describe the Servant Word Serial, Longword Serial, and Extended Longword Serial Protocol functions. The descriptions are grouped by functionality and are presented at a functional level describing the operation of each of the functions.

DefaultWSScmdHandler (cmd)

DefaultWSScmdHandler is the default handler for the WSSwrt interrupt, which InitVXIlibrary automatically installs when it initializes the NI-VXI software. The current WSScmdHandler is called whenever the local CPU Commander sends any Word Serial Protocol command or query (other than BAV or BREQ). While Word Serial operations are enabled, the WSScmd interrupt handler is called every time a Word Serial command is received (other than BAV if a WSSrd call is pending, or BREQ if a WSSwrt call is pending). DefaultWSScmdHandler parses the commands and takes appropriate action. If it is a query, it returns a response using the WSSsendResp function. If it is a command, it calls the WSSnoResp function to acknowledge it. If either a BREQ or BAV command is received via this handler, it calls GenProtError with the corresponding protocol error code (DOR violation or DIR violation). For unsupported commands, the protocol error code sent to GenProtError is UnSupCom.

DefaultWSSEcmdHandler (cmdExt, cmd)

DefaultWSSEcmdHandler is the default handler for the WSSwrt interrupt, which InitVXIlibrary automatically installs when it initializes the NI-VXI software. The current WSSEcmdHandler is called whenever the local CPU Commander sends any Extended Longword Serial Protocol command or query. While Word Serial operations are enabled, the WSSEcmdHandler is called whenever a Longword Serial command is received. WSSEcmdHandler must parse the commands and take the appropriate action. Because the VXI specification does not define any Extended Longword Serial commands, DefaultWSSEcmdHandler calls GenProtError with a protocol error code of UnSupCom for every Extended Longword Serial command received.

DefaultWSSLcmdHandler (cmd)

DefaultWSSLcmdHandler is the default handler for the WSSwrt interrupt, which InitVXIlibrary automatically installs when it initializes the NI-VXI software. The current WSSLcmdHandler is called whenever the local CPU Commander sends any Longword Serial Protocol command or query. While Word Serial operations are enabled, the WSSLcmdHandler is called whenever a Longword Serial command is received. The WSSLcmdHandler must parse the commands and take the appropriate action. Because the VXI specification does not define any Longword Serial commands,

DefaultWSSLcmdHandler calls GenProtError with a protocol error code of UnSupCom for every Longword Serial command received.

DefaultWSSrdHandler (status, count)

DefaultWSSrdHandler is the default handler for the WSSrd interrupt, which InitVXIlibrary automatically installs when it initializes the NI-VXI software. When WSSrd reaches the specified count or an END bit, or an error occurs, it calls the WSSrd interrupt handler with the status of the call. The default handler sets the global variables WSSrdDone, WSSrdDoneStatus, and WSSrdDoneCount. You can use the variable WSSrdDone to poll until the operation is complete. Afterwards, you can inspect WSSrdDoneStatus and WSSrdDoneCount to see the outcome of the call. If you want, you can use the SetWSSrdHandler function to install an alternate handler.

DefaultWSSwrtHandler (status, count)

DefaultWSSwrtHandler is the default handler for the WSSwrt interrupt, which InitVXIlibrary automatically installs when it initializes the NI-VXI software. When WSSwrt reaches the specified count or an error occurs, it calls the WSSwrt interrupt handler with the status of the call. The default handler sets the global variables WSSwrtDone, WSSwrtDoneStatus, and WSSwrtDoneCount. You can use the variable WSSwrtDone to poll until the operation is complete. Afterwards, you can inspect WSSwrtDoneStatus and WSSwrtDoneCount to see the outcome of the call. If you want, you can use the SetWSSwrtHandler function to install an alternate handler.

GenProtError (proterr)

In response to a Word Serial Protocol Error, the application should call GenProtError to generate the error. Generating the error consists of preparing the response to a future *Read Protocol Error* query (saving the value in a global variable) and setting the ERR* bit in the local Response register. The RespProtError function actually generates the response when the *Read Protocol Error* query is received later.

GetWSScmdHandler ()

GetWSScmdHandler returns the address of the current WSScmd interrupt handler function. While Word Serial operations are enabled, the WSScmd interrupt handler is called whenever a Word Serial command (other than BREQ and BAV) is received.

GetWSSEcmdHandler ()

`GetWSSEcmdHandler` returns the address of the current `WSSEcmd` interrupt handler function. While Word Serial operations are enabled, the `WSSEcmd` interrupt handler will be called every time an Extended Longword Serial command is received.

GetWSSLcmdHandler ()

`GetWSSLcmdHandler` returns the address of the current `WSSLcmd` interrupt handler function. While Word Serial operations are enabled, the `WSSLcmd` interrupt handler is called whenever a Longword Serial command is received.

GetWSSrdHandler ()

`GetWSSrdHandler` returns the address of the current `WSSrd` interrupt handler function. When `WSSrd` reaches the specified count or an END bit, or an error occurs, it calls the `WSSrd` interrupt handler with the status of the call.

GetWSSwrtHandler ()

`GetWSSwrtHandler` returns the address of the current `WSSwrt` interrupt handler function. When `WSSwrt` reaches the specified count or an error occurs, it calls the `WSSwrt` interrupt handler with the status of the call.

RespProtError ()

When the Word Serial *Read Protocol Error* query is received, `RespProtError` places the saved error response in the Data Low register, sets the saved error response to ffffh (no error), unasserts ERR*, and sets RR. If no previous error is pending, the value ffffh (no error) is returned.

SetWSScmdHandler (func)

`SetWSScmdHandler` replaces the current `WSScmd` interrupt handler with an alternate handler. While Word Serial operations are enabled, the `WSScmd` interrupt handler is called whenever a Word Serial command is received (other than BAV if a `WSSrd` call is pending, or BREQ if a `WSSwrt` call is pending). A default handler, `DefaultWSScmdHandler`, is supplied in source code as an example, and is automatically installed when `InitVXIlibrary` is called. The

default handler provides examples of how to parse commands, respond to queries, and generate protocol errors.

SetWSSEcmdHandler (func)

`SetWSSEcmdHandler` replaces the current `WSSEcmd` interrupt handler with an alternate handler. While Word Serial operations are enabled, the `WSSEcmd` interrupt handler is called whenever an Extended Longword Serial command is received. A default handler, `DefaultWSSEcmdHandler`, is supplied in source code as an example, and is automatically installed when `InitVXIlibrary` is called.

SetWSSLcmdHandler (func)

`SetWSSLcmdHandler` replaces the current `WSSLcmd` interrupt handler with an alternate handler. While Word Serial operations are enabled, the `WSSLcmd` interrupt handler is called whenever a Longword Serial command is received. A default handler, `DefaultWSSLcmdHandler`, is supplied in source code as an example, and is automatically installed when `InitVXIlibrary` initializes the NI-VXI software.

SetWSSrdHandler (func)

`SetWSSrdHandler` replaces the current `WSSrd` interrupt handler with an alternate handler. When `WSSrd` reaches the specified count or an END bit, or an error occurs, it calls the `WSSrd` interrupt handler with the status of the call. A default handler, `DefaultWSSrdHandler`, is automatically installed when `InitVXIlibrary` is called. The default handler puts the status and read count in a global variable and flags the operation complete.

SetWSSwrtHandler (func)

`SetWSSwrtHandler` replaces the current `WSSwrt` interrupt handler with an alternate handler. When `WSSwrt` reaches the specified count or an error occurs, it calls the `WSSwrt` interrupt handler with the status of the call. The DOR bit will be cleared before WR is set on the last byte of transfer. `InitVXIlibrary` automatically installs a default handler, `DefaultWSSwrtHandler`, when it initializes the NI-VXI software. The default handler puts the status and read count in a global variable and flags the operation complete.

WSSabort (abortop)

`WSSabort` aborts the Servant Word Serial operation(s) in progress. It returns an error code of `ForcedAbort` to the `WSSrd` or `WSSwrt` interrupt handlers in response to the corresponding pending functions. This may be necessary if the application needs to abort for some application-specific reason, or if the Commander of this device sends a Word Serial *Clear*, *End Normal Operation*, or *Abort* command.

WSSdisable ()

`WSSdisable` disables all Servant Word Serial functions from being used. More precisely, this function desensitizes the local CPU to interrupts generated when writing a Word Serial command to the Data Low register or reading a response from the Data Low register.

WSSenable ()

`WSSenable` enables all Servant Word Serial functions. More precisely, this function sensitizes the local CPU to interrupts generated when writing a Word Serial command to the Data Low register or reading a response from the Data Low register. By default, the Servant Word Serial functions are disabled. At any time after `InitVXIlibrary` initializes the NI-VXI software, you can call `WSSenable` to set up processing of Servant Word Serial commands and queries.

WSSLnoResp ()

`WSSLnoResp` sets the WR bit so that it is ready to accept any further Longword Serial commands. The `WSSLcmd` interrupt handler should call `WSSLnoResp` after processing a Longword Serial command (it calls `WSSLsendResp` for Longword Serial queries).

WSSLsendResp (response)

`WSSLsendResp` responds to a Longword Serial Protocol query from a VXI message-based Commander device. The `WSSLcmd` interrupt handler calls this function to respond to a Longword Serial query. If a previous response has not been read yet, a `WSSLsendResp` call generates a Multiple Query Error (MQE). Otherwise, it writes a response value to the Data High and Data Low registers and sets the RR bit. It also sets the WR bit so that it is ready to accept any further Word Serial commands.

WSSnoResp ()

`WSSnoResp` sets the WR bit so that it is ready to accept any further Word Serial commands. The `WSScmd` interrupt handler should call `WSSnoResp` after processing a Word Serial command (it calls `WSSsendResp` for a Word Serial query, which requires a response).

WSSrd (buf, count, modevalue)

`WSSrd` is the buffer read function. `WSSrd` receives a specified number of bytes from a VXI message-based Commander device and places the bytes into a memory buffer, using the VXIbus Byte Transfer Protocol. The process involves setting the DIR and WR bits on the local CPU Response register and building a buffer out of data bytes received via a series of *Byte Available* (BAV) Word Serial commands. When `WSSrd` reaches the specified count or an END bit, or an error occurs, it calls the `WSSrd` interrupt handler with the status of the call. It clears the DIR bit before setting the WR on the last byte of transfer.

WSSsendResp (response)

`WSSsendResp` responds to a Word Serial Protocol query from a VXI message-based Commander device. The `WSScmd` interrupt handler calls this function to respond to a Word Serial query. If a previous response has not been read yet, a `WSSsendResp` call generates a Multiple Query Error (MQE). Otherwise, it writes a response value to the Data Low register and sets the RR bit is. It also sets the WR bit so that it is ready to accept any further Word Serial commands.

WSSwrt (buf, count, modevalue)

`WSSwrt` sends a specified number of bytes to a VXI message-based Commander device, using the VXIbus Byte Transfer Protocol. The process involves setting the DOR and WR bits in the local Response register and responding to a series of *Byte Request* (BREQ) Word Serial commands. When the data output completes or an error occurs, `WSSwrt` calls its interrupt handler with the status of the call. Before responding to the last byte of the write, it clears DOR to prevent another BREQ from being sent before the application is able to handle the BREQ properly.

High-Level VXI/VMEbus Access Functions

You can use both low-level and high-level VXI/VMEbus access functions to read or write to VXI/VMEbus addresses. These are required in many situations, including the following:

- Register-based device/instrument drivers
- Non-VXI/VME device/instrument drivers
- Accessing device-dependent registers on any type of VXI/VME device
- Implementing shared memory protocols

Low-level and high-level access to the VXI/VMEbus, as the NI-VXI interface defines them, are very similar. Both sets of functions can perform direct reads of and writes to any VXI/VMEbus address space with any privilege state or byte order. However, the two interfaces have different emphases with respect to user protection, error checking, and access speed. For example, your application must check error conditions such as Bus Error (BERR*) separately when using low-level accesses.

High-level VXI/VMEbus access functions need not take into account any of the considerations that are required by the low-level VXIbus access functions. The high-level VXI/VMEbus access functions have all necessary information for accessing a particular VXI/VMEbus address wholly contained within the function parameters. The parameters prescribe the address space, privilege state, byte order, and offset within the address space. High-level VXI/VMEbus access functions automatically trap bus errors and return an appropriate error status. Using the high-level VXI/VMEbus access functions involves more overhead, but if overall throughput of a particular access (for example, configuration or small number of accesses) is not the primary concern, the high-level VXI/VMEbus access functions act as an easy-to-use interface for VXI/VMEbus accesses.

Programming Considerations

All accesses to the VXI/VMEbus address spaces performed by use of the high-level VXI/VMEbus access functions are fully protected. The hardware interface settings (*context*) for the applicable window are saved on entry to the function and restored upon exit. No other functions in the NI-VXI interface, including the low-level VXI/VMEbus access functions, will conflict with the high-level

VXI/VMEbus access functions. You can use both high-level and low-level VXI/VMEbus access functions at the same time.

The following paragraphs describe the high-level VXI/VMEbus access functions.

VXIin (accessparms, address, width, value)

`VXIin` reads a single byte, word, or longword from a particular VXI/VME address in one of the VXI address spaces. The parameter **accessparms** specifies the VXI/VME address space, the VXI privilege access, and the byte order to use with the access. The **address** parameter specifies the offset within the particular VXI/VME address space. The **width** parameter selects either byte, word, or longword transfers. The value read from the VXI/VMEbus returns in the output parameter **value**. If the VXI/VME address selected has no device residing at the address and a bus error occurs, `VXIin` traps the bus error condition and indicates the error through the return status.

VXIinReg (la, reg, value)

`VXIinReg` reads a single 16-bit value from a particular VXI device's VXI registers within the logical address space (the upper 16 KB of VXI A16 address space). The function sets the VXI access privilege to Nonprivileged Data and the byte order to Motorola. If the VXI address selected has no device residing at the address and a bus error occurs, `VXIinReg` traps the bus error condition and indicates the error through the return status. This function is mainly for convenience and is a layer on top of `VXIinLR` and `VXIin`. If the **la** specified is the local CPU logical address, it calls the `VXIinLR` function. Otherwise, it calculates the A16 address of the VXI device's register and calls `VXIin`.

☞ **Note:** `VXIinReg` ***is designed to access a VXIbus device configuration register and therefore is not applicable to VME devices.***

VXImove (srcparms, srcaddr, destparms, destaddr, length, width)

`VXImove` moves a block of bytes, words, or longwords from a particular address in one of the available address spaces (local, A16, A24, A32) to any other address in any one of the address spaces. The parameters **srcparms** and **destparms** specify the address space, the privilege access, and the byte order used to perform the access for the source address and the destination address, respectively. The **srcaddr** and **destaddr** parameters specify the offset within the particular

address space for the source and destination, respectively. The **width** parameter selects either byte, word, or longword transfers. If one of the addresses selected has no device residing at the address and a bus error occurs, `VXImove` traps the bus error condition and indicates the error through the return status.

VXIout (accessparms, address, width, value)

`VXIout` writes a single byte, word, or longword to a particular VXI/VME address in one of the VXI/VME address spaces. The parameter **accessparms** specifies the VXI address space, the VXI privilege access, and the byte order to use with the access. The **address** parameter specifies the offset within the particular VXI/VME address space. The **width** parameter selects either byte, word, or longword transfers. If the VXI/VME address selected has no device residing at the address and a bus error occurs, `VXIout` traps the bus error condition and indicates the error through the return status.

VXIoutReg (la, reg, value)

`VXIoutReg` writes a single word to a particular VXI device's VXI registers within the logical address space (the upper 16 KB of VXI A16 address space). The function sets the VXI access privilege to Nonprivileged Data and the byte order to Motorola. If the VXI address selected has no device residing at the address and a bus error occurs, `VXIinReg` traps the bus error condition and indicates the error through the return status. This function is mainly for convenience and is a layer on top of `VXIoutLR` and `VXIout`. If the **la** specified is the local CPU logical address, it calls the `VXIoutLR` function. Otherwise, it calculates the A16 address of the VXI device's register and calls `VXIout`.

☞ **Note:** `VXIoutReg` ***is designed to access a VXIbus device configuration register and therefore is not applicable to VME devices.***

Low-Level VXI/VMEbus Access Functions

This section describes the use of the low-level VXI/VMEbus access functions. You can use both low-level and high-level VXI/VMEbus access functions to directly read or write to VXI/VMEbus addresses. Some of the situations that require direct reads and writes to the different VXI/VMEbus address spaces include the following:

- Register-based device/instrument drivers
- Non-VXI device/instrument drivers
- Accessing device-dependent registers on any type of VXI/VME device
- Implementing shared memory protocols

Low-level and high-level access to the VXI/VMEbus, as the NI-VXI interface defines them, are very similar in nature. Both sets of functions can perform direct reads of and writes to any VXI/VMEbus address space with any privilege state or byte order. However, the two interfaces have different emphases with respect to user protection, error checking, and access speed.

Low-level VXI/VMEbus access is the fastest access method (in terms of overall throughput to the device) for directly reading or writing to/from any of the VXI/VMEbus address spaces with random memory accesses. Under many platforms, the high-level operation `VXImove` provides the fastest access in terms of block moves. As such, however, it is more detailed and leaves more issues for the application to resolve. You can use these functions to obtain pointers that are directly mapped to a particular VXI/VMEbus address with a particular VXI/VME access privilege and byte ordering. You need to consider a number of issues when using the direct pointers:

- You need to determine bounds for the pointers.
- Based on the methods in which a particular hardware platform sets up access to VXI/VME address spaces, using more than one pointer can result in conflicts.
- Your application must check error conditions such as Bus Error (BERR*) separately.

Programming Considerations

All accesses to the VXI/VMEbus address spaces are performed by reads and writes to particular offsets within the local CPU address space, which are made to correspond to addresses on the VXI/VMEbus (using a hardware interface). The areas where the address space of the local CPU is mapped onto the VXI/VMEbus are referred to as *windows*. The sizes and numbers of windows present vary depending on the hardware being used. The size of the window is always a power of two, where a multiple of the size of the window would encompass an entire VXI/VMEbus address space. The multiple for which a window currently can access is determined by modifying a *window base* register.

The constraints of a particular hardware platform lead to restrictions on the area of address space reserved for windows into VXI/VMEbus address spaces. Be sure to take into account the number and size of the windows provided by a particular platform. If a mapped pointer is to be incremented or decremented, the bounds for accessing within a particular address space must be tested before accessing within the space.

NI-VXI uses a term within this chapter called the hardware (or window) *context*. The hardware context for window to VXI/VME consists of the VXI/VME address space being accessed, the base offset into the address space, the access privilege, and the byte order for the accesses through the window. Before accessing a particular address, you must set up the window with the appropriate hardware context. You can use the `MapVXIAddress` function for this purpose. This function returns a pointer that you can use for subsequent accesses to the window with the `VXIpeek` and `VXIpoke` functions.

On most systems, `VXIpeek` and `VXIpoke` are really C macros (`#defines`) that dereference the pointer. It is highly recommended to use these functions instead of performing the direct dereference within the application. If your application does not use `VXIpeek` and `VXIpoke`, it might not be portable between different platforms. In addition, `VXIpeek` and `VXIpoke` allow for compatibility between the C language and other languages such as BASIC.

Multiple-Pointer Access for a Window

Application programmers can encounter a potential problem when the application requires different privilege states, byte orders, and/or base addresses within the same window. If the hardware context changes due to a subsequent call to `MapVXIAddress` or other calls such as `SetPrivilege` or `SetByteOrder`, previously mapped pointers would not have their intended access parameters. This problem is greater in a multitasking system, where independent and conflicting processes can change the hardware context. Two types of access privileges to a window are available to aid in solving this problem: *Owner Privilege*, and *Access-Only Privilege*. These two privileges define which caller of the `MapVXIAddress` function can change the settings of the corresponding window.

Owner Privilege

A caller can obtain Owner Privilege to a window by requesting owner privilege in the `MapVXIAddress` call (via the **accessparms** parameter). This call will not succeed if another process already has either Owner Privilege or Access-Only Privilege to that window. If the call succeeds, the function returns a valid pointer and a non-negative return value. The 32-bit **windowId** output parameter returned from the `MapVXIAddress` call associates the C pointer returned from the function with a particular window and also signifies Owner Privilege to that window. Owner Privilege access is complete and exclusive. The caller can use `SetPrivilege`, `SetByteOrder`, and `SetContext` with this **windowId** to dynamically change the access privileges.

Notice that if the call to `MapVXIAddress` succeeds for either Owner Privilege or Access-Only Privilege, the pointer remains valid in both cases until an explicit `UnMapVXIAddress` call is made for the corresponding window. The pointer is guaranteed to be a valid pointer in either multitasking systems or nonmultitasking systems. The advantage with Owner Privilege is that it gives complete and exclusive access for that window to the caller, so you can dynamically change the access privileges. Because no other callers can succeed, there is no problem with either destroying another caller's access state or having an inconsistent pointer environment.

Access-Only Privilege

A process can obtain Access-Only Privilege by requesting access-only privileges in the `MapVXIAddress` call. With this privilege mode, you

can have multiple pointers in the same process or over multiple processes to access a particular window simultaneously, while still guaranteeing that the hardware context does not change between accesses. The call succeeds under either of the following conditions:

- No processes are mapped for the window (first caller for Access-Only Privilege for this window). The hardware context is set as requested in the call. The call returns a successful status and a valid C pointer and **windowId** for Access-Only Privilege.
- No process currently has Owner Privilege to the required window. There *are* processes with Access-Only Privilege, but they are using the same hardware context (privilege state, byte order, address range) for their accesses to the window. Because the hardware context is compatible, it does not need to be changed. The call returns a successful status and a valid C pointer and **windowId** for Access-Only Privilege.

The successful call returns a valid pointer and a non-negative return value. The 32-bit window number signifies that the access privileges to the window are Access-Only Privilege.

With Access-Only Privilege, you cannot use the `SetPrivilege`, `SetByteOrder`, and `SetContext` calls in your application to dynamically change the hardware context. No Access-Only accessor can change the state of the window. The initial Access-Only call sets the hardware context for the window, which cannot be changed until all Access-Only accessors have called `UnMapVXIAddress` to free the window. The functions `GetPrivilege`, `GetByteOrder`, and `GetContext` will succeed regardless of whether the caller has Owner Privilege or Access-Only Privilege.

The following paragraphs describe the low-level VXIbus access functions. The descriptions are presented at a functional level describing the operation of each of the functions. The functions are grouped by area of functionality.

☞ **Note:** ***On MITE-based platforms,*** `MapVXIAddress` ***cannot be called while the CPU is in interrupt context. For this reason, it is recommended that you not use the*** `SaveContext` ***and*** `RestoreContext` ***functions. Due to the multiple window support of the MITE, you should not need these functions.***

GetByteOrder (window, ordermode)

`GetByteOrder` retrieves the byte/word order of data transferred into or out of the specified window. The two possible settings are Motorola (most significant byte/word first) or Intel (least significant byte/word first). The application can have either Owner-Access Privilege or Access-Only Privilege to the applicable window for this function to execute successfully.

GetContext (window, context)

`GetContext` retrieves all of the hardware interface settings (context) for a particular VXI/VME window. The application can have either Owner Access Privilege or Access-Only Privilege to the applicable window for this function to execute successfully. Any application can use `GetContext` along with `SetContext` to save and restore the VXI/VME interface hardware state (context) for a particular window.

GetPrivilege (window, priv)

`GetPrivilege` retrieves the current windowing hardware VXI/VMEbus access privileges for the specified window. The possible privileges include Nonprivileged Data, Supervisory Data, Nonprivileged Program, Supervisory Program, Nonprivileged Block, and Supervisory Block access. The application can have either Owner-Access Privilege or Access-Only Privilege to the applicable window for this function to execute successfully.

GetVXIbusStatus (controller, status)

`GetVXIbusStatus` retrieves information about the current state of the VXI/VMEbus.

The information that is returned includes the state of the Sysfail, ACfail, VXI/VME interrupt, TTL trigger, and ECL trigger lines as well as the number of VXI signals on the global signal queue. This information returns in a C structure containing all of the known information. An individual hardware platform might not support all of the different hardware signals polled. In this case, a value of -1 is returned for the corresponding field in the structure. Interrupt service routines can automatically handle all of the conditions retrieved from this function, if enabled to do so. You can use this function for simple polled operations.

GetVXIbusStatusInd (controller, field, status)

`GetVXIbusStatusInd` retrieves information about the current state of the VXI/VMEbus.

The information that can be returned includes the state of the Sysfail, ACfail, VXI interrupt, TTL trigger, or ECL trigger lines as well as the number of VXI signals on the global signal queue. The specified information returns in a single integer value. The **field** parameter specifies the particular VXI/VMEbus information to be returned. An individual hardware platform might not support the specified hardware signals polled. In this case, a value of -1 is returned in **status**. Interrupt service routines can automatically handle all of the conditions retrieved from this function, if enabled to do so. You can use this function for simple polled operations.

GetWindowRange (window, windowbase, windowend)

`GetWindowRange` retrieves the range of addresses that a particular VXI/VMEbus window can currently access within a particular VXI/VMEbus address space. The **windowbase** and **windowend** output parameters are based on VXI/VME addresses (not local CPU addresses). The **window** parameter value should be the value returned from a `MapVXIAddress` call. The VXI/VME address space being accessed is inherent in the **window** parameter.

MapVXIAddress (accessparms, address, timo, window, ret)

`MapVXIAddress` sets up a window into one of the VXI/VME address spaces and returns a pointer to a local address that will access the specified VXI/VME address. The **accessparms** parameter specifies Owner Privilege/Access-Only Privilege, the VXI/VME address space, the VXI/VME access privilege, and the byte ordering. The value of the **timo** parameter gives the time (in milliseconds) that the process will wait checking for window availability. The function returns immediately if the window is already available, or if the **timo** value is 0. The **timo** field is ignored in a uniprocess (nonmultitasking) system. The return value in **window** gives a unique window identifier that various calls such as `GetWindowRange` or `GetContext` use to get window settings. When a request for Owner Privilege is granted, you can also use this window identifier with calls such as `SetContext` or `SetPrivilege` to change the hardware context for that window.

MapVXIAddressSize (size)

`MapVXIAddressSize` sets the size for mapping user windows. The subsequent calls to `MapVXIAddress` will attempt to map a window of the size passed to `MapVXIAddressSize`. `MapVXIAddressSize` only provides a preferred size to the `MapVXIAddress`. If it is not possible to map a window to the given size, `MapVXIAddress` can use a different size. To determine the exact size of window mapped, use the `GetWindowRange` function.

☞ **Note:** ***Not all platforms support*** `MapVXIAddressSize`**.**

SetByteOrder (window, ordermode)

`SetByteOrder` sets the byte/word order of data transferred into or out of the specified window. The two possible settings are Motorola (most significant byte/word first) or Intel (least significant byte/word first). The application must have Owner-Access Privilege to the applicable window for this function to execute successfully. Notice that some hardware platforms do not allow you to change the byte order of a window, which is reflected in the return code of the call to `SetByteOrder`. Most Intel processor-based hardware platforms support both byte order modes. Most Motorola processor-based hardware platforms support only the Motorola byte order mode, because the VXI/VMEbus is based on Motorola byte order.

SetContext (window, context)

`SetContext` sets all of the hardware interface settings (context) for a particular VXI/VME window. The application must have Owner-Access Privilege to the applicable window for this function to execute successfully. Any application can use `GetContext` along with `SetContext` to save and restore the VXI/VME interface hardware state (context) for a particular window. As a result, the application can set the hardware context associated with a particular pointer into VXI/VME address spaces (obtained from `MapVXIAddress`). After making a `MapVXIAddress` call for Owner Access to a particular window (and possibly calls to `SetPrivilege` and `SetByteOrder`), you can call `GetContext` to save this context for later restoration by `SetContext`.

SetPrivilege (window, priv)

`SetPrivilege` sets the VXI/VMEbus windowing hardware to access the specified window with the specified VXI/VMEbus access privilege. The possible privileges include Nonprivileged Data, Supervisory Data, Nonprivileged Program, Supervisory Program, Nonprivileged Block, and Supervisory Block access. The application must have Owner-Access Privilege to the applicable window for this function to execute successfully. Notice that some platforms may not support all of the privilege states. This is reflected in the return code of the call to `SetPrivilege`. Nonprivileged Data transfers must be supported within the VXI/VME environment, and are supported on all hardware platforms.

UnMapVXIAddress (window)

`UnMapVXIAddress` reallocates the window mapped using the `MapVXIAddress` function. If the caller is an Owner-Privilege accessor (only one is permitted), the window is free to be remapped. If the caller is an Access-Only Privilege accessor, the window can be remapped only if the caller is the last Access-Only accessor. After a call is made to `UnMapVXIAddress`, the pointer obtained from `MapVXIAddress` is no longer valid. You should no longer use the pointer because a subsequent call may have changed the settings for the particular window, or the window may no longer be accessible at all.

VXIpeek (addressptr, width, value)

`VXIpeek` reads a single byte, word, or longword from a particular address obtained by `MapVXIAddress`. On most platforms using C language interfaces, `VXIpeek` is a macro. It is recommended, however, that you use `VXIpeek` instead of a direct dereference of the pointer, as it supports portability between different platforms and programming languages.

VXIpoke (addressptr, width, value)

`VXIpoke` writes a single byte, word, or longword to a particular address obtained by `MapVXIAddress`. On most platforms using C language interfaces, `VXIpoke` is a macro. It is recommended, however, that you use `VXIpoke` instead of a direct dereference of the pointer, as it supports portability between different platforms and programming languages.

Local Resource Access Functions

Local resources are hardware and/or software capabilities that are reserved for the local CPU (the CPU on which the NI-VXI interface resides). You can use these functions to gain access to miscellaneous local resources such as the local CPU register set and the local CPU Shared RAM. These functions are useful for shared memory type communication, non-Resource Manager operation, and debugging purposes.

The following paragraphs describe the local resource access functions. The descriptions are presented at a functional level describing the operation of each of the functions. The functions are grouped by area of functionality.

GetMyLA

`GetMyLA` retrieves the logical address of the local VXI/VME device. The local CPU VXI/VME logical address is required for retrieving configuration information with one of the `GetDevInfo` functions. The local CPU VXI/VME logical address is also required for creating correct VXI signal values to send to other devices.

ReadMODID (modid)

`ReadMODID` senses the MODID line drivers of the local CPU when configured as a VXI Slot 0 device. The **modid** output parameter returns the polarity of each of the slot's MODID lines.

SetMODID (enable, modid)

`SetMODID` controls the MODID line drivers of the local CPU when configured as a VXI Slot 0 device. The **enable** parameter enables the MODID drivers for all the slots. The **modid** parameter specifies which slots should have their corresponding MODID lines asserted.

☞ **Note:** ***The*** `MODID` ***lines are unique to the VXIbus and has no meaning on a VMEbus.***

VXIinLR (reg, width, value)

VXIinLR reads a single byte, word, or longword from the local CPU VXI/VME registers. On many CPUs, the local CPU VXI/VME registers cannot be accessed from the local CPU in the A16 address space window (due to hardware limitations). Another area in the local CPU address space is reserved for accessing the local CPU VXI registers. VXIinLR is designed to read these local registers. The VXI/VME access privilege is not applicable but can be assumed to be Nonprivileged Data. The byte order is Motorola. Unless otherwise specified, reads should always be performed as words. This function can be used to read configuration information (manufacturer, model code, and so on) for the local CPU.

VXImemAlloc (size, useraddr, vxiaddr)

VXImemAlloc allocates physical RAM from the operating system's dynamic memory pool. This RAM will reside in the VXI/VME Shared RAM region of the local CPU. VXImemAlloc returns not only the user address that the application uses, but also the VXI/VME address that a remote device would use to access this RAM. This function is very helpful on virtual memory systems, which require contiguous, locked-down blocks of virtual-to-physical RAM. On non-virtual memory systems, this function is a *malloc* (standard C dynamic allocation routine) and an address translation. When the application is finished using the memory, it should make a call to VXImemFree to return the memory to the operating system's dynamic memory pool.

VXImemCopy (useraddr, bufaddr, size, dir)

VXImemCopy copies blocks of memory to or from the local user's address space into the local shared memory region. On some interfaces, your application cannot directly access local shared memory. VXImemCopy gives you fast access to this local shared memory.

VXImemFree (useraddr)

VXImemFree reallocates physical RAM from the operating system's dynamic memory pool allocated using VXImemAlloc. VXImemAlloc returns not only the user address that the application uses, but also the VXI address that a remote device would use to access this RAM. When the application is through using the memory, it should make a call to VXImemFree (with the user address) to return the memory to the operating system's dynamic memory pool.

VXIoutLR (reg, width, value)

VXIoutLR writes a single byte, word, or longword to the local CPU VXI/VME registers. On many CPUs, the local CPU VXI/VME registers cannot be accessed from the local CPU in the A16 address space window (due to hardware limitations). Another area in the local CPU address space is reserved for accessing the local CPU VXI registers. VXIoutLR is designed to write to these local VXI/VME registers. The VXI/VME access privilege is not applicable but can be assumed to be Nonprivileged Data. The byte order is Motorola. Unless otherwise specified, writes should always be performed as words. This function can be used to write application specific registers (A24 pointer register, A32 pointer register, and so on) for the local CPU.

VXI Signal Functions

With these functions, VXI/VME bus master devices can interrupt another device. VXI signal functions can specify the signal routing, manipulate the global signal queue, and wait for a particular signal value (or set of values) to be received.

VXI signals are a basic form of asynchronous communication used by VXI/VME bus master devices. A VXI signal is a 16-bit value written to the Signal register of a VXI message-based device. Normally, the write to the Signal register generates a local CPU interrupt, and the local CPU then acquires the signal value in some device-specific manner. All National Instruments hardware platforms have a hardware FIFO to accumulate signal values while waiting for the local CPU to retrieve them. The format of the 16-bit signal value is defined by the VXIbus specification and is the same as the format used for the VXI interrupt status/ID word that is returned during a VXI interrupt acknowledge cycle. All VXI signals and status/ID values contain the VXI logical address of the sending device in the lower 8 bits of the VXI signal or status/ID value. The upper 8 bits of the 16-bit value depends on the VXI device type.

☞ **Note:** ***For VME bus master devices, the VXI signal register can be considered a general purpose notification register. Although the*** VXIbus ***specification defines the use for this register, you can program the application on the controller to respond to write to this register in any manner you require.***

VXI signals from message-based devices can be one of two types: *Response* signals and *Event* signals (bit 15 distinguishes between the two). *Response* signals are used to report changes in Word Serial communication status between a Servant and its Commander. *Event* signals are used to inform another device of other asynchronous changes. The four *Event* signals currently defined by the VXIbus specification (other than *Shared Memory* Events) are *No Cause Given*, *Request for Service True* (REQT), *Request for Service False* (REQF), and *Unrecognized Command*. REQT and REQF are used to manipulate the SRQ condition (RSV bit assertion in the IEEE 488/488.2 status byte) while *Unrecognized Command* is used to report unsupported Word Serial commands (only in VXIbus specification, Revision 1.2). If the sender of a signal (or VXI interrupt status/ID) value is a register-based device, the upper 8 bits are device dependent. Consult your device manual for definitions of these values.

Two methods are available to handle VXI signals under the NI-VXI software interface. Signals can be handled either by calling a handler or by queuing on a global signal queue. The `RouteSignal` function specifies which types of signals are handled by the handlers, and which are queued onto the global signal queue for each VXI logical address. A separate handler can be installed for each VXI logical address present (see the description of `SetSignalHandler`). The `InitVXIlibrary` function automatically installs a default handler, `DefaultSignalHandler`, for every VXI logical address. If signals are queued, the application can use the `SignalDeq` function to selectively retrieve a signal off a global signal queue by VXI logical address and/or type of signal.

In another method for handling signals (and VXI/VME interrupts routed to signals) other than the two previous methods, you can use the function `WaitForSignal`. This function can suspend a process/function until a particular signal (or one of a set of signals) arrives. A multitasking operating system lets you have any number of `WaitForSignal` calls pending. A non-multitasking operating system permits only one pending `WaitForSignal` call. Notice that even on a multitasking operating system, there is only one signal queue for the entire system. Therefore, if two applications both wait on the same logical address, it will be a race condition as to which process will receive the signal.

Programming Considerations

The global signal queue used to hold signal values is of a finite length. If the application is not handling signals fast enough, it is theoretically possible to fill the global signal queue. If the global signal queue becomes full, `DisableSignalInt` is called to inhibit more signals from being received. Under the VXIbus specification, if the local CPU signal FIFO becomes full (in which case a signal be lost if another signal is written), the local CPU must return a bus error on any subsequent writes to its Signal register. This bus error condition notifies the sending CPU that the signal transfer needs to be retried. This guarantees the application that, even if the global signal queue becomes full, no signals will be lost.

In addition to `DisableSignalInt`, the `DisableVXItoSignalInt` function is also called to disable VXI/VME interrupts from occurring on levels that are routed to the signal Processor. When `SignalDeq` is called to remove a signal from the global signal queue, the interrupts for the Signal register and the VXI/VME interrupt levels routed to the signal handler are automatically re-enabled.

If signals received never get dequeued, the global signal queue eventually becomes full and the interrupts will be disabled forever. If the signals were routed to the `DefaultSignalHandler`, all except *Unrecognized Command* Events from message-based devices perform no operation. *Unrecognized Command* Events call the function `WSabort` to abort the current Word Serial operation in progress.

Figure 3-3 provides a graphical overview of the NI-VXI interrupt and signal functions.

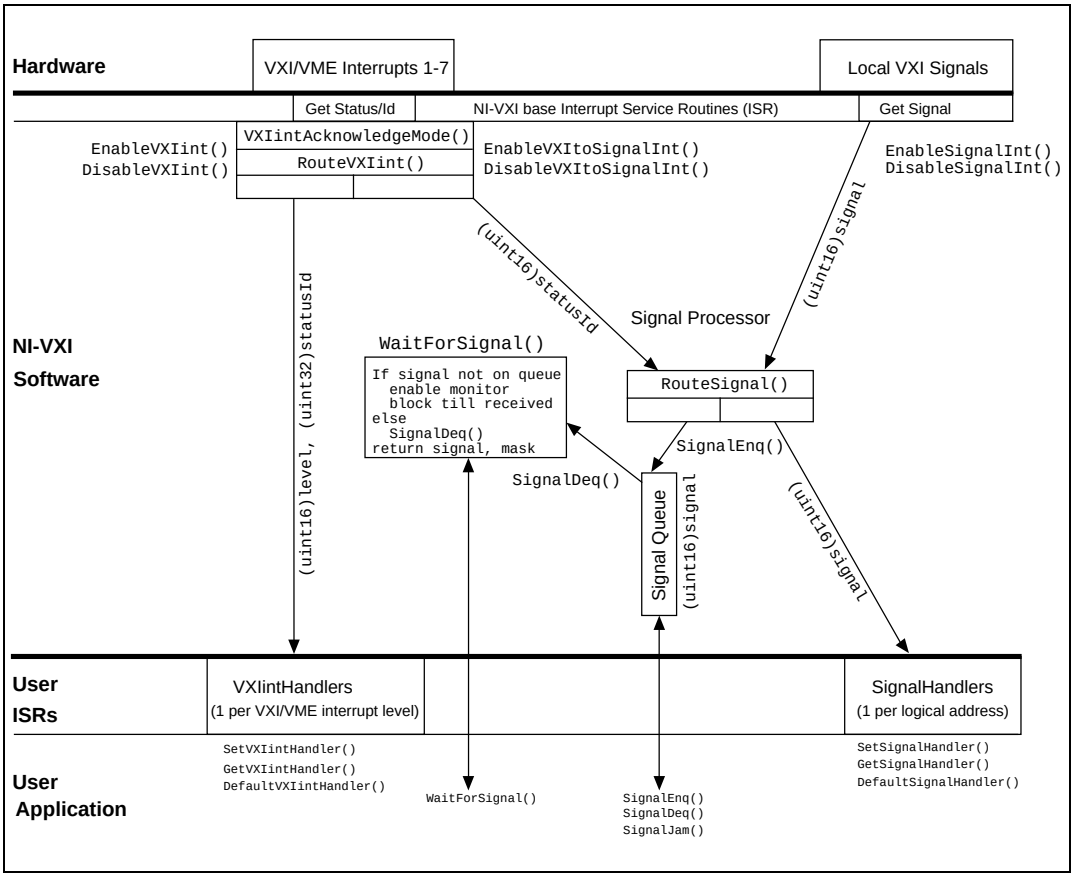


Figure 3-3. NI-VXI Interrupt and Signal Model

WaitForSignal Considerations

The function `WaitForSignal` can be used to suspend a process/function until a particular VXI signal (or one of a set of signals) arrives. Any signals to be waited on should be routed to the global signal queue. If the `RouteSignal` function has specified for the signal to be handled by the interrupt service routine, the `WaitForSignal` call will not detect that the signal and the process/function may block until a timeout. `WaitForSignal` attempts to dequeue a signal of the specified type before the process/function is suspended. If an appropriate signal can be dequeued, the signal is

returned immediately to the caller and the process/function is not suspended.

The following paragraphs describe the VXI signal functions and default handler. The descriptions are presented at a functional level describing the operation of each of the functions. The functions are grouped by area of functionality.

DefaultSignalHandler (signal)

`DefaultSignalHandler` is the sample handler for VXI signals that is installed when the `InitVXIlibrary` function is called for every applicable VXI logical address. The default handler performs no action on the signals except when *Unrecognized Command* Events are received. For these events, it calls the function `WSabort` with an **abortop** of `UnSupCom` to abort the current Word Serial transfer in progress.

DisableSignalInt ()

`DisableSignalInt` desensitizes the application to local signal interrupts. While signal interrupts are disabled, a write to the local CPU VXI Signal register does not cause an interrupt on the local CPU; instead, the local CPU hardware signal FIFO begins to fill up. If the hardware FIFO becomes full, bus errors will occur on subsequent writes to the Signal register. This function is automatically called when the global signal queue becomes full, and is automatically re-enabled on a call to `SignalDeq`. `DisableSignalInt` along with `EnableSignalInt` can be used to temporarily suspend the generation of signal interrupts.

EnableSignalInt ()

`EnableSignalInt` sensitizes the application to local signal interrupts. When signal interrupts are enabled, any write to the local CPU VXI Signal register causes an interrupt on the local CPU. The internal signal router then routes the signal value to the handler or to the global signal queue, as specified by the `RouteSignal` function. `EnableSignalInt` must be called after `InitVXIlibrary` to begin the reception of signals. Calls to `RouteSignal` and/or `SetSignalHandler` must be made before the signal interrupt is enabled to guarantee proper signal routing of the first signals.

GetSignalHandler (la)

`GetSignalHandler` returns the address of the current signal handler for the specified VXI logical address. If signal interrupts are enabled (via `EnableSignalInt`), the signal handler for a specific logical address is called if the `RouteSignal` function has been set up to route signals to the handler (as opposed to the global signal queue). The `InitVXIlibrary` function automatically installs a default handler, `DefaultSignalHandler`, for every VXI logical address.

RouteSignal (la, modemask)

`RouteSignal` specifies how to route VXI signals for the application. Two methods are available to handle VXI signals. You can handle the signals either at interrupt service routine time or by queueing on a global signal queue. For each VXI logical address, the `RouteSignal` function specifies which types of signals should be handled by the handlers, and which should be queued on the global signal queue. A separate handler can be installed for each VXI logical address present (see the description of `SetSignalHandler`). The `InitVXIlibrary` function automatically installs a default handler, `DefaultSignalHandler`, for every VXI logical address. If signals are queued, the application can use the `SignalDeq` or `WaitForSignal` function to selectively return a signal off a global signal queue by VXI logical address and/or type of signal. The default for `RouteSignal` is to have all signals routed to interrupt service routines.

SetSignalHandler (la, func)

`SetSignalHandler` replaces the current signal handler for the specified VXI logical address with an alternate handler. If signal interrupts are enabled (via `EnableSignalInt`), the signal handler for a specific logical address is called if the `RouteSignal` function has been set up to route signals to the handler (as opposed to the global signal queue). The `InitVXIlibrary` function automatically installs a default handler, `DefaultSignalHandler`, for every VXI logical address. The logical address (**la**) value of -2 is a special case and is provided to specify a handler to capture signals from devices not known to the device information table. This should occur only when the local CPU is not the Resource Manager or VME devices not listed in the Non-VXI Device Editor in `VXIedit`. Support is not provided to handle these signals via the global signal queue or the `WaitForSignal` function.

SignalDeq (la, signalmask, signal)

`SignalDeq` retrieves signals from the global signal queue. Two methods are available to handle VXI signals. You can handle the signals either by handlers or by queueing on a global signal queue. The `RouteSignal` function specifies which types of signals should be handled by which of the two methods for each VXI logical address. You can use `SignalDeq` to selectively dequeue a signal off of the global signal queue. The signal specified by **signalmask** for the specified logical address (**la**) is dequeued and returned in the output parameter **signal**.

SignalEnq (signal)

`SignalEnq` places signals at the end of the global signal queue. You can use `SignalEnq` within a signal handler to queue a signal or to simulate the reception of a signal by placing a value on the global signal queue that was not actually received as a signal.

SignalJam (signal)

`SignalJam` places signals at the front of the global signal queue. `SignalJam` can be used to simulate the reception of a signal by placing a value on the global signal queue that was not actually received as a signal. Because `SignalJam` places signal values on the front of the global signal queue, the signal is guaranteed to be the first of its type to be dequeued.

☞ **Note:** ***This function is intended only for debugging purposes.***

WaitForSignal (la, signalmask, timeout, retsignal, retsignalmask)

`WaitForSignal` waits for the specified maximum amount of time for a particular signal (or class of signals) to be received. `Signalmask` defines the type(s) of signals that the application program waits for. The **timeout** value specifies the maximum amount of time (in milliseconds) to wait until the signal occurs. The signal that unblocks the `WaitForSignal` call returns in the output parameter **retsignal**. You should use the `WaitForSignal` function only when signals are queued. A multitasking operating system lets you have any number of `WaitForSignal` calls pending. A non-multitasking operating system permits only one pending `WaitForSignal` call.

VXI Interrupt Functions

VXI/VME interrupts are a basic form of asynchronous communication used by devices with interrupter support. In VME, a device asserts a VME interrupt line and the VME interrupt handler device acknowledges the interrupt. During the VME interrupt acknowledge cycle, an 8-bit status/ID value is returned. Most 680*x*0-based VME CPUs use this 8-bit value as a local interrupt vector value routed directly to the 680*x*0 processor. This value specifies which interrupt service routine to invoke.

In VXI systems, however, the VXI interrupt acknowledge cycle returns (at a minimum) a 16-bit status/ID value. This 16-bit status/ID value is data, not a vector base location. The definition of the 16-bit value is specified by the VXIbus specification and is the same as for the VXI signal. The lower 8 bits of the status/ID value form the VXI logical address of the interrupting device, while the upper 8 bits specify the reason for interrupting.

VXI status/ID values from message-based devices can be one of two types: *Response* status/IDs and *Event* status/IDs (bit 15 distinguishes between the two). Response status/IDs are used to report changes in Word Serial communication status between a Servant and its Commander. Event status/IDs are used to inform another device of other asynchronous changes. The four Event status/IDs currently defined by the VXIbus specification (other than *Shared Memory* Events) are *No Cause Given*, *Request for Service True* (REQT), *Request for Service False* (REQF), and *Unrecognized Command*. REQT and REQF are used to manipulate the SRQ condition (RSV bit assertion in the IEEE 488/488.2 status byte), while *Unrecognized Command* is used to report unsupported Word Serial commands (only in VXIbus specification, Revision 1.2). If the VXI interrupt status/ID value is from a register-based device, the upper 8 bits are device dependent. Consult your device manual for definitions of these values.

Because the VXI interrupt status/ID has the same format as the VXI signal, your application can handle VXI interrupts as VXI signals. However, because VME interrupters may be present in a VXI system, the VXI/VME interrupt handler functions are included with the NI-VXI software. The `RouteVXIint` function specifies whether the status/ID value should be handled as a signal or handled by a VXI/VME interrupt handler. Two methods are available to handle VXI signals. Signals can be handled either by calling a signal handler, or by queueing on a global signal queue. The `RouteSignal` function

specifies which types of signals are handled by signal handlers, and which are queued onto the global signal queue for each VXI logical address. A separate handler can be installed for each VXI logical address present (refer to the description for `SetSignalHandler`). A default handler, `DefaultSignalHandler`, is automatically installed when `InitVXIlibrary` is called from the application for every VXI logical address. If signals are queued, the application can use the `SignalDeq` function to selectively return a signal off a global signal queue by VXI logical address and/or type of signal.

Another method for handling signals (and VXI/VME interrupts routed to signals) can be used instead of the two previous methods, and involves using the `WaitForSignal` function. `WaitForSignal` can be used to suspend a process/function until a particular signal (or one of a set of signals) arrives. In a multitasking operating system, any number of `WaitForSignal` calls can be pending. In a nonmultitasking operating system, only one `WaitForSignal` call can be pending.

If the `RouteVXIint` has specified that a status/ID value should be handled by the VXI/VME interrupt handler and not by the signal handler, the specified callback handler is invoked. The VXI/VME interrupt handler for a particular level is called with the VXI interrupt level and the status/ID without any interpretation of the status/ID value. The callback handler can do whatever is necessary with the status/ID value. The `SetVXIintHandler` function can be called to change the current callback handler for a particular level. A default handler, `DefaultVXIintHandler` is automatically installed with a call to `InitVXIlibrary` at the start of the application. `EnableVXIint` and `DisableVXIint` are used to sensitize and desensitize the application to VXI/VME interrupts routed to the VXI/VME interrupt handlers. `EnableVXItoSignalInt` and `DisableVXItoSignalInt` are used to sensitize and desensitize the application to VXI/VME interrupts routed to be processed as VXI signals.

When you are testing VXI/VME interrupt handlers or creating a message-based interrupter, you must assert a VXI/VMEbus interrupt line and present a valid status/ID value. The `AssertVXIint` function asserts an interrupt on the local CPU or on the specified extended controller. `DeAssertVXIint` can be used to unassert a VXI/VME interrupt that was asserted using the `AssertVXIint` function. `AcknowledgeVXIint` can be used to acknowledge VXI/VME interrupts that the local CPU is not enabled to automatically handle via `EnableVXIint` or `EnableVXItoSignalInt`. Both

`DeAssertVXIint` and `AcknowledgeVXIint` are intended only for debugging purposes.

Programming Considerations

Figure 3-4 is a graphical overview of the NI-VXI interrupt and signal model.

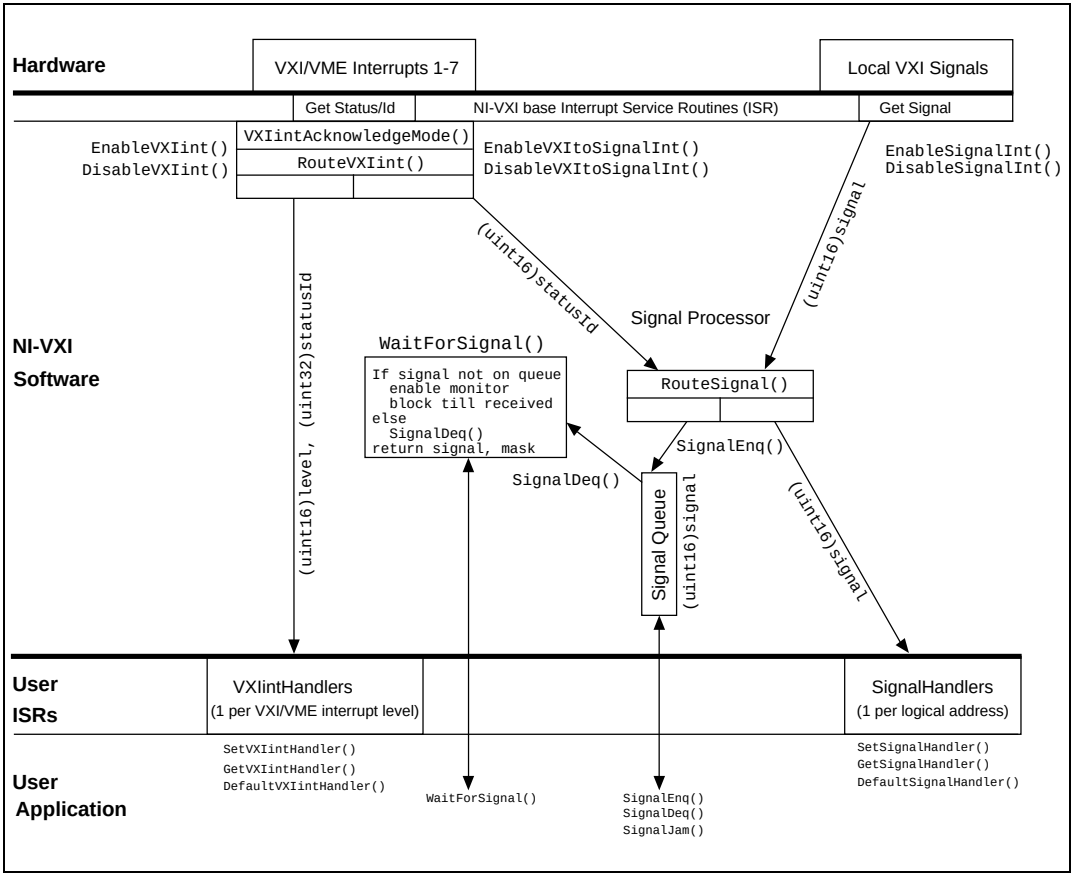


Figure 3-4. NI-VXI Interrupt and Signal Model

ROAK Versus RORA VXI/VME Interrupters

There are two types of VXI/VME interrupters. The Release On Acknowledge (ROAK) interrupter is the more common. A ROAK interrupter automatically unasserts the VXI/VME interrupt line it is asserting when an interrupt acknowledge cycle on the backplane occurs on the corresponding level. The VXIbus specification requires that all message-based devices be ROAK interrupters. It is recommended that all other types of VXI devices also be ROAK interrupters.

The Release On Register Access (RORA) interrupt is the second type of VXI/VME interrupter. The RORA interrupter continues to assert the VXI/VME interrupt line after the interrupt acknowledge cycle is complete. The RORA interrupter will unassert the interrupt only when some device-specific interaction is performed. There is no standard method to cause a RORA interrupter to unassert its interrupt line. Because a RORA interrupt remains asserted on the backplane, the local CPU interrupt generation must be inhibited until the device-dependent acknowledgment is complete.

The function `VXIintAcknowledgeMode` specifies whether a VXI/VME interrupt level for a particular controller (embedded or extended) is to be handled as a RORA or ROAK interrupt. If the VXI/VME interrupt is specified to be handled as a RORA interrupt, the local CPU automatically inhibits VXI/VME interrupt generation for the corresponding controller and levels whenever the corresponding VXI/VME interrupt occurs. After the application has handled and caused the RORA interrupter to unassert the interrupt line, either `EnableVXIint` or `EnableVXItoSignalInt` must be called to re-enable local CPU interrupt generation.

The following paragraphs describe the VXI/VME interrupt functions and default handler. The descriptions are presented at a functional level describing the operation of each of the functions. The functions are grouped by area of functionality.

AcknowledgeVXIint (controller, level, statusId)

`AcknowledgeVXIint` performs a VXI/VME interrupt acknowledge (IACK cycle) on the backplane in the specified controller and VXI/VME interrupt level.

☞ **Note:** ***This function is intended only for debugging purposes.***

Normally, VXI/VME interrupts are automatically acknowledged when enabled via the function `EnableVXIint`. However, if the interrupts are not enabled and the assertion of an interrupt is detected through some method (such as `GetVXIbusStatus`), you can use `AcknowledgeVXIint` to acknowledge an interrupt and return the status/ID value. If the **controller** parameter specifies an extended controller, `AcknowledgeVXIint` specifies hardware on the VXI/VME frame extender (if present) to acknowledge the specified interrupt.

AssertVXIint (controller, level, statusId)

`AssertVXIint` asserts a particular VXI/VME interrupt level on a specified controller (embedded or extended) and returns the specified status/ID value when acknowledged. You can use `AssertVXIint` to send any status/ID value to the VXI/VME interrupt handler configured for the specified VXI/VME interrupt level. `AssertVXIint` returns immediately (that is, it does not wait for the interrupt to be acknowledged). You can call `GetVXIbusStatus` to detect if the interrupt has been serviced. Use `DeAssertVXIint` to unassert a interrupt that had been asserted using `AssertVXIint` but has not yet been acknowledged.

DeAssertVXIint (controller, level)

`DeAssertVXIint` unasserts the VXI/VME interrupt level on a given controller that was previously asserted using the `AssertVXIint` function. You can use `AssertVXIint` to send any status/ID value to the VXI/VME interrupt handler configured for the specified interrupt level. You can call `GetVXIbusStatus` to detect if the interrupt has been serviced. Use `DeAssertVXIint` to unassert a VXI/VME interrupt that had been asserted using `AssertVXIint` but has not yet been acknowledged.

☞ **Note:** ***Unasserting an interrupt may violate the VME and VXIbus specifications if the interrupt has not yet been acknowledged by the interrupt handler.***

DefaultVXIintHandler (controller, level, statusId)

`DefaultVXIintHandler` is the sample handler for VXI/VME interrupts, which is installed when the function `InitVXIlibrary` is called. If VXI/VME interrupts are enabled (via `EnableVXIint`), the VXI/VME interrupt handler for a specific logical address is called. You must first call `RouteVXIint` to route VXI/VME interrupts to the callback handler (as opposed to the signal processing routine).

DefaultVXIintHandler sets the global variables VXIintController, VXIintLevel, and VXIintStatusId. You can leave this default handler installed or install a completely new handler using SetVXIintHandler.

DisableVXIint (controller, levels)

DisableVXIint desensitizes the application to specified VXI interrupt levels being processed as VXI/VME interrupts (not as VXI signals). EnableVXIint enables VXI/VME interrupts to be handled as VXI/VME interrupts (not as VXI signals). A -1 (negative one) or local logical address in the **controller** parameter specifies the local frame (for an embedded CPU) or the first extended controller (in an external CPU situation).

DisableVXItoSignalInt (controller, levels)

DisableVXItoSignalInt desensitizes the application to specified VXI/VME interrupt levels being processed as VXI/VME signals. An EnableVXItoSignalInt call enables VXI/VME interrupt levels that are routed to VXI signals. Use DisableVXItoSignalInt to disable these interrupts. Use EnableVXIint to enable interrupts not routed to signals. A -1 (negative one) or local logical address in the **controller** parameter specifies the local frame (for an embedded CPU) or the first extended controller (in an external CPU situation). If a RouteVXIint call has specified to route a particular VXI/VME interrupt level to the VXI signal processing routine and the global signal queue becomes full, DisableVXItoSignalInt is automatically called to inhibit these VXI/VME interrupts from being received from the appropriate levels. EnableVXItoSignalInt is automatically called to enable interrupt reception when SignalDeq is called.

EnableVXIint (controller, levels)

EnableVXIint sensitizes the application to specified VXI/VME interrupt levels being processed as VXI/VME interrupts (not as VXI signals). After calling InitVXIlibrary, the application can sensitize itself to interrupt levels for which it is configured to handle. RouteVXIint specifies whether interrupts are to be handled as VXI/VME interrupts or as VXI signals (the default is VXI signals). You must then call EnableVXIint to enable interrupts to be handled as VXI/VME interrupts (not as VXI signals). A -1 (negative one) or local logical address in the **controller** parameter specifies the local

frame (for an embedded CPU) or the first extended controller (in an external CPU situation).

EnableVXItoSignalInt (controller, levels)

`EnableVXItoSignalInt` is used to sensitize the application to specified interrupt levels being processed as VXI signals. After calling `InitVXIlibrary`, the application can sensitize itself to interrupt levels for which it is configured to handle. `RouteVXIint` specifies whether interrupts are to be handled as VXI/VME interrupts or as VXI signals (the default is VXI signals). An `EnableVXItoSignalInt` call enables interrupt levels that are routed to VXI signals. Use `DisableVXItoSignalInt` to disable these VXI interrupts. Use `EnableVXIint` to enable interrupts not routed to VXI signals. A -1 (negative one) or local logical address in the **controller** parameter specifies the local embedded controller or the first extended controller (in an external controller situation). If a `RouteVXIint` call has specified to route a particular VXI/VME interrupt level to the VXI signal processing routine and the global signal queue becomes full, `DisableVXItoSignalInt` is automatically called to inhibit these VXI interrupts from being received from the appropriate levels. `EnableVXItoSignalInt` is automatically called to enable VXI/VME interrupt reception when `SignalDeq` is called.

GetVXIintHandler (level)

`GetVXIintHandler` returns the address of the current VXI/VME interrupt handler routine for the specified interrupt level. If interrupts are enabled (via `EnableVXIint`), the callback handler for a specific logical address is called. You must first call `RouteVXIint` to route VXI/VME interrupts to the callback handler (as opposed to the signal processing routine). A default handler, `DefaultVXIintHandler`, is automatically installed for every applicable VXI interrupt level when the `InitVXIlibrary` function is called.

RouteVXIint (controller, Sroute)

`RouteVXIint` specifies whether status/ID values returned from a VXI/VME interrupt acknowledge cycle are routed to a VXI/VME interrupt handler or to the VXI signal processing routine. The function `RouteVXIint` specifies whether the status/ID value should be handled as a signal or handled locally by a VXI/VME interrupt handler. Two methods are available to handle VXI signals. Signals can be handled either by signal handlers (as signals) or by queueing on a global signal

queue. The RouteSignal function specifies which types of signals should be handled by signal handlers, and which should be queued on the global signal queue for each VXI logical address. If the VXI/VME interrupt status/IDs are specified to be handled by a VXI/VME interrupt handler, the level and status/ID value is sent to the appropriate callback handler when an interrupt occurs. An individual handler can be installed for each of the seven VXI/VME interrupt levels. EnableVXIint and EnableVXItoSignalInt must be used to sensitize the local CPU to interrupts generated by VXI/VME interrupts. Only the levels routed to the appropriate handlers (VXI/VME interrupts or VXI signals) via the RouteVXIint function are enabled.

SetVXIintHandler (levels, func)

SetVXIintHandler replaces the current callback handler for the specified VXI/VME interrupt levels with an alternate callback handler. If VXI/VME interrupts are enabled (via EnableVXIint), the VXI/VME interrupt handler for a specific logical address is called. The RouteVXIint function must first be called to route VXI/VME interrupts to the callback handler (as opposed to the signal processing routine). A default handler, DefaultVXIintHandler is automatically installed when the InitVXIlibrary function is called for every applicable VXI/VME interrupt level. You can use SetVXIintHandler to install a new callback handler.

VXIintAcknowledgeMode (controller, modes)

VXIintAcknowledgeMode specifies whether to handle the VXI/VME interrupt acknowledge cycle for the specified controller (embedded or extended) for the specified levels as ROAK interrupts or as RORA interrupts. If the VXI/VME interrupt level is handled as a RORA interrupt, the local interrupt generation is automatically inhibited during the interrupt acknowledgment. After device-specific interaction has caused the deassertion of the interrupt on the backplane, your application must call EnableVXIint to re-enable the appropriate VXI/VME interrupt level.

VXI Trigger Functions

VXI triggers are a backplane feature that VXI added to the VME standard. Tight timing and signaling is important between many types of controllers and/or instruments. In the past, clumsy cables of specified length had to be connected between controllers and/or instruments to get the required timing. For many systems, phase shifting and propagation delays had to be calculated precisely, based on the instrument connection scheme. This limited the architecture of many systems.

In VXI however, every VXI board with a P2 connector has access to eight 10 MHz TTL trigger lines. If the VXI board has a P3 connector, it has access to six 100 MHz ECL trigger lines. The phase shifting and propagation delays can be held to a known maximum, based on the VXIbus specification's rigid requirement on backplanes. The VXIbus specification does not currently prescribe an allocation method for TTL or ECL trigger lines. The application must decide how to allocate required trigger lines.

The VXIbus specification specifies several trigger protocols that can be supported, thereby promoting compatibility among the various VXI devices. The following is a description of the four basic protocols.

- *SYNC*—SYNC protocol is the most basic protocol. SYNC protocol is a pulse of a minimum time (30 ns on TTL, 8 ns on ECL) on any one of the trigger lines.
- *ASYNC*—ASYNC is a two-device, two-line handshake protocol. ASYNC uses two consecutive even/odd trigger lines (a source/acceptor line and an acknowledge line, respectively). The sourcing device sources a trigger pulse (30 ns TTL, 8 ns ECL minimum) on the even trigger line (TTL0, TTL2, TTL4, TTL6, ECL0, ECL2, or ECL4) and waits for the acknowledge pulse on the next highest odd trigger line (TTL1, TTL3, TTL5, TTL7, ECL1, ECL3, or ECL5). The acceptor waits for the source pulse on the even trigger line. Sometime after the source pulse is sensed (no maximum time is specified), the acceptor sends an acknowledge pulse back on the next highest odd trigger line to complete the handshake.
- *SEMI-SYNC*—SEMI-SYNC is a one-line, open collector, multiple-device handshake protocol. The sourcing device sources a trigger pulse (50 ns TTL, 20 ns ECL minimum) on any one of the trigger lines. The accepting device(s) must begin to assert the same trigger line upon reception (within 40 ns TTL, 15 ns ECL

maximum time from source assertion edge). The accepting device(s) can later unassert the trigger line (no maximum time is specified) to complete the handshake.

- *START/STOP*—START/STOP is a one-line, multiple-device protocol. START/STOP can be sourced only by the VXI Slot 0 device and sensed by any other devices on the VXI backplane. The START/STOP protocol is synchronized with the backplane clock (CLK10 for TTL, CLK100 and SYNC100 for ECL) onto any one of the trigger lines. A START condition is generated on the assertion edge on the trigger line, and a STOP condition is generated on the unassertion edge of the trigger line.
- *ON/OFF*—ON/OFF protocol is identical to the START/STOP protocol. The VXIbus specification, however, defines START/STOP such that only Slot 0 may assert START/STOP. Therefore, ON/OFF protocols are outside the VXIbus specifications but provide similar functionality.

You can use these protocols in any way that your application requires. You can use them for device synchronization, for stepping through tests, or for a command path. The NI-VXI trigger functions have been designed to accommodate all trigger lines and the protocols for all appropriate TTL and ECL VXI trigger lines (SYNC, ASYNC, SEMI-SYNC, START/STOP, and ON/OFF).

The VXI trigger functions have been grouped into the following four categories:

- Source trigger functions
- Acceptor trigger functions
- Map trigger functions
- Trigger configuration functions

The actual capabilities of specific systems are based on the triggering capabilities of the hardware devices involved (both the sourcing and accepting devices). All of the NI-VXI functions have appropriate error response for unsupported capabilities.

Capabilities of the National Instruments Triggering Hardware

The NI-VXI trigger functions are a general-purpose interface designed to accommodate most uses of VXI triggers. The actual capabilities of a particular platform will always be a subset of these capabilities. In

general, however, National Instruments hardware has two current configurations that provide triggering functionality:

- Trigger control used on a VXI-MXI-1 frame extender when used as an extending controller (under direct control of a root-level MXI-1 controller interface, such as an AT-MXI-1). These configurations *do not* have the National Instruments Trigger Interface Chip (TIC) on them.

☞ **Note:** ***VXI-MXI-1 and VXI-MXI-2 controllers that are configured for extender only (that is, not extending controllers), as well as external MXI-1 controllers, do not have trigger functionality. See the section,*** *Multiple Mainframe Support,* ***in Chapter 2,*** *Introduction to the NI-VXI Functions,* ***for more information.***

- An embedded controller, external MXI-2 controllers, or VXI-MXI-2 remote controllers. These configurations *do* have the National Instruments Trigger Interface Chip (TIC) on them.

External Controller/VXI-MXI-1 Trigger Capabilities

All National Instruments external controllers (such as the AT-MXI-1) that are connected to VXI-MXI-1 extending controllers have the same basic trigger capabilities:

- Source a single TTL or ECL (0 and 1 only) trigger using any protocol on any one of the backplane TTL trigger lines
- Accept a single backplane TTL or ECL (0 and 1 only) trigger using any protocol (as long as it does not source SEMI-SYNC and ASYNC protocols at the same time)
- Map a front panel In connector to a TTL or ECL (0 or 1 only) trigger line (sourcing will be disabled)
- Source a TTL or ECL (0 or 1 only) trigger out the front panel
- Map a TTL or ECL (0 or 1 only) trigger line from the backplane out the front panel Out connector (accepting disabled) (Some platforms do not have this capability.)

The following capabilities are not supported:

- Multiple-line support
- Crosspoint switching
- Signal conditioning
- External connections other than the front panel In/Out

Embedded, External MXI-2, and Remote Controller Trigger Capabilities

National Instruments has developed a highly functional ASIC specifically designed for use within the VXIbus triggering environment called the Trigger Interface Chip (TIC).

☞ **Note:** ***In MXI-2 and the latest embedded systems, the TIC has been incorporated into the MANTIS ASIC.***

The TIC chip has access to all of the eight VXI TTL trigger lines, two ECL trigger lines (ECL0 and ECL1), and 10 external or General-Purpose Input/Output (GPIO) connections simultaneously. The TIC also contains a 16-bit counter and a dual 5-bit scaler tick timer. It contains a full crosspoint switch for routing trigger lines and GPIOs (as well as the counter and the tick timers) between one another.

If you want more information on triggering or if you plan to use any of the advanced features of the TIC, please contact National Instruments for the technical note, *Triggering with NI-VXI*.

Acceptor Trigger Functions

The NI-VXI acceptor trigger functions act as a standard interface for sensing (accepting) TTL and ECL triggers, as well as for sending acknowledgments back to the sourcing device. These functions can sense any of the VXI-defined trigger protocols on the local embedded controller or external extended controller(s). Use the `EnableTrigSense` function to prepare for the sensing of any of the trigger protocols. If the protocol requires an acknowledgment, you should call the `AcknowledgeTrig` function when appropriate. You can use `SetTrigHandler` to install a callback handler for the specified trigger line. A default handler, `DefaultTrigHandler`, is installed for each one of the trigger lines when `InitVXIlibrary` is called and will call `AcknowledgeTrig` for you. You can use the `SetTrigHandler` function at any time to replace the default handlers. In addition, you can use the `WaitForTrig` function to accommodate applications that do not want to install callback handlers.

AcknowledgeTrig (controller, line)

`AcknowledgeTrig` performs the required trigger acknowledgments for the ASYNC or SEMI-SYNC VXI-defined protocol, as configured via the `EnableTrigSense` function.

DefaultTrigHandler (controller, line, type)

`DefaultTrigHandler` is the sample handler for the receiving acknowledges and sensing triggers, and is automatically installed after a call to `InitVXIlibrary`. After a call to `EnableTrigSense` for a particular VXI trigger line protocol, the trigger handler for a specific trigger line is called when the sourced trigger is sensed from the sourcing device. If the configured VXI trigger protocol requires an acknowledgment (either ASYNC or SEMI-SYNC), you must call the `AcknowledgeTrig` function to perform the acknowledgment. `DefaultTrigHandler` calls the `AcknowledgeTrig` function if the `type` parameter specifies that an acknowledge interrupt occurred. Otherwise, `DefaultTrigHandler` performs no operations.

DefaultTrigHandler2 (controller, line, type)

`DefaultTrigHandler2` is a sample handler for receiving trigger interrupt sources similar to `DefaultTrigHandler`. `DefaultTrigHandler2` performs no operations. Any required acknowledgments must be performed by the application.

DisableTrigSense (controller, line)

`DisableTrigSense` unconfigures and desensitizes the triggering hardware that was enabled by the `EnableTrigSense` function to generate interrupts when any VXI-defined trigger protocol is sensed on the specified trigger line.

EnableTrigSense (controller, line, prot)

`EnableTrigSense` configures and sensitizes the triggering hardware to generate interrupts when the specified VXI-defined trigger protocol is sensed on the specified trigger line. When `EnableTrigSense` has configured and enabled the triggering hardware to generate interrupts, and the specified trigger protocol is sensed, a local CPU interrupt is generated. The trigger handler installed is automatically called when a trigger interrupt occurs.

GetTrigHandler (line)

`GetTrigHandler` returns the address of the current trigger handler for the specified VXI trigger line.

SetTrigHandler (lines, func)

`SetTrigHandler` replaces the current trigger handler for the specified VXI trigger lines with an alternate handler.

WaitForTrig (controller, line, timeout)

You can use the `WaitForTrig` function to suspend operation until it receives a trigger configured by the `EnableTrigSense` function. After a call to `EnableTrigSense` for a particular VXI trigger line protocol, the trigger handler for a specific trigger line is called when the sourced trigger is sensed from the sourcing device. You can use `WaitForTrig` as an alternate method for receiving sensed triggers by having the caller wait until the trigger occurs instead of installing a callback handler. The current handler is invoked regardless of whether a `WaitForTrig` call is pending.

Map Trigger Functions

You can use the NI-VXI map trigger functions as configuration tools for multiframe and local support for VXI triggers. You can configure the triggering hardware to route specified source trigger locations to destination trigger locations by using the `MapTrigToTrig` and `UnMapTrigToTrig` functions.

MapTrigToTrig (controller, srcTrig, destTrig, mode)

`MapTrigToTrig` configures triggering hardware to route specified source trigger locations to destination trigger locations with some possible signal conditioning. The possible values for source or destination locations are the TTL trigger lines, ECL trigger lines, Star X lines, Star Y lines, or miscellaneous external sources. Miscellaneous external sources include front panel trigger ins, front panel trigger outs, local clocks, and crosspoint switch locations. The **mode** parameter specifies how the line is to be routed to the destination. You can manipulate the line in various ways, including inverting it, synchronizing it with the CLK10, or stretching it to a minimum time. In this way, `MapTrigToTrig` can be used as a simple map from an external source to a trigger line, or as a complex

crosspoint switch configurator (depending on the hardware capabilities of the applicable device).

UnMapTrigToTrig (controller, srcTrig, destTrig)

`UnMapTrigToTrig` unconfigures triggering hardware that was configured by the `MapTrigToTrig` function to route specified source trigger locations to destination trigger locations.

Source Trigger Functions

The NI-VXI source trigger functions act as a standard interface for asserting (sourcing) TTL and ECL triggers, as well as for detecting acknowledgments from accepting devices. These functions can source any of the VXI-defined trigger protocols from the local embedded controller or external extended controller(s). You can use the `SrcTrig` function to initiate any of the trigger protocols. If the protocol requires an acknowledgment and your application is required to know when the acknowledgment occurs, you must use the `SetTrigHandler` function to install a callback handler for the specified trigger line. A default handler, `DefaultTrigHandler`, is installed for each one of the trigger lines when `InitVXIlibrary` is called. You can use the `SetTrigHandler` function at any time to replace the default handlers.

SrcTrig (controller, line, prot, timeout)

Use `SrcTrig` to source any one of the VXI-defined trigger protocols from the local CPU or from any remote frame extender device that supports trigger assertion. For protocols that require an acknowledgment from the accepting device (ASYNC or SEMI-SYNC), you need to specify whether to wait for an acknowledgment (with a timeout) or return immediately and let the trigger handler get called when the acknowledgment is received. Another option is available in which you can assert or unassert any of the trigger lines continuously, or have an external trigger (possibly from the front panel) routed to the VXIbus backplane.

Trigger Configuration Functions

You can use the NI-VXI trigger configuration functions to configure not only the general settings of the trigger inputs and outputs, but also the TIC counter and tick timers.

TrigAssertConfig (controller, trigline, mode)

`TrigAssertConfig` configures the local triggering generation method for the TTL/ECL triggers. You can decide on an individual basis whether to synchronize the triggers to CLK10. You can globally select the synchronization to be the rising or falling edge of CLK10. In addition, you can specify the trigger line to partake in automatic external SEMI-SYNC acknowledgment. In this mode, when a trigger is sensed on the line, the line is asserted until an external (GPIO) trigger line which is mapped to the corresponding trigger line is pulsed. You can also use `AcknowledgeTrig` to manually acknowledge a pending SEMI-SYNC trigger configured in this fashion.

TrigCntrConfig (controller, mode, source, count)

`TrigCntrConfig` configures the TIC chip's 16-bit counter. You can use this function to initialize, reload, or disable the current counter settings. If the counter is initialized, you must call either `SrcTrig` or `EnableTrigSense` to actually start the counter. You can use any trigger line, CLK10, or EXTCLK as the source of the counter. The count range is 1 to 65535. You can use the counter to source multiple sync or multiple semi-sync triggers to one or more trigger lines. You can also use it to accept multiple sync or multiple semi-sync triggers from one trigger line. The counter has two outputs: TCNTR and GCNTR. The TCNTR signal pulses for 100 ns every time a source pulse occurs. You can use `MapTrigToTrig` to map the TCNTR signal to one or more trigger lines. The GCNTR signal stays unasserted until the counter goes from 1 to 0. It then becomes asserted until the counter is disabled. You can use the `MapTrigToTrig` function to directly map the GCNTR signal to one or more GPIO lines.

TrigExtConfig (controller, extline, mode)

`TrigExtConfig` configures the way the external trigger sources (General-Purpose Inputs and Outputs, or *GPIOs*) are configured. The TIC chip has 10 GPIO lines. Typically, GPIO 0 is connected to the front panel In connector. GPIO 1 is connected to the front panel Out connector. GPIO 2 is connected to a direct ECL bypass from the front

panel. GPIO 3 is fed back in as the EXTCLK signal used for signal conditioning modes with `MapTrigToTrig`. The six remaining GPIOs are dependent upon the hardware platform. Regardless of the sources connected to the GPIOs, `TrigExtConfig` configures several aspects of the connection. You can disconnect and feed back the connection for use as a crosspoint switch. You can also choose whether to invert the external input. In addition, you can configure the GPIO to be asserted high or low continuously. In this configuration, no input mapping is possible (that is, no trigger line can be mapped to the GPIO).

TrigTickConfig (controller, mode, source, tcount1, tcount2)

`TrigTickConfig` configures the TIC chip's dual 5-bit tick timers. This function can initialize with auto reload, initialize with manual reload, do a manual reload, or disable the current tick timer settings. If the tick timer is initialized, you must call either `EnableTrigSense` or `SrcTrig` to start the tick timer. You can use any GPIO line, CLK10, or EXTCLK as the source of the tick timer. Both tick timers—TICK1 and TICK2—count independently from the same internal counter. The range for each tick timer is specified as a power of two from 0 to 31. If you did not select auto reload, the timer stops when TICK1 has counted to zero. You can use `MapTrigToTrig` to map the TICK1 output signal to one or more trigger lines, or to map the TICK2 output signal to one or more trigger lines or GPIO lines. Both TICK1 and TICK2 outputs are square wave outputs. The signal is asserted for the duration of the corresponding tick count and then unasserted for the duration of the count.

System Interrupt Handler Functions

With these functions, you can handle miscellaneous system conditions that can occur in the VXI/VME environment, such as Sysfail, ACfail, Sysreset, Bus Error, and/or Soft Reset interrupts. The NI-VXI software interface can handle all of these system conditions for the application through the use of callback routines. The NI-VXI software handles all system interrupt handlers in the same manner. Each type of interrupt has its own specified default handler, which is installed when `InitVXIlibrary` initializes the NI-VXI software. If your application program requires a different interrupt handling algorithm, it can call the appropriate `SetHandler` function to install a new callback handler. All system interrupt handlers are initially disabled (except for Bus Error). The corresponding enable function for each handler must be called in order to invoke the default or user-installed handler.

The following paragraphs describe the system interrupt handler functions and default handlers. The descriptions are presented at a functional level describing the operation of each of the functions. The functions are grouped by area of functionality.

AssertSysreset (controller, mode)

`AssertSysreset` asserts the SYSRESET* signal on the specified controller. You can use this function to reset the local CPU, individual mainframes, all mainframes, or the entire system. If you reset the system but not the local CPU, you will need to re-execute all device configuration programs.

☞ **Note:** ***Due to the operation of some operating systems, not all platforms support resetting the local CPU.***

DefaultACfailHandler (controller)

`DefaultACfailHandler` is the sample handler for the ACfail interrupt, and is installed as a default handler when `InitVXIlibrary` initializes the NI-VXI software. It increments the global variable `ACfailRecv`. The VXI/VMEbus specification allows for a minimum amount of time after a power failure condition occurs for the system to remain operational. The detection of a power failure in a VXI/VME system asserts the backplane signal ACFAIL*. An ACfail condition detected on the local CPU generates an interrupt that calls the current ACfail interrupt handler. Your application can take any appropriate action within the allotted time period before complete power failure. Your application must then call `EnableACfail` to enable ACfail interrupts after the `InitVXIlibrary` call.

DefaultBusErrorHandler ()

`DefaultBusErrorHandler` is the sample handler for the bus error exception, and is installed as a default handler when `InitVXIlibrary` initializes the NI-VXI software. During an access to the VXI/VMEbus, the BERR* signal (bus error) is asserted to end the bus cycle if the address or mode of access is determined to be invalid. The bus error exception condition generates an exception on the local CPU, which can be trapped by the bus error handler. Your application should include a retry mechanism if it is possible for a particular access to generate bus errors at times and valid results at other times. Because bus errors can occur at any time, a corresponding enable and disable function is not possible. Notice that only BERRs occurring via

low-level VXI/VMEbus access functions will be reported to this handler. See also the descriptions of `SetBusErrorHandler` and `GetBusErrorHandler`.

DefaultSoftResetHandler ()

`DefaultSoftResetHandler` is the sample handler for the Soft Reset interrupt, and is installed as a default handler when `InitVXIlibrary` initializes the NI-VXI software. It increments the global variable `SoftResetRecv`. When the Reset bit in the VXI Control register of the local CPU is written, the VXI interface (if an embedded CPU) and the VXI register sets are reset (VXI logical address and address base are retained). The write to the Reset bit causes an interrupt on the local CPU, which can be handled in any appropriate manner. The CPU cannot restart operation until the Reset bit is cleared. After the Reset bit is cleared, the local CPU can go through a reinitialization process or reboot altogether. If the local CPU is the Resource Manager (and top-level Commander), the Reset bit should never be written. Writing the Reset bit of any device should be reserved for the Commander of the device. `EnableSoftReset` must be called to enable writes to the Reset bit to generate interrupts to the local CPU after the `InitVXIlibrary` call.

Note: ***The Soft Reset interrupt does not apply to VME.***

DefaultSysfailHandler (controller)

`DefaultSysfailHandler` is the sample handler for the Sysfail interrupt, and is installed as a default handler when `InitVXIlibrary` initializes the NI-VXI software. The VXIbus specification requires that all VXI Commanders monitor the PASSed or FAILed state of their VXI Servants. When a VXIbus device is in the FAILed state, the failed device clears its PASS bit (in its Status register) and asserts the SYSFAIL* signal on the VXIbus backplane. A Sysfail condition detected on the local CPU generates an interrupt that calls the current Sysfail interrupt handler. The failed Servant device must be forced offline or brought back online in an orderly fashion. `DefaultSysfailHandler` scans the local CPU Servants and if a Servant is detected to have failed, the Servant's Sysfail Inhibit bit in its Control register is set. In addition, the global variable `SysfailRecv` is incremented.

DefaultSysresetHandler (controller)

`DefaultSysresetHandler` is the sample handler for the Sysreset interrupt, and is installed as a default handler when `InitVXIlibrary` initializes the NI-VXI software. It increments the global variable `SysresetRecv`.

DisableACfail (controller)

`DisableACfail` desensitizes the application to ACfail interrupts from embedded controller or extended controller(s) ACfail conditions (dependent on the hardware platform). The VXI/VMEbus specification allows for a minimum amount of time after a power failure condition occurs for the system to remain operational. The detection of the power failure asserts the VXI/VMEbus backplane signal ACFAIL*. An ACfail condition detected on the local CPU generates an interrupt that calls the current ACfail interrupt handler. Your application can take any appropriate action within the allotted time period before complete power failure.

DisableSoftReset ()

`DisableSoftReset` desensitizes the application to Soft Reset conditions on the local CPU. When the Reset bit in the VXI Control register of the local CPU is written, the VXI interface (if an embedded CPU) and the VXI register sets are reset (VXI logical address and address base are retained). The write to the Reset bit causes an interrupt on the local CPU, which can be handled in any appropriate manner. The CPU cannot restart operation until the Reset bit is cleared. After the Reset bit is cleared, the local CPU can go through a reinitialization process or reboot altogether. If the local CPU is the Resource Manager (and top-level Commander), the Reset bit should never be written. Writing the Reset bit of any device should be reserved for the Commander of the device.

☞ **Note:** ***The Soft Reset interrupt does not apply to VME.***

DisableSysfail (controller)

`DisableSysfail` desensitizes the application to Sysfail interrupts from embedded controller or extended controller(s) Sysfail conditions (dependent on the hardware platform). The VXIbus specification requires that all VXI Commanders monitor the PASSed or FAILed state of their VXI Servants. When a VXIbus device is in the FAILed

state, the failed device clears its PASS bit (in its Status register) and asserts the SYSFAIL* signal on the VXIbus backplane.

DisableSysreset (controller)

`DisableSysreset` desensitizes the application to Sysreset interrupts from embedded or extended controller(s) (dependent on the hardware platform).

EnableACfail (controller)

`EnableACfail` sensitizes the application to ACfail interrupts from embedded controller or extended controller(s) ACfail conditions (dependent on the hardware platform). The VXI/VMEbus specification allows for a minimum amount of time after a power failure condition occurs for the system to remain operational. The detection of the power failure asserts the VXI/VMEbus backplane signal ACFAIL*. An ACfail condition detected on the local CPU generates an interrupt that calls the current ACfail interrupt handler. Your application can take any appropriate action within the allotted time period before complete power failure.

EnableSoftReset ()

`EnableSoftReset` sensitizes the application to Soft Reset conditions on the local CPU. When the Reset bit in the VXI Control register of the local CPU is written, the VXI interface (if an embedded CPU) and the VXI register sets are reset (VXI logical address and address base are retained). The write to the Reset bit causes an interrupt on the local CPU, which can be handled in any appropriate manner. The CPU cannot restart operation until the Reset bit is cleared. After the Reset bit is cleared, the local CPU can go through a reinitialization process or reboot altogether. If the local CPU is the Resource Manager (and top-level Commander), the Reset bit should never be written. Writing the Reset bit of any device should be reserved for the Commander of the device.

☞ **Note:** ***The Soft Reset interrupt does not apply to VME.***

EnableSysfail (controller)

`EnableSysfail` sensitizes the application to Sysfail interrupts from embedded controller or extended controller(s) Sysfail conditions (dependent on the hardware platform and configuration). The VXIbus

specification requires that all VXI Commanders monitor the PASSed or FAILed state of their VXI Servants. When a VXIbus device is in the FAILed state, the failed device clears its PASS bit (in its Status register) and asserts the SYSFAIL* signal on the VXIbus backplane. When a Sysfail condition is detected on the local CPU, an interrupt is generated, and the current Sysfail interrupt handler is called. The failed Servant device must be forced offline or brought back online in an orderly fashion.

EnableSysreset (controller)

`EnableSysreset` sensitizes the application to Sysreset interrupts from embedded or extended controller(s) (dependent on the hardware platform). Notice that if the local CPU is configured to be reset by Sysreset conditions on the backplane, the interrupt handler will not get invoked (the CPU will reboot).

GetACfailHandler ()

`GetACfailHandler` returns the address of the current ACfail interrupt handler. An ACfail condition detected on the local CPU generates an interrupt that calls the current ACfail interrupt handler. Your application can take any appropriate action within the allotted time period before complete power failure. The `InitVXIlibrary` function automatically installs a default handler, `DefaultACfailHandler`, when it initializes the NI-VXI software.

GetBusErrorHandler ()

`GetBusErrorHandler` returns the address of the current bus error interrupt handler. During an access to the VXI/VMEbus, the BERR* signal (bus error) is asserted to end the bus cycle if the address or mode of access is determined to be invalid. The bus error exception condition generates an exception on the local CPU, which can be trapped by the bus error handler. Your application should include a retry mechanism if it is possible for a particular access to generate bus errors at times and valid results at other times. The `InitVXIlibrary` function automatically installs a default handler, `DefaultBusErrorHandler`, when it initializes the NI-VXI software. It increments the global variable `BusErrorRecv`. Because bus errors can occur at any time, a corresponding enable and disable function is not possible.

GetSoftResetHandler ()

`GetSoftResetHandler` returns the address of the current Soft Reset interrupt handler. A default handler, `DefaultSoftResetHandler`, is automatically installed when `InitVXIlibrary` initializes the NI-VXI software.

☞ **Note:** ***The Soft Reset interrupt does not apply to VME.***

GetSysfailHandler ()

`GetSysfailHandler` returns the address of the current Sysfail interrupt handler. A Sysfail condition detected on the local CPU generates an interrupt that calls the current Sysfail interrupt handler. A default handler, `DefaultSysfailHandler`, is automatically installed when `InitVXIlibrary` initializes the NI-VXI software.

GetSysresetHandler ()

`GetSysresetHandler` returns the address of the current Sysreset interrupt handler. The `InitVXIlibrary` function automatically installs a default handler, `DefaultSysresetHandler`, when it initializes the NI-VXI software.

SetACfailHandler (func)

`SetACfailHandler` replaces the current ACfail interrupt handler with an alternate handler. An ACfail condition detected on the local CPU generates an interrupt that calls the current ACfail interrupt handler. Your application can take any appropriate action within the allotted time period before complete power failure. The `InitVXIlibrary` function automatically installs a default handler, `DefaultACfailHandler`, when it initializes the NI-VXI software. Your application must then call `EnableACfail` to enable ACfail interrupts.

SetBusErrorHandler (func)

`SetBusErrorHandler` replaces the current bus error interrupt handler with an alternate handler. During an access to the VXI/VMEbus, the BERR* signal (bus error) is asserted to end the bus cycle if the address or mode of access is determined to be invalid. The bus error exception condition generates an exception on the local CPU, which can be trapped by the bus error handler. Your application should include a retry mechanism if it is possible for a particular access to generate bus

errors at times and valid results at other times. The `InitVXIlibrary` function automatically installs a default handler, `DefaultBusErrorHandler`, when it initializes the NI-VXI software. Because bus errors can occur at any time, a corresponding enable and disable function is not possible.

SetSoftResetHandler (func)

`SetSoftResetHandler` replaces the current Soft Reset interrupt handler with an alternate handler. A default handler, `DefaultSoftResetHandler`, is automatically installed when `InitVXIlibrary` initializes the NI-VXI software. `EnableSoftReset` must be called to enable writes to the Reset bit to generate interrupts to the local CPU after the `InitVXIlibrary` call.

☞ **Note:** ***The Soft Reset interrupt does not apply to VME.***

SetSysfailHandler (func)

`SetSysfailHandler` replaces the current Sysfail interrupt handler with an alternate handler. A Sysfail condition detected on the local CPU generates an interrupt that calls the current Sysfail interrupt handler. A default handler, `DefaultSysfailHandler`, is automatically installed when `InitVXIlibrary` initializes the NI-VXI software. `EnableSysfail` must be called to enable Sysfail interrupts after the `InitVXIlibrary` call.

SetSysresetHandler (func)

`SetSysresetHandler` replaces the current SYSRESET* interrupt handler with an alternate handler. The `InitVXIlibrary` function automatically installs a default handler, `DefaultSysresetHandler`, when it initializes the NI-VXI software. Your application must then call `EnableSysreset` to enable writes to the Reset bit to generate interrupts to the local CPU.

VXI/VMEbus Extender Functions

The NI-VXI software interface fully supports the standard VXIbus extension method presented in the *VXIbus Mainframe Extender Specification*. When the National Instruments Resource Manager (RM) completes its configuration, all default transparent extensions are complete. The transparent extensions include extensions of VXI/VME interrupt, TTL trigger, ECL trigger, Sysfail, ACfail, and Sysreset signals. The VXI/VMEbus extender functions are used to dynamically change the default RM settings if the application has such a requirement. Usually, the application never needs to change the default settings. Consult your utilities manual on how to use the NI-VXI resource editor utility, either `VXIedit` or `VXItedit`, to change the default extender settings.

☞ **Note:** ***The MXIbus, which is used as the transparent mainframe extender bus, extends both VXI and VME chassis and even allows a system consisting of both VXI and VME chassis.***

The following paragraphs describe the VXI/VMEbus extender functions. The descriptions are presented at a functional level describing the operation of each of the functions.

MapECLtrig

`MapECLtrig` configures mainframe extender triggering hardware to map the specified ECL triggers for the specified mainframe in the specified direction (into or out of the mainframe). If the specified frame extender can extend VXI ECL triggers between the mainframes, you can use `MapECLtrig` to configure the mainframe-to-mainframe mapping. The NI-VXI Resource Manager automatically configures a default mapping based on the user-modifiable configuration files. The `MapECLtrig` function can dynamically reconfigure the ECL trigger mapping. Only special circumstances should require any changes to the default configuration.

MapTTLtrig

`MapTTLtrig` configures mainframe extender triggering hardware to map the specified TTL triggers for the specified mainframe in the specified direction (into or out of the mainframe). If the specified frame extender can extend VXI TTL triggers between the mainframes,

you can use MapTTLtrig to configure the mainframe-to-mainframe mapping. The NI-VXI Resource Manager automatically configures a default mapping based on the user-modifiable configuration files. The MapTTLtrig function can dynamically reconfigure the TTL trigger mapping. Only special circumstances should require any changes to the default configuration.

MapUtilBus (extender, modes)

MapUtilBus configures mainframe extender utility bus hardware to map Sysfail, ACfail, and/or Sysreset for the specified mainframe into and/or out of the mainframe. If the specified frame extender can extend the VXI/VME utility signals between mainframes, you can use MapUtilBus to configure the mainframe-to-mainframe mapping. The NI-VXI Resource Manager automatically configures a default mapping based on user-modifiable configuration files. The MapUtilBus function can dynamically reconfigure the utility bus mapping. Only special circumstances should require any changes to the default configuration.

MapVXIint (extender, levels, directions)

MapVXIint changes the VXI/VME interrupt extension configuration in multiple mainframe configurations. If the specified frame extender can extend the VXI/VME interrupts between mainframes, you can use MapVXIint to configure the mainframe-to-mainframe mapping. The NI-VXI Resource Manager automatically configures a default mapping based on user-modifiable configuration files. The MapVXIint function can dynamically reconfigure the utility bus mapping. Only special circumstances should require any changes to the default configuration.

Function Classification Reference

Appendix

A

This appendix contains two tables you can use as a quick reference. Table A-1, *Function Listing by Group*, lists the NI-VXI functions by their group association. This arrangement can help you determine easily which functions are available within each group. Table A-2, *Function Listing by Name*, lists each function alphabetically. You can refer to this table if you don't remember the group association of a particular function. Both tables use checkmarks to denote whether a VXI function also applies to VME and also whether it is associated with C/C++ and/or BASIC.

Table A-1. Function Listing by Group

Group	Function	VXI	VME	C/C++	BASIC
System Configuration	`CloseVXIlibrary`	✔	✔	✔	✔
	`CreateDevInfo`	✔	✔	✔	✔
	`FindDevLA`	✔	✔	✔	✔
	`GetDevInfo`	✔	✔	✔	
	`GetDevInfoLong`	✔	✔	✔	✔
	`GetDevInfoShort`	✔	✔	✔	✔
	`GetDevInfoStr`	✔	✔	✔	✔
	`InitVXIlibrary`	✔	✔	✔	✔
	`SetDevInfo`	✔	✔	✔	
	`SetDevInfoLong`	✔	✔	✔	✔
	`SetDevInfoShort`	✔	✔	✔	✔
	`SetDevInfoStr`	✔	✔	✔	✔

Table A-1. Function Listing by Group

Group	Function	VXI	VME	C/C++	BASIC
Commander Word Serial Protocol	WSabort	✔		✔	✔
	WSclr	✔		✔	✔
	WScmd / WSEcmd/ WSLcmd	✔		✔	✔
	WSgetTmo	✔		✔	✔
	WSrd / WSrdi / WSrdl	✔		✔	✔
	WSrdf	✔		✔	✔
	WSresp/WSLresp	✔		✔	✔
	WSsetTmo	✔		✔	✔
	WStrg	✔		✔	✔
	WSwrt / WSwrti / WSwrtl	✔		✔	✔
	WSwrtf	✔		✔	✔
Servant Word Serial Protocol	DefaultWSScmdHandler	✔		✔	
	DefaultWSSEcmdHandler	✔		✔	
	DefaultWSSLcmdHandler	✔		✔	
	DefaultWSSrdHandler	✔		✔	
	DefaultWSSwrtHandler	✔		✔	
	GenProtError	✔		✔	
	GetWSScmdHandler	✔		✔	
	GetWSSEcmdHandler	✔		✔	
	GetWSSLcmdHandler	✔		✔	
	GetWSSrdHandler	✔		✔	
	GetWSSwrtHandler	✔		✔	
	RespProtError	✔		✔	
	SetWSScmdHandler	✔		✔	
	SetWSSEcmdHandler	✔		✔	
	SetWSSLcmdHandler	✔		✔	
	SetWSSrdHandler	✔		✔	

Table A-1. Function Listing by Group

Group	Function	VXI	VME	C/C++	BASIC
Servant Word Serial Protocol (continued)	SetWSSwrtHandler	✔		✔	
	WSSabort	✔		✔	
	WSSdisable	✔		✔	
	WSSenable	✔		✔	
	WSSnoResp /WSSLnoResp	✔		✔	
	WSSrd / WSSrdi / WSSrdl	✔		✔	
	WSSsendResp / WSSLsendResp	✔		✔	
	WSSwrt / WSSwrti / WSSwrtl	✔		✔	
High-Level VXI/VMEbus Access	VXIin	✔	✔	✔	✔
	VXIinReg	✔		✔	✔
	VXImove	✔	✔	✔	✔
	VXIout	✔	✔	✔	✔
	VXIoutReg	✔		✔	✔
Low-Level VXI/VMEbus Access	GetByteOrder	✔	✔	✔	✔
	GetContext	✔	✔	✔	✔
	GetPrivilege	✔	✔	✔	✔
	GetVXIbusStatus	✔	✔	✔	
	GetVXIbusStatusInd	✔	✔	✔	✔
	GetWindowRange	✔	✔	✔	✔
	MapVXIAddress	✔	✔	✔	✔
	MapVXIAddressSize	✔	✔	✔	✔
	SetByteOrder	✔	✔	✔	✔
	SetContext	✔	✔	✔	✔
	SetPrivilege	✔	✔	✔	✔
	UnMapVXIAddress	✔	✔	✔	✔
	VXIpeek	✔	✔	✔	✔
	VXIpoke	✔	✔	✔	✔

Table A-1. Function Listing by Group

Group	Function	VXI	VME	C/C++	BASIC
Local Resource Access	GetMyLa	✔	✔	✔	✔
	ReadMODID	✔		✔	✔
	SetMODID	✔		✔	✔
	VXIinLR	✔	✔	✔	✔
	VXImemAlloc	✔	✔	✔	✔
	VXImemCopy	✔	✔	✔	✔
	VXImemFree	✔	✔	✔	✔
	VXIoutLR	✔	✔	✔	✔
VXI Signal	DefaultSignalHandler	✔	✔	✔	✔
	DisableSignalInt	✔		✔	✔
	EnableSignalInt	✔		✔	✔
	GetSignalHandler	✔	✔	✔	
	RouteSignal	✔	✔	✔	✔
	SetSignalHandler	✔	✔	✔	
	SignalDeq	✔	✔	✔	✔
	SignalEnq	✔	✔	✔	✔
	SignalJam	✔	✔	✔	✔
	WaitForSignal	✔	✔	✔	✔
VXI/VME Interrupt	AcknowledgeVXIint	✔	✔	✔	✔
	AssertVXIint	✔	✔	✔	✔
	DeAssertVXIint	✔	✔	✔	✔
	DefaultVXIintHandler	✔	✔	✔	✔
	DisableVXIint	✔	✔	✔	✔
	DisableVXItoSignalInt	✔	✔	✔	✔
	EnableVXIint	✔	✔	✔	✔
	EnableVXItoSignalInt	✔	✔	✔	✔
	GetVXIintHandler	✔	✔	✔	

Table A-1. Function Listing by Group

Group	Function	VXI	VME	C/C++	BASIC
VXI/VME Interrupt (continued)	RouteVXIint	✔	✔	✔	✔
	SetVXIintHandler	✔	✔	✔	
	VXIintAcknowledgeMode	✔	✔	✔	✔
Triggers	AcknowledgeTrig	✔		✔	✔
	DefaultTrigHandler	✔		✔	✔
	DefaultTrigHandler2	✔		✔	✔
	DisableTrigSense	✔		✔	✔
	EnableTrigSense	✔		✔	✔
	GetTrigHandler	✔		✔	
	MapTrigToTrig	✔		✔	✔
	SetTrigHandler	✔		✔	
	SrcTrig	✔		✔	✔
	TrigAssertConfig	✔		✔	✔
	TrigCntrConfig	✔		✔	✔
	TrigExtConfig	✔		✔	✔
	TrigTickConfig	✔		✔	✔
	UnMapTrigToTrig	✔		✔	✔
	WaitForTrig	✔		✔	✔
System Interrupt Handler	AssertSysreset	✔	✔	✔	✔
	DefaultACfailHandler	✔	✔	✔	✔
	DefaultBusErrorHandler	✔	✔	✔	✔
	DefaultSoftResetHandler	✔		✔	✔
	DefaultSysfailHandler	✔	✔	✔	✔
	DefaultSysresetHandler	✔	✔	✔	✔
	DisableACfail	✔	✔	✔	✔
	DisableSoftReset	✔		✔	✔
	DisableSysfail	✔	✔	✔	✔

Table A-1. Function Listing by Group

Group	Function	VXI	VME	C/C++	BASIC
System Interrupt Handler (continued)	DisableSysreset	✔	✔	✔	✔
	EnableACfail	✔	✔	✔	✔
	EnableSoftReset	✔		✔	✔
	EnableSysfail	✔	✔	✔	✔
	EnableSysreset	✔	✔	✔	✔
	GetACfailHandler	✔	✔	✔	
	GetBusErrorHandler	✔	✔	✔	
	GetSoftResetHandler	✔		✔	
	GetSysfailHandler	✔	✔	✔	
	GetSysresetHandler	✔	✔	✔	
	SetACfailHandler	✔	✔	✔	
	SetBusErrorHandler	✔	✔	✔	
	SetSoftResetHandler	✔		✔	
	SetSysfailHandler	✔	✔	✔	
	SetSysresetHandler	✔	✔	✔	
VXI/VMEbus Extender	MapECLtrig	✔		✔	✔
	MapTTLtrig	✔		✔	✔
	MapUtilBus	✔	✔	✔	✔
	MapVXIint	✔	✔	✔	✔

Table A-2. Function Listing by Name

Function	Group	VXI	VME	C/C++	BASIC
AcknowledgeTrig	Triggers	✔		✔	✔
AcknowledgeVXIint	VXI/VME Interrupt	✔	✔	✔	✔
AssertSysreset	System Interrupt Handler	✔	✔	✔	✔
AssertVXIint	VXI/VME Interrupt	✔	✔	✔	✔
CloseVXIlibrary	System Configuration	✔	✔	✔	✔
CreateDevInfo	System Configuration	✔	✔	✔	✔
DeAssertVXIint	VXI/VME Interrupt	✔	✔	✔	✔
DefaultACfailHandler	System Interrupt Handler	✔	✔	✔	✔
DefaultBusErrorHandler	System Interrupt Handler	✔	✔	✔	✔
DefaultSignalHandler	VXI Signal	✔	✔	✔	✔
DefaultSoftResetHandler	System Interrupt Handler	✔		✔	✔
DefaultSysfailHandler	System Interrupt Handler	✔	✔	✔	✔
DefaultSysresetHandler	System Interrupt Handler	✔	✔	✔	✔
DefaultTrigHandler	Triggers	✔		✔	✔
DefaultTrigHandler2	Triggers	✔		✔	✔
DefaultVXIintHandler	VXI/VME Interrupt	✔	✔	✔	✔
DefaultWSScmdHandler	Servant Word Serial Protocol	✔		✔	
DefaultWSSEcmdHandler	Servant Word Serial Protocol	✔		✔	
DefaultWSSLcmdHandler	Servant Word Serial Protocol	✔		✔	
DefaultWSSrdHandler	Servant Word Serial Protocol	✔		✔	
DefaultWSSwrtHandler	Servant Word Serial Protocol	✔		✔	

Table A-2. Function Listing by Name

Function	Group	VXI	VME	C/C++	BASIC
DisableACfail	System Interrupt Handler	✔	✔	✔	✔
DisableSignalInt	VXI Signal	✔		✔	✔
DisableSoftReset	System Interrupt Handler	✔		✔	✔
DisableSysfail	System Interrupt Handler	✔	✔	✔	✔
DisableSysreset	System Interrupt Handler	✔	✔	✔	✔
DisableTrigSense	Triggers	✔		✔	✔
DisableVXIint	VXI/VME Interrupt	✔	✔	✔	✔
DisableVXItoSignalInt	VXI/VME Interrupt	✔	✔	✔	✔
EnableACfail	System Interrupt Handler	✔	✔	✔	✔
EnableSignalInt	VXI Signal	✔		✔	✔
EnableSoftReset	System Interrupt Handler	✔		✔	✔
EnableSysfail	System Interrupt Handler	✔	✔	✔	✔
EnableSysreset	System Interrupt Handler	✔	✔	✔	✔
EnableTrigSense	Triggers	✔		✔	✔
EnableVXIint	VXI/VME Interrupt	✔	✔	✔	✔
EnableVXItoSignalInt	VXI/VME Interrupt	✔	✔	✔	✔
FindDevLA	System Configuration	✔	✔	✔	✔
GenProtError	Servant Word Serial Protocol	✔		✔	
GetACfailHandler	System Interrupt Handler	✔	✔	✔	
GetBusErrorHandler	System Interrupt Handler	✔	✔	✔	
GetByteOrder	Low-Level VXI/VMEbus Access	✔	✔	✔	✔

Table A-2. Function Listing by Name

Function	Group	VXI	VME	C/C++	BASIC
GetContext	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
GetDevInfo	System Configuration	✔	✔	✔	
GetDevInfoLong	System Configuration	✔	✔	✔	✔
GetDevInfoShort	System Configuration	✔	✔	✔	✔
GetDevInfoStr	System Configuration	✔	✔	✔	✔
GetMyLa	Local Resource Access	✔	✔	✔	✔
GetPrivilege	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
GetSignalHandler	VXI Signal	✔	✔	✔	
GetSoftResetHandler	System Interrupt Handler	✔		✔	
GetSysfailHandler	System Interrupt Handler	✔	✔	✔	
GetSysresetHandler	System Interrupt Handler	✔	✔	✔	
GetTrigHandler	Triggers	✔		✔	
GetVXIbusStatus	Low-Level VXI/VMEbus Access	✔	✔	✔	
GetVXIbusStatusInd	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
GetVXIintHandler	VXI/VME Interrupt	✔	✔	✔	
GetWindowRange	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
GetWSScmdHandler	Servant Word Serial Protocol	✔		✔	
GetWSSEcmdHandler	Servant Word Serial Protocol	✔		✔	
GetWSSLcmdHandler	Servant Word Serial Protocol	✔		✔	
GetWSSrdHandler	Servant Word Serial Protocol	✔		✔	
GetWSSwrtHandler	Servant Word Serial Protocol	✔		✔	

Table A-2. Function Listing by Name

Function	Group	VXI	VME	C/C++	BASIC
InitVXIlibrary	System Configuration	✔	✔	✔	✔
MapECLtrig	VXI/VMEbus Extender	✔		✔	✔
MapTrigToTrig	Triggers	✔		✔	✔
MapTTLtrig	VXI/VMEbus Extender	✔		✔	✔
MapUtilBus	VXI/VMEbus Extender	✔	✔	✔	✔
MapVXIAddress	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
MapVXIAddressSize	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
MapVXIint	VXI/VMEbus Extender	✔	✔	✔	✔
ReadMODID	Local Resource Access	✔		✔	✔
RespProtError	Servant Word Serial Protocol	✔		✔	
RouteSignal	VXI Signal	✔	✔	✔	✔
RouteVXIint	VXI/VME Interrupt	✔	✔	✔	✔
SetACfailHandler	System Interrupt Handler	✔	✔	✔	
SetBusErrorHandler	System Interrupt Handler	✔	✔	✔	
SetByteOrder	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
SetContext	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
SetDevInfo	System Configuration	✔	✔	✔	
SetDevInfoLong	System Configuration	✔	✔	✔	✔
SetDevInfoShort	System Configuration	✔	✔	✔	✔
SetDevInfoStr	System Configuration	✔	✔	✔	✔
SetMODID	Local Resource Access	✔		✔	✔
SetPrivilege	Low-Level VXI/VMEbus Access	✔	✔	✔	✔

Table A-2. Function Listing by Name

Function	Group	VXI	VME	C/C++	BASIC
SetSignalHandler	VXI Signal	✔	✔	✔	
SetSoftResetHandler	System Interrupt Handler	✔		✔	
SetSysfailHandler	System Interrupt Handler	✔	✔	✔	
SetSysresetHandler	System Interrupt Handler	✔	✔	✔	
SetTrigHandler	Triggers	✔		✔	
SetVXIintHandler	VXI/VME Interrupt	✔	✔	✔	
SetWSScmdHandler	Servant Word Serial Protocol	✔		✔	
SetWSSEcmdHandler	Servant Word Serial Protocol	✔		✔	
SetWSSLcmdHandler	Servant Word Serial Protocol	✔		✔	
SetWSSrdHandler	Servant Word Serial Protocol	✔		✔	
SetWSSwrtHandler	Servant Word Serial Protocol	✔		✔	
SignalDeq	VXI Signal	✔	✔	✔	✔
SignalEnq	VXI Signal	✔	✔	✔	✔
SignalJam	VXI Signal	✔	✔	✔	✔
SrcTrig	Triggers	✔		✔	✔
TrigAssertConfig	Triggers	✔		✔	✔
TrigCntrConfig	Triggers	✔		✔	✔
TrigExtConfig	Triggers	✔		✔	✔
TrigTickConfig	Triggers	✔		✔	✔
UnMapTrigToTrig	Triggers	✔		✔	✔
UnMapVXIAddress	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
VXIin	High-Level VXI/VMEbus Access	✔	✔	✔	✔
VXIinLR	Local Resource	✔	✔	✔	✔

Table A-2. Function Listing by Name

Function	Group	VXI	VME	C/C++	BASIC
	Access				
VXIinReg	High-Level VXI/VMEbus Access	✔		✔	✔
VXIintAcknowledgeMode	VXI/VME Interrupt	✔	✔	✔	✔
VXImemAlloc	Local Resource Access	✔	✔	✔	✔
VXImemCopy	Local Resource Access	✔	✔	✔	✔
VXImemFree	Local Resource Access	✔	✔	✔	✔
VXImove	High-Level VXI/VMEbus Access	✔	✔	✔	✔
VXIout	High-Level VXI/VMEbus Access	✔	✔	✔	✔
VXIoutLR	Local Resource Access	✔	✔	✔	✔
VXIoutReg	High-Level VXI/VMEbus Access	✔		✔	✔
VXIpeek	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
VXIpoke	Low-Level VXI/VMEbus Access	✔	✔	✔	✔
WaitForSignal	VXI Signal	✔	✔	✔	✔
WaitForTrig	Triggers	✔		✔	✔
WSabort	Commander Word Serial Protocol	✔		✔	✔
WSclr	Commander Word Serial Protocol	✔		✔	✔
WScmd/ WSEcmd/ WSLcmd	Commander Word Serial Protocol	✔		✔	✔
WSgetTmo	Commander Word Serial Protocol	✔		✔	✔
WSresp/ WSLresp	Commander Word Serial Protocol	✔		✔	✔
WSrd / WSrdi / WSrdl	Commander Word Serial Protocol	✔		✔	✔

Table A-2. Function Listing by Name

Function	Group	VXI	VME	C/C++	BASIC
WSrdf	Commander Word Serial Protocol	✔		✔	✔
WSSabort	Servant Word Serial Protocol	✔		✔	
WSSdisable	Servant Word Serial Protocol	✔		✔	
WSSenable	Servant Word Serial Protocol	✔		✔	
WSsetTmo	Word Serial Protocol	✔		✔	✔
WSSnoResp/ WSSLnoResp	Servant Word Serial Protocol	✔		✔	
WSSrd / WSSrdi / WSSrdl	Servant Word Serial Protocol	✔		✔	
WSSsendResp/ WSSLsendResp	Servant Word Serial Protocol	✔		✔	
WSSwrt / WSSwrti / WSSwrtl	Servant Word Serial Protocol	✔		✔	
WStrg	Commander Word Serial Protocol	✔		✔	✔
WSwrt / WSwrti / WSwrtl	Commander Word Serial Protocol	✔		✔	✔
WSwrtf	Commander Word Serial Protocol	✔		✔	✔

Appendix

B

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422 or (800) 327-3077
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 1 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as anonymous and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

FaxBack Support

FaxBack is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access FaxBack from a touch-tone telephone at (512) 418-1111.

E-Mail Support (currently U.S. only)

You can submit technical support questions to the appropriate applications engineering team through e-mail at the Internet addresses listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabWindows: lw.support@natinst.com
LabVIEW: lv.support@natinst.com
HiQ: hiq.support@natinst.com
VISA: visa.support@natinst.com
Lookout: lookout.support@natinst.com

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	Telephone	Fax
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Title _____

Company _____

Address _____

_____

Fax (____) _____ Phone (____) _____

Computer brand _____ Model _____ Processor _____

Operating system (include version number) _____

RAM _____ MB Display adapter _____

Mouse ____ yes _____ no Other adapters installed _____

Hard disk capacity _____ MB Brand _____

Instruments used _____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is _____

_____

_____

_____

List any error messages _____

_____

_____

The following steps will reproduce the problem _____

_____

_____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *NI-VXI™ User Manual*

Edition Date: July 1996

Part Number: 371702A-01

Please comment on the completeness, clarity, and organization of the manual.

_____

_____

_____

_____

_____

_____

_____

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

Phone (____) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Glossary

Prefix	Meaning	Value
n-	nano-	10^{-9}
m-	milli-	10^{-3}
K-	kilo-	10^{3}
M-	mega-	10^{6}
G-	giga-	10^{9}

A

A16 space — One of the VXIbus address spaces. Equivalent to the VME 64 KB short address space. In VXI, the upper 16 KB of A16 space is allocated for use by VXI devices configuration registers. This 16 KB region is referred to as VXI configuration space.

A24 space — One of the VXIbus address spaces. Equivalent to the VME 16 MB standard address space.

A32 space — One of the VXIbus address spaces. Equivalent to the VME 4 GB extended address space.

ACFAIL* — A VMEbus backplane signal that is asserted when a power failure has occurred (either AC line source or power supply malfunction), or if it is necessary to disable the power supply (such as for a high temperature condition).

address — Character code that identifies a specific location (or series of locations) in memory.

address modifier — One of six signals in the VMEbus specification used by VMEbus masters to indicate the address space and mode (supervisory/nonprivileged, data/program/block) in which a data transfer is to take place.

address space — A set of 2^n memory locations differentiated from other such sets in VXI/VMEbus systems by six signal lines known as address modifiers. *n* is the number of address lines required to uniquely specify a byte location in a given space. Valid numbers for *n* are 16, 24, and 32.

address window — A range of address space that can be accessed from the application program.

ANSI — American National Standards Institute

ASCII — American Standard Code for Information Interchange. A 7-bit standard code adopted to facilitate the interchange of data among various types of data processing and data communications equipment.

ASIC — Application-Specific Integrated Circuit (a custom chip)

asserted — A signal in its active true state.

asynchronous — Not synchronized; not controlled by periodic time signals, and therefore unpredictable with regard to the timing of execution of commands.

ASYNC Protocol — A two-device, two-line handshake trigger protocol using two consecutive even/odd trigger lines (a source/acceptor line and an acknowledge line).

B

backplane — An assembly, typically a PCB, with 96-pin connectors and signal paths that bus the connector pins. A C-size VXIbus system will have two sets of bused connectors called the J1 and J2 backplanes. A D-size VXIbus system will have three sets of bused connectors called the J1, J2, and J3 backplane.

base address — A specified address that is combined with a *relative* address (or offset) to determine the *absolute* address of a data location. All VXI address windows have an associated base address for their assigned VXI address spaces.

BAV	Word Serial Byte Available command. Used to transfer 8 bits of data from a Commander to its Servant under the Word Serial Protocol.
BERR*	Bus error signal. This signal is asserted by either a slave device or the bus time out (BTO) unit when an incorrect transfer is made on the Data Transfer Bus (DTB). The BERR* signal is also used in VXI for certain protocol implementations such as writes to a full Signal register and synchronization under the Fast Handshake Word Serial Protocol.
binary	A numbering system with a base of 2.
bit	Binary digit. The smallest possible unit of data: a two-state, yes/no, 0/1 alternative. The building block of binary coding and numbering systems. Several bits make up a *byte*.
bit vector	A string of related bits in which each bit has a specific meaning.
BREQ	Word Serial Byte Request query. Used to transfer 8 bits of data from a Servant to its Commander under the Word Serial Protocol.
BTO	See *bus timeout unit*.
buffer	Temporary memory/storage location for holding data before it can be transmitted elsewhere.
bus master	A device that is capable of requesting the Data Transfer Bus (DTB) for the purpose of accessing a slave device.
bus timeout unit	A VMEbus functional module that times the duration of each data transfer on the Data Transfer Bus (DTB) and terminates the DTB cycle if the duration is excessive. Without the termination capability of this module, a bus master could attempt to access a nonexistent slave, resulting in an indefinitely long wait for a slave response.
byte	A grouping of adjacent binary digits operated on by the computer as a single unit. A byte consists of 8 bits.
byte order	How bytes are arranged within a word or how words are arranged within a longword. Motorola ordering stores the most significant byte (MSB) or word first, followed by the least significant byte (LSB) or word. Intel ordering stores the LSB or word first, followed by the MSB or word.

C

clearing	Replacing the information in a register, storage location, or storage unit with zeros or blanks.
CLK10	A 10 MHz, ± 100 ppm, individually buffered (to each module slot), differential ECL system clock that is sourced from Slot 0 and distributed to Slots 1 through 12 on P2. It is distributed to each slot as a single-source, single-destination signal with a matched delay of under 8 ns.
command	A directive to a device. In VXI, three types of commands are as follows: • In Word Serial Protocol, a 16-bit imperative to a servant from its Commander (written to the Data Low register); • In Shared Memory Protocol, a 16-bit imperative from a client to a server, or vice versa (written to the Signal register); • In Instrument devices, an ASCII-coded, multi-byte directive.
Commander	A message-based device which is also a bus master and can control one or more Servants.
communications registers	In message-based devices, a set of registers that are accessible to the device's Commander and are used for performing Word Serial Protocol communications.
configuration registers	A set of registers through which the system can identify a module device type, model, manufacturer, address space, and memory requirements. In order to support automatic system and memory configuration, the VXIbus specification requires that all VXIbus devices have a set of such registers.
controller	A device that is capable of controlling other devices. A desktop computer with a MXI interface board, an embedded computer in a VXI chassis, a VXI-MXI, and a VME-MXI may all be controllers depending on the configuration of the VXI system.
CR	Carriage Return; the ASCII character 0Dh.

D

Data Transfer Bus	One of four buses on the VMEbus backplane. The DTB is used by a bus master to transfer binary data between itself and a slave device.
decimal	Numbering system based upon the 10 digits 0 to 9. Also known as base 10.
de-referencing	Accessing the contents of the address location pointed to by a pointer.
default handler	Automatically installed at startup to handle associated interrupt conditions; the software can then replace it with a specified handler.
DIR	Data In Ready. This is a bit in the Response register of a message-based device that indicates that the device is ready to accept data from its Commander.
DIRviol	Data In Ready violation. A type of word serial protocol error that occurs when the Commander attempts to write data to the device when the device is not ready.
DOR	Data Out Ready. This is a bit in the Response register of a message-based device that indicates that the device is ready to output data to its Commander.
DORviol	Data Out Ready violation. A type of word serial protocol error that occurs when the Commander attempts to read data from the device when the device is not ready.
DRAM	Dynamic RAM (Random Access Memory); storage that the computer must refresh at frequent intervals.
DTB	See *Data Transfer Bus*.

E

ECL	Emitter-Coupled Logic
embedded controller	A computer plugged directly into the VXI backplane. An example is the National Instruments VXIpc-850.
END	Signals the end of a data string.

EOS	End Of String; a character sent to designate the last byte of a data message.
ERR	Protocol error
Event signal	A 16-bit value written to a message-based device's Signal register in which the most significant bit (bit 15) is a 1, designating an Event (as opposed to a Response signal). The VXI specification reserves half of the Event values for definition by the VXI Consortium. The other half are user defined.
Extended Class device	A class of VXIbus device defined for future expansion of the VXIbus specification. These devices have a subclass register within their configuration space that defines the type of extended device.
extended controller	The external controller plus all of the extending controllers to which it is directly connected. An example is an AT-MXI connected to a VXI-MXI.
Extended Longword Serial Protocol	A form of Word Serial communication in which Commanders and Servants communicate with 48-bit data transfers.
extending controller	A mainframe extender that has additional VXIbus controller capabilities. An example is the VXI-MXI.
external controller	A desktop computer or workstation connected to the VXI system via a MXI interface board. An example is a standard personal computer with a PCI-MXI-2 installed.

F

FHS	Fast Handshake; a mode of the Word Serial Protocol which uses the VXIbus signals DTACK* and BERR* for synchronization instead of the Response register bits.
FIFO	First In-First Out; a method of data storage in which the first element stored is the first one retrieved.

G

GPIB	General-Purpose Interface Bus; the industry-standard IEEE 488 bus.

GPIO	General-Purpose Input Output, a module within the National Instruments TIC chip which is used for two purposes. First, GPIOs are used for connecting external signals to the TIC chip for routing/conditioning to the VXIbus trigger lines. Second, GPIOs are used as part of a crosspoint switch matrix.

H

handshaking	A type of protocol that makes it possible for two devices to synchronize operations.
hardware context	The hardware setting for address space, access privilege, and byte ordering.
hex	Hexadecimal; the numbering system with base 16, using the digits 0 to 9 and letters A to F.
high-level	Programming with instructions in a notation more familiar to the user than machine code. Each high-level statement corresponds to several low-level machine code instructions and is machine-independent, meaning that it is portable across many platforms.
Hz	Hertz; a measure of cycles per second.

I

I/O	Input/output; the techniques, media, or devices used to achieve communication between entities.
IACK	Interrupt Acknowledge
IEEE	Institute of Electrical and Electronics Engineers
IEEE 1014	The VME specification. Its full title is *ANSI/IEEE 1014-1987, IEEE Standard for a Versatile Backplane Bus: VMEbus.*
IEEE 1155	The VXI specification. Its full title is *ANSI/IEEE 1155-1992, VMEbus Extensions for Instrumentation: VXIbus*.
IEEE 488	Standard 488-1978, which defines the GPIB. Its full title is *IEEE Standard Digital Interface for Programmable Instrumentation*. Also referred to as IEEE 488.1 since the adoption of IEEE 488.2.

IEEE 488.2	A supplemental standard for GPIB. Its full title is *Codes, Formats, Protocols and Common Commands.*
INT16	A 16-bit signed integer; may also be called a *short integer* or *word*.
INT32	A 32-bit signed integer; may also be called a *long* or *longword*.
INT8	An 8-bit signed integer; may also be called a *char*.
interrupt	A means for a device to notify another device that an event occurred.
interrupt handler	A functional module that detects interrupt requests generated by interrupters and performs appropriate actions.
interrupter	A device capable of asserting interrupts and responding to an interrupt acknowledge cycle.
INTX	Interrupt and Timing Extension; a daughter card option for MXI mainframe extenders that extends interrupt lines and reset signals on VME boards. On VXI boards it also extends trigger lines and the VXIbus CLK10 signal.

K

KB	1,024 or 2^{10}
kilobyte	A thousand bytes.

L

LF	Linefeed; the ASCII character 0Ah.
logical address	An 8-bit number that uniquely identifies the location of each VXIbus device's configuration registers in a system. The A16 register address of a device is C000h + Logical Address * 40h.
longword	Data type of 32-bit integers.
Longword Serial Protocol	A form of Word Serial communication in which Commanders and Servants communicate with 32-bit data transfers instead of 16-bit data transfers as in the normal Word Serial Protocol.
low-level	Programming at the system level with machine-dependent commands.

M

MB	1,048,576 or 2^{20}
mapping	Establishing a range of address space for a one-to-one correspondence between each address in the window and an address in VXIbus memory.
master	A functional part of a MXI/VME/VXIbus device that initiates data transfers on the backplane. A transfer can be either a read or a write.
megabyte	A million bytes.
Memory Class device	A VXIbus device that, in addition to configuration registers, has memory in VME A24 or A32 space that is accessible through addresses on the VME/VXI data transfer bus.
message-based device	An intelligent device that implements the defined VXIbus registers and communication protocols. These devices are able to use Word Serial Protocol to communicate with one another through communication registers.
MODID	A set of 13 signal lines on the VXI backplane that VXI systems use to identify which modules are located in which slots in the mainframe.
MQE	Multiple Query Error; a type of Word Serial Protocol error. If a Commander sends two Word Serial queries to a Servant without reading the response to the first query before sending the second query, a MQE is generated.
multitasking	The ability of a computer to perform two or more functions simultaneously without interference from one another. In operating system terms, it is the ability of the operating system to execute multiple applications/processes by time-sharing the available CPU resources.
MXIbus	Multisystem eXtension Interface Bus; a high-performance communication link that interconnects devices using round, flexible cables.
ms	Milliseconds

N

NI-VXI	The National Instruments bus interface software for VME/VXIbus systems.
nonprivileged access	One of the defined types of VMEbus data transfers; indicated by certain address modifier codes. Each of the defined VMEbus address spaces has a defined nonprivileged access mode.
NULL	A special value to denote that the contents (usually of a pointer) are invalid or zero.

O

octal	Numbering system with base 8, using numerals 0 to 7.

P

parse	The act of interpreting a string of data elements as a command to perform a device-specific action.
peek	To read the contents.
pointer	A data structure that contains an address or other indication of storage location.
poke	To write a value
ppm	Parts per million
privileged access	See *Supervisory Access*.
propagation	Passing of signal through a computer system.
protocol	Set of rules or conventions governing the exchange of information between computer systems.

Q

query	Like command, causes a device to take some action, but requires a response containing data or other information. A command does not require a response.

queue	A group of items waiting to be acted upon by the computer. The arrangement of the items determines their processing priority. Queues are usually accessed in a FIFO fashion.

R

read	To get information from any input device or file storage media.
register	A high-speed device used in a CPU for temporary storage of small amounts of data or intermediate results during processing.
register-based device	A Servant-only device that supports only the four basic VXIbus configuration registers. Register-based devices are typically controlled by message-based devices via device-dependent register reads and writes.
remote controller	A device in the VXI system that has the capability to control the VXIbus, but has no intelligent CPU installed. An example is the VXI-MXI-2.
REQF	Request False; a VXI Event condition transferred using either VXI signals or VXI interrupts, indicating that a Servant no longer has a need for service.
REQT	Request True; a VXI Event condition transferred using either VXI signals or VXI interrupts, indicating that a Servant has a need for service.
resman	The name of the National Instruments Resource Manager application in the NI-VXI bus interface software. See *Resource Manager*.
Resource Manager	A message-based Commander located at Logical Address 0, which provides configuration management services such as address map configuration, Commander and Servant mappings, and self-test and diagnostic management.
Response signal	Used to report changes in Word Serial communication status between a Servant and its Commander.
ret	Return value.
RM	See *Resource Manager*

ROAK	Release On Acknowledge; a type of VXI interrupter which always deasserts its interrupt line in response to an IACK cycle on the VXIbus. All message-based VXI interrupters must be ROAK interrupters.
ROR	Release On Request; a type of VME bus arbitration where the current VMEbus master relinquishes control of the bus only when another bus master requests the VMEbus.
RORA	Release On Register Access; a type of VXI/VME interrupter which does not deassert its interrupt line in response to an IACK cycle on the VXIbus. A device-specific register access is required to remove the interrupt condition from the VXIbus. The VXI specification recommends that VXI interrupters be only ROAK interrupters.
RR	Read Ready; a bit in the Response register of a message-based device used in Word Serial Protocol indicating that a response to a previously sent query is pending.
RRviol	Read Ready protocol violation; a type of Word Serial Protocol error. If a Commander attempts to read a response from the Data Low register when the device is not Read Ready (does not have a response pending), a Read Ready violation may be generated.
rsv	Request Service; a bit in the status byte of an IEEE 488.1 and 488.2 device indicating a need for service. In VXI, whenever a new need for service arises, the rsv bit should be set and the REQT signal sent to the Commander. The rsv bit should be automatically deasserted when the Word Serial Read Status Byte query is sent.

S

s	Seconds
SEMI-SYNC Protocol	A one-line, open collector, multiple-device handshake trigger protocol.
Servant	A device controlled by a Commander.
setting	To place a binary cell into the 1 (non-zero) state.
Shared Memory Protocol	A communications protocol for message-based devices that uses a block of memory that is accessible to both a client and a server. The memory block acts as the medium for the protocol transmission.
short integer	Data type of 16 bits, same as *word*.

signal	Any communication between message-based devices consisting of a write to a Signal register. Sending a signal requires that the sending device have VMEbus master capability.
signed integer	*n* bit pattern, interpreted such that the range is from $-2^{(n-1)}$ to $+2^{(n-1)} -1$.
slave	A functional part of a MXI/VME/VXIbus device that detects data transfer cycles initiated by a VMEbus master and responds to the transfers when the address specifies one of the device's registers.
SMP	See *Shared Memory Protocol*.
SRQ	Service Request
status/ID	A value returned during an IACK cycle. In VME, usually an 8-bit value which is either a status/data value or a vector/ID value used by the processor to determine the source. In VXI, a 16-bit value used as a data; the lower 8 bits form the VXI logical address of the interrupting device and the upper 8 bits specify the reason for interrupting.
STST	START/STOP trigger protocol; a one-line, multiple-device protocol that can be sourced only by the VXI Slot 0 device and sensed by any other device on the VXI backplane.
supervisory access	One of the defined types of VMEbus data transfers; indicated by certain address modifier codes.
synchronous communications	A communications system that follows the command/response cycle model. In this model, a device issues a command to another device; the second device executes the command and then returns a response. Synchronous commands are executed in the order they are received.
SYNC Protocol	The most basic trigger protocol, simply a pulse of a minimum duration on any one of the trigger lines.
SYSFAIL*	A VMEbus signal that is used by a device to indicate an internal failure. A failed device asserts this line. In VXI, a device that fails also clears its PASSed bit in its Status register.
SYSRESET*	A VMEbus signal that is used by a device to indicate a system reset or power-up condition.
system clock driver	A VMEbus functional module that provides a 16 MHz timing signal on the utility bus.

System Controller	A functional module that has arbiter, daisy-chain driver, and MXIbus cycle timeout responsibility. Always the first device in the MXIbus daisy-chain.
system hierarchy	The tree structure of the Commander/Servant relationships of all devices in the system at a given time. In the VXIbus structure, each Servant has a Commander. A Commander can in turn be a Servant to another Commander.

T

TIC	Trigger Interface Chip; a proprietary National Instruments ASIC used for direct access to the VXI trigger lines. The TIC contains a 16-bit counter, a dual 5-bit tick timer, and a full crosspoint switch.
tick	The smallest unit of time as measured by an operating system.
trigger	Either TTL or ECL lines used for intermodule communication.
tristated	Defines logic that can have one of three states: low, high, and high-impedance.
TTL	Transistor-Transistor Logic

U

unasserted	A signal in its inactive false state.
UINT8	An 8-bit unsigned integer; may also be called an *unsigned char*.
UINT16	A 16-bit unsigned integer; may also be called an *unsigned short* or *word*.
UINT32	A 32-bit unsigned integer; may also be called an *unsigned long* or *longword*.
unsigned integer	*n* bit pattern interpreted such that the range is from 0 to 2^n -1.
UnSupCom	Unsupported Command; a type of Word Serial Protocol error. If a Commander sends a command or query to a Servant which the Servant does not know how to interpret, an Unsupported Command protocol error is generated.

V

VIC	VXI Interactive Control program, a part of the NI-VXI bus interface software package. Used to program VXI devices, and develop and debug VXI application programs. Called *VICtext* when used on text-based platforms.
VME	Versa Module Eurocard or IEEE 1014
VMEbus Class device	Also called non-VXIbus or foreign devices when found in VXIbus systems. They lack the configuration registers required to make them VXIbus devices.
void	In the C language, a generic data type that can be cast to any specific data type.
VXIbus	VMEbus Extensions for Instrumentation
VXIedit	VXI Resource Editor program, a part of the NI-VXI bus interface software package. Used to configure the system, edit the manufacturer name and ID numbers, edit the model names of VXI and non-VXI devices in the system, as well as the system interrupt configuration information, and display the system configuration information generated by the Resource Manager. Called *vxitedit* when used on text-based platforms.
VXItedit	Text based version of `VXIedit`

W

Word Serial Protocol	The simplest required communication protocol supported by message-based devices in the VXIbus system. It utilizes the A16 communication registers to perform 16-bit data transfers using a simple polling handshake method.
word	A data quantity consisting of 16 bits.
write	Copying data to a storage device.
WR	Write Ready; a bit in the Response register of a message-based device used in Word Serial Protocol indicating the ability for a Servant to receive a single command/query written to its Data Low register.

WRviol	Write Ready protocol violation; a type of Word Serial Protocol error. If a Commander attempts to write a command or query to a Servant that is not Write Ready (already has a command or query pending), a Write Ready protocol violation may be generated.
WSP	See *Word Serial Protocol*

Index

D

E

F

G

H

I

L

M

N

O

R

S

T

U

V

W