NI-VXI™

C Software Reference Manual for VME

November 1993 Edition

Part Number 320389-01

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices:
Australia (03) 879 9422, Austria (0662) 435986, 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, Italy 02/48301892, Japan (03) 3788-1921, Netherlands 03480-33466, Norway 32-848400, Spain (91) 640 0085, Sweden 08-730 49 70, Switzerland 056/20 51 51, U.K. 0635 523545

Limited 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

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.

Contents

Chapter 8
System Interrupt Handler Functions

Chapter 9
VXI/VMEbus Extender Functions

Figures

About This Manual

This manual describes in detail the features of the NI-VXI software and the function calls for programming VME systems in C language.

Organization of This Manual

The *NI-VXI C Software Reference Manual for VME* is organized as follows:

- Chapter 1, *Introduction to NI-VXI for VME*, introduces you to the concepts of VME, VXI (VME eXtensions for Instrumentation), and MXI (Multisystem eXtension Interface), and to the NI-VXI software.
- Chapter 2, *Introduction to the NI-VXI Functions for VME*, introduces you to the NI-VXI functions and their capabilities for programming VME systems, discusses the use of function parameters, and describes application environments for which the functions are designed.
- Chapter 3, *System Configuration Functions*, describes the C syntax and use of the VXI system configuration functions. These functions copy all of the Resource Manager 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. This chapter defines the parameters and shows examples of the use of each function. The descriptions are listed alphabetically for easy reference.
- Chapter 4, *Low-Level VMEbus Access Functions*, describes the C syntax and use of the low-level VMEbus access functions. Low-level VMEbus access is the fastest access method for directly reading from or writing to any of the VMEbus address spaces. This chapter defines the parameters and shows examples of the use of each function. The descriptions are listed alphabetically for easy reference.
- Chapter 5, *High-Level VMEbus Access Functions*, describes the C syntax and use of the high-level VMEbus access functions. With high-level VMEbus access functions, you have direct access to the VMEbus address spaces. You can use these functions to read, write, and move blocks of data between any of the VXIbus address spaces. When execution speed is not a critical issue, these functions provide an easy-to-use interface. This chapter defines the parameters and shows examples of the use of each function. The descriptions are listed alphabetically for easy reference.
- Chapter 6, *Local Resource Access Functions*, describes the C syntax and use of the VME local resource access functions. With these functions, you have access to miscellaneous local resources such as the local CPU VME register set and the local CPU VME Shared RAM. These functions are useful for shared memory type communication and debugging purposes. This chapter defines the parameters and shows examples of the use of each function. The descriptions are listed alphabetically for easy reference.
- Chapter 7, *VME Interrupt Functions*, describes the C syntax and use of the VME interrupt functions and default handler. VME interrupts are a basic form of asynchronous communication used by VME devices with VME interrupter support. These functions can specify the status/ID processing method, install interrupt service routines, and assert specified VME interrupt lines with a specified status/ID value. This chapter defines the parameters and shows examples of the use of each function. The descriptions are listed alphabetically for easy reference.
- Chapter 8, *System Interrupt Handler Functions*, describes the C syntax and use of the VME system interrupt handler functions and default handlers. With these functions, you can handle miscellaneous system conditions that can occur in the VME environment. This chapter defines the parameters and shows examples of the use of each function. The descriptions are listed alphabetically for easy reference.

- Chapter 9, *VXI/VMEbus Extender Functions*, describes the C syntax and use of the VXI/VMEbus extender functions. These functions can be used to dynamically reconfigure multi-mainframe transparent mapping of the VME interrupt and utility bus signals. This chapter defines the parameters and shows examples of the use of each function. The descriptions are listed alphabetically for easy reference.
- The appendix, *Customer Communication*, directs you where you can find forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- 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

Throughout this manual, the following conventions are used to distinguish elements of text:

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 NI-VXI function calls, sections of code, function syntax, parameter names, console responses, and syntax examples.
bold italic	Text in this font denotes an important note.

Numbers in this manual are base 10 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).

Terminology that is specific to a chapter or section is defined at its first occurrence.

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 1.2 (part number 340007-01)
- *NI-VXI Software Reference Manual for C* (for VXI systems, part number 320307-01)
- VXI-1, *VXIbus System Specification*, Revision 1.4, VXIbus Consortium
  (available from National Instruments, part number 350083-01)
- VXI-6, *VXIbus Mainframe Extender Specification*, Revision 1.0, VXIbus Consortium
  (available from National Instruments, part number 340258-01)

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 and your Getting Started manual contain comment and configuration forms for you to complete. These forms are in the appendix, *Customer Communication*, at the end of this manual.

Chapter 1
Introduction to NI-VXI for VME

This chapter introduces you to the concepts of VME, VXI (VME eXtensions for Instrumentation), and MXI (Multisystem eXtension Interface), and to the NI-VXI software. You can use this chapter as an overview of the VXIbus, because the NI-VXI software was actually designed for use in VXI systems. Because VXI is a superset of VME, the NI-VXI software includes a comprehensive set of programming tools that work with VME systems. This manual describes the NI-VXI functions that apply to the initialization of your VME interface and the programming of your VME system.

About the NI-VXI Functions for VME

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

This manual describes the NI-VXI functions in groups based on their functionality. Chapter 2, *Introduction to the NI-VXI Functions for VME*, describes these groups and explains how you can use the functions within a group in VME systems. Chapters 3 through 9 completely define each function within a group.

VXIbus Overview

This section introduces some of the concepts of the VXIbus specification.

VXI is a superset of VME. VXI defines additional board sizes beyond VME and defines the cooling and EMC specifications for both the mainframe chassis and the modules installed in the system.

Most importantly, VXI reserves a portion of VME address space (the upper 16 KB of Short (A16) space) as VXI configuration space. All VXI devices have configuration registers that reside in this portion of memory. These configuration registers are used to programmatically configure the system. In addition, VXI devices may optionally have a standardized message-passing protocol called *Word Serial Protocol*. Because VME systems do not implement the VXI-defined Word Serial Protocol, they do not require the NI-VXI Word Serial functions.

In a VXI system, a system initialization program called the *Resource Manager* configures the system on power-up or after a backplane RESET. A VME system uses the Resource Manager program only to initialize the hardware interface between your computer and the VMEbus.

VXI Devices

Unlike VME devices, a VXI device has a unique logical address, which serves as a means of accessing the device in the VXI system. This logical address is analogous to a GPIB device address. Because VXI uses an 8-bit logical address, you can have up to 256 VXI devices in a VXI system. The logical address specifies the 64-byte boundary in the upper 16 KB of Short (A16) space where the VXI device's configuration/communication registers reside. It is important that none of your VME devices occupies any of the addresses within this space.

Each VXI device must have a specific set of registers, called *configuration registers* (see Figure 1-1).

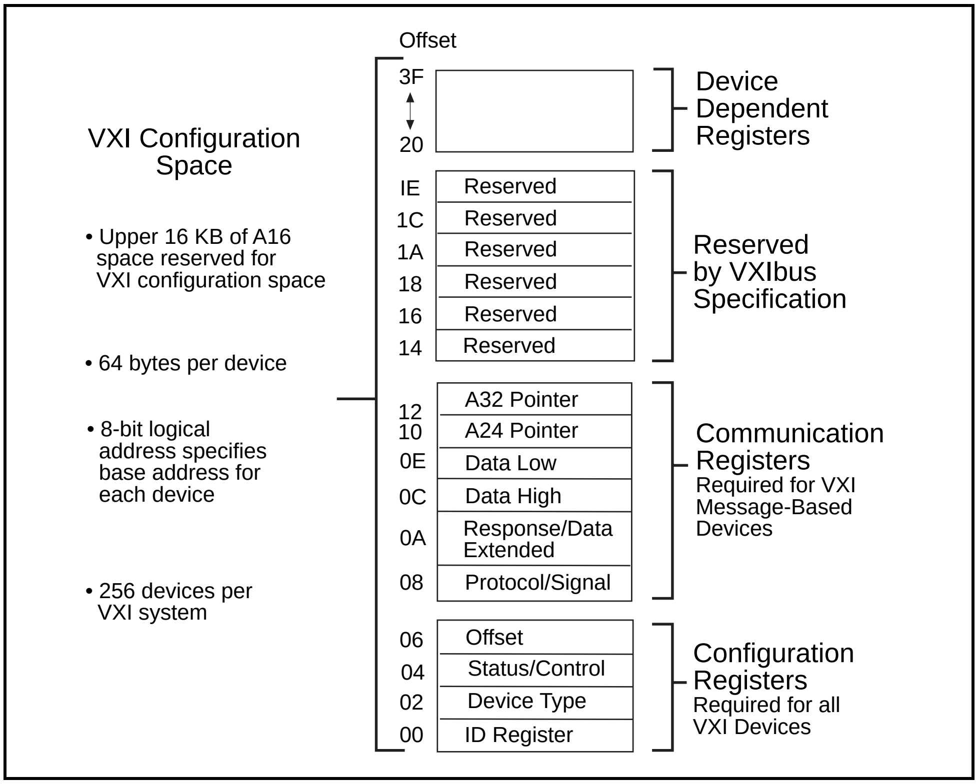

Figure 1-1. VXI Configuration Registers

Register-Based VXI 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 VXI *Resource Manager* (RM), essentially a software module, can perform automatic system and memory configuration when a VXI system is initialized. A VME system uses the Resource Manager program only to configure the hardware interface between your computer and the VMEbus.

Message-Based Devices

In addition to Register-Based devices, the VXIbus specification also defines *Message-Based* devices, which are required to have *communication registers* as well as the configuration registers. All Message-Based VXIbus devices, regardless of the manufacturer, can communicate at a minimum level using the VXI-specified Word Serial Protocol, as shown in Figure 1-2. In addition, you can establish higher-performance communication channels, such as shared-memory channels, to take advantage of the VXIbus bandwidth capabilities.

Note: ***VME devices do not implement the VXI-defined Word Serial Protocol. For this reason, the NI-VXI Word Serial functions are not required in VME systems.***

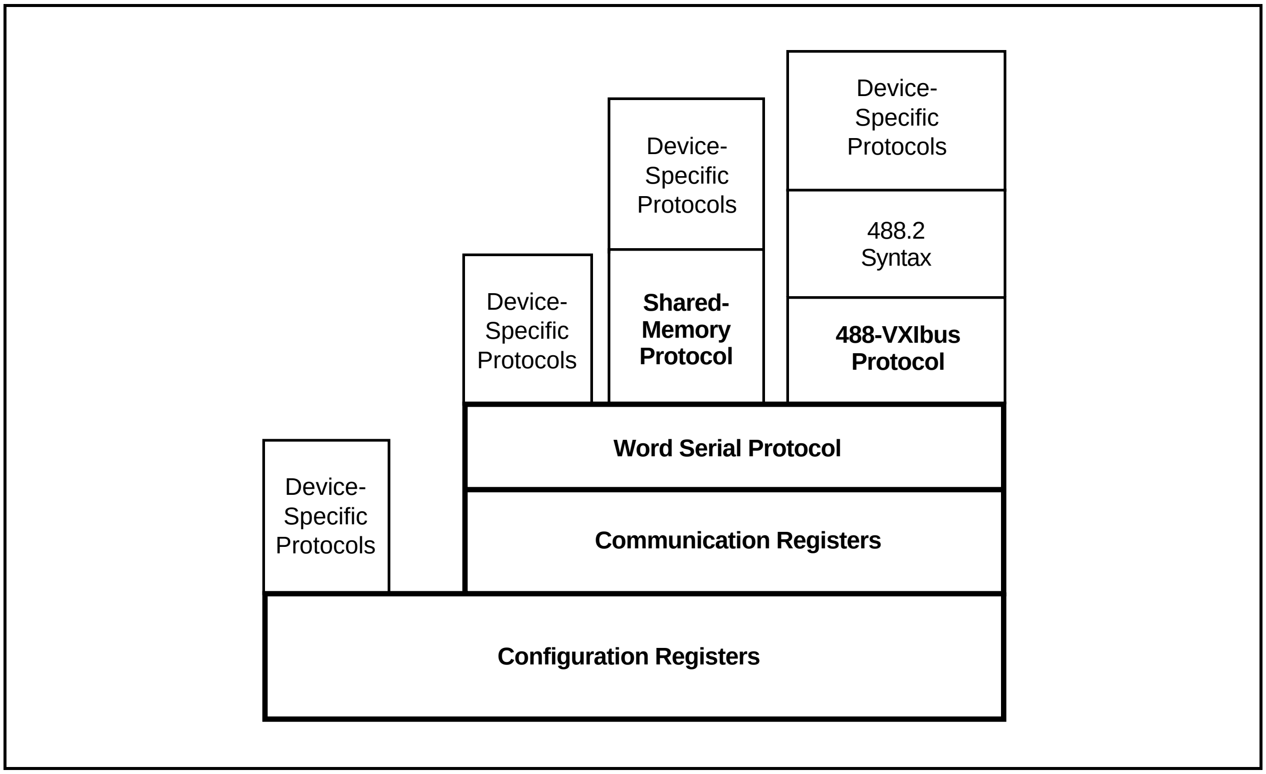

Figure 1-2. VXI Software Protocols

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, Word Serial Protocol has messages for configuring Message-Based devices and the system resources.

All VXI Message-Based devices are required to use Word Serial Protocol and communicate in a standard way. The protocol is called *word serial*, because communication with a Message-Based device is performed by reading and writing 16-bit words one at a time to and from the Data Out (read Data Low) and Data In (write 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 Universal Asynchronous Receiver Transmitter (UART) on a serial port.

As mentioned earlier, VME devices do not implement the VXI-defined Word Serial Protocol and do not use the NI-VXI Word Serial functions.

Commander/Servant Hierarchies

The VXIbus specification defines a Commander/Servant communication protocol you can use to construct hierarchical systems with conceptual layers of VXI devices. This structure is like an inverted 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 Word Serial Protocol to communicate with Servants through the Servants' communication registers. Servants communicate with their Commander by responding to the Word Serial commands and queries from their Commander. Servants can also communicate asynchronous status and events to their Commander through VME hardware interrupts, or by writing specific messages directly to their Commander's Signal register.

Note: ***VME devices do not support the VXI-defined concept of a Commander/Servant hierarchy, so it does not apply to VME systems.***

Interrupts and Asynchronous Events

In a VXI system, Servant devices communicate asynchronous status and events to their Commander device either through traditional VME hardware interrupts or by writing specific messages (signals) directly to their Commander's hardware Signal register. Devices that are *not* bus masters always transmit such information via interrupts, whereas VXI devices that *do* have bus master capability can either use interrupts or send signals. Some devices can receive only signals, while others might be only interrupt handlers.

The VXIbus specification defines Word Serial commands that a Commander uses to 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.

Note: ***These capabilities do not apply to VME systems.***

Although the Word Serial Protocol is reserved for Commander/Servant communications, you can establish peer-to-peer communication between two VXI devices through a specified shared-memory protocol or simply by writing specific messages directly to the device's Signal register. While these peer-peer protocols are possible in VME, they cannot be standardized at the same level as VXI.

MXIbus Overview

The MXIbus is a high-performance communication link that interconnects devices using round, flexible cables. MXI operates like modern backplane computer buses, but is 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, recognized that VXI requires a new generation of connectivity for the instrumentation systems of the future. National Instruments developed the MXIbus specification over a period of two years and announced it in April 1989 as an open industry standard.

National Instruments offers MXI interface products for a variety of platforms, including the VXIbus and VMEbus backplane systems, and the PC AT, EISA, PS/2, Sun SPARCstation, Macintosh, DECstation 5000, and IBM RISC System/6000 computer systems. These MXI products directly and transparently couple these industry-standard computers to the VXIbus and the VMEbus backplanes. They also transparently extend VXI/VME across multiple mainframes, and seamlessly integrate external devices that cannot physically fit on a plug-in module into a VXI/VME system.

Chapter 2
Introduction to the NI-VXI Functions for VME

This chapter introduces you to the NI-VXI functions and their capabilities for programming VME systems, discusses the use of function parameters, and describes application environments for which the functions are designed. The NI-VXI functions were actually designed for use in VXI systems. Because VXI is a superset of VME, you can also use the NI-VXI functions as a comprehensive set of programming tools for VME systems. This manual describes the NI-VXI functions that apply to the initialization of your VME interface and the programming of your VME system. The NI-VXI C language interface is independent of the hardware platform and the operating system environment.

NI-VXI Functions for VME

This manual describes in detail only the types of NI-VXI functions that you can use with VME systems. These functions fall into the following groups:

- **System Configuration Functions**
  The System Configuration functions provide the lowest level initialization of your VME interface. In addition, the System Configuration functions can retrieve or modify device configuration information.

- **Low-Level VMEbus Access Functions**
  Low-Level VMEbus access is the fastest access method for directly reading from or writing to any of the VMEbus address spaces. You can use these functions to obtain a pointer that is directly mapped to a particular VMEbus address. How the pointer is used is at the discretion of the application. When using the Low-Level Access functions, your application must take into account certain programming constraints and error conditions such as Bus Error (BERR*).

- **High-Level VMEbus Access Functions**
  Similar to the Low-Level VMEbus Access functions, the High-Level VMEbus Access functions give you direct access to the VMEbus address spaces. You can use these functions to read, write, and move blocks of data between any of the VMEbus address spaces. You can specify any VMEbus privilege mode or byte order. The functions trap and report Bus Errors. When the execution speed is not the most critical issue, the High-Level VMEbus Access functions provide an easy-to-use interface.

- **Local Resource Access Functions**
  Local Resource Access functions let you access 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, for non-Resource Manager operation (when the local CPU is not the Resource Manager), and for debugging purposes.

- **Interrupt Functions**
  The Interrupt functions let you process individual interrupt status/ID values from VME interrupters as VME status/IDs, VXI status/IDs, or VXI signals. By default, status/IDs are processed as VXI signals (either with an interrupt service routine or by queuing on the global signal queue). For a VME system, you must specify that the interrupt functions process the status/IDs as VME status/IDs. The interrupt functions can specify the status/ID processing method and install interrupt service routines. In addition, the interrupt functions can assert specified interrupt lines on the backplane with a specified status/ID value.

- **System Interrupt Handler Functions**
  The System Interrupt Handler functions let you install interrupt service routines for the system interrupt conditions. These conditions include Sysfail, ACfail, Bus Error, and Sysreset interrupts.

- **VXI/VMEbus Extender Functions**
  The VXI/VMEbus Extender functions can dynamically reconfigure multiple-mainframe transparent mapping of the interrupt lines and the utility bus signals between multiple VME chassis interconnected via the MXIbus. They also include VXI-specific functions for extending the VXI TTL triggers and ECL triggers between multiple mainframes in a VXI system. The trigger functions are VXI-specific and are not used in VME systems. For this reason, they are not fully documented in this manual. The National Instruments Resource Manager configures the mainframe extenders with settings based on user-modifiable configuration files.

VXI-Specific Functions

Your NI-VXI distribution diskette contains a variety of programming functions that apply only to VXI systems. Because these VXI-specific functions are not used in VME systems, they are not documented comprehensively in this manual. For your information, however, the following paragraphs briefly describe these functions. For a complete description of these VXI-specific functions, refer to the *NI-VXI Software Reference Manual for C* (part number 320307-01).

- **Commander Word Serial Protocol Functions**
  Word Serial is the minimal mode of communication between VXI Message-Based devices. Commander Word Serial functions give you all of 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 command/query sending and buffer reads/writes.

- **Servant Word Serial Protocol Functions**
  Servant Word Serial functions give you all of the necessary capabilities 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 Signal Functions**
  VXI Signals are an alternate method for VXI Bus Masters to interrupt another device. The value written to a device's Signal register has the same format as the status/ID value returned during a VXI interrupt acknowledge cycle. You can route VXI signals to an interrupt service routine or queue them on a global signal queue. You can use these functions to specify the signal routing, install interrupt service routines, manipulate the global signal queue, and specify how long to wait for a particular signal value (or set of values) to be received.

- **VXI Trigger Functions**
  The VXI Trigger functions are a standard interface for sourcing and accepting any of the VXIbus TTL or ECL trigger lines. The VXI trigger functions support all VXI-defined trigger protocols. The actual capabilities depend on the specific hardware platform. The VXI trigger functions can install interrupt service routines for various trigger interrupt conditions.

Calling Syntax

This manual uses a generic syntax to describe each function and its arguments. The function syntaxes used are C programming language specific. The C language 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 source level-compatible), whether it is non-multitasking (for example, MS-DOS), pseudo multitasking (such as MS Windows or Macintosh OS), multitasking (for example, UNIX or OS/2), or real time (such as pSOS+ 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 a complete, full-function 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, and the applications you develop are portable across either platform.

National Instruments offers VXI development systems for these two platforms that link the NI-VXI driver software into LabWindows/CVI to control VXI instruments from either embedded VXI 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 parameter types are used for all the NI-VXI functions in the following chapters and in the actual NI-VXI libraries 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.

```
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 function calls pass function parameters by value (not by reference), you need to specify the address of the parameter when the parameter is an output parameter. The C "&" operator accomplishes this task.

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

Because `value` is an output parameter, `&value` is sent to the function instead of just `value`. `la` and `reg` would be considered input parameters.

Return Parameters and System Errors

All NI-VXI functions return a status indicating a degree of 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. This return value usually occurs only when an operating system IOCTL call to the driver fails, but could also occur because of system errors. This error is specific to the operating system on which the NI-VXI interface is running. If your system is configured correctly and does not conflict with other operating system drivers, this error should never occur. On systems in which NI-VXI is a linkable library, this error code is never returned.

Multiple Mainframe Support

The NI-VXI functions described in this manual fully 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 system. Refer to VXI-6, *VXIbus Mainframe Extender Specification*, Revision 1.0, for more details on multiple-mainframe systems.

If you have a multiple-mainframe system, please continue with the following sections in this chapter. If you have a single-mainframe system, you can proceed to the other chapters in this manual.

Embedded Versus External and Extended Controllers

The two basic types of multiple-mainframe configurations are the *embedded* CPU (controller) configuration and the *external* CPU (controller) configuration (see Figure 2-1 for examples). The embedded CPU configuration is an intelligent CPU interface directly plugged into the backplane. The embedded CPU must have all of its required VXI/VME interface capabilities built onto the embedded CPU itself. An embedded CPU has direct access to the VXI/VME backplane for which it is installed. Access to other mainframes is done through the use of mainframe extenders.

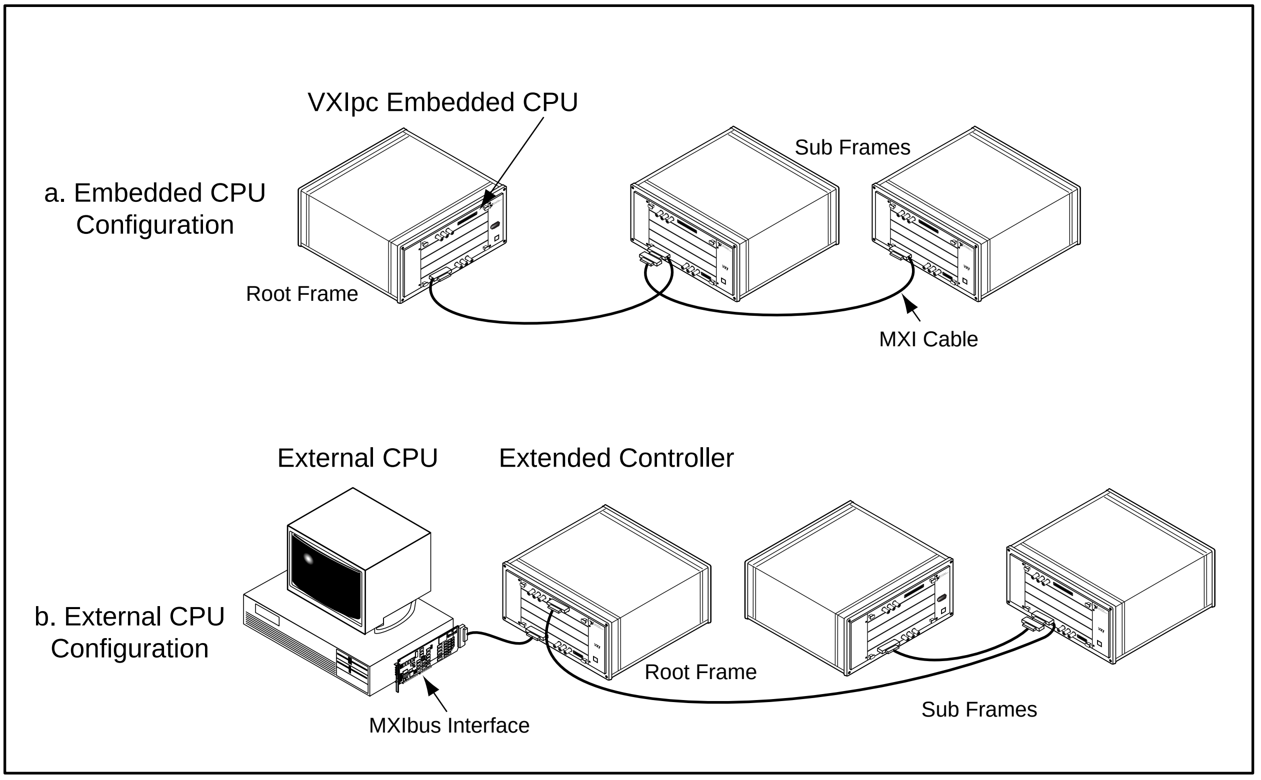

Figure 2-1. Embedded Versus External CPU Configurations

The external CPU configuration involves plugging an interface board into an existing computer that connects the external CPU to VXI/VME mainframes via one or more VXI/VME extended controllers. An extended controller is a mainframe extender with additional VXI/VMEbus controller capabilities.

Special features outside of the scope of the *VXIbus Mainframe Extender Specification* have been added to National Instruments MXIbus mainframe extender products for more complete support of the VXI/VMEbus capabilities. These features give the external CPU all of the features of an embedded CPU, including VXI interrupt, TTL trigger, and ECL trigger for VXI systems, and Sysfail, ACfail, and Sysreset support for VXI/VME systems. The external computer uses these features to interrupt on, sense, and/or assert these backplane signals. The specific capabilities of the MXIbus mainframe extender are dependent upon the specific product and configuration.

Extended controllers exist only on the first level of mainframe hierarchy, as Figure 2-1 illustrates. The first level of hierarchy for the embedded CPU is always the local mainframe. Because of this, the embedded CPU will never have any extended controllers. An external CPU along with an extended controller is functionally equivalent to an embedded CPU configuration. An external CPU with more than one extended controller is a superset of the embedded CPU configuration. If the application requires the local CPU (external or embedded) to receive VXI interrupts, triggers (in VXI systems), and utility signals from below the first level of mainframe hierarchy, you should extend these signals using the transparent extender method (requiring INTX support on MXI extender products) via the Resource Manager configuration or Extender functions.

The Extender Versus Controller Parameters

This document uses the `extender` and `controller` parameters to specify the VXI/VME mainframe to which a particular function applies. In general, the value of the `extender` or `controller` parameter is either the local CPU or the logical address of the VXI/VME mainframe extender device that is used to access the particular mainframe (for example, a VXI-MXI or VME-MXI). Refer to Figure 2-2 for an example of mainframe extenders used with the `extender` and/or `controller` parameters.

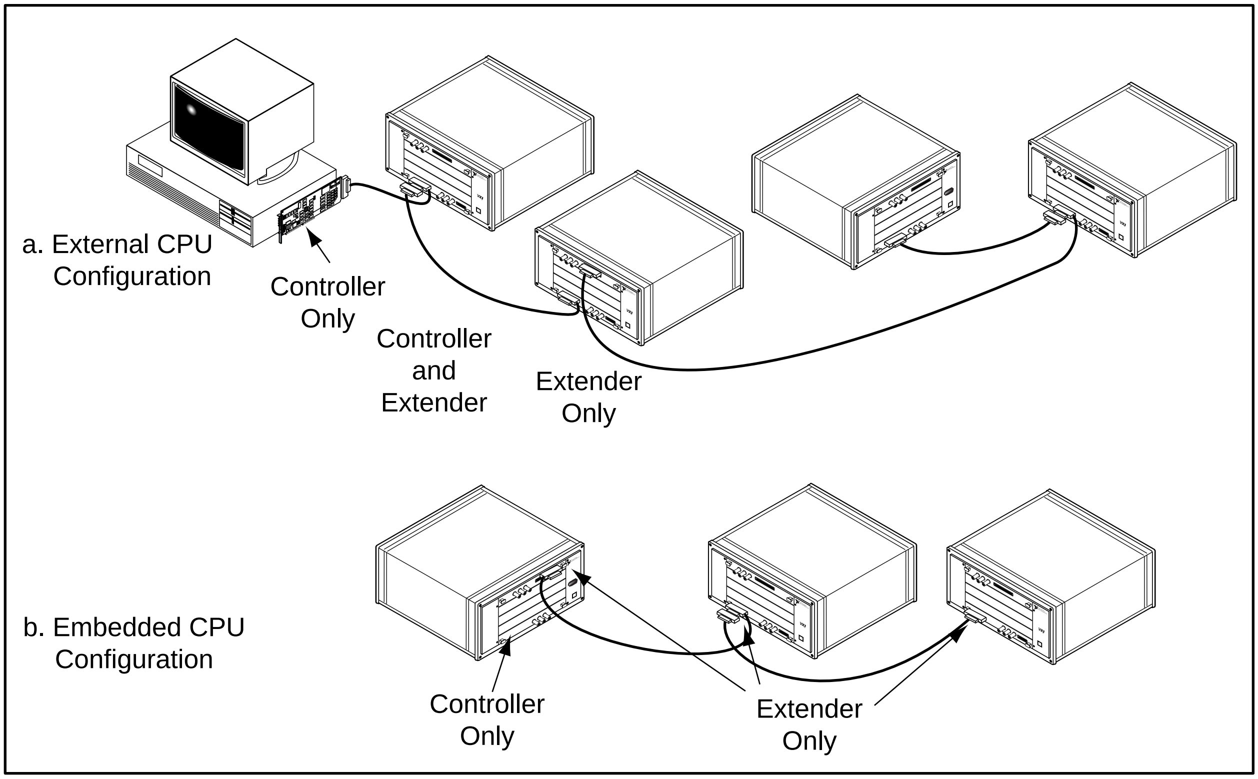

Figure 2-2. Extender Versus Controller Parameters

You can use the `extender` parameter only with the VXI/VME Extender functions, which are fully described in Chapter 9, *VXIbus Extender Functions*. With these functions, you can reconfigure the transparent mainframe extension configured by the Resource Manager. The extensions included are interrupts, the utility bus (Sysfail, ACfail, and Sysreset), and TTL and ECL triggers (for VXI systems only). The capabilities of the VXI/VME Extender functions are mapped directly onto the capabilities of the individual mapping registers of the standard VXIbus or VMEbus mainframe extender. Because the Resource Manager configures the mainframe extenders with settings based on user-modifiable configuration files, your application probably will never need the VXI/VMEbus Extender functions.

You will find the `controller` parameter only in NI-VXI functions that apply to embedded or extended controller capabilities. These capabilities include interrupt, ACfail, Sysfail, and TTL/ECL trigger (only for VXI systems) services. In embedded CPU configurations, you must always use a -1 or local CPU logical address for the `controller` parameter to specify the local resources of the embedded CPU. For external CPU configurations, a -1 or local CPU logical address specifies the first extended controller (mainframe extender with the lowest logical address).

You can use other values in external CPU configurations that have more than one extended controller. In this case, the `controller` parameter value specifies the particular extended controller for which the functions should apply. As a result, you can use different sets of VXI/VMEbus resources within individual first-level mainframes (for example, different interrupt levels handled on a per-mainframe basis). Notice that having more than one extended controller is not directly portable to the embedded CPU configuration.

NI-VXI Multiple Mainframe Portability

You should aim to achieve full portability between an external CPU configuration and an embedded CPU configuration in any multiple-platform application. Assuming that the extended controller and the embedded CPU have the required hardware support, single-mainframe systems have no configuration portability problems. Single-mainframe systems do not require functions that use the `extender` parameter for multiple-mainframe extension, and functions that use the `controller` parameter always specify the single extended controller or embedded CPU by default.

However, for direct portability of a multiple-mainframe configuration, you should probably not use multiple mainframes (extended controllers) on the first level of the hierarchy. Because the first link into VXI/VME for an embedded CPU is a single backplane interface (and not multiple backplane interfaces), there is no functional equivalent to the external CPU multiple extended controller configuration. Figure 2-3 shows an example of this type of configuration.

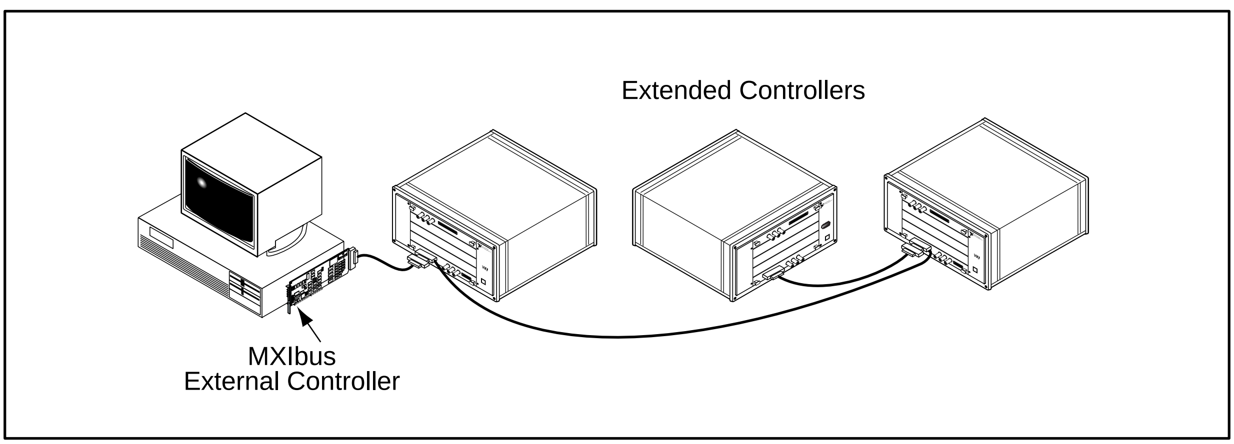

Figure 2-3. External CPU Configuration with Multiple Extended Controllers

While this configuration may be advantageous for certain applications, it is not directly portable to an embedded CPU configuration (the embedded CPU configuration is more restrictive). For external CPU configurations, the only equivalent configuration is one extended controller on the first link from the external CPU. You should extend any additional mainframes out of the first (root) frame. Figure 2-1 illustrates this type of configuration. When looking for portability problems between the two types of configurations, always consider the combination of the external CPU and its associated mainframe extender as equivalent to an embedded CPU. The special features of the MXI mainframe extenders give the external CPU the extended VXI/VMEbus capabilities of an embedded CPU (on a per-mainframe basis). The NI-VXI interface treats the combination of the external CPU and the MXI mainframe extenders (extended controllers) as equivalent to an embedded CPU.

It is possible to change the external CPU configuration shown in Figure 2-1 into a multiple first-level mainframe configuration. Figure 2-3 shows how you could arrange the three mainframes. Notice that the first (root) mainframe has two mainframe extenders in Figure 2-1 in order to make a two-level mainframe hierarchy, whereas the configuration in Figure 2-3 has only one. The multiple first-level case always saves one mainframe extender interface. This savings may overcome the portability advantages for your application.

On the other hand, it is possible to make a multiple-mainframe configuration such as the system in Figure 2-3 fully compatible with the embedded CPU configuration in Figure 2-1. Multiple mainframes on the first level in an external CPU situation are not software-compatible with the embedded CPU situation for one reason. Any functions that use the `controller` parameter with values other than -1 or the local CPU logical address would return error codes when used in the embedded CPU configuration. Using these `controller` parameter values implies that more than one extended controller has interrupts, Sysfail, ACfail, and/or trigger (for VXI systems only) conditions controlled directly by the external CPU. For full portability, you need to avoid this situation, which you can do by transparently mapping the Resource Manager and the VXI/VMEbus Extender functions (requiring INTX support for MXIbus mainframe extenders). You must map all first-level mainframe interrupts, Sysfails, ACfails, and triggers (for VXI systems only) into the first-level mainframe with the lowest logical address (the default extended controller). From this point, the only value of the `controller` parameter required is -1 or the local CPU logical address. You can then achieve transparent operation of the `controller` parameter functions and direct portability to the embedded CPU configuration.

Chapter 3
System Configuration Functions

This chapter describes the C syntax and use of the VXI system configuration functions. These functions copy all of the Resource Manager (RM) table information into data structures at startup and help you 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. Only in very special cases should you modify the information in the table, which you can do using the `SetDevInfo` function. In this manner, both the application and the driver functions can directly access all the necessary VXI system information. Your application must call the `CloseVXIlibrary` function upon exit to free all data structures and disable interrupts.

Functional Overview

The following paragraphs describe the system configuration 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.

InitVXIlibrary ()

`InitVXIlibrary` is the application startup 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.

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`.

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 or `""` specifies to ignore the corresponding field in the attribute comparison. After finding the VXI logical address, you can use one of the `DevInfo` functions to get any information about the specified device.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

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 `field`s include the device name, Commander's logical address, mainframe number, slot, manufacturer ID number, model code, model name, device class, address space/base/size allocated, 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.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

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 `fieldvalue`s of `GetDevInfo`. `GetDevInfoShort` returns only the `field`s of `GetDevInfo` that are *16-bit integers*. Possible `field`s include the Commander's logical address, mainframe number, slot, manufacturer ID number, manufacturer name, model code, device class, address space allocated, interrupt lines/handlers allocated, protocols supported, and so on.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

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 `fieldvalue`s of `GetDevInfo`. `GetDevInfoLong` returns only the `field`s of `GetDevInfo` that are *32-bit integers*. Possible `field`s include the address base and size allocated to the device by the RM.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

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 `fieldvalue`s of `GetDevInfo`. `GetDevInfoStr` returns only the `field`s of `GetDevInfo` that are *character strings*. Possible `field`s include the device name, manufacturer name, and model name.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

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 `field`s include the device name, Commander's logical address, mainframe number, slot, manufacturer ID number, manufacturer name, model code, model name, device class, address space/base/size allocated, 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 system. No initial changes are necessary for VXI/VME devices.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

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 `fieldvalue`s of `SetDevInfo`. `SetDevInfoShort` changes only the `field`s of `SetDevInfo` that are *16-bit integers*. Possible `field`s include the Commander's logical address, mainframe number, slot, manufacturer ID number, model code, device class, address space allocated, 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 system. No initial changes are necessary for VXI/VME devices.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

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 `fieldvalue`s of `SetDevInfo`. `SetDevInfoLong` returns only the `field`s of `SetDevInfo` that are *32-bit integers*. Possible `field`s 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 system. No initial changes are necessary for VXI/VME devices.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

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 `fieldvalue`s of `SetDevInfo`. `SetDevInfoStr` returns only the `field`s of `SetDevInfo` that are *character strings*. Possible `field`s 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 system. No initial changes are necessary for VXI/VME devices.

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

CreateDevInfo (la)

`CreateDevInfo` creates a new entry in the NI-VXI RM table for the specified logical address. It installs default null values into the entry. You must use one of the `DevInfo` functions after this point to change any of the device information as needed. Use this function only in very special situations. At the startup of your application, `InitVXIlibrary` completely initializes the RM table to how the RM configured the system. No initial changes/creations are necessary for VXI/VME devices. You can use `CreateDevInfo` to add non-VXI devices or pseudo devices (future expansion).

In a VXI system, device information is encoded in the VXI-defined register set that is required for all VXI devices. Because VME devices do not have these VXI-required registers, specific device information will not be automatically recorded by the Resource Manager. For this reason, you must enter this information manually by using the Non-VXI Device Editor of the NI-VXI software. When the Resource Manager executes, it recognizes information regarding VME devices that you configured with the Non-VXI Device Editor. You can then use this function to access the information during run-time.

Function Descriptions

The following paragraphs describe the system configuration functions. The descriptions are explained at the C syntax level and are listed in alphabetical order.

CloseVXIlibrary

Syntax: `ret = CloseVXIlibrary ()`

Action: Disables interrupts and frees dynamic memory allocated for the internal device information table. This function should be called before the application is exited.

Remarks: Parameters:

none

Return value:

`ret`	`int16`	Return Status

0 = NI-VXI library closed successfully
1 = Successful; previous `InitVXIlibrary` calls still pending
-1 = NI-VXI library was not open

Example:

```
/* Close the NI-VXI library. */

main()
{
   int16 ret;

   ret = InitVXIlibrary();
   if (ret < 0)
         /* RM table memory allocation or file open failed. */;

   /*
         Application-specific program.
   */

   ret = CloseVXIlibrary();
}
```

CreateDevInfo

Syntax: `ret = CreateDevInfo (la)`

Action: Allocates space in the device information table for a new entry with logical address `la`. It sets fields for the entry in the device information table to default values (null or unasserted values).

Remarks: Input parameters:

`la`	`int16`	Logical address of device for which to create entry

Return value:

`ret`	`int16`	Return Status
		0 = Entry successfully created -1 = `la` already exists -2 = `la` out of range 0 to 511 -3 = Dynamic memory allocation failure

Example:

```
/* Create a new entry for pseudo logical address 298. */

int16     ret;
uint16    la;

la = 298;
ret = CreateDevInfo (la);
if (ret != 0)
   /* Error creating new entry. */;
```

FindDevLA

Syntax: `ret = FindDevLA (namepat, manid, modelcode, devclass, slot, mainframe, cmdrla, la)`

Action: Finds a device with the specified attributes in the device information table and returns its logical address. If the `namepat` parameter is `""` or any other attribute is -1, that attribute is not used in the matching algorithm. For `namepat`, it accepts a partial name (for example, for `GPIB-VXI` it will accept `GPI`). If two or more devices match, it returns the logical address of the first device found.

Remarks: Input parameters:

`namepat`	`int8[14]`	Name Pattern
`manid`	`int16`	VXI Manufacturer ID number
`modelcode`	`int16`	Manufacturer's 12-bit model number
`devclass`	`int16`	Device class of the device -1 = Any 0 = Memory Class Device 1 = Extended Class Device 2 = Message-Based Device 3 = Register-Based Device
`slot`	`int16`	Slot location of the device
`mainframe`	`int16`	Mainframe location of device (logical address of extender)
`cmdrla`	`int16`	Commander's logical address

Output parameter:

`la`	`int16*`	Logical address of the device found

Return value:

`ret`	`int16`	Return Status 0 = A device matching the specification was found -1 = No device matching the specification was found

Example:

```
/* Find the logical address of a device with manid = 0xff6
   (National Instruments) and modelcode = 0xff (GPIB-VXI). */

int16     ret;
int8      *namepat;
int16     manid;
int16     modelcode;
int16     devclass;
int16     mainframe;
int16     slot;
int16     cmdrla;
int16     la;

namepat = "";
manid = 0xff6;
modelcode = 0xff;
devclass = -1;
mainframe = -1;
slot = -1;
cmdrla = -1;
ret = FindDevLA (namepat, manid, modelcode, devclass, mainframe,
slot, cmdrla, &la);
if (ret != 0)
    /* No device with manid = 0xff6 and modelcode = 0xff was
       found. */;
else
    /* Device was found; logical address in la. */;
```

GetDevInfo

Syntax: `ret = GetDevInfo (la, field, fieldvalue)`

Action: Gets device information about a specified device.

Remarks: Input parameters:

`la`	`int16`	Logical address of device to get information about
`field`	`uint16`	Field identification number

Field	Type	Description
0	`struct`	Retrieve entire RM table entry for the specified device (structure of all of the following)
1	`int8[14]`	Device name
2	`int16`	Commander's logical address
3	`int16`	Mainframe
4	`int16`	Slot
5	`uint16`	Manufacturer identification number
6	`int8[14]`	Manufacturer name
7	`uint16`	Model code
8	`int8[14]`	Model name
9	`uint16`	Device class
10	`uint16`	Extended subclass (if extended class device)
11	`uint16`	Address space used
12	`uint32`	Base of A24/A32 memory
13	`uint32`	Size of A24/A32 memory
14	`uint16`	Memory type and access time
15	`uint16`	Bit vector list of VXI interrupter lines
16	`uint16`	Bit vector list of VXI interrupt handler lines
17	`uint16`	Mainframe extender, controller information

Bits	Description
15 to 13	Reserved
12	1 = Child side extender 0 = Parent side extender
11	1 = Frame extender 0 = Not frame extender
10	1 = Extended controller
9	1 = Embedded controller
8	1 = External controller
7 to 0	Frame extender towards root frame

Field	Type	Description
18	`uint16`	Asynchronous mode control state
19	`uint16`	Response enable state
20	`uint16`	Protocols supported
21	`uint16`	Capability/status flags
22	`uint16`	Status state (Pass/Fail, Ready/Not Ready)

Output parameter:

`fieldvalue`	`void*`	Information for that field (size dependent on field)

Return value:

`ret`	`int16`	Return Status

0 = The specified information was returned
-1 = Device not found
-2 = Invalid `field` specified

Example:

```
/* Get the model code of a device at Logical Address 4. */

int16      ret;
int16      la;
uint16     field;
uint16     fieldvalue;

la = 4;
field = 7;
ret = GetDevInfo (la, field, &fieldvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

GetDevInfoLong

Syntax: `ret = GetDevInfoLong (la, field, longvalue)`

Action: Gets information about a specified device from the device information table. This function is layered on top of `GetDevInfo` and returns only those fields that are 32-bit integers.

Remarks: Input parameters:

`la`	`int16`	Logical address of device to get information about
`field`	`uint16`	Field identification number

Field	Description
12	Base of A24/A32 memory
13	Size of A24/A32 memory

Output parameter:

`longvalue`	`uint32*`	Information for that field

Return value:

`ret`	`int16`	Return Status

0 = The specified information was returned
-1 = Device not found
-2 = Invalid `field`

Example:

```
/* Get the A24 base of a device at Logical Address 4. */

int16     ret;
uint16    la;
uint16    field;
uint32    longvalue;

la = 4;
field = 12;
ret = GetDevInfoLong (la, field, &longvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

GetDevInfoShort

Syntax: `ret = GetDevInfoShort (la, field, shortvalue)`

Action: Gets information about a specified device from the device information table. This function is layered on top of `GetDevInfo` and returns only those fields that are 16-bit integers.

Remarks: Input parameters:

`la`	`int16`	Logical address of device to get information about
`field`	`uint16`	Field identification number

Field	Description
2	Commander's logical address
3	Mainframe
4	Slot
5	Manufacturer identification number
7	Model code
9	Device class
10	Extended subclass (if extended class device)
11	Address space used
14	Memory type and access time
15	Bit vector list of VXI interrupter lines
16	Bit vector list of VXI interrupt handler lines
17	Mainframe extender and controller information

Bits	Description
15 to 13	Reserved
12	1 = Child side extender 0 = Parent side extender
11	1 = Frame extender 0 = Not frame extender
10	1 = Extended controller
9	1 = Embedded controller
8	1 = External controller
7 to 0	Frame extender towards root frame

Field	Description
18	Asynchronous mode control state
19	Response enable state
20	Protocols supported
21	Capability/status flags
22	Status state (Passed/Failed, Ready/Not Ready)

Output parameter:

`shortvalue`	`uint16*`	Information for that field

Return value:

`ret`	`int16`	Return Status

0 = The specified information was returned
-1 = Device not found
-2 = Invalid `field`

Example:

```
/* Get the model code of a device at Logical Address 4. */

int16     ret;
uint16    la;
uint16    field;
uint16    shortvalue;

la = 4;
field = 7;
ret = GetDevInfoShort (la, field, &shortvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

GetDevInfoStr

Syntax: `ret = GetDevInfoStr (la, field, stringvalue)`

Action: Gets information about a specified device from the device information table. This function is layered on top of `GetDevInfo` and returns only those fields that are character strings.

Remarks: Input parameters:

`la`	`int16`	Logical address of device to get information about
`field`	`uint16`	Field identification number

Field	Description
1	Device name
6	Manufacturer name
8	Model name

Output parameter:

`stringvalue`	`uint8*`	Buffer to receive information for that field

Return value:

`ret`	`int16`	Return Status

0 = The specified information was returned
-1 = Device not found
-2 = Invalid `field`

Example:

```
/* Get the model name of a device at Logical Address 4. */

int16      ret;
int16      la;
uint16     field;
uint8      stringvalue[14];

la = 4;
field = 8;
ret = GetDevInfoStr (la, field, stringvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

InitVXIlibrary

Syntax: `ret = InitVXIlibrary ()`

Action: Allocates and initializes the data structures required by the NI-VXI library functions. This function reads the RM table file and copies all of the device information into data structures in local memory. It also performs other initialization operations, such as installing the default interrupt handlers and initializing their associated global variables.

Remarks: Parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = NI-VXI library initialized 1 = NI-VXI library already initialized (repeat call) -1 = RM table memory allocation failed

Example:

```
/* Initialize for using the library functions. */

main()
{
   int16 ret;

   ret = InitVXIlibrary();
   if (ret < 0)
         /* RM table memory allocation or file open failed. */;

   /*
         Application-specific program.
   */

   ret = CloseVXIlibrary();
}
```

SetDevInfo

Syntax: `ret = SetDevInfo (la, field, fieldvalue)`

Action: Sets information about a specified device in the device information table.

Remarks: Input parameters:

`la`	`int16`	Logical address of device to set information for
`field`	`uint16`	Field identification number

Field	Type	Description
0	`struct`	Retrieve entire RM table entry for the specified device (structure of all of the following)
1	`int8[14]`	Device name
2	`int16`	Commander's logical address
3	`int16`	Mainframe
4	`int16`	Slot
5	`uint16`	Manufacturer identification number
6	`int8[14]`	Manufacturer name
7	`uint16`	Model code
8	`int8[14]`	Model name
9	`uint16`	Device class
10	`uint16`	Extended subclass (if extended class device)
11	`uint16`	Address space used
12	`uint32`	Base of A24/A32 memory
13	`uint32`	Size of A24/A32 memory
14	`uint16`	Memory type and access time
15	`uint16`	Bit vector list of VXI interrupter lines
16	`uint16`	Bit vector list of VXI interrupt handler lines
17	`uint16`	Mainframe extender, controller information

Bits	Description
15 to 13	Reserved
12	1 = Child side extender 0 = Parent side extender
11	1 = Frame extender 0 = Not frame extender
10	1 = Extended controller
9	1 = Embedded controller
8	1 = External controller
7 to 0	Frame extender towards root frame

Field	Type	Description
18	`uint16`	Asynchronous mode control state
19	`uint16`	Response enable state
20	`uint16`	Protocols supported
21	`uint16`	Capability/status flags
22	`uint16`	Status state (Pass/Fail, Ready/Not Ready)

`fieldvalue`	`void*`	Information for that field (size dependent on field)

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = The specified information was returned -1 = Device not found -2 = Invalid `field` specified

Example:

```
/* Set the model code of a device at Logical Address 4. */

int16     ret;
int16     la;
uint16    field;
uint32    fieldvalue;

la = 4;
field = 7;
fieldvalue = 0xffffL;
ret = SetDevInfo (la, field, &fieldvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

SetDevInfoLong

Syntax: `ret = SetDevInfoLong (la, field, longvalue)`

Action: Sets information about a specified device in the device information table. This function is layered on top of `SetDevInfo` and changes only those fields that are 32-bit integers.

Remarks: Input parameters:

`la`	`int16`	Logical address of device to set information for
`field`	`uint16`	Field identification number
	<u>Field</u>	<u>Description</u>
	12	Base of A24/A32 memory
	13	Size of A24/A32 memory
`longvalue`	`uint32`	Information for that field

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = The specified information was returned -1 = Device not found -2 = Invalid `field`

Example:

```
/* Set the A24 base of a device at Logical Address 4. */

int16     ret;
int16     la;
uint16    field;
uint32    longvalue;

la = 4;
field = 12;
longvalue = 0x200000L;
ret = SetDevInfoLong (la, field, longvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

SetDevInfoShort

Syntax: `ret = SetDevInfoShort (la, field, shortvalue)`

Action: Sets information about a specified device in the device information table. This function is layered on top of `SetDevInfo` and changes only those fields that are 16-bit integers.

Remarks: Input parameters:

`la`	`int16`	Logical address of device device to set information for
`field`	`uint16`	Field identification number

Field	Description
2	Commander's logical address
3	Mainframe
4	Slot
5	Manufacturer identification number
7	Model code
9	Device class
10	Extended subclass (if extended class device)
11	Address space used
14	Memory type and access time
15	Bit vector list of VXI interrupter lines
16	Bit vector list of VXI interrupt handler lines
17	Mainframe extender and controller information

Bits	Description
15 to 13	Reserved
12	1 = Child side extender 0 = Parent side extender
11	1 = Frame extender 0 = Not frame extender
10	1 = Extended controller
9	1 = Embedded controller
8	1 = External controller
7 to 0	Frame extender towards root frame

Field	Description
18	Asynchronous mode control state
19	Response enable state
20	Protocols supported
21	Capability/status flags
22	Status state (Passed/Failed, Ready/Not Ready)

`shortvalue`	`uint16`	Information for that field

Output parameters:

none

Return value:

`ret`	`int16`	Return Status

0 = The specified information was returned
-1 = Device not found
-2 = Invalid `field`

Example:

```
/* Set the model code of a device at Logical Address 4. */

int16     ret;
int16     la;
uint16    field;
uint16    shortvalue;

la = 4;
field = 7;
shortvalue = 0xffff;
ret = SetDevInfoShort (la, field, shortvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

SetDevInfoStr

Syntax: `ret = SetDevInfoStr (la, field, stringvalue)`

Action: Sets information about a specified device in the device information table. This function is layered on top of `SetDevInfo` and changes only those fields that are character strings.

Remarks: Input parameters:

`la`	`int16`	Logical address of device to set information for
`field`	`uint16`	Field identification number
	Field	Description
	1	Device name
	6	Manufacturer name
	8	Model name
`stringvalue`	`uint8*`	Buffer to receive information for that field

Output parameters:

none

Return value:

`ret`	`int16`	Return Status
		0 = The specified information was returned -1 = Device not found -2 = Invalid `field`

Example:

```
/* Set the model name of a device at Logical Address 4. */

int16     ret;
int16     la;
uint16    field;
uint8     stringvalue[14];

la = 4;
field = 8;
strcpy (stringvalue, "Device 1");
ret = SetDevInfoStr (la, field, stringvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

Chapter 4
Low-Level VMEbus Access Functions

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

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

Low-level and high-level access to the 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 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 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 VMEbus address spaces. 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 VMEbus address with a particular VME access privilege and byte ordering. How the C pointers are used is at the discretion of the application. You need to consider a number of issues when using the direct pointers:

- Byte, word, or longword accesses are made based on the de-reference of the C pointer.
- You need to determine bounds for the pointers.
- Based on the methods in which a particular hardware platform sets up access to VME address spaces, using more than one pointer can also result in conflicts.
- Your application must check error conditions such as Bus Error (BERR*) separately.

High-level VMEbus access functions need not take into account any of the considerations that are required by the low-level VMEbus access functions. The high-level VMEbus access functions have all necessary information for accessing a particular 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 VMEbus access functions automatically trap bus errors and return an appropriate error status. Using the high-level 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 VMEbus access functions act as an easy-to-use interface that can do any VMEbus accesses necessary for an application. For more information, refer to Chapter 5, *High-Level VMEbus Access Functions*.

Programming Considerations

All accesses to the 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 VMEbus (using a complex hardware interface). The areas where the address space of the local CPU is mapped onto the 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 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 VMEbus address spaces. Be sure to take into account the number and size of the windows provided by a particular platform. If mapping a pointer requires the use of the same window as another pointer already in existence, the window context must be saved and restored. 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. Based on your knowledge of the platform, you can make assumptions about the sizes of windows. If you are more concerned with portability of code, however, you should base your assumptions on the minimal support all of the target platforms. Not all platforms support all access modes (for example, 680X0 platforms do not support Intel byte ordering).

Note: ***We strongly recommend that your devices have all of the same access privileges and byte orders. The VXIbus specification requires that VXI devices respond to nonprivileged data privilege state (address modifier codes) with Motorola byte order. Following this principle will greatly increase overall throughput of the program. Otherwise, the application must keep saving and restoring the state of the windows into VMEbus address spaces.***

NI-VXI uses a term within this chapter called the hardware (or window) *context*. The hardware context for window to VME consists of the 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 (`#define`s) that simply de-reference the pointer. It is highly recommended to use these functions instead of performing the direct de-reference 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 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 `window` 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 `window` 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 non-multitasking systems. The advantage with Owner Privilege is that it gives the caller complete and exclusive access for that window, 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:

1. 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 `window` for Access Only Privilege.

2. 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 `window` 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.

Owner and Access Only Privilege Versus Interrupt Service Routines

Regardless of whether a window has Owner Privilege or Access Only Privilege, you may find it necessary to temporarily control a particular window for a period of time. An interrupt service routine is a good example of this type of situation. Because an interrupt service routine cannot *wait* for an `UnMapVXIAddress` call, the interrupt service routine must be able to temporarily take control of a particular window. To accomplish this task, you can use the `SaveContext` and `RestoreContext` functions. `SaveContext` logs the current settings of the windows and `RestoreContext` returns the windows to their old settings. Because an interrupt service routine can be suspended only by a higher level interrupt service routine, there is never any problem with the usage of `SaveContext` and `RestoreContext`.

Functional Overview

The following paragraphs describe the low-level VMEbus 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.

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

`MapVXIAddress` sets up a window into one of the VME address spaces and returns a pointer to a local address that will access the specified VME address. The `accessparms` parameter specifies Owner Privilege/Access Only Privilege, the VME address space, the 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 is used in various calls such as `GetWindowRange` or `GetContext` to get window settings. When a request for Owner Privilege is granted, you can also use this window identifier to change the hardware context for that window through the use of calls such as `SetContext` or `SetPrivilege`.

UnMapVXIAddress (window)

`UnMapVXIAddress` deallocates 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.

GetWindowRange (window, windowbase, windowend)

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

Note: ***Take into account that the Resource Manager assigns all windows to VME based on a power of two. The application can reduce or altogether exclude overhead for testing window bounds by keeping this in mind.***

VXIpeek (addressptr, width, value)

`VXIpeek` is a simple macro that reads a single byte, word, or longword from a particular address obtained by `MapVXIAddress`. On most systems using C language interfaces, `VXIpeek` is simply a macro to de-reference a C pointer. We recommend, however, that you use `VXIpeek` instead of a direct de-reference, as it supports portability between different platforms and programming languages.

VXIpoke (addressptr, width, value)

`VXIpoke` is a simple function that writes a single byte, word, or longword to a particular address obtained by `MapVXIAddress`. On most systems using C language interfaces, `VXIpoke` is simply a macro to de-reference a C pointer. We recommend, however, that you use `VXIpoke` instead of a direct de-reference, as it supports portability between different platforms and programming languages.

SaveContext (contextlist)

`SaveContext` retrieves the hardware interface settings (context) for all VME windows and unlocks all windows, effectively making it appear as if there are no Owner Privilege or Access Only Privilege accessors using any windows. In some applications, especially within an interrupt service routine, the application cannot wait for a process to unmap a particular window. You can use `SaveContext` along with `RestoreContext` to globally save and restore the hardware context for all the windows, while guaranteeing access to a particular VME window. `RestoreContext` restores the window settings to what they were before the interrupt service routine was called (from the point in which `SaveContext` was called).

RestoreContext (contextlist)

`RestoreContext` restores the hardware interface settings (context) for all VME windows from a previously saved context (via `SaveContext`). In some applications, especially within an interrupt service routine, the application cannot wait for a process to unmap a particular window. You can use `SaveContext` along with `RestoreContext` to globally save and restore the hardware context for all the windows, while guaranteeing access to a particular VME window.

SetContext (window, context)

`SetContext` sets all of the hardware interface settings (context) for a particular 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 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 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`.

GetContext (window, context)

`GetContext` retrieves all of the hardware interface settings (context) for a particular 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 VME interface hardware state (context) for a particular window.

SetPrivilege (window, priv)

`SetPrivilege` sets the VMEbus windowing hardware to access the specified window with the specified 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 VME environment, and are supported on all hardware platforms.

GetPrivilege (window, priv)

`GetPrivilege` retrieves the current windowing hardware 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.

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 VMEbus is based on Motorola byte order.

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.

GetVXIbusStatus (controller, status)

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

Note: ***This function is for debug purposes only.***

The information that is returned includes the state of the Sysfail, ACfail, and interrupt lines. For VXI systems only (not for VME systems), it also includes the state of the VXIbus, 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 VMEbus.

Note: ***This function is for debug purposes only.***

The information that can be returned includes the state of the Sysfail, ACfail, and interrupt lines. For VXI systems only (not for VME systems), it also includes the state of the VXIbus, 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 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.

Function Descriptions

The following paragraphs describe the low-level VMEbus access functions. The descriptions are explained at the C syntax level and are listed in alphabetical order.

GetByteOrder

Syntax: `ret = GetByteOrder (window, ordermode)`

Action: Gets the byte/word order of data transferred into or out of the specified window.

Remarks: Input parameter:

`window`	`uint32`	Window number as returned from `MapVXIAddress`

Output parameter:

`ordermode`	`uint16*`	Contains the byte/word ordering 0 = Motorola byte ordering 1 = Intel byte ordering

Return value:

`ret`	`int16`	Return Status 0 = Successful 1 = Byte order returned successfully; same for all -1 = Invalid `window`

Example:

```
/* Get the byte order for the specified window. */

int16     ret;
uint32    window;
uint16    ordermode;

/* Window value is set in MapVXIAddress. */

ret = GetByteOrder (window, &ordermode);
```

GetContext

Syntax: `ret = GetContext (window, context)`

Action: Gets the current hardware interface settings (context) for the specified window.

Remarks: Input parameter:

`window`	`uint32`	Window number as returned from `MapVXIAddress`

Output parameter:

`context`	`uint32*`	Returned VME hardware access context

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Invalid `window`

Example:

```
/* Get or set the context for a window. */

int16      ret;
uint32     window;
uint32     context;

   /* Window ID set in MapVXIAddress call. */

ret = GetContext (window, &context);

   /* Change window settings as needed. */

ret = SetContext (window, context);
```

GetPrivilege

Syntax: `ret = GetPrivilege (window, priv)`

Action: Gets the current VME access privilege for the specified window.

Remarks: Input parameter:

`window`	`uint32`	Window number as returned from `MapVXIAddress`

Output parameter:

`priv`	`uint16*`	Access Privilege 0 = Nonprivileged data access 1 = Supervisory data access 2 = Nonprivileged program access 3 = Supervisory program access 4 = Nonprivileged block access 5 = Supervisory block access

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Invalid `window`

Example:

```
/* Get the privilege for a window. */

int16     ret;
uint32    window;
uint16    priv;

   /* Window value is returned from MapVXIAddress. */

ret = GetPrivilege (window, &priv);
if (ret != 0)
   /* Error occurred in GetPrivilege. */;
```

GetVXIbusStatus

Syntax: `ret = GetVXIbusStatus (controller, status)`

Action: Gets information about the state of the VMEbus in a specified controller (either an embedded CPU or an extended controller).

Remarks: Input parameter:

`controller`	`int16`	Controller to get status from (-2 = OR of all)

Output parameter:

`status` Structure containing VMEbus status

Structure is as follows:

```
struct BusStat {
   int16  BusError;  /* 1 = Last access BERRed          */
   int16  Sysfail;   /* 1 = SYSFAIL* asserted           */
   int16  ACfail;    /* 1 = ACFAIL* asserted            */
   int16  SignalIn;  /* Number of signals queued
                           (not used in VME)            */
   int16  VXIints;   /* Bit vector 1 = int asserted     */
   int16  ECLtrigs;  /* Bit vector 1 = trig asserted
                           (not used in VME)            */
   int16  TTLtrigs;  /* Bit vector 1 = trig asserted
                           (not used in VME)            */
   }
```

A value of -1 returned in any of the fields signifies that there is no hardware support to retrieve information for that particular state.

Return value:

`ret`	`int16`	Return Status

0 = Status information received successfully
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`

Example:

```
/* Get the VMEbus status from local (or first) controller. */

int16      ret;
int16      controller;
BusStat    status;

controller = -1;
ret = GetVXIbusStatus (controller, &status);
if (ret < 0)
   /* Error in GetVXIbusStatus. */;
```

GetVXIbusStatusInd

Syntax: `ret = GetVXIbusStatusInd (controller, field, status)`

Action: Gets information about the state of the VMEbus for the specified field in a particular controller.

Remarks: Input parameters:

`controller`	`int16`	Controller to get status from (-2 = OR of all)
`field`	`uint16`	Number of field to return information on

1	`BusError;`	/* 1 = Last access BERRed	*/
2	`Sysfail;`	/* 1 = SYSFAIL* asserted	*/
3	`ACfail;`	/* 1 = ACFAIL* asserted	*/
4	`SignalIn;`	/* Number of signals queued (not used in VME)	*/
5	`VXIints;`	/* Bit vector 1 = int asserted	*/
6	`ECLtrigs;`	/* Bit vector 1 = trig asserted (not used in VME)	*/
7	`TTLtrigs;`	/* Bit vector 1 = trig asserted (not used in VME)	*/

Output parameter:

`status`	`int16*`	VMEbus Status A value of -1 in any of the fields means that there is no hardware support for that particular state.

Return value:

`ret`	`int16`	Return Status

0 = Status information received successfully
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `field`

Example:

```
/* Get the VMEbus status for Sysfail on local (or first)
   controller. */

int16     ret;
int16     controller;
uint16    field;
int16     status;

controller = -1;
field = 2;
ret = GetVXIbusStatusInd (controller, field, &status);
if (ret < 0)
   /* Error in GetVXIbusStatusInd. */;
```

GetWindowRange

Syntax: `ret = GetWindowRange (window, windowbase, windowend)`

Action: Gets the range of addresses that a particular window, allocated with the `MapVXIAddress` function, can currently access within a particular VMEbus address space.

Remarks: Input parameter:

`window`	`uint32`	Window number obtained from `MapVXIAddress`

Output parameters:

`windowbase`	`uint32*`	Base VME Address
`windowend`	`uint32*`	End VME Address

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Invalid `window`

Example:

```
/*  Get the range for the window obtained from MapVXIAddress. */

uint16    accessparms;
uint32    address;
int32     timo;
uint32    window;
uint32    windowbase;
uint32    windowend;
int16     ret;
void      *addr;

accessparms = 1;
address = 0xc100L;
timo = 0L;
addr = MapVXIAddress (accessparms, address, timo, &window, &ret);
if (ret < 0)
{       /* Map failed; handle error. */;
}

ret = GetWindowRange (window, &windowbase, &windowend);
```

MapVXIAddress

Syntax: `addr = MapVXIAddress (accessparms, address, timo, window, ret)`

Action: Sets up a window into one of the VME address spaces according to the access parameters specified, and returns a pointer to a local CPU address that accesses the specified VME address. This function also returns the window ID associated with the window, which is used with all other low-level VMEbus access functions.

Remarks: Input parameters:

`accessparms`	`uint16`	(Bits 0 to 1) VME Address Space 1 = A16 2 = A24 3 = A32 (Bits 2 to 4) Access Privilege 0 = Nonprivileged data access 1 = Supervisory data access 2 = Nonprivileged program access 3 = Supervisory program access 4 = Nonprivileged block access 5 = Supervisory block access (Bit 5) 0 (Bit 6) Access Mode 0 = Access Only 1 = Owner Access (Bit 7) Byte Order 0 = Motorola 1 = Intel (Bits 8 to 15) 0
`address`	`uint32`	Address within A16, A24, or A32
`timo`	`int32`	Timeout (in milliseconds)

Output parameters:

`window`	`uint32`	Window number for use with other functions
`ret`	`int16`	Return Status 0 = Map successful -2 = Invalid/unsupported `accessparms` -3 = Invalid `address` -5 = Byte order not supported -6 = Offset not accessible with this hardware -7 = Privilege not supported -8 = Window still in use; must use `UnMapVXIAddress`

Return value:

`addr`	`void*`	Pointer to local address for specified VME address; 0 if unable to get pointer.

Note: ***To maintain compatibility and portability, the pointer obtained by calling this function should be used only with the functions `VXIpeek` and `VXIpoke`.***

Example:

```
/* Get the local address pointer for address 0xc100 in the A16
   space with nonprivileged data and Motorola byte order.  Wait up
   to 5 seconds to get "Access Only" access to the window. */

uint16    accessparms;
uint32    address;
int32     timo;
uint32    window;
int16     ret;
void      *addr;

accessparms = 1;
address = 0xc100L;
timo    = 5000L;
addr = MapVXIAddress (accessparms, address, timo, &window, &ret);
if (ret < 0)
   /* Unable to get the pointer. */;
```

RestoreContext

Syntax: `ret = RestoreContext (contextlist)`

Action: Restores hardware context for all of the VME windows. The `contextlist` parameter should contain values set within the function `SaveContext`.

Remarks: Input parameters:

none

Output parameter:

`contextlist`	`ContextStruct*`	Pointer to structure created by `SaveContext`

Return value:

`ret`	`int16`	Return Status 0 = Successful -2 = Null `contextlist` pointer

Example:

```
/* Restore the context for all the windows. */

int16            ret;
ContextStruct    contextlist;

ret = SaveContext (&contextlist);

   /*
      Interrupt service routine code.
   */

ret = RestoreContext (&contextlist);
```

SaveContext

Syntax: `ret = SaveContext (contextlist)`

Action: Saves the hardware context for all of the VME windows. The `contextlist` parameter will be filled with a list of the contexts for all of the VME windows. This function is recommended for use only within interrupt service routines to guarantee access to a particular VME window.

Remarks: Input parameters:

none

Output parameter:

`contextlist`	`ContextStruct*`	Pointer to allocated structure to hold all contexts

Return value:

`ret`	`int16`	Return Status 0 = Successful -2 = Null `contextlist` pointer

Example:

```
/* Save the context for all the windows. */

int16           ret;
ContextStruct   contextlist;

ret = SaveContext (&contextlist);

   /*
      Interrupt service routine code.
   */

ret = RestoreContext (&contextlist);
```

SetByteOrder

Syntax: `ret = SetByteOrder (window, ordermode)`

Action: Sets the byte/word order of data transferred into or out of the specified window.

Remarks: Input parameters:

`window`	`uint32`	Window number as returned from `MapVXIAddress`
`ordermode`	`uint16`	Sets the byte/word ordering 0 = Motorola byte ordering 1 = Intel byte ordering

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful; byte order set for specific window only 1 = Successful; byte order set for all windows -1 = Invalid `window` -2 = Invalid `ordermode` -5 = `ordermode` not supported -9 = `window` is not Owner Access

Example:

```
/*  Set the byte order to Motorola for a window. */

int16      ret;
uint32     window;
uint16     ordermode;

/* Window set in call to MapVXIAddress(). */
ordermode = 0;
ret = SetByteOrder (window, ordermode);
if (ret == -1)
    /* Capability not present. */;
```

SetContext

Syntax: `ret = SetContext (window, context)`

Action: Sets the current hardware interface settings (context) for the specified window. The value for `context` should have been set previously by the function `GetContext`.

Remarks: Input parameters:

`window`	`uint32`	Window number as returned from `MapVXIAddress`
`context`	`uint32`	VME hardware context to install (context returned from `GetContext`)

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Invalid `window` -2 = Invalid/unsupported `context` -9 = `window` is not Owner Access

Example:

```
/* Get or set the context for a window. */

int16     ret;
uint32    window;
uint32    context;

   /* Window ID set in MapVXIAddress call. */
ret = GetContext (window, &context);

   /* Change window settings as needed. */

ret = SetContext (window, context);
```

SetPrivilege

Syntax: `ret = SetPrivilege (window, priv)`

Action: Sets the VME access privilege for the specified window to the specified privilege state.

Remarks: Input parameters:

`window`	`uint32`	Window number as returned from `MapVXIAddress`
`priv`	`uint16`	Access Privilege 0 = Nonprivileged data access 1 = Supervisory data access 2 = Nonprivileged program access 3 = Supervisory program access 4 = Nonprivileged block access 5 = Supervisory block access

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Invalid `window` -2 = Invalid `priv` -7 = `priv` not supported -9 = `window` is not Owner Access

Example:

```
/*  Set nonprivileged data access for a window. */

int16      ret;
uint32     window;
uint16     priv;

/* Window ID set in MapVXIAddress call. */
priv = 0;
ret = SetPrivilege (window, priv);
if (ret != 0)
   /* Error occurred in SetPrivilege. */;
```

UnMapVXIAddress

Syntax: `ret = UnMapVXIAddress (window)`

Action: Deallocates a window that was allocated using the `MapVXIAddress` function.

Remarks: Input parameter:

`window`	`uint32`	Window number obtained from `MapVXIAddress`

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 1 = Access Only released (accessors remain) 0 = `window` successfully unmapped -1 = Invalid `window`

Example:

```
/*  Unmap the window obtained from MapVXIAddress. */

uint16    accessparms;
uint32    address;
int32     timo;
uint32    window;
int16     ret;
void      *addr;

accessparms = 1;
address = 0xc100L;
timo   = 0L;
addr = MapVXIAddress (accessparms, address, timo, &window, &ret);
if (addr != null)
{
   /**
      Use the pointer here.
   **/
   ret = UnMapVXIAddress (window);
   if (ret >= 0)
        /** Unmap successful. **/
}
```

VXIpeek

Syntax: `VXIpeek (addressptr, width, value)`

Action: Reads a single byte, word, or longword from a specified VME address by de-referencing a C pointer obtained from `MapVXIAddress.`

Remarks: Input parameters:

`addressptr`	`void*`	Address pointer obtained from `MapVXIAddress`
`width`	`uint16`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword

Output parameter:

`value`	`void*`	Data value read (uint8, uint16, or uint32)

Return value:

none

Example:

```
/* Read the value from address 0xc106 in VME A16 space. */

uint16    accessparms;
uint32    window;
int16     ret;
uint16    *addressptr;
uint16    value;

accessparms = 1;
addressptr =
   (uint16 *)MapVXIAddress(accessparms,(uint32)0xc106,
   (int32)0x7fffffff, &window, &ret);
if (ret >= 0)  /* If a valid pointer was returned. */
{
   VXIpeek (addressptr, 2, &value);
}
```

VXIpoke

Syntax: `VXIpoke (addressptr, width, value)`

Action: Writes a single byte, word, or longword to a specified VME address by de-referencing a C pointer obtained from `MapVXIAddress`.

Remarks: Input parameters:

`addressptr`	`void*`	Address pointer obtained from `MapVXIAddress`
`width`	`uint16`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword
`value`	`uint32`	Data value to write

Output parameters:

none

Return value:

none

Example:

```
/* Write the value 0x2000 to address 0xc106 in VME A16 space. */

uint16    accessparms;
uint32    window;
int16     ret;
uint16    *addressptr;
uint32    value;

accessparms = 1;
addressptr =
   (uint16 *)MapVXIAddress(accessparms,(uint32)0xc106,
   (int32)0x7fffffff, &window, &ret);
if (ret >= 0)  /* If a valid pointer was returned. */
{
   value = 0x2000L;
   VXIpoke (addressptr, 2, value);
}
```

Chapter 5
High-Level VMEbus Access Functions

This chapter describes the C syntax and use of the high-level VMEbus access functions. You can use both low-level and high-level VMEbus access functions to directly read or write to VMEbus addresses. Direct reads and writes to the different VMEbus address spaces are required in many situations, including the following:

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

Low-level and high-level access to the 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 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 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 VMEbus address spaces. 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 VMEbus address with a particular VME access privilege and byte ordering. How the C pointers are used is at the discretion of the application. You need to consider a number of issues when using the direct pointers:

- Byte, word, or longword accesses are made based on the de-reference of the C pointer.
- You need to determine bounds for the pointers.
- Based on the methods in which a particular hardware platform sets up access to VME address spaces, using more than one pointer can also result in conflicts.
- Your application must check error conditions such as Bus Error (BERR*) separately.

For more information, refer to *Chapter 4, Low-Level VMEbus Access Functions*.

High-level VMEbus access functions need not take into account any of the considerations that are required by the low-level VMEbus access functions. The high-level VMEbus access functions have all necessary information for accessing a particular 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 VMEbus access functions automatically trap bus errors and return an appropriate error status. Using the high-level 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 VMEbus access functions act as an easy-to-use interface that can do any VMEbus accesses necessary for an application.

Programming Considerations for High-Level VMEbus Access Functions

All accesses to the VMEbus address spaces performed by use of the high-level 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 VMEbus access functions, will conflict with the high-level VMEbus access functions. You can use both high-level and low-level VMEbus access functions at the same time.

Functional Overview

The following paragraphs describe the high-level VMEbus 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.

VXIin (accessparms, address, width, value)

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

VXIout (accessparms, address, width, value)

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

VXIinReg (la, reg, value)

`VXIinReg` reads a single word 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 VME access privilege to Nonprivileged Data and the byte order to Motorola. If the VME address selected has no device residing at the address and a bus error occurs, `VXIinReg` traps the bus error condition and returns a corresponding return status. This function is mainly for convenience and is simply 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`.

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 VME access privilege to Nonprivileged Data and the byte order to Motorola. If the VME address selected has no device residing at the address and a bus error occurs, `VXIinReg` traps the bus error condition and returns a corresponding return status. This function is mainly for convenience and is simply 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`.

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 returns a corresponding return status.

Function Descriptions

The following paragraphs describe the high-level VMEbus access functions. The descriptions are explained at the C syntax level and are listed in alphabetical order.

VXIin

Syntax: `ret = VXIin (accessparms, address, width, value)`

Action: Reads a single byte, word, or longword from a specified VME address with the specified byte order and privilege state.

Remarks: Input parameters:

`accessparms`	`uint16`	(Bits 0, 1) VME Address Space 1 = A16 2 = A24 3 = A32 (Bits 2 to 4) Access Privilege 0 = Nonprivileged data access 1 = Supervisory data access 2 = Nonprivileged program access 3 = Supervisory program access 4 = Nonprivileged block access 5 = Supervisory block access (Bits 5, 6) Reserved (should be 0) (Bit 7) Byte Order 0 = Motorola 1 = Intel (Bits 8 to 15) Reserved (should be 0)
`address`	`uint32`	VME address within specified space
`width`	`uint16`	Read Width 1 = Byte 2 = Word 4 = Longword

Output parameter:

`value`	`void*`	Value read (uint8, uint16, or uint32)

Return value:

`ret`	`int16`	Return Status

0 = Read completed successfully
-1 = Bus error occurred during transfer
-2 = Invalid parms
-3 = Invalid `address`
-4 = Invalid `width`
-5 = Byte order not supported
-6 = `address` not accessible with this hardware
-7 = Privilege not supported
-9 = `width` not supported

Example:

```
/*  Read 16-bit value from address 0xc1000 from A16 space. */

int16      ret;
uint16     accessparms;
uint32     address;
uint16     width;
uint16     value;

accessparms = 1;
address = 0xc100L;
width = 2;
ret = VXIin (accessparms, address, width, &value);
if (ret != 0)
   /* Error occurred during read. */;
```

VXIinReg

Syntax: `ret = VXIinReg (la, reg, value)`

Action: Reads a single word from a specified VXI register offset on the specified VXI device. The register is read in Motorola byte order and as nonprivileged data.

Remarks: Input parameters:

`la`	`int16`	Logical address of the device to read from
`reg`	`uint16`	Offset within VXI logical address registers

Output parameter:

`value`	`uint16*`	Value read from device's VXI register

Return value:

`ret`	`int16`	Return Status

0 = Read completed successfully
-1 = Bus error occurred during transfer
-3 = Invalid `reg` specified

Example:

```
/*  Read ID register of the device at Logical Address 4. */

int16     ret;
uint16    la;
uint16    reg;
uint16    value;

la = 4;
reg = 0;
ret = VXIinReg (la, reg, &value);
if (ret != 0)
   /* Error occurred during read. */;
```

VXImove

Syntax: `ret = VXImove (srcparms, srcaddr, destparms, destaddr, length, width)`

Action: Copies a block of memory from a specified source location in any address space (local, A16, A24, A32) to a specified destination in any address space.

Remarks: Input parameters:

`srcparms`	`uint16`	(Bits 0, 1) Source Address Space 0 = Local (bits 2, 3, 4, and 7 should be 0) 1 = A16 2 = A24 3 = A32 (Bits 2 to 4) Access Privilege 0 = Nonprivileged data access 1 = Supervisory data access 2 = Nonprivileged program access 3 = Supervisory program access 4 = Nonprivileged block access 5 = Supervisory block access (Bits 5, 6) Reserved (should be 0) (Bit 7) Byte Order 0 = Motorola 1 = Intel (Bits 8 to 15) Reserved (should be 0)
`srcaddr`	`uint32`	Address within source address space. This address is a long integer value if it represents a VME space (1, 2, 3) or an array address for a local address space (0).
`destparms`	`uint16`	(Bits 0, 1) Destination Address Space 0 = Local (bits 2, 3, 4, and 7 should be 0) 1 = A16 2 = A24 3 = A32 (Bits 2 to 4) Access Privilege 0 = Nonprivileged data access 1 = Supervisory data access 2 = Nonprivileged program access 3 = Supervisory program access 4 = Nonprivileged block access 5 = Supervisory block access (Bits 5, 6) Reserved (should be 0) (Bit 7) Byte Order 0 = Motorola 1 = Intel (Bits 8 to 15) Reserved (should be 0)
`destaddr`	`uint32`	Address within destination address space. This address is a long integer value if it represents a VME space (1, 2, 3) or an array address for a local address space (0).

`length`	`uint32`	Number of elements to transfer
`width`	`uint16`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Transfer completed successfully -1 = Bus error occurred -2 = Invalid `srcparms` or `destparms` -3 = Invalid `srcaddr` or `destaddr` -4 = Invalid `width` -5 = Byte order not supported -6 = Address not accessible with this hardware -7 = Privilege not supported -8 = Timeout, DMA aborted (if applicable) -9 = `width` not supported

Example:

```
/* Move 1 kilobyte from A24 space at 0x200000 to a local
   buffer. */

int16      ret;
uint16     srcparms;
uint32     srcaddr;
uint16     destparms;
uint32     destaddr;
uint32     length;
uint16     width;

srcparms = 2;            /* A24, nonprivileged data, Motorola */
srcaddr = 0x200000L;
destparms = 0;           /* Local space. */
length = 0x400L;         /* 1 kilobyte. */
destaddr = (uint32)malloc(length);  /* Allocate local buffer. */
width = 2;               /* Transfer as words. */
ret = VXImove (srcparms, srcaddr, destparms, destaddr, length,
width);
if (ret < 0)
   /* Error occurred during VXImove. */;
```

VXIout

Syntax: `ret = VXIout (accessparms, address, width, value)`

Action: Writes a single byte, word, or longword to a specified VME address with the specified byte order and privilege state.

Remarks: Input parameters:

`accessparms`	`uint16`	(Bits 0, 1) VME Address Space 1 = A16 2 = A24 3 = A32 (Bits 2 to 4) Access Privilege 0 = Nonprivileged data access 1 = Supervisory data access 2 = Nonprivileged program access 3 = Supervisory program access 4 = Nonprivileged block access 5 = Supervisory block access (Bits 5, 6) Reserved (should be 0) (Bit 7) Byte Order 0 = Motorola 1 = Intel (Bits 8 to 15) Reserved (should be 0)
`address`	`uint32`	VME address within specified address space
`width`	`uint16`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword
`value`	`uint32`	Data value to write

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Write completed successfully -1 = Bus error occurred during transfer -2 = Invalid `accessparms` -3 = Invalid `address` -4 = Invalid `width` -5 = Byte order not supported -6 = Address not accessible with this hardware -7 = Privilege not supported -9 = `width` not supported

Example:

```
/* Write the 16-bit value 0x2000 to address 0xc10a in A16
   space. */

int16     ret;
uint16    accessparms;
uint32    address;
uint16    width;
uint32    value;

accessparms = 1;
address = 0xc10aL;
width = 2;
value = 0x2000L;
ret = VXIout (accessparms, address, width, value);
if (ret < 0)
   /* Error occurred during write. */;
```

VXIoutReg

Syntax: `ret = VXIoutReg (la, reg, value)`

Action: Writes a single word to a specified VXI register offset on the specified VXI device. The register is written in Motorola byte ordering and as nonprivileged data.

Remarks: Input parameters:

`la`	`int16`	Logical address of the device to write to
`reg`	`uint16`	Offset within VXI logical address registers
`value`	`uint16`	Value written to device's VXI register

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Write completed successfully -1 = Bus error occurred during transfer -3 = Invalid `reg` specified

Example:

```
/* Write Signal register of the device at Logical Address 10 with
   the value 0xfd0a (REQT). */

int16      ret;
uint16     la;
uint16     reg;
uint16     value;

la = 10;
reg = 8;
value = 0xfd0a;
ret = VXIoutReg (la, reg, value);
if (ret != 0)
   /* Error occurred during write. */;
```

Chapter 6
Local Resource Access Functions

This chapter describes the C syntax and use of the VME 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.

Reading local registers is required for retrieving configuration information. Writing to the A24 and A32 pointer registers is required for using shared memory communication.

Functional Overview

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

VXIinLR (reg, width, value)

`VXIinLR` reads a single byte, word, or longword from the local CPU VXI registers. On many CPUs, the local CPU VXI registers cannot be accessed from the local CPU in the VXI 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 VXI registers. The VXI 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.

VXIoutLR (reg, width, value)

`VXIoutLR` writes a single byte, word, or longword to the local CPU VXI registers. On many CPUs, the local CPU VXI registers cannot be accessed from the local CPU in the VXI 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 registers. The VXI 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. You can use this function to write application-specific registers (A24 pointer register, A32 pointer register, 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 Shared RAM region of the local CPU. `VXImemAlloc` returns not only the user address that the application uses, but also the 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 simply a `malloc` (standard C dynamic allocation routine) and an address translation. When the application is finished using the memory, it must call `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` deallocates physical RAM from the operating system's dynamic memory pool that had been allocated using `VXImemAlloc`. `VXImemAlloc` returns not only the user address that the application uses, but also the VME address that a remote device would use to access this RAM. When the application is through using the memory, it must call `VXImemFree` (with the user address) to return the memory to the operating system's dynamic memory pool.

Function Descriptions

The following paragraphs describe the local resource access functions. The descriptions are explained at the C syntax level and are listed in alphabetical order.

GetMyLA

Syntax: `la = GetMyLA ()`

Action: Gets the logical address of the local VXI device (the VXI device on which this copy of the NI-VXI software is running).

Remarks: Parameters:

none

Return value:

`la`	`int16`	Logical address of the local device

Example:

```
/*  Get my logical address. */

int16      la;

la = GetMyLA();
```

VXIinLR

Syntax: `ret = VXIinLR (reg, width, value)`

Action: Reads a single byte, word, or longword from a particular VXI register on the local VME device. The register is read in Motorola byte order and as nonprivileged data.

Remarks: Input parameters:

`reg`	`uint16`	Offset within VXI logical address registers
`width`	`uint16`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword

Output parameter:

`value`	`void*`	Data value read (uint8, uint16, or uint32)

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Bus error -3 = Invalid `reg` -4 = Invalid `width` -9 = `width` not supported

Example:

```
/*  Read the value of the local Offset register. */

int16      ret;
uint16     reg;
uint16     width;
uint16     value;

reg = 6;           /* Offset register offset within registers. */
width = 2;         /* Read word register. */
ret = VXIinLR (reg, width, &value);
if (ret != 0)
   /* Error in VXIinLR. */;
```

VXImemAlloc

Syntax: `ret = VXImemAlloc (size, useraddr, vxiaddr)`

Action: Allocates dynamic system RAM from the Shared RAM area of the local CPU and returns both the local and remote VME addresses. The VME address space is the same as the space for which the local device is dual-porting memory. You can use this function for setting up shared memory transfers.

Remarks: Input parameter:

`size`	`uint32`	Number of bytes to allocate

Output parameters:

`useraddr`	`void*`	Returned application memory buffer address
`vxiaddr`	`uint32`	Returned remote VME memory buffer address

Return value:

`ret`	`int16`	Return Status

0 = Successful; memory can be accessed directly
1 = Successful; memory must be accessed using `VXImemCopy`
-1 = Memory allocation failed
-2 = Local CPU is A16 only

Example:

```
/*  Allocate, use, and free 32 kilobytes of Shared system RAM. */

uint32    size;
void      *useraddr;
uint32    vxiaddr;
int16     ret;

size = 0x8000;       /* 32 kilobytes  */
ret = VXImemAlloc (size, &useraddr, &vxiaddr);
if (ret != 0)
   /* Error in VXImemAlloc. */;

/*
   Use buffer.
*/

ret = VXImemFree (useraddr);
if (ret != 0)
   /* Error in VXImemFree. */;
```

VXImemCopy

Syntax: `ret = VXImemCopy (useraddr, bufaddr, size, dir)`

Action: Copies an application buffer to or from the local shared memory. On some systems, local shared memory cannot be accessed directly by an application. `VXImemCopy` provides a fast access method to local shared memory.

Remarks: Input parameter:

`useraddr`	`void*`	VXI shared memory buffer address
`bufaddr`	`void*`	Address of application buffer to copy into or out of
`size`	`uint32`	Number of bytes to copy
`dir`	`uint16`	Copy direction 1 = Copy from `bufaddr` to `useraddr` 0 = Copy from `useraddr` to `bufaddr`

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Copy failed -5 = Invalid `dir`

Example:

```
/*  Allocate, copy, use, and free 32 kilobytes of VXI Shared
    system RAM. */

uint32    size;
void      *useraddr;
uint32    *vxiaddr;
int16     ret;
void      *bufaddr;

size = 0x8000;       /* 32 kilobytes. */
ret = VXImemAlloc (size, &useraddr, &vxiaddr);
if (ret < 0)
   /* Error in VXImemAlloc. */;

/*
   Tell remote bus master to copy 32 kilobytes to local
   shared memory by writing to VXI address "vxiaddr."
*/

   /* Copy to application. */;
bufaddr = malloc(size);
VXImemCopy (useraddr, bufaddr, size, 0);

/*
   Use buffer.
*/

ret = VXImemFree (useraddr);
if (ret != 0)
   /* Error in VXImemFree. */;
```

VXImemFree

Syntax: `ret = VXImemFree (useraddr)`

Action: Deallocates dynamic system RAM from the Shared RAM area of the local CPU that was allocated using the `VXImemAlloc` function.

Remarks: Input parameter:

`useraddr`	`void*`	Application memory buffer address to free

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Memory deallocation failed

Example:

```
/*  Allocate, use, and free 32 kilobytes of VME Shared system
    RAM. */

uint32    size;
void      *useraddr;
uint32    *vxiaddr;
int16     ret;

size = 0x8000;       /* 32 kilobytes. */
ret = VXImemAlloc (size, &useraddr, &vxiaddr);
if (ret != 0)
   /* Error in VXImemAlloc. */;

/*
   Use buffer.
*/

ret = VXImemFree (useraddr);
if (ret != 0)
   /* Error in VXImemFree. */;
```

VXIoutLR

Syntax: `ret = VXIoutLR (reg, width, value)`

Action: Writes a single byte, word, or longword to a particular VXI register on the local VME device. The register is written in Motorola byte order and as nonprivileged data.

Remarks: Input parameters:

`reg`	`uint16`	Offset within VXI logical address registers
`width`	`uint16`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword
`value`	`void`	Data value to write

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Bus error -3 = Invalid `reg` -4 = Invalid `width` -9 = `width` not supported

Example:

```
/*  Write the value of 0xfd00 (REQT) to the local Signal
    register. */

int16     ret;
uint16    reg;
uint16    width;
uint16    value;

reg = 8;             /* Register offset for Signal register. */
width = 2;           /* Word register. */
value = 0xfd00L;
ret = VXIoutLR (reg, width, (void)value);
if (ret != 0)
   /* Error in VXIoutLR. */;
```

Chapter 7
VME Interrupt Functions

This chapter describes the C syntax and use of the VME interrupt functions and default handler. VME interrupts are a basic form of asynchronous communication used by VME devices with VME 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 680X0-based VME CPUs use this 8-bit value as a local interrupt vector value routed directly to the 680X0 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 vector is specified by the VXIbus specification. 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. Because the NI-VXI functions were designed for VXI, which is a superset of VME, the interrupt functions are configured by default to use 16-bit VXI interrupt status/ID values. You will need to use the VME mode when you use the NI-VXI interrupt functions.

The main use for the VME interrupt handler functions is to handle VME interrupters. The VME interrupt handler function for a particular level is called with the VME interrupt level and the status/ID without any interpretation of the status/ID value. The VME interrupt handler can do whatever is necessary with the status/ID value. The `SetVXIintHandler` function can be called to change the current VXI interrupt handler for a particular level. A default handler, `DefaultVXIintHandler`, is given in source code as an example, and 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 VME interrupts routed to the VME interrupt handlers.

When you are testing VME interrupt handlers, you must assert a 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. Use the `DeAssertVXIint` function to deassert a VME interrupt that was asserted using the `AssertVXIint` function. `AcknowledgeVXIint` acknowledges VME interrupts that the local CPU is not enabled to automatically handle via `EnableVXIint`. Both `DeAssertVXIint` and `AcknowledgeVXIint` are intended for debug use only.

ROAK Versus RORA VME Interrupters

In VME, there are two types of interrupters. The Release On Acknowledge (ROAK) interrupter is the more common. A ROAK interrupter automatically deasserts the VME interrupt line it is asserting when an interrupt acknowledge cycle on the VME backplane occurs on the corresponding level. The Release On Register Access (RORA) interrupt is the second type of interrupter. The RORA interrupter continues to assert the VME interrupt line after the interrupt acknowledge cycle is complete. The RORA interrupter will deassert the VME interrupt only when some device-specific interaction is performed. There is no standard method to cause a RORA interrupter to deassert its interrupt line. Because a RORA interrupt remains asserted on the VME backplane, the local CPU interrupt generation must be inhibited until the device-dependent acknowledgement is complete. The function `VXIintAcknowledgeMode` specifies that a VME interrupt level for a particular controller (embedded or extended) be handled as a RORA or ROAK interrupt. If the VME interrupt is specified to be handled as a RORA interrupt, the local CPU automatically inhibits VME interrupt generation for the corresponding controller and levels whenever the corresponding VME interrupt occurs. After the application has handled and caused the RORA interrupter to deassert the interrupt line, the application must call either `EnableVXIint` or `EnableVXItoSignalInt` to re-enable local CPU interrupt generation.

Functional Overview

The following paragraphs describe the 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.

RouteVXIint (controller, Sroute)

`RouteVXIint` specifies whether to route status/ID values returned from an interrupt acknowledge cycle to a VME interrupt service routine or to the VXI signal processing routine. Because VXI devices have a 16-bit interrupt status/ID, the interrupt status/ID format for VXI interrupts can be processed as VXI signals that are written directly to a register on the controller, as opposed to the traditional method of asserting a backplane interrupt line and performing an interrupt acknowledge cycle on the backplane to obtain the status/ID value. This capability is not available in VME systems. For this reason, in VME systems, you need to call the `RouteVXIint` function to configure your computer to handle interrupts as VME interrupts.

EnableVXItoSignalInt (controller, levels)

`EnableVXItoSignalInt` is used to sensitize the application to specified VME 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 to handle VME interrupts as VXI/VME interrupts or as VXI signals (the default is VXI signals). A `EnableVXItoSignalInt` call enables VME interrupt levels that are routed to VXI signals. Use `DisableVXItoSignalInt` to disable these VME interrupts. Use `EnableVXIint` to enable VME interrupts not routed to VXI signals. A -1 (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 VME interrupt level to the VXI signal processing routine and the global signal queue becomes full, `DisableVXItoSignalInt` is automatically called to inhibit these VME interrupts from being received from the appropriate levels. When `SignalDeq` is called, `EnableVXItoSignalInt` is automatically called to enable VXI interrupt reception.

DisableVXItoSignalInt (controller, levels)

`DisableVXItoSignalInt` desensitizes the application to specified VME interrupt levels being processed as VXI signals. A `EnableVXItoSignalInt` call enables VME interrupt levels that are routed to VXI signals. Use `DisableVXItoSignalInt` to disable these VME interrupts. Use `EnableVXIint` to enable VME interrupts not routed to VXI signals. A -1 (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 `RouteVXIint` has been called to specify that a particular VME interrupt level be routed to the VXI signal processing routine and the global signal queue becomes full, `DisableVXItoSignalInt` is automatically called to inhibit these VME interrupts from being received from the appropriate levels. `EnableVXItoSignalInt` is automatically called to enable VME interrupt reception when `SignalDeq` is called.

EnableVXIint (controller, levels)

`EnableVXIint` sensitizes the application to specified VME interrupt levels being processed as 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 to handle interrupts as VME interrupts or as VXI signals (the default is VXI signals). You must first call the `RouteVXIint` function to instruct your system to handle interrupts as VME interrupts (not as VXI signals). Then call `EnableVXIint` to enable VME interrupts to be handled as VME interrupts (not as VXI signals). A -1 (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).

DisableVXIint (controller, levels)

`DisableVXIint` desensitizes the application to specified VME interrupt levels being processed as VME interrupts (not as VXI signals). `EnableVXIint` enables VXI interrupts handled as VME interrupts (not as VXI signals). A -1 (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).

VXIintAcknowledgeMode (controller, modes)

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

SetVXIintHandler (levels, func)

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

GetVXIintHandler (level)

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

DefaultVXIintHandler (controller, level, statusId)

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

AssertVXIint (controller, level, statusId)

`AssertVXIint` asserts a particular 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 VME interrupt handler configured for the specified VME interrupt level. `AssertVXIinterrupt` returns immediately (that is, it does not wait for the VME interrupt to be acknowledged). You can call the function `GetVXIbusStatus` to detect if the VME interrupt has been serviced. Use `DeAssertVXIint` to deassert a VME interrupt that had been asserted using `AssertVXIint` but not yet acknowledged.

DeAssertVXIint (controller, level)

`DeAssertVXIint` deasserts the 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 VME interrupt handler configured for the specified VME interrupt level. You can call the function `GetVXIbusStatus` to detect if the VME interrupt has been serviced. Use `DeAssertVXIint` to deassert a VME interrupt that had been asserted using `AssertVXIint` but not yet acknowledged.

Note: ***Deasserting a VME interrupt may violate the VME (and VXIbus) specifications if the interrupt has not yet been acknowledged by the interrupt handler.***

AcknowledgeVXIint (controller, level, statusId)

`AcknowledgeVXIint` performs an VME interrupt acknowledge (IACK cycle) on the VMEbus backplane in the specified controller and VME interrupt level.

Note: ***This function is for debug purposes only.***

Normally, VME interrupts are automatically acknowledged when enabled via the function `EnableVXIint`. However, if the VME 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.

Function Descriptions

The following paragraphs describe the VME interrupt functions and default handler. The descriptions are explained at the C syntax level and are listed in alphabetical order.

AcknowledgeVXIint

Syntax: `ret = AcknowledgeVXIint (controller, level, statusId)`

Action: Performs an IACK cycle on the VMEbus on the specified controller (either an embedded CPU or an extended controller) for a particular VME interrupt level. VME interrupts are automatically acknowledged when enabled by `EnableVXIint`. Use this function to manually acknowledge VME interrupts that the local device is not enabled to receive.

Note: ***This function is for debug purposes only.***

Remarks: Input parameters:

`controller`	`int16`	Controller on which to acknowledge interrupt
`level`	`uint16`	Interrupt level to acknowledge

Output parameter:

`statusId`	`uint32`	Status/ID obtained during IACK cycle

Return value:

`ret`	`int16`	Return Status

0 = IACK cycle completed successfully
-1 = Unsupportable function (no hardware support for IACK)
-2 = Invalid `controller`
-3 = Invalid `level`
-4 = Bus error occurred during IACK cycle

Example:

```
/* Acknowledge Interrupt 4 on the local CPU (or first extended
   controller). */

int16     controller;
uint16    level;
uint32    statusId;
int16     ret;

controller = -1;
level = 4;
ret = AcknowledgeVXIint (controller, level, &statusId);
```

AssertVXIint

Syntax: `ret = AssertVXIint (controller, level, statusId)`

Action: Asserts a VME interrupt line on the specified controller (either an embedded CPU or an extended controller). When the VME interrupt is acknowledged (a VME IACK cycle occurs), the specified status/ID is passed to the device that acknowledges the VME interrupt.

Remarks: Input parameters:

`controller`	`int16`	Controller on which to assert interrupt
`level`	`uint16`	Interrupt level to assert
`statusId`	`uint32`	Status/ID to present during IACK cycle

Output parameters:

none

Return value:

`ret`	`int16`	Return Status

0 = Interrupt line asserted successfully
-1 = Unsupportable function (no hardware support for VME interrupter)
-2 = Invalid `controller`
-3 = Invalid `level`
-5 = VME interrupt still pending from previous `AssertVXIint`

Example:

```
/*  Assert Interrupt 4 on the local CPU (or first extended
    controller) with status/ID of 0x1111. */

int16      ret;
int16      controller;
uint16     level;
uint32     statusId;

controller = -1;
level = 4;
statusId = 0x1111L;
ret = AssertVXIint (controller, level, statusId);
```


DeAssertVXIint

Syntax: `ret = DeAssertVXIint (controller, level)`

Action: Asynchronously deasserts a VME interrupt line on the specified controller (either an embedded CPU or an extended controller) previously asserted by the function `AssertVXIint`.

Note: ***This function is for debug purposes only. Deasserting a VME interrupt can cause a violation of the VME and VXIbus specifications.***

Remarks: Input parameters:

`controller`	`int16`	Controller on which to deassert interrupt
`level`	`uint16`	Interrupt level to deassert

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Interrupt line deasserted successfully -1 = Unsupportable function (no hardware support) -2 = Invalid `controller` -3 = Invalid `level`

Example:

```
/* Deassert Interrupt 4 on the local CPU (or first extended
   controller). */

int16     controller;
uint16    level;
int16     ret;

controller = -1;
level = 4;
ret = DeAssertVXIint (controller, level);
```

DisableVXIint

Syntax: `ret = DisableVXIint (controller, levels)`

Action: Desensitizes the local CPU to specified VME interrupts generated in the specified controller, which the `RouteVXIint` function routed to be handled as VME interrupts (not as VXI signals). The RM assigns the interrupt levels automatically. Use the `GetDevInfo` functions to retrieve the assigned levels.

Remarks: Input parameters:

`controller`	`int16`	Controller (embedded or extended) to disable interrupts
`levels`	`uint16`	Vector of VME interrupt levels to disable. Bits 6 to 0 correspond to VME interrupt levels 7 to 1, respectively. 1 = Disable for appropriate level 0 = Leave at current setting

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = VME interrupt disabled -1 = No hardware support -2 = Invalid `controller`

Example:

```
/* Disable VME Interrupt 4 on the local CPU (or first extended
   controller). */

int16      controller;
uint16     levels;
int16      ret;

controller = -1;             /** Local CPU or first frame. **/
levels = (uint16)(1<<3);     /** Interrupt level 4. **/
ret = DisableVXIint (controller, levels);
```

DisableVXItoSignalInt

Syntax: `ret = DisableVXItoSignalInt (controller, levels)`

Action: Desensitizes the local CPU to specified VME interrupts generated in the specified controller, which the `RouteVXIint` function routed to be handled as VXI signals.

Remarks: Input parameters:

`controller`	`int16`	Controller (embedded or extended) to disable interrupts
`levels`	`uint16`	Vector of VME interrupt levels to disable. Bits 6 to 0 correspond to VME interrupt levels 7 to 1, respectively. 1 = Disable for appropriate level 0 = Leave at current setting

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = VME interrupt disabled -1 = No hardware support -2 = Invalid `controller` specified

Example:

```
/* Disable VME Interrupt 6 on the local CPU (or first extended
   controller). */

int16     controller;
uint16    levels;
int16     ret;

controller = -1;             /** Local CPU or first frame. **/
levels = (uint16)(1<<5);     /** Interrupt level 6. **/
ret = DisableVXItoSignalInt (controller, levels);
```

EnableVXIint

Syntax: `ret = EnableVXIint (controller, levels)`

Action: Sensitizes the local CPU to specified VME interrupts generated in the specified controller, which the `RouteVXIint` function routed to be handled as VME interrupts (not as VXI signals). The RM assigns the interrupt levels automatically. Use the `GetDevInfo` functions to retrieve the assigned levels. Notice that each VME interrupt is physically enabled only if the `RouteVXIint` function has specified that the VME interrupt be routed to be handled as a VME interrupt (not as a VXI signal).

Remarks: Input parameters:

`controller`	`int16`	Controller (embedded or extended) to enable interrupts
`levels`	`uint16`	Vector of VME interrupt levels to enable. Bits 6 to 0 correspond to VME interrupt levels 7 to 1, respectively. 1 = Enable for appropriate level 0 = Leave at current setting

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = VME interrupt enabled -1 = No hardware support -2 = Invalid `controller` specified

Example:

```
/* Enable VME Interrupt 4 on the local CPU (or first extended
   controller). */

int16     controller;
uint16    levels;
int16     ret;

controller = -1;            /** Local CPU or first frame. **/
levels = (uint16)(1<<3);    /** Interrupt level 4. **/
ret = EnableVXIint (controller, levels);
```

EnableVXItoSignalInt

Syntax: `ret = EnableVXItoSignalInt (controller, levels)`

Action: Sensitizes the local CPU to specified VME interrupts generated in the specified controller, which the `RouteVXIint` function routed to be handled as VXI signals. The RM assigns the interrupt levels automatically. Use the `GetDevInfo` functions to retrieve the assigned levels. Notice that each VME interrupt is physically enabled only if the `RouteVXIint` function has specified that the VME interrupt be routed to be handled as a VXI signal.

Remarks: Input parameters:

`controller`	`int16`	Controller (embedded or extended) to enable interrupts
`levels`	`uint16`	Vector of VME interrupt levels to enable. Bits 6 to 0 correspond to VME interrupt levels 7 to 1, respectively. 1 = Enable for appropriate level 0 = Leave at current setting

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 1 = Signal queue full, will enable after a `SignalDeq()` 0 = VME interrupt enabled -1 = No hardware support -2 = Invalid `controller` specified

Example:

```
/* Enable VME Interrupt 6 on the local CPU (or first extended
   controller). */

int16     controller;
uint16    levels;
int16     ret;

controller = -1;            /** Local CPU or first frame. **/
levels = (uint16)(1<<5);    /** Interrupt level 6. **/
ret = EnableVXItoSignalInt (controller, levels);
```

GetVXIintHandler

Syntax: `func = GetVXIintHandler (level)`

Action: Returns the address of the current interrupt handler for a specified VMEbus interrupt level.

Remarks: Input parameter:

`level`	`uint16`	VME interrupt level associated with the handler

Output parameters:

none

Return value:

`func`	`void`	Pointer to the current interrupt handler for a specified VMEbus interrupt level (Null = invalid `level` or no hardware support)

Example:

```
/*  Get the address of the interrupt handler for VME interrupt
    level 4. */

void       (*func)();
uint16     level;

level = 4;
func = GetVXIintHandler (level);
```

RouteVXIint

Syntax: `ret = RouteVXIint (controller, Sroute)`

Action: Specifies whether to route the status/ID value retrieved from a VME interrupt acknowledge cycle to a VME interrupt handler or to a VXI-specific signal processing routine. `RouteVXIint` dynamically enables and disables the appropriate VME interrupts based on the current settings from calls to `EnableVXIint`. For most VME systems, you wil first call `RouteVXIint` to instruct your system to handle interrupts as VME interrupts.

Remarks: Input parameters:

`controller`	`int16`	Controller (embedded or extended) to specify route for
`Sroute`	`uint16`	A bit vector that specifies whether to handle interrupts as VXI signals or route them to a VME interrupt handler routine. Bits 6 to 0 correspond to VME interrupt levels 7 to 1, respectively. 1 = Handle interrupt for this level as a VXI signal 0 = Handle interrupt as a VME interrupt

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = No hardware support -2 = Invalid `controller`

Example:

```
/*  Route VME interrupts for level 4 (on the local controller) to the VME

int16     controller;
uint16    Sroute;
int16     ret;

controller = -1;
Sroute = ~(1<<3);
ret = RouteVXIint (controller, Sroute);
```

SetVXIintHandler

Syntax: `ret = SetVXIintHandler (levels, func)`

Action: Replaces the current interrupt handler for the specified VMEbus interrupt levels with a specified handler.

Remarks: Input parameters:

`levels`	`uint16`	Bit vector of VME interrupt levels. Bits 6 to 0 correspond to VME interrupt levels 7 to 1, respectively. 1 = Set 0 = Do not set handler
`func`	`void`	Pointer to the new VME interrupt handler (Null = `DefaultVXIintHandler`)

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = No hardware support

Example:

```
/*  Set the VME interrupt handler for VME interrupt level 4. */

void      func (int16, uint16, uint32);
uint16    levels;
int16     ret;

levels = (uint16)(1<<3);
ret = SetVXIintHandler (levels, func);



   /* This is a sample VME interrupt handler. */
void func (controller, level, statusId)
int16 controller;   /* Controller VME interrupt received from. */
uint16 level;       /* VME interrupt level. */
uint32 statusId;    /* 32-bit VME interrupt acknowledge (IACK)
                       status/ID. Lower 8 bits are the VME
                       interrupt status/ID value received from
                       the device. */
{
}
```

VXIintAcknowledgeMode

Syntax: `ret = VXIintAcknowledgeMode (controller, modes)`

Action: Specifies whether to handle the VME interrupt acknowledge cycle for the specified controller (embedded or extended) for the specified levels as Release On AcKnowledge (ROAK) interrupts or as Release On Register Access (RORA) interrupts. If the VME interrupt level is handled as a RORA VME interrupt, further local interrupt generation is automatically inhibited while the VME interrupt acknowledge is performed. `EnableVXIint` must be called to re-enable the appropriate VME interrupt level whenever a RORA VME interrupt occurs.

Remarks: Input parameters:

`controller`	`int16`	Controller (embedded or extended) to specify route for
`modes`	`uint16`	Vector of VME interrupt levels to set to RORA/ROAK interrupt acknowledge mode. Bits 6 to 0 correspond to VME interrupt levels 7 to 1, respectively. 0 = Set to ROAK VME interrupt for corresponding level 1 = Set to RORA VME interrupt for corresponding level

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = VME interrupt enabled -1 = No hardware support -2 = Invalid `controller` specified

Example:

```
/* Set VME Interrupt levels 2 and 3 on the local CPU (or first
   extended controller) to be RORA interrupters--set reset to
   ROAK. */

int16     controller;
uint16    modes;
int16     ret;

controller = -1;            /** Local CPU or first frame. **/
   /** Levels 2 and 3 are RORA mode. **/
modes = (uint16)((1<<1) | (1<<2));
ret = RORAint (controller, modes);
```

Default Handler for VME Interrupt Functions

The NI-VXI software provides the following default handler for the VME interrupts. This is a sample handler that `InitVXIlibrary` installs when it initializes the software at the beginning of the application program. Default handlers give you the minimal and most common functionality required for a VME system. They are given in source code form on your NI-VXI distribution media to be used as examples/prototypes for extending their functionality to a particular application.

DefaultVXIintHandler

Syntax: `DefaultVXIintHandler (controller, level, statusId)`

Action: Handles the VME interrupts. The global variable `VXIintController` is set to `controller`. `VXIintLevel` is set to `level`. `VXIintStatusId` is set to `statusId`.

Remarks: Input parameters:

`controller`	`int16`	Controller (embedded or extended) that interrupted
`level`	`uint16`	The received VME interrupt level
`statusId`	`uint32`	Status/ID obtained during IACK cycle (for VME systems, the lower 8 bits are the 8-bit status/ID value received from the interrupting device during the IACK cycle)

Output parameters:

none

Return value:

none

Chapter 8
System Interrupt Handler Functions

This chapter describes the C syntax and use of the VME system interrupt handler functions and default handlers. You can use these functions to handle miscellaneous system conditions that can occur in the VME environment, such as Sysfail, ACfail, Sysreset, and/or Bus Errors. The NI-VXI software interface can handle all of these system conditions for the application through the use of interrupt service 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 interrupt handler. All system interrupt handlers are initially disabled (except for Bus Error). It is necessary to call the corresponding enable function for each handler to invoke the default or user-installed handler.

Functional Overview

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.

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). A Sysfail condition detected on the local CPU generates an interrupt that calls the current Sysfail interrupt handler. The failed device must be forced offline or brought back online in an orderly fashion.

DisableSysfail (controller)

`DisableSysfail` desensitizes the application to Sysfail interrupts from embedded controller or extended controller(s) Sysfail conditions (dependent on the hardware platform).

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.

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.

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. A Sysfail condition detected on the local CPU generates an interrupt that calls the current Sysfail interrupt handler. The failed device must be forced offline or brought back online in an orderly fashion.

EnableACfail (controller)

`EnableACfail` sensitizes the application to ACfail interrupts from embedded controller or extended controller(s) ACfail conditions (dependent on the hardware platform). The detection of a power failure in a 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.

DisableACfail (controller)

`DisableACfail` desensitizes the application to ACfail interrupts from embedded controller or extended controller(s) ACfail conditions (dependent on the hardware platform). The detection of a power failure in a 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.

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.

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.

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 simply increments the global variable `ACfailRecv`. The detection of a power failure in a 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.

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).

DisableSysreset (controller)

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

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.

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.

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.

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 simply increments the global variable `SysresetRecv`.

SetBusErrorHandler (func)

`SetBusErrorHandler` replaces the current Bus Error interrupt handler with an alternate handler. During an access to the 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 the Bus Error handler can trap. 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.

GetBusErrorHandler ()

`GetBusErrorHandler` returns the address of the current Bus Error interrupt handler. During an access to the 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 the Bus Error handler can trap. In cases where it is possible for a particular access to generate Bus Errors at times and valid results at other times, a retry mechanism should be written. A default handler, `DefaultBusErrorHandler`, is automatically installed when the `InitVXIlibrary` function is called. It simply increments the global variable `BusErrorRecv`. Because Bus Errors can occur at any time, a corresponding enable and disable function is not possible.

DefaultBusErrorHandler ()

`DefaultBusErrorHandler` is the sample handler for the Bus Error exception, and is installed as a default handler when the function `InitVXIlibrary` is called. During an access to the 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.

Function Descriptions

The following paragraphs describe the system interrupt handler functions and default handlers. The descriptions are explained at the C syntax level and are listed in alphabetical order.

AssertSysreset

Syntax: `ret = AssertSysreset (controller, mode)`

Action: Asserts the SYSRESET* signal in the mainframe specified by `controller`.

Remarks: Input parameter:

`controller`	`int16`	Logical address of mainframe extender on which to assert SYSRESET* -1 = From the local CPU or first extended controller -2 = All extenders
`mode`	`uint16`	Mode of execution 0 = Do not disturb original configuration 1 = Force link between SYSRESET* and local reset (SYSRESET* resets local CPU) 2 = Break link between SYSRESET* and local reset (SYSRESET* does *not* reset local CPU)

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = SYSRESET* signal successfully asserted -1 = `AssertSysreset` not supported -2 = Invalid `controller`

Example:

```
/* Assert SYSRESET* on the first extended controller (or local
   CPU) without changing the current configuration. */

int16      controller;
uint16     mode;
int16      ret;

controller = -1;
mode = 0;
ret = AssertSysreset (controller, mode);
```

DisableACfail

Syntax: `ret = DisableACfail (controller)`

Action: Desensitizes the local CPU from interrupts generated from ACfail conditions on the embedded CPU VMEbus backplane or from the specified extended controller VME backplane (if external CPU).

Remarks: Input parameter:

`controller`	`int16`	Logical address of mainframe extender to disable

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = ACfail interrupt successfully disabled -1 = ACfail interrupts not supported -2 = Invalid `controller`

Example:

```
/* Disable the ACfail interrupt on the first frame (or local
   CPU). */

int16     controller;
int16     ret;

controller = -1;
ret = DisableACfail (controller);
```

DisableSysfail

Syntax: `ret = DisableSysfail(controller)`

Action: Desensitizes the local CPU from interrupts generated from Sysfail conditions on the embedded CPU VMEbus backplane or from the specified extended controller VME backplane (if external CPU).

Remarks: Input parameter:

`controller`	`int16`	Logical address of mainframe extender to disable

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Sysfail interrupt successfully disabled -1 = Sysfail interrupts not supported -2 = Invalid `controller`

Example:

```
/* Disable the Sysfail interrupt. */

int16     controller;
int16     ret;

controller = -1;
ret = DisableSysfail();
```

DisableSysreset

Syntax: `ret = DisableSysreset(controller)`

Action: Desensitizes the application from Sysreset interrupts from the embedded CPU VMEbus backplane or from the specified extended controller VME backplane (if external CPU).

Remarks: Input parameter:

`controller`	`int16`	Logical address of mainframe extender to disable

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Sysreset interrupt successfully disabled -1 = Sysreset interrupts not supported -2 = Invalid `controller`

Example:

```
/* Disable the Sysreset interrupt. */

int16     controller;
int16     ret;

controller = -1;
ret = DisableSysreset(controller);
```

EnableACfail

Syntax: `ret = EnableACfail (controller)`

Action: Sensitizes the local CPU to interrupts generated from ACfail conditions on the embedded CPU VMEbus backplane or from the specified extended controller VME backplane (if external CPU).

Remarks: Input parameter:

`controller`	`int16`	Logical address of mainframe extender to enable

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = ACfail interrupt successfully enabled -1 = ACfail interrupts not supported -2 = Invalid `controller`

Example:

```
/* Enable the ACfail interrupt on the first frame (or local
   CPU). */

int16      controller;
int16      ret;

controller = -1;
ret = EnableACfail (controller);
```

EnableSysfail

Syntax: `ret = EnableSysfail (controller)`

Action: Sensitizes the local CPU to interrupts generated from Sysfail conditions on the embedded CPU VMEbus backplane or from the specified extended controller VME backplane (if external CPU).

Remarks: Input parameter:

`controller`	`int16`	Logical address of mainframe extender to enable

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Sysfail interrupt successfully enabled -1 = Sysfail interrupts not supported -2 = Invalid `controller`

Example:

```
/*  Enable the Sysfail interrupt in the local CPU (or first
    frame). */

int16     controller;
int16     ret;

controller = -1;
ret = EnableSysfail (controller);
```

EnableSysreset

Syntax: `ret = EnableSysreset (controller)`

Action: Sensitizes the application to Sysreset interrupts from the embedded CPU's VMEbus backplane or from the specified extended controller's VME backplane (if external CPU).

Remarks: Input parameter:

`controller`	`int16`	Logical address of mainframe extender to enable

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Sysreset interrupt successfully enabled -1 = Sysreset interrupts not supported -2 = Invalid `controller`

Example:

```
/*  Enable the Sysreset interrupt in the local CPU (or first
    frame). */

int16     controller;
int16     ret;

controller = -1;
ret = EnableSysreset (controller);
```

GetACfailHandler

Syntax: `func = GetACfailHandler ()`

Action: Returns the address of the current ACfail interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`void (*)()`	Pointer to the current ACfail interrupt handler Null = ACfail interrupt not supported

Example:

```
/* Get the address of the ACfail handler. */

void      (*func)();

func = GetACfailHandler();
```

GetBusErrorHandler

Syntax: `func = GetBusErrorHandler()`

Action: Returns the address of the current user Bus Error interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`void (*)()`	Pointer to the current Bus Error interrupt handler

Example:

```
/* Get the address of the Bus Error handler. */

void      (*func)();

func = GetBusErrorHandler();
```

GetSysfailHandler

Syntax: `func = GetSysfailHandler ()`

Action: Returns the address of the current Sysfail interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`void (*)()`	Pointer to the current Sysfail interrupt handler Null = Sysfail interrupt not supported

Example:

```
/* Get the address of the Sysfail handler. */

void      (*func)();

func = GetSysfailHandler ();
```

GetSysresetHandler

Syntax: `func = GetSysresetHandler ()`

Action: Returns the address of the current SYSRESET* interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`void (*)()`	Pointer to the current SYSRESET* interrupt handler Null = SYSRESET* interrupt not supported

Example:

```
/* Get the address of the SYSRESET* handler. */

void      (*func)();

func = GetSysresetHandler();
```

SetACfailHandler

Syntax: `ret = SetACfailHandler (func)`

Action: Replaces the current ACfail interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`void (*)()`	Pointer to the new ACfail interrupt handler Null = `DefaultACfailHandler`

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = ACfail interrupt not supported

Example:

```
/* Set the ACfail handler. */

void      func (int16);
int16     ret;

ret = SetACfailHandler (func);
```


SetBusErrorHandler

Syntax: `ret = SetBusErrorHandler(func)`

Action: Replaces the current Bus Error handler with a specified handler.

Remarks: Input parameter:

`func`	`void (*)()`	Pointer to the new Bus Error interrupt handler Null = `DefaultBusErrorHandler`

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful

Example:

```
/* Set the Bus Error handler. */

void      func();
int16     ret;

ret = SetBusErrorHandler(func);
```

SetSysfailHandler

Syntax: `ret = SetSysfailHandler (func)`

Action: Replaces the current Sysfail interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`void (*)()`	Pointer to the new Sysfail interrupt handler Null = `DefaultSysfailHandler`

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Sysfail interrupt not supported

Example:

```
/* Set the Sysfail handler. */

void      func (int16);
int16     ret;

ret = SetSysfailHandler (func);
```

SetSysresetHandler

Syntax: `ret = SetSysresetHandler (func)`

Action: Replaces the current SYSRESET* interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`void (*)()`	Pointer to the new SYSRESET* interrupt handler Null = `DefaultSysresetHandler`

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = SYSRESET* interrupt not supported

Example:

```
/* Set the SYSRESET* handler. */

void      func ();
int16     ret;

ret = SetSysresetHandler (func);
```

Default Handlers for the System Interrupt Handler Functions

The NI-VXI software provides the following default handlers for the system interrupt handler functions. These are sample handlers that `InitVXIlibrary` installs when it initializes the software at the beginning of the application program. Default handlers give you the minimal and most common functionality required for a VME/VXI system. They are given in source code form on your NI-VXI distribution media to be used as examples/prototypes for extending their functionality to a particular application.

DefaultACfailHandler

Syntax: `DefaultACfailHandler (controller)`

Action: This default handler simply increments the global variable `ACfailRecv`.

Remarks: Input parameter:

`controller`	`int16`	Logical address of controller interrupting

Output parameters:

none

Return value:

none

DefaultBusErrorHandler

Syntax: `DefaultBusErrorHandler ()`

Action: This default handler simply increments the global variable `BusErrorRecv`.

Remarks: Parameters:

none

Return value:

none

DefaultSysfailHandler

Syntax: `DefaultSysfailHandler (controller)`

Action: Handles the interrupt generated when the SYSFAIL* signal on the VME backplane is asserted.

Remarks: Input parameter:

`controller`	`int16`	Logical address of controller interrupting

Output parameters:

none

Return value:

none

DefaultSysresetHandler

Syntax: `DefaultSysresetHandler (controller)`

Action: Handles the interrupt generated when the SYSRESET* signal on the VME backplane is asserted (and the local CPU is not configured to be reset itself). This default handler simply increments the global variable `SysresetRecv`.

Remarks: Input parameter:

`controller`	`int16`	Logical address of controller interrupting

Output parameters:

none

Return value:

none

Chapter 9
VXI/VMEbus Extender Functions

This chapter describes the C syntax and use of the VXI/VMEbus extender functions. The NI-VXI software interface fully supports the standard VXI/VMEbus 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 interrupt, Sysfail, ACfail, and Sysreset signals. For VXIbus systems, it also includes transparent extensions of all VXI TTL and ECL trigger lines. You can use the VXI/VMEbus extender functions to dynamically change the default RM settings if your application has such a requirement. Usually, the application never needs to change the default settings. Consult your utilities manual on how to use `vxiedit` or `vxitedit` to change the default extender settings.

Functional Overview

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.

MapUtilBus (extender, modes)

`MapUtilBus` configures chassis extender utility bus hardware to map Sysfail, ACfail, and/or Sysreset for the specified chassis into and/or out of the chassis. If the specified chassis extender can extend the VME utility signals between chassis, you can use the `MapUtilBus` function to configure the chassis-to-chassis 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 VME interrupt extension configuration in multiple-chassis configurations. If the specified chassis extender can extend the VME interrupts between chassis, you can use the `MapVXIint` function to configure the chassis-to-chassis 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 Descriptions

The following paragraphs describe the system configuration functions. The descriptions are explained at the C syntax level and are listed in alphabetical order.

MapUtilBus

Syntax: `ret = MapUtilBus (extender, modes)`

Action: Maps the specified VME utility bus signal for the specified chassis into and/or out of the chassis. The utility bus signals include Sysfail, ACfail, and Sysreset.

Remarks: Input parameters:

`extender`	`int16`	Chassis extender for which to map utility bus signals
`modes`	`uint16`	Bit vector of utility bus signals corresponding to the utility bus signals.

1 = Enable for corresponding signal and direction
0 = Disable for corresponding signal and direction

Bit	Utility Bus Signal and Direction
5	ACfail into the chassis
4	ACfail out of the chassis
3	Sysfail into the chassis
2	Sysfail out of the chassis
1	Sysreset into the chassis
0	Sysreset out of the chassis

Output parameters:
none

Return value:

`ret`	`int16`	Return Status

0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `extender`

Example:

```
/* Map Sysfail into Chassis 5.  Map Sysreset into and out of
   Chassis 5.  Do not map ACfail at all. */

int16     extender;
uint16    modes;
int16     ret;

extender= 5;
modes = (uint16)((1<<3) | (1<<1) | (1<<0));
ret = MapUtilBus (extender, modes);
```

MapVXIint

Syntax: `ret = MapVXIint (extender, levels, directions)`

Action: Maps the specified VME interrupt levels for the specified chassis in the specified direction (into or out of the chassis).

Remarks: Input parameters:

`extender`	`int16`	Chassis extender for which to map VME interrupt levels
`levels`	`uint16`	Bit vector of VME interrupt levels. Bits 6 to 0 correspond to VME interrupt levels 7 to 1 respectively. 1 = Enable for appropriate level 0 = Disable for appropriate level
`directions`	`uint16`	Bit vector of directions for VME interrupt levels. Bits 6 to 0 correspond to VME interrupt levels 7 to 1, respectively. 1 = Into the chassis 0 = Out of the chassis

Output parameters:

none

Return value:

`ret`	`int16`	Return Status 0 = Successful -1 = Unsupportable function (no hardware support) -2 = Invalid `extender`

Example:

```
/* Map VME interrupt levels 4 and 7 on the chassis extender at
   Logical Address 5 to go out of the chassis.  Map VME interrupt
   level 1 to go into the chassis. */

int16     extender;
uint16    levels;
uint16    directions;
int16     ret;

extender= 5;
levels = (uint16)((1<<0) | (1<<3) | (1<<6)); /** Levels 1, 4, 7. **/
directions = (uint16)(1<<0);        /* Level 1 only one in. */
ret = MapVXIint (extender, levels, directions);
```

Appendix
Customer Communication

For your convenience, this appendix and your Getting Started manual contain 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. Completing the forms 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.

Corporate Headquarters
(512) 795-8248
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices	**Phone Number**	**Fax Number**
Australia	(03) 879 9422	(03) 879 9179
Austria	(0662) 435986	(0662) 437010-19
Belgium	02/757.00.20	02/757.03.11
Denmark	45 76 26 00	45 76 71 11
Finland	(90) 527 2321	(90) 502 2930
France	(1) 48 14 24 00	(1) 48 14 24 14
Germany	089/741 31 30	089/714 60 35
Italy	02/48301892	02/48301915
Japan	(03) 3788-1921	(03) 3788-1923
Netherlands	03480-33466	03480-30673
Norway	32-848400	32-848600
Spain	(91) 640 0085	(91) 640 0533
Sweden	08-730 49 70	08-730 43 70
Switzerland	056/20 51 51	056/20 51 55
U.K.	0635 523545	0635 523154

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™ C Software Reference Manual for VME**

Edition Date: **November 1993**

Part Number: **320389-01**

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, MS 53-02
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
MS 53-02
(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.
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 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 tmeout 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. In VXI systems, 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

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 nsec.

controller	An intelligent device (usually involving a CPU) that is capable of controlling other devices.
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.
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 ten 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
DIRviol	Data In Ready violation
DOR	Data Out Ready
DORviol	Data Out Ready violation
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	An intelligent CPU (controller) interface plugged directly into the VXI backplane, giving it direct access to the VXIbus. It must have all of its required VXI interface capabilities built in.
END	Signals the end of a data string.

EOS	End Of String; a character sent to designate the last byte of a data message.
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 Longword Serial Protocol	A form of Word Serial communication in which Commanders and Servants communicate with 48-bit data transfers.
extended controller	A mainframe extender with additional VXIbus controller capabilities.
external controller	In this configuration, a plug-in interface board in a computer is connected to the VXI mainframe via one or more VXIbus extended controllers. The computer then exerts overall control over VXIbus system operations.

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

IACK — Interrupt Acknowledge

I/O — Input/output; the techniques, media, or devices used to achieve communication between entities.

IEEE — Institute of Electrical and Electronics Engineers

IEEE 1014 — The VME specification.

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*.

int8 — An 8-bit signed integer; may also be called a *char*.

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*.

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.

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.
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.
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.
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.

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.
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.
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

sec	Seconds
SEMI-SYNC Protocol	A one-line, open collector, multiple-device handshake trigger protocol.
Servant	A device controlled by a Commander.
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 which 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

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.
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.
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.

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

E

F

G

H

I

L

M

N

P

R

S

T

U

V

W