COMPREHENSIVE SERVICES

We offer competitive repair and calibration services, as well as easily accessible documentation and free downloadable resources.

SELL YOUR SURPLUS

We buy new, used, decommissioned, and surplus parts from every NI series. We work out the best solution to suit your individual needs.

Sell For Cash   Get Credit   Receive a Trade-In Deal

OBSOLETE NI HARDWARE IN STOCK & READY TO SHIP

We stock **New, New Surplus, Refurbished,** and **Reconditioned** NI Hardware.

APEX WAVES

Bridging the gap *between the manufacturer and your legacy test system.*

1-800-915-6216

www.apexwaves.com

sales@apexwaves.com

All trademarks, brands, and brand names are the property of their respective owners.

Request a Quote   CLICK HERE   ***SB-MXI***

NI-VXI™
Software Reference Manual for C

October 1994 Edition

Part Number 371693A-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, Mexico 95 800 010 0793,
Netherlands 03480-33466, Norway 32-84 84 00, Singapore 2265886, Spain (91) 640 0085, Sweden 08-730 49 70,
Switzerland 056/20 51 51, Taiwan 02 377 1200, U.K. 0635 523545

Contents

Chapter 5
Servant Word Serial Protocol Functions 5-1

Figures

About This Manual

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

Organization of This Manual

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

- Chapter 1, *Introduction to VXI,* introduces you to the concepts of VXI and MXI, and to the NI-VXI software.
- Chapter 2, *Introduction to the NI-VXI Functions,* introduces you to the NI-VXI functions and their capabilities, discusses the use of function parameters, describes application environments for which the functions are designed, and concludes with an overview on using NI-VXI.
- 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.
- Chapter 4, *Commander Word Serial Protocol Functions*, describes the C syntax and use of the VXI Commander Word Serial Protocol functions. Word serial communication is the minimal mode of communication between VXI Message-Based devices within the VXI Commander/Servant hierarchy. Commander Word Serial functions let the local CPU (the CPU on which the NI-VXI interface resides) perform VXI Message-Based Commander Word Serial communication with its Servants. This chapter defines the parameters and shows examples of the use of each function.
- Chapter 5, *Servant Word Serial Protocol Functions*, describes the C syntax and use of the VXI Servant Word Serial Protocol functions and default handlers. Word serial communication is the minimal mode of communication between VXI Message-Based devices within the VXI Commander/Servant hierarchy. The local CPU (the CPU on which the NI-VXI interface resides) uses the Servant Word Serial functions to perform VXI Message-Based Servant Word Serial communication with its Commander. This chapter defines the parameters and shows examples of the use of each function.
- Chapter 6, *Low-Level VXIbus Access Functions*, describes the C syntax and use of the VXI low-level VXIbus access functions. Low-level VXIbus access is the fastest access method for directly reading from or writing to any of the VXIbus address spaces. This chapter defines the parameters and shows examples of the use of each function.
- Chapter 7, *High-Level VXIbus Access Functions*, describes the C syntax and use of the VXI high-level VXIbus access functions. With high-level VXIbus access functions, you have direct access to the VXIbus 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.
- Chapter 8, *Local Resource Access Functions*, describes the C syntax and use of the VXI local resource access functions. With these functions, you have access to miscellaneous local resources such as the local CPU VXI register set, Slot 0 MODID operations, and the local CPU VXI Shared RAM. These functions are useful for shared memory type communication, non-Resource Manager operation, and debugging purposes. This chapter defines the parameters and shows examples of the use of each function.
- Chapter 9, *VXI Signal Functions*, describes the C syntax and use of the VXI signal functions and default handler. With these functions, VXI bus master devices can interrupt another device. VXI signal functions can specify the signal routing, manipulate the global signal queue, and wait for a particular signal value (or set of values) to be received. This chapter defines the parameters and shows examples of the use of each function.

- Chapter 10, *VXI Interrupt Functions*, describes the C syntax and use of the VXI interrupt functions and default handler. VXI interrupts are a basic form of asynchronous communication used by VXI devices with VXI interrupter support. These functions can specify the status/ID processing method, install interrupt service routines, and assert specified VXI interrupt lines with a specified status/ID value. This chapter defines the parameters and shows examples of the use of each function.

- Chapter 11, *VXI Trigger Functions*, describes the C syntax and use of the VXI trigger functions and default handlers. These functions provide a standard interface to source and accept any of the VXIbus TTL or ECL trigger lines. VXI trigger functions support all VXI-defined trigger protocols, with the actual capabilities dependent on the specific hardware platform. This chapter defines the parameters and shows examples of the use of each function.

- Chapter 12, *System Interrupt Handler Functions*, describes the C syntax and use of the VXI system interrupt handler functions and default handlers. With these functions, you can handle miscellaneous system conditions that can occur in the VXI environment. This chapter defines the parameters and shows examples of the use of each function.

- Chapter 13, *VXIbus Extender Functions*, describes the C syntax and use of the VXI extender functions. These functions can be used to dynamically reconfigure multi-mainframe transparent mapping of the VXI interrupt and trigger lines and utility bus signals. This chapter defines the parameters and shows examples of the use of each function.

- 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, and metric prefixes.

- The *Index* contains an alphabetical list of key terms and topics used in this manual, including the page where each one can be found.

Conventions Used in This Manual

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. In this manual, italics are also used to denote Word Serial commands, queries, and signals.
`monospace`	Text in this font denotes the names of all 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
- VXI-1, *VXIbus System Specification*, Revision 1.4, VXIbus Consortium
- VXI-6, *VXIbus Mainframe Extender Specification*, Revision 1.0, VXIbus Consortium

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual 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 our manuals.

Chapter 1
Introduction to VXI

This chapter introduces you to the concepts of VXI (VME eXtensions for Instrumentation) and MXI (Multisystem eXtension Interface), and to the NI-VXI software.

About the NI-VXI Functions

The comprehensive functions for programming the 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 line of 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*, describes these groups and explains when the functions within a group are normally used. Chapters 3 through 13 completely define each function within a group.

VXIbus Overview

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

VXI 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. VXI uses an 8-bit logical address, allowing for up to 256 VXI devices in a VXI system.

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

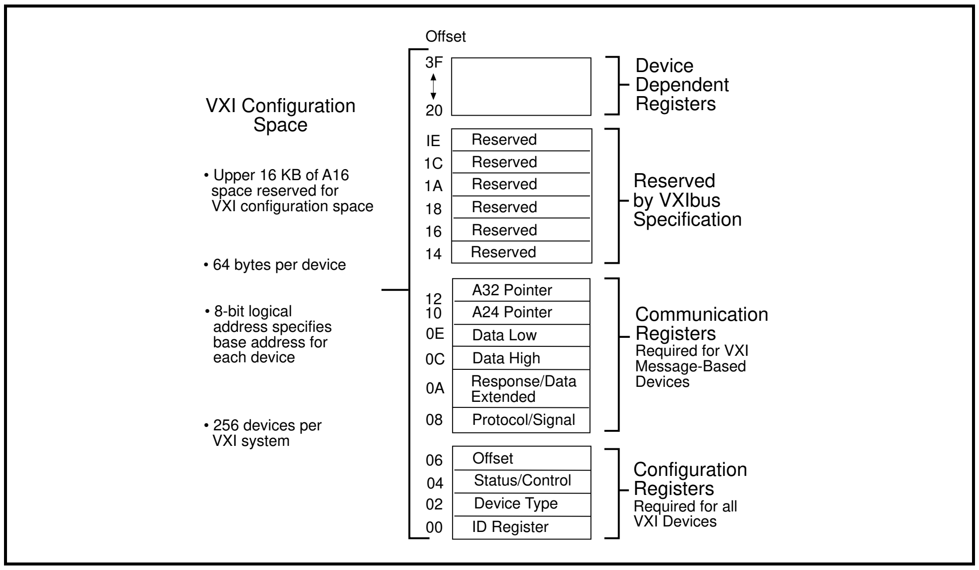
VXI Configuration Space
• Upper 16 KB of A16 space reserved for VXI configuration space
• 64 bytes per device
• 8-bit logical address specifies base address for each device
• 256 devices per VXI system
Offset
3F
20
IE
1C
1A
18
16
14
12
10
0E
0C
0A
08
06
04
02
00
Reserved
Reserved
Reserved
Reserved
Reserved
Reserved
A32 Pointer
A24 Pointer
Data Low
Data High
Response/Data Extended
Protocol/Signal
Offset
Status/Control
Device Type
ID Register
Device Dependent Registers
Reserved by VXIbus Specification
Communication Registers Required for VXI Message-Based Devices
Configuration Registers Required for all VXI Devices

Figure 1-1. VXI Configuration Registers

Register-Based Devices

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

Message-Based Devices

In addition to register-based devices, the VXIbus specification also defines *message-based* devices, which are required to have *communication registers* 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.

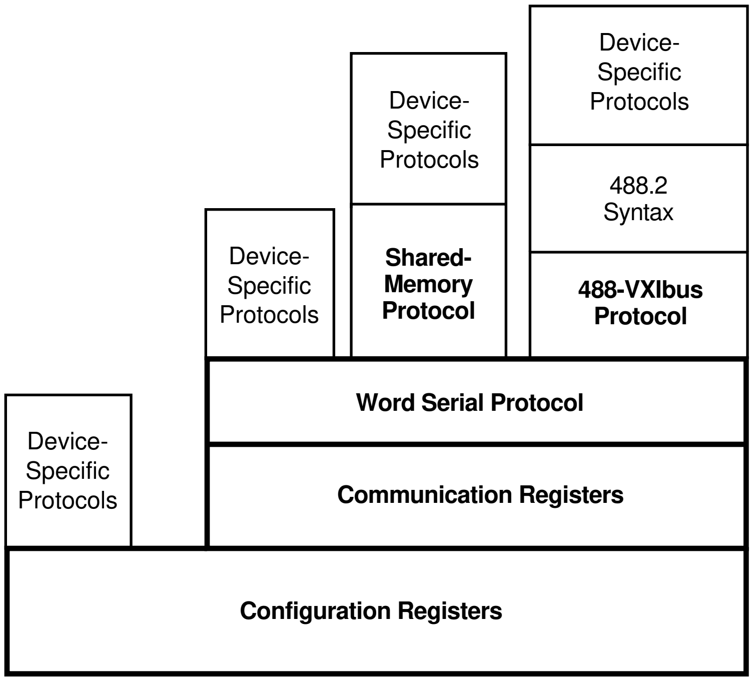

Figure 1-2. VXI Software Protocols

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 if you want to communicate with a message-based device, you do so by writing and reading 16-bit words one at a time to and from the Data In (write Data Low) and Data Out (read Data Low) hardware registers located on the device itself. Word serial communication is paced by bits in the device's response register that indicate whether the Data In register is empty and whether the Data Out register is full. This operation is very similar to Universal Asynchronous Receiver Transmitter (UART) on a serial port.

Commander/Servant Hierarchies

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

Interrupts and Asynchronous Events

Servants can communicate asynchronous status and events to their Commander through hardware interrupts or by writing specific messages (signals) directly to their Commander's hardware Signal register. Devices that do *not* have bus master capability always transmit such information via interrupts, whereas devices that *do* have bus master capability can use either interrupts or send signals. Some devices can receive only signals, some only interrupts, while some others can receive both signals and interrupts.

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

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

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

This chapter introduces you to the NI-VXI functions and their capabilities, discusses the use of function parameters, describes application environments for which the functions are designed, and concludes with an overview on using NI-VXI. You can find additional summaries on each class of function at the beginning of the function description chapters.

The NI-VXI functions are a set of C language functions you can use to perform operations in the VXI environment. The NI-VXI C language interface is independent of the hardware platform and the operating system environment.

The NI-VXI functions are divided into the following groups:

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

- **Commander Word Serial Protocol Functions**
  Word Serial is the minimal mode of communication between VXI message-based devices. Commander Word Serial functions give you the necessary capabilities to communicate with a message-based Servant device using the Word Serial, Longword Serial, or Extended Longword Serial protocols. These capabilities include command/query sending and buffer reads/writes.

- **Servant Word Serial Protocol Functions**
  Servant Word Serial functions give you 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.

- **Low-Level VXIbus Access Functions**
  Low-level VXIbus access is the fastest access method for directly reading from or writing to any of the VXIbus address spaces. You can use these functions to obtain a pointer that is directly mapped to a particular VXIbus address. Then you use the pointer with the low-level VXIbus access functions to read from or write to the VXIbus address space. When using these functions in your application, you need to consider certain programming constraints and error conditions such as bus errors (BERR*).

- **High-Level VXIbus Access Functions**
  Similar to the low-level VXIbus access functions, the high-level VXIbus access functions give you direct access to the VXIbus address spaces. You can use these functions to read, write, and move blocks of data between any of the VXIbus address spaces. You can specify any VXIbus privilege mode or byte order. The functions trap and report bus errors. When the execution speed is not a critical issue, the high-level VXIbus 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 VXI register set, Slot 0 MODID operations (when the local device is configured for Slot 0 operation), and the local CPU VXI Shared RAM. These functions are useful for shared memory type communication, for non-Resource Manager operation (when the local CPU is not the Resource Manager), and for debugging purposes.

- **VXI Signal Functions**
  VXI signals are a method for VXI bus masters to interrupt another device. 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 a handler or queue them on a global signal queue. You can use these functions to specify the signal routing, install handlers, manipulate the global signal queue, and wait for a particular signal value (or set of values) to be received.

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

- **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 work with all VXI-defined trigger protocols. The actual capabilities depend on the specific hardware platform. The VXI trigger functions can install handlers for various trigger interrupt conditions.

- **System Interrupt Handler Functions**
  The system interrupt handler functions let you install handlers for the system interrupt conditions. These conditions include Sysfail, ACfail, bus error, and soft reset interrupts.

- **VXIbus Extender Functions**
  The VXIbus extender functions can dynamically reconfigure multiple-mainframe transparent mapping of the VXI interrupt lines, TTL triggers, ECL triggers, and utility bus signals. The National Instruments Resource Manager configures the mainframe extenders with settings based on user-modifiable configuration files.

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 LynxOS or VxWorks). The NI-VXI interface includes most of the mutual exclusion necessary for a multitasking environment. Each individual platform has been optimized within the boundaries of the particular hardware and operating system environment.

LabWindows®/CVI

You can use the functions described in this manual with LabWindows/CVI. LabWindows/CVI is 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 must specify the address of the parameter when the parameter is an output parameter. The C "`&`" operator accomplishes this task.

For 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 occur because of system errors as well. 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 VXI system. Refer to *VXIbus Mainframe Extender Specification*, Revision 1.3 or later, for more details on multiple mainframe systems.

If you have a multiple-mainframe VXI 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. The embedded CPU configuration is an intelligent CPU interface directly plugged into the VXI backplane. The embedded CPU must have all of its required VXI interface capabilities built onto the embedded CPU itself. An embedded CPU has direct access to the VXIbus backplane for which it is installed. Access to other mainframes is done through the use of mainframe extenders.

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

Figure 2-1 illustrates the embedded and external CPU configurations.

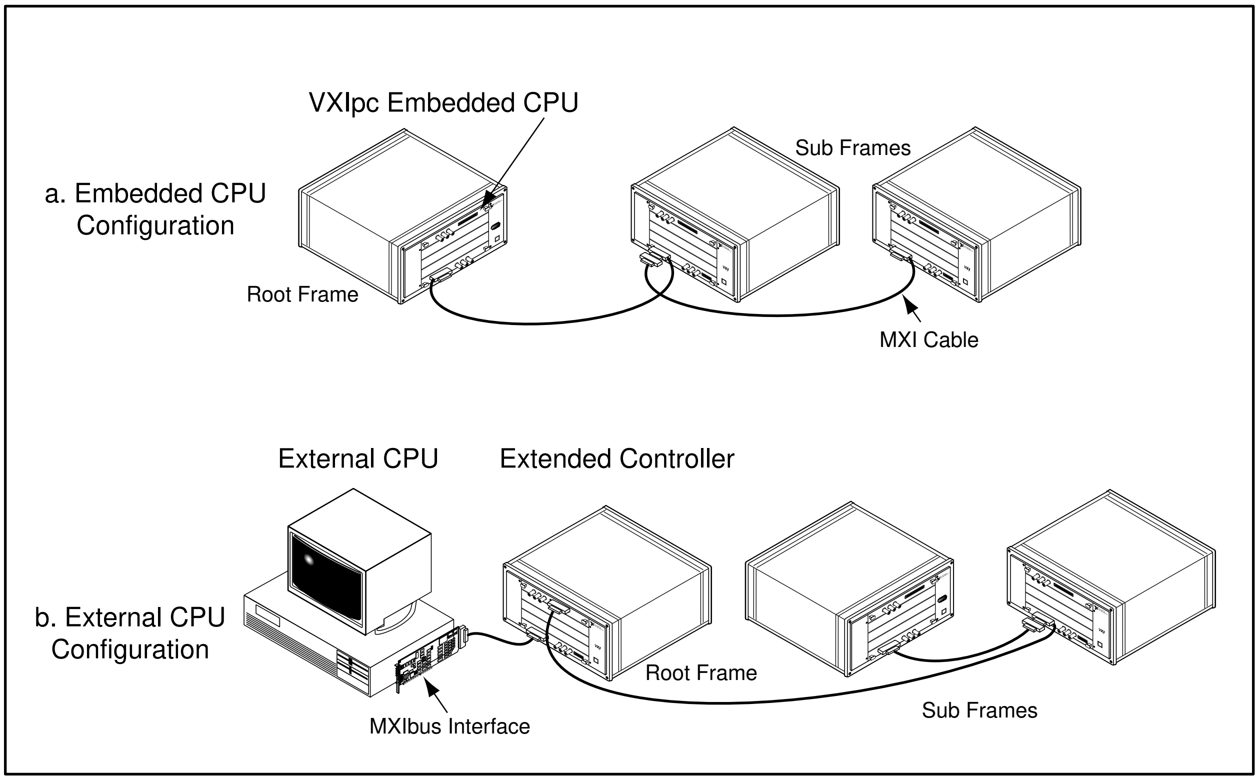

Figure 2-1. Embedded Versus External CPU Configurations

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 VXIbus capabilities. These features give the external CPU all of the features of an embedded CPU, including VXI interrupt, TTL trigger, ECL trigger, Sysfail, ACfail, and Sysreset support for VXI 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, and utility signals from below the first level of mainframe hierarchy, you should extend these VXIbus signals using the transparent VXIbus extender method (requiring INTX support on MXI extender products) via the Resource Manager configuration or VXIbus extender functions.

The Extender Versus Controller Parameters

This document uses the `extender` and `controller` parameters to specify the VXI 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 mainframe extender device that is used to access the particular mainframe (for example, a VXI-MXI or VME-MXI). Figure 2-2 shows an example of mainframe extenders used with the `extender` and/or `controller` parameters.

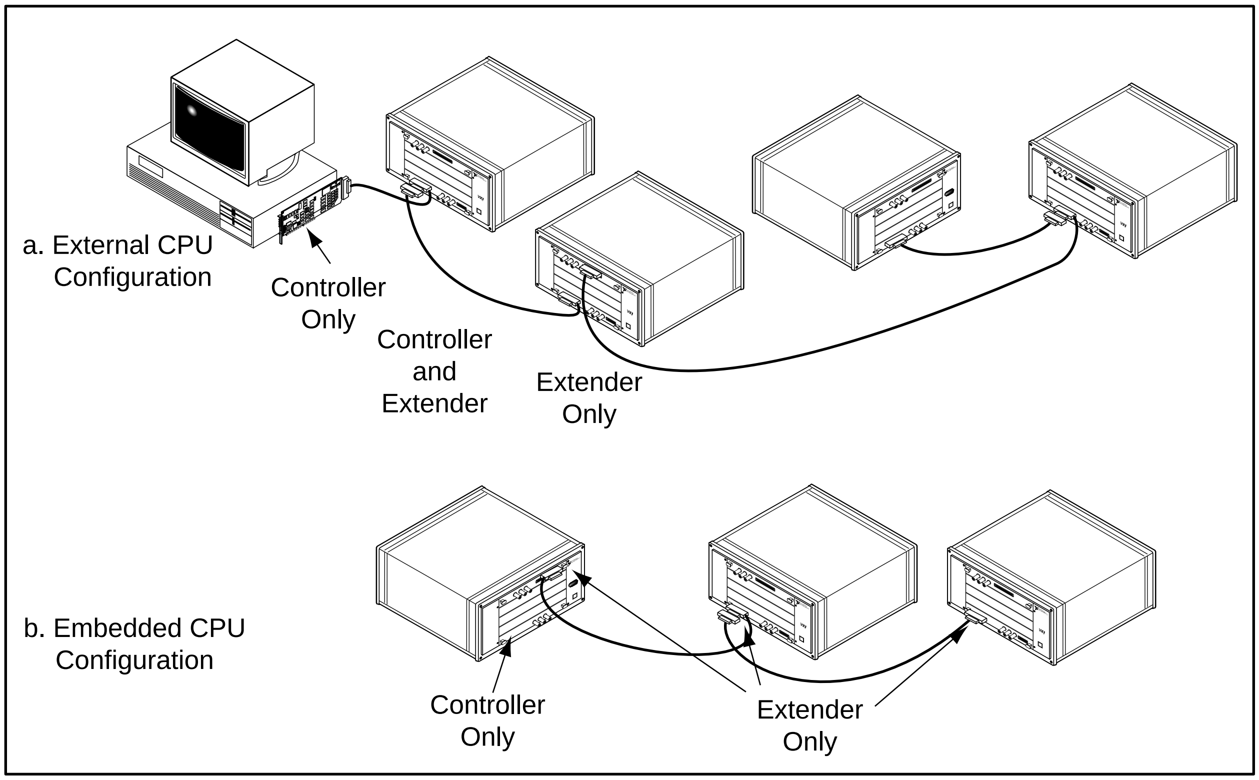

Figure 2-2. Extender Versus Controller Parameters

You can use the `extender` parameter only with the VXIbus extender functions, which are fully described in Chapter 13, *VXIbus Extender Functions*. With these functions, you can reconfigure the transparent mainframe extension configured by the Resource Manager. The extensions included are VXI interrupts, TTL and ECL triggers, and utility bus (Sysfail, ACfail, and Sysreset). The capabilities of the VXIbus extender functions are mapped directly onto the capabilities of the individual mapping registers of the standard VXIbus 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 VXIbus extender functions.

You will find the `controller` parameter only in NI-VXI functions that apply to embedded or extended controller capabilities. These capabilities include VXI interrupt, ACfail, Sysfail, and TTL/ECL trigger 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 is the logical address of the particular extended controller for which the functions should apply. As a result, you can use different sets of VXIbus 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 for an embedded CPU is a single VXI 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.

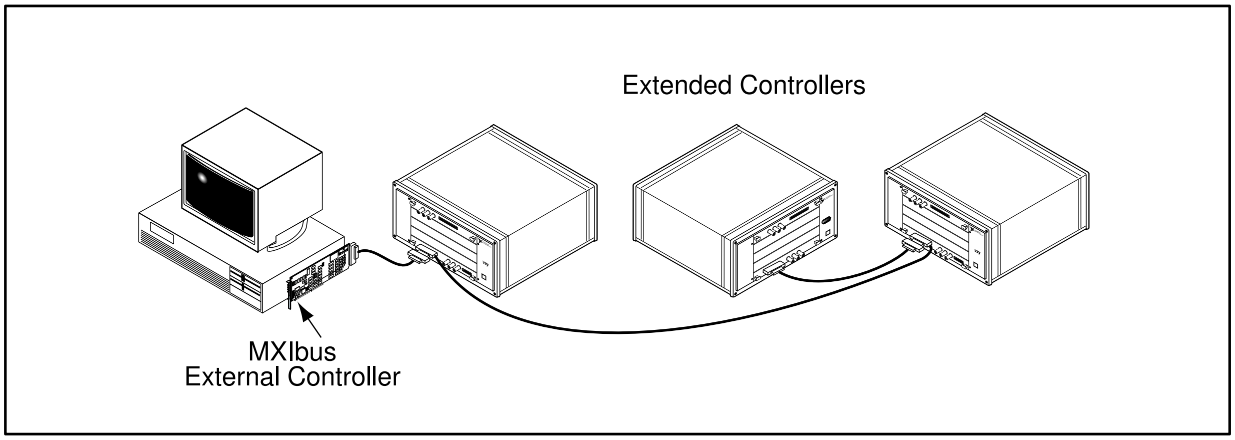

Figure 2-3. External CPU Configuration with Multiple Extended Controllers

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 VXIbus 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 cause error codes to be returned when used in the embedded CPU configuration. Using these `controller` parameter values implies that more than one extended controller has VXI interrupts, triggers, Sysfail, and/or ACfail conditions controlled directly by the external CPU. For full portability, you need to avoid this situation. You can do so by using the transparent mapping of the Resource Manager and the VXIbus extender functions (requiring INTX support for MXIbus mainframe extenders). Map all first-level mainframe VXI interrupts, triggers, Sysfails, and ACfails 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.

Using NI-VXI

This manual is designed as a reference for looking up specific information about a function or a class of functions (such as high-level VXIbus access or Commander Word Serial). Each section assumes a certain amount of knowledge on how to use these functions in your program. In contrast, this section presents a general overview of the more commonly used class of functions available in NI-VXI, summarizes how you can use them to perform certain tasks, and describes a general structure of programming using NI-VXI. For more information, please see the reference chapters on these functions, and review the example programs and README file that came with your driver. You can also contact National Instruments for a list of application notes available on VXI.

Variable Types

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

The datasize.h File

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

In addition to the integers, `datasize.h` defines several types for other uses such as interrupt handlers. For example, NIVXI_HVXIINT is an interrupt handler type. Merely defining a variable with this type is sufficient to create the function prototype necessary for your interrupt handler. Also, different platforms require different flags for use with interrupt handlers. To simplify this problem, `datasize.h` defines NIVXI_HQUAL and NIVXI_HSPEC, which are used in the handler definition and take care of the platform dependencies. See the *Interrupts and Signals* section later in this chapter and your README file for more information. In addition, refer to Chapter 10, *VXI Interrupt Functions*, and Chapter 12, *System Interrupt Handler Functions*.

The busacc.h File

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

The devinfo.h File

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

The Beginning and End

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

An important note about these two functions is that the internal structure maintains a record of the number of calls to `InitVXIlibrary()` and `CloseVXIlibrary()`. Although `InitVXIlibrary()` needs to be called only once, the structure of your program may cause the function to be called multiple times. A successful call to `InitVXIlibrary()` returns either a zero or a one. A zero indicates that the structure has been created, and a one indicates that the structure was created by an earlier call so no action was taken (other than incrementing the count of the number of `InitVXIlibrary()` calls). When `CloseVXIlibrary()` returns a successful code, it also returns either a zero or a one. A zero indicates that the structure has been successfully destroyed, and a one indicates that there are still outstanding calls to `InitVXIlibrary()` that must be closed before the structure is destroyed. The outcome of all of this is that when exiting a program, you should call `CloseVXIlibrary()` *the same number of times* that you have called `InitVXIlibrary()`.

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

Useful Tools

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

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

Consider the case of wanting to locate a device with manufacturer’s code ABC hex and model number 123 hex. You could use the following code to determine the logical address.

Note: ***In the examples in this section, most of the return codes from the functions are not checked for either warnings nor errors. This step is omitted only to simplify the example programs. We strongly recommend that your own programs include error checking.***

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

      ret = InitVXIlibrary();

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

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

Word Serial Communication

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

Word Serial Protocol is based on a Commander writing 16-bit commands to a Servant register (See Chapter 4, *Commander Word Serial Protocol Functions*, and Chapter 5, *Servant Word Serial Protocol Functions*, for more information on the protocol). The VXIbus specification has defined several commands, such as *Byte Available*, *Byte Request*, and *Clear*. The bit patterns for Word Serial commands have been laid out in the VXIbus specification, and your application can send these commands to a Servant via the `WScmd()` function. However, because string communication is the most common use for Word Serial Protocol, the functions `WSwrt()` and `WSrd()` use the Word Serial commands *Byte Available* (for sending a byte to a servant) and *Byte Request* (for retrieving a byte from a Servant) repetitively to send or receive strings as defined by the Word Serial Protocol. In addition, other common commands such as *Clear* have been encapsulated in their own functions, such as `WSclr()`.

Chapter 4 describes all NI-VXI functions pertaining to message-based communication for the Commander. However, there are times when you want the controller to operate as a Word Serial Servant. NI-VXI allows for the controller to accept Word Serial commands from a Commander. Chapter 5 describes a different set of functions that a Servant uses for message-based communication with its Commander.

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

Master Memory Access

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

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

You can use the low-level VXIbus access functions if you want to access VXI memory with as little overhead as possible. Although you now have to perform such actions as bus error handling and mapping—which are handled automatically by the high-level functions—you can experience a performance gain if you optimize for the particular accesses you are performing. Consider the following sample code, which performs a memory access using the low-level functions. Notice that there is no bus error handler installed by the program (See the *Interrupts and Signals* section). Instead, the program uses the NI-VXI default bus error handler. This handler automatically increments the BusErrorRecv global variable (See the description of `DefaultBusErrorHandler()` in Chapter 12).

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

main() {
      INT16 ret;
      UINT16 *addrptr, svalue;
      UINT32 addr, window1;
      INT32 timeout;
      void *addptr1;

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

      /* The following code maps the A16 space with the Access Only */
      /* access in order to access the A16 space directly.  */
      addr = 0xc000L;  /* Map upper 16 KB of the A16 space */
      timeout = 2000L; /* 2 seconds */

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

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

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

Notice that the return variable for the `MapVXIAddress()` function is a pointer. You can dereference this pointer directly on some platforms—for example, by writing `*addrptr = 0` rather than `VXIpoke(addrptr, 2, 0)`. This applies currently to any of the embedded computers and the SB-MXI. We recommend that you do not dereference the pointer directly on any other platform; due to various hardware or software constraints, NI-VXI cannot guarantee the success of the I/O operation.

Note: ***The AT-MXI and VXIpc-486 embedded computers have a compiler option known as binary compatibility, available on certain operating systems. With this feature, object code can use the most efficient methods for*** `VXIpeek()` ***and*** `VXIpoke()` ***depending on the platform—either a function call or a pointer dereference. Please see the README file with these platforms for more information.***

Slave Memory Access

It is possible to share local resources such as RAM and port I/O space with the VXIbus. You can accomplish this functionality by setting the appropriate fields in the `VXIedit` or `VXItedit` program to instruct the controller to respond to bus accesses as a slave. What address space is used is dependent on the settings in the `VXIedit` or `VXItedit` program. However, the actual VXIbus memory addresses are assigned by `RESMAN` and should be read by the program through the `GetDevInfo()` function. See the *Useful Tools* section for more information on `GetDevInfo()`.

Keep in mind that when the controller shares its resources, it does not allocate them from the local system first. For example, if you instruct the system to share 1 MB of RAM, the controller will map VXI addresses (as defined by `RESMAN`) to 1 MB of local memory. However, at no point has the controller prevented the local system from also using this space. For example, on a IBM compatible PC, the first 1 MB of address space contains not only user RAM, but also the interrupt vector table, video memory, BIOS, and so on. Therefore, it is important that you first use `VXImemAlloc()` to reserve a portion of the shared memory, and then communicate this address to the remote master that will be accessing the slave memory. For example, assume that the following code will run on a controller that has shared 1 MB of local RAM.

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

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

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

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

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

Interrupts and Signals

In NI-VXI, you can set up your controller to function as both an interrupt handler and an interrupter. You can also have your controller respond to writes to its signal register, if present. Signaling another device requires the high-level or low-level VXIbus access functions, as discussed earlier. In addition, NI-VXI lets you configure both interrupts and signals to be handled either through handlers or through the signal queue. Chapter 9, *VXI Signal Functions*, goes into greater detail on the signal queue, but for now you can look upon it as a FIFO (first-in, first-out) queue that you can access via the signal queue management functions, such as `SignalDeq()`. Both the signal queue and the interrupt handler will provide the status/ID obtained from the interrupt acknowledge or from the signal register. You can use this value to determine which device generated the interrupt/signal as well as the cause of the event. See Chapter 10, *VXI Interrupt Functions*, for more information.

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

Note: `RESMAN` ***allocates interrupt lines and devices should use only those interrupt lines allocated to them. Again, you can use*** `GetDevInfo()` ***to determine what interrupts lines have been allocated to the controller.***

Alternatively, you can choose to handle either signals or interrupts with an interrupt handler. You can use `RouteSignal()` to specify that the events (signals and/or interrupts) should be handled by the interrupt handlers rather than the signal queue. (`RouteVXIint()` is not necessary because the default for VXI interrupts is interrupt handlers). After you have enabled the event handler through either the `EnableSignalInt()` or the `EnableVXIint()` call, the callback function will be invoked when the event occurs. Installing and using interrupt handlers is very simple with NI-VXI because all of the operating system interaction is handled for you. The following section of code gives an example of using an interrupt handler.

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

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

main () {
      INT16 ret, controller;

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

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

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

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

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

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

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

Note: ***Although NI-VXI simplifies the installation and use of interrupt handlers, it cannot affect how the system handles interrupts. It is the programmer's responsibility to follow programming guidelines set by the operating system being used, such as use of reentrant code only, and/or timing restrictions, and on Macintosh computers, regaining access to global variables.***

Triggers

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

Using NI-VXI to source or acknowledge triggers is very simple. The `SrcTrig()` function can generate any of the VXIbus-defined trigger protocols, and `AcknowledgeTrig()` can perform the acknowledgment to the ASYNC and SEMI-SYNC protocols if you do not want NI-VXI to acknowledge automatically for you. In addition, you can generate interrupts on certain trigger conditions to promote a faster response to triggers. The use of these interrupts is the same as described in the previous section, except that you would use the `GetTrigHandler()` and `SetTrigHandler()` functions to install and remove the interrupt handlers, respectively. Please refer to Chapter 11, *VXI Trigger Functions*, for more information on triggering with NI-VXI and the TIC chip.

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 so that you can find device names or logical addresses by specifying certain attributes of the device for identification purposes.

Initializing and closing the NI-VXI software interface, and getting information about devices in the system are among the most important aspects of the NI-VXI software. All applications need to use the system configuration functions at one level or another. When the NI-VXI RM runs, it logs the system configuration information in the RM table file, `resman.tbl.` The `InitVXIlibrary` function reads the information from `resman.tbl` into data structures accessible from the `GetDevInfo` and `SetDevInfo` functions. From this point on, you can retrieve any device-related information from the entry in the table. Only in very special cases should you modify the information in the table, which you can do using one of the `SetDevInfo` functions. In this manner, both the application and the driver functions have direct access to 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 (negative one) or `""` specifies to ignore the corresponding field in the attribute comparison. After finding the VXI logical address, you can use one of the `DevInfo` functions to get any information about the specified device.

GetDevInfo (la, field, fieldvalue)

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

GetDevInfoShort (la, field, shortvalue)

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

GetDevInfoLong (la, field, longvalue)

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

GetDevInfoStr (la, field, stringvalue)

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

SetDevInfo (la, field, fieldvalue)

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

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, VXI address space allocated, VXI interrupt lines/handlers allocated, protocols supported, and so on. You should use this function only in very special situations, because it updates information in the NI-VXI interface and can affect execution. At the startup of your application, `InitVXIlibrary` completely initializes the RM table to how the RM configured the VXI system. No initial changes are necessary for VXI devices.

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 VXI system. No initial changes are necessary for VXI devices.

SetDevInfoStr (la, field, stringvalue)

`SetDevInfoStr` changes information about the specified device in the NI-VXI RM table. The `field` parameter specifies the attribute of the information to change. `SetDevInfoStr` is a function layered on top of `SetDevInfo` for languages (such as BASIC) that cannot typecast the `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 VXI system. No initial changes are necessary for VXI devices.

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 VXI system. No initial changes/creations are necessary for VXI devices. You can use `CreateDevInfo` to add non-VXI devices or pseudo devices (future expansion).

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 1 = Successful; previous `InitVXIlibrary` calls still pending 0 = NI-VXI library closed successfully -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 the fields in the device information table for the entry 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;
INT16      la;

la = 298;
ret = CreateDevInfo (la);
if (ret < 0)
   /* An error occurred creating new entry. */;
```

FindDevLA

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

Action: Finds a VXI 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, the function 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      slot;
INT16      mainframe;
INT16      cmdrla;
INT16      la;

namepat = "";
manid = 0xff6;
modelcode = 0xff;
devclass = -1;
slot = -1;
mainframe = -1;
cmdrla = -1;
ret = FindDevLA (namepat, manid, modelcode, devclass, slot,
mainframe, 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	`UserLaEntry`	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;
INT16      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`	`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;
INT16      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 1 = NI-VXI library already initialized (repeat call) 0 = NI-VXI library initialized -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	`UserLaEntry`	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 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
	<u>Field</u>	<u>Description</u>
	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, "DMM0");
ret = SetDevInfoStr (la, field, stringvalue);
if (ret != 0)
   /* Invalid logical address or field specified. */;
```

Chapter 4
Commander Word Serial Protocol Functions

This chapter describes the C syntax and use of the VXI Commander Word Serial Protocol functions. Word Serial communication is the minimal mode of communication between VXI Message-Based devices within the VXI Commander/Servant hierarchy. The Commander Word Serial functions let the local CPU (the CPU on which the NI-VXI interface resides) perform VXI Message-Based Commander Word Serial communication with its Servants. The four basic types of Commander Word Serial transfers are as follows:

- Command sending
- Query sending
- Buffer writes
- Buffer reads

Word Serial Protocol is a simple 16-bit transfer protocol between a Commander and its Servants. The Commander polls specific bits in the Servant's VXI Response register to determine when it can write a command , when it can read a response from the Data Low register, and when a Word Serial protocol error occurs. Before a Commander can send a Word Serial command to a Servant, it must first poll the Write Ready (WR) bit until it is asserted (set to 1). The Commander can then write the command to the Data Low register. If the Commander is sending a query, it first sends the query in the same manner as sending a command, but then continues by polling the Read Ready (RR) bit until it is asserted. It then reads the response from the Data Low register. A buffer write simply involves sending a series of *Byte Available* (BAV) Word Serial commands to the Servant, with the additional constraint that the Data In Ready (DIR) bit as well as the WR bit must be asserted before sending the *Byte Available*. The lower 8 bits (bits 0 to 7) of the 16-bit command contain a single byte of data (bit 8 is the END bit). Therefore, one *Byte Available* is sent for each data byte in the buffer written. A buffer read simply involves sending a series of *Byte Request* (BREQ) Word Serial queries to the Servant, with the additional constraint that the Data Out Ready (DOR) bit as well as the WR bit must be asserted before sending the *Byte Request*. The lower 8 bits (bits 0 to 7) of the 16-bit response contain a single byte of data (bit 8 is the END bit). Therefore, one *Byte Request* is sent for each data byte in the buffer read.

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

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

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

Programming Considerations

The Commander Word Serial functions provide a flexible and very easy-to-use interface. Depending upon the hardware and software platforms involved in your system, however, certain issues need to be taken into account.

Interrupt Service Routine Support

If portability between operating systems is essential (or a single-tasking/real-time operating system is used), the Word Serial Protocol functions should not be called from an interrupt service routine. Only for operating systems in which the user-installed handlers are run at process level (most UNIX and OS/2 systems) is it possible to initiate a Word Serial operation. The Commander Word Serial functions require operating system support only provided at the application (process) level of execution. Calling these functions from CPU interrupt level will have undetermined results. The `WSabort` function is the only exception to this. `WSabort` is used to abort various Word Serial transfers in progress and will usually be called from an interrupt service routine (although it is not limited to interrupt service routines). The most common example of this is with the *Unrecognized Command* events from devices implementing Word Serial to VXIbus specification 1.2. When an *Unrecognized Command* event is received by the NI-VXI VXI interrupt or Signal interrupt handler, `WSabort` must be called to abort the current Word Serial command transfer in progress that caused the generation of the *Unrecognized Command* event.

Single-Tasking Operating System Support

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

Multitasking Support (Non-Preemptive Operating System)

The Word Serial Protocol functions have extensive mutual exclusion support when running in non-preemptive multitasking operating systems. In a non-preemptive operating system, an operating system call may not be forcefully suspended (preempted) by a higher level process. Once an operating system call has been initiated, it will run to completion unless the call itself decides to give up the processor. The Commander Word Serial functions allow read and write or trigger calls to be made at the same time. Command transfers will automatically suspend a read, write, or trigger call in progress. Figure 4-1 gives a precise description of how this exclusion works. If the application is to be compatible with IEEE 488.2, the application must perform trigger and write calls in sequential order. Notice that this exclusion is on a per logical address basis. Any number of logical addresses can have Word Serial transfers in progress without conflict. For each logical address, however, the restriction in the model presented in Figure 4-1 must be followed. When this model is followed exactly, its effectiveness is limited by the non-preemptive nature of the operating system. Switches between different processes (Word Serial function calls) will only occur when the function currently executing willfully relinquishes the processor. This happens only at regular designated points in the Word Serial operation.

The Commander Word Serial functions are fully reentrant and preemptive on a per logical address basis. Because of the nature of a non-preemptive operating system, the number of calls pending will be very limited. Any number of logical addresses can have Commander Word Serial functions in progress without conflict. A higher level process may make a Commander Word Serial call when others are already in progress as long as the restrictions in Figure 4-1 are followed. Again, however, the preemptive nature of the operating system will greatly reduce the ability of the application to initiate multiple Word Serial operations.

Because Commander Word Serial is a protocol involving extensive polling, support has been added for *round robining* of Commander Word Serial function calls with other processes. If a particular logical address takes more than one millisecond to accept or respond to a particular Word Serial command or query, the process is suspended and another process (possibly with a different Commander Word Serial call in progress) can continue to execute. The amount of time for which the process is suspended is dependent upon the operating system (usually 10 to 40 ms). When the process is resumed, the polling will continue. The polling will continue until the transfer is complete or a timeout occurs. This support also keeps a *hung* device from hanging the operating system on the local CPU.

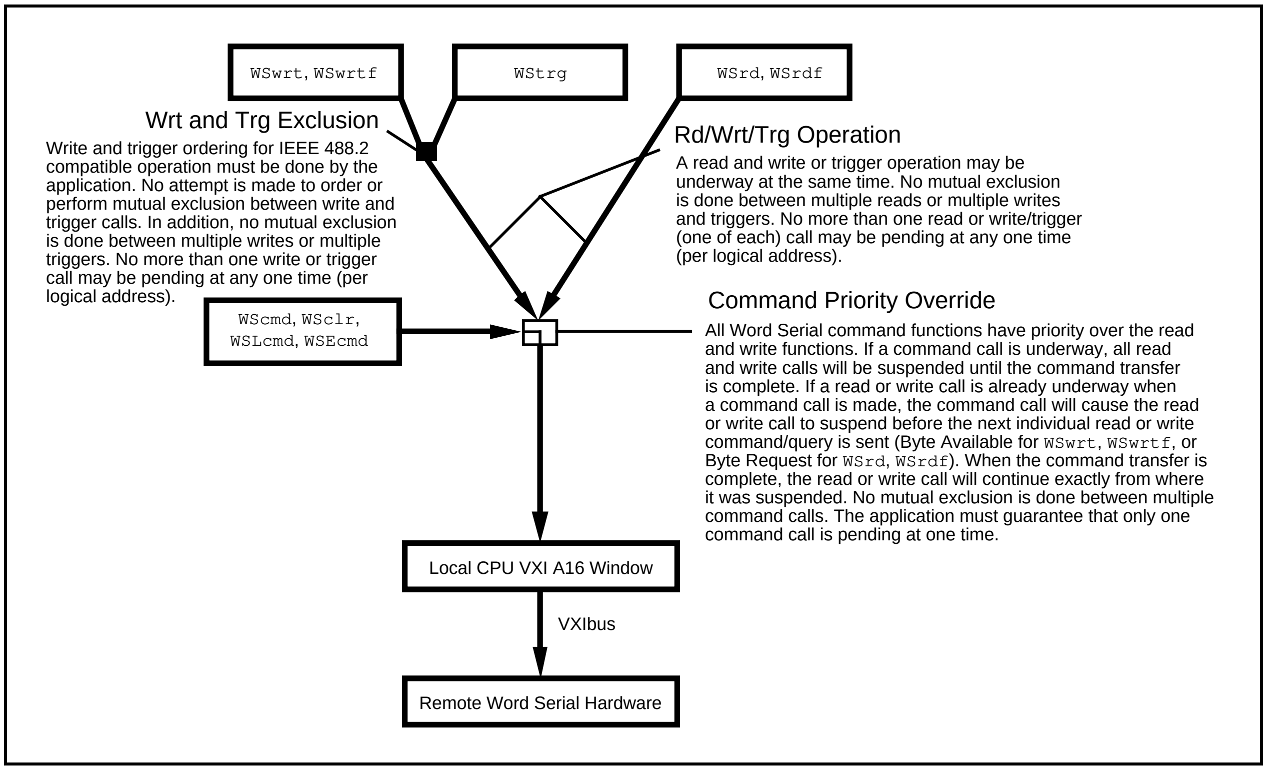
WSwrt, WSwrtf
WStrg
WSrd, WSrdf
Wrt and Trg Exclusion
Write and trigger ordering for IEEE 488.2 compatible operation must be done by the application. No attempt is made to order or perform mutual exclusion between write and trigger calls. In addition, no mutual exclusion is done between multiple writes or multiple triggers. No more than one write or trigger call may be pending at any one time (per logical address).
Rd/Wrt/Trg Operation
A read and write or trigger operation may be underway at the same time. No mutual exclusion is done between multiple reads or multiple writes and triggers. No more than one read or write/trigger (one of each) call may be pending at any one time (per logical address).
WScmd, WSclr, WSLcmd, WSEcmd
Command Priority Override
All Word Serial command functions have priority over the read and write functions. If a command call is underway, all read and write calls will be suspended until the command transfer is complete. If a read or write call is already underway when a command call is made, the command call will cause the read or write call to suspend before the next individual read or write command/query is sent (Byte Available for WSwrt, WSwrtf, or Byte Request for WSrd, WSrdf). When the command transfer is complete, the read or write call will continue exactly from where it was suspended. No mutual exclusion is done between multiple command calls. The application must guarantee that only one command call is pending at one time.
Local CPU VXI A16 Window
VXIbus
Remote Word Serial Hardware

Figure 4-1. Non-Preemptive Word Serial Mutual Exclusion (Per Logical Address)

Real-Time Multitasking Support (Preemptive Operating System)

The Word Serial Protocol functions have extensive mutual exclusion support when running on a preemptive multi-tasking operating system (most real-time operating systems). A two-level mutual exclusion algorithm is used to allow read and write or trigger calls to be made at the same time. Command transfers will automatically suspend a read, write, or trigger call in progress. Figure 4-2 gives a precise description of this two-level mutual exclusion algorithm. Notice that this mutual exclusion is on a per logical address basis. Any number of logical addresses can have Word Serial transfers in progress without conflict. For each logical address, however, the restriction presented in Figure 4-1 must be followed. If the application is to be compatible with IEEE 488.2, the application must perform trigger and write calls in sequential order.

The Commander Word Serial Functions are fully reentrant and preemptive on a per logical address basis. Any number of logical addresses can have Commander Word Serial functions in progress without conflict. A higher level process can make a Commander Word Serial call when others are already in progress so long as the restrictions in Figure 4-1 are followed.

Because Commander Word Serial is a protocol involving extensive polling, support has been added for *round robining* of Commander Word Serial function calls with other processes. If a particular logical address takes more than one millisecond to accept or respond to a particular Word Serial command or query, the process is suspended and another process (possibly with a different Commander Word Serial call in progress) can continue to execute. The amount of time for which the process is suspended is dependent upon the operating system (usually 10 to 40 ms). When the process is resumed, the polling will continue. The polling will continue until the transfer is complete or a timeout occurs. This support also keeps a *hung* device from hanging the operating system on the local CPU.

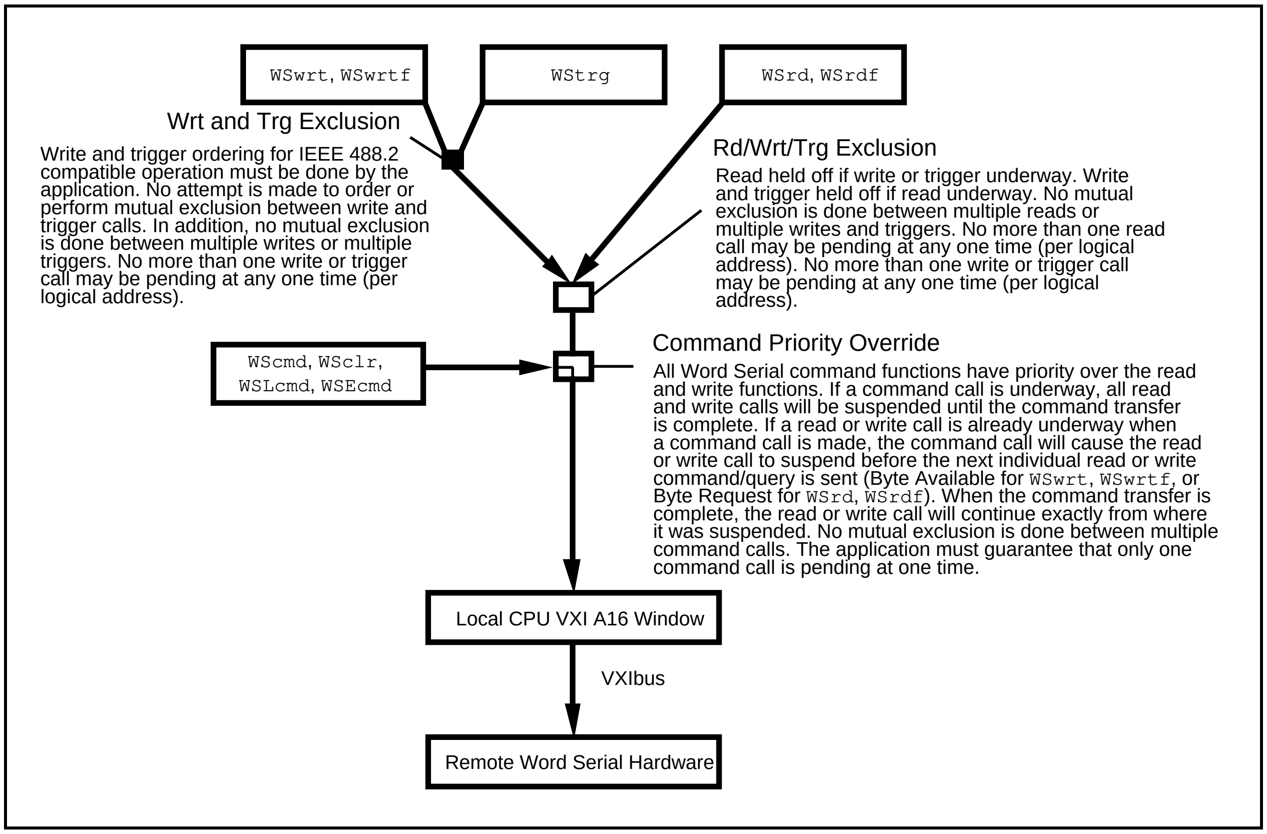
WSwrt, WSwrtf
WStrg
WSrd, WSrdf
Wrt and Trg Exclusion
Write and trigger ordering for IEEE 488.2 compatible operation must be done by the application. No attempt is made to order or perform mutual exclusion between write and trigger calls. In addition, no mutual exclusion is done between multiple writes or multiple triggers. No more than one write or trigger call may be pending at any one time (per logical address).
Rd/Wrt/Trg Exclusion
Read held off if write or trigger underway. Write and trigger held off if read underway. No mutual exclusion is done between multiple reads or multiple writes and triggers. No more than one read call may be pending at any one time (per logical address). No more than one write or trigger call may be pending at any one time (per logical address).
WScmd, WSclr, WSLcmd, WSEcmd
Command Priority Override
All Word Serial command functions have priority over the read and write functions. If a command call is underway, all read and write calls will be suspended until the command transfer is complete. If a read or write call is already underway when a command call is made, the command call will cause the read or write call to suspend before the next individual read or write command/query is sent (Byte Available for WSwrt, WSwrtf, or Byte Request for WSrd, WSrdf). When the command transfer is complete, the read or write call will continue exactly from where it was suspended. No mutual exclusion is done between multiple command calls. The application must guarantee that only one command call is pending at one time.
Local CPU VXI A16 Window
VXIbus
Remote Word Serial Hardware

Figure 4-2. Preemptive Word Serial Mutual Exclusion (Per Logical Address)

Functional Overview

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

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

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

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

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

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

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

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

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

WScmd (la, cmd, respflag, response)

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

WSresp (la, response)

`WSresp` retrieves a response to a previously sent Word Serial Protocol query from a VXI Message-Based Servant.

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

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

WStrg (la)

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

WSclr (la)

`WSclr` sends the Word Serial *Clear* command to a Message-Based Servant. The *Clear* command clears any pending protocol error on the receiving device. The ERR* bit is ignored during the transfer so as not to generate a protocol error. The WR bit is polled until asserted after the *Clear* command is sent to verify that the command executed properly.

WSabort (la, abortop)

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

WSLcmd (la, cmd, respflag, response)

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

WSLresp (la, response)

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

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

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

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

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

WSsetTmo (timo, actualtimo)

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

WSgetTmo (actualtimo)

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

Function Descriptions

The following paragraphs describe the Commander Word Serial, Longword Serial, and Extended Longword Serial Protocol functions. The descriptions are explained at the C syntax level and are listed in alphabetical order.

WSabort

Syntax: `ret = WSabort (la, abortop)`

Action: Performs a Forced or Unrecognized (Unsupported) Command abort of a Commander Word Serial operation(s) in progress.

Remarks: Input parameters:

`la`	`INT16`	Logical address of the Message-Based device
`abortop`	`UINT16`	The operation to abort 1 = `Forced Abort:` aborts `WSwrt`, `WSrd`, and `WStrg` 2 = `UnSupCom`: aborts `WScmd`, `WSLcmd`, and `WSEcmd` 3 = `Forced Abort:` aborts `WScmd`, `WSLcmd`, and `WSEcmd` 4 = `Forced Abort:` aborts all Word Serial operations 5 = `Async Abort:` aborts all Word Serial operations immediately. Be careful when using this option. During a Word Serial query, the Servant may be left in an invalid state if the operation is aborted after writing the query and before reading the response register. When using this option, the Word Serial operation aborts immediately as compared to using options 1, 3, and 4, where the operation does not abort until reading the response.

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successfully aborted -1 = Invalid `la` -2 = Invalid `abortop`

Example:

```
/* Perform Unsupported Command abort on Logical Address 5. */

INT16      ret;
INT16      la;
UINT16     abortop;

la = 5;
abortop = 2;
ret = WSabort (la, abortop);
if (ret < 0)
   /* An error occurred during WSabort. */;
```

WSclr

Syntax: `ret = WSclr (la)`

Action: Sends the Word Serial *Clear* command to a Message-Based device.

Remarks: Input parameter:

`la`	`INT16`	Logical address of the Message-Based device

Output parameters:

none

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
7	BERR	Bus error occurred during transfer
5	InvalidLA	Invalid `la` specified
2	TIMO_DONE	Timed out before WR set (clear complete)
1	TIMO_SEND	Timed out before able to send *Clear*
Successful Transfer (Bit 15 = 0)		
0	IODONE	Command transfer successfully completed

Example:

```
/* Send Clear command to Logical Address 5. */

INT16      ret;
INT16      la;

la = 5;
ret = WSclr (la);
if (ret < 0)
   /* An error occurred during the command transfer. */;
```

WScmd

Syntax: `ret = WScmd (la, cmd, respflag, response)`

Action: Sends a Word Serial command or query to a Message-Based device.

Remarks: Input parameters:

`la`	`INT16`	Logical address of the Message-Based device
`cmd`	`UINT16`	Word Serial command value
`respflag`	`UINT16`	Non-0 = Get a response (query) 0 = Do not get a response

Output parameter:

`response`	`UINT16*`	16-bit location to store response

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device did not recognize the command
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
2	TIMO_RES	Timed out before response received
1	TIMO_SEND	Timed out before able to send command
Successful Transfer (Bit 15 = 0)		
0	IODONE	Command transfer successfully completed

Example:

```
/* Send the Word Serial command Read STB to a device at Logical
   Address 5, and get the response. */

INT16      ret;
INT16      la;
UINT16     cmd;
UINT16     respflag;
UINT16     response;

la = 5;
cmd = 0xcfff;
respflag = 1;
ret = WScmd (la, cmd, respflag, &response);
if ( ret < 0)
   /* An error occurred during WS command transfer. */;
```

WSEcmd

Syntax: `ret = WSEcmd (la, cmdExt, cmd, respflag, response)`

Action: Sends an Extended Longword Serial command or query to a Message-Based device.

Remarks: Input parameters:

`la`	`INT16`	Logical address of the Message-Based device
`cmdExt`	`UINT16`	Upper 16 bits of 48-bit Extended Longword Serial command value
`cmd`	`UINT32`	Lower 32 bits of 48-bit Extended Longword Serial command value
`respflag`	`UINT16`	Non-0 = Get a response (query) 0 = Do not get a response

Output parameter:

`response`	`UINT32*`	32-bit location to store response

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device did not recognize the command
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
2	TIMO_RES	Timed out before response received
1	TIMO_SEND	Timed out before able to send command
Successful Transfer (Bit 15 = 0)		
0	IODONE	Command transfer successfully completed

Example:

```
/* Send the Extended Longword Serial command FFFCFFFDFFFE hex to a
   device at Logical Address 5, and get the response. */

INT16      ret;
INT16      la;
UINT16     cmdExt;
UINT32     cmd;
UINT16     respflag;
UINT32     response;

la = 5;
cmdExt = 0xfffc;
cmd = 0xfffdfffeL;
respflag = 1;
ret = WSEcmd (la, cmdExt, cmd, respflag, &response);
if ( ret < 0)
   /* An error occurred during command transfer. */;
```

WSgetTmo

Syntax: `ret = WSgetTmo(actualtimo)`

Action: Gets the actual time period to wait before aborting a Word Serial, Longword Serial, or Extended Longword Serial Protocol transfer.

Remarks: Input parameters:

none

Output parameter:

`actualtimo`	`INT32*`	Timeout period in milliseconds

Return value:

`ret`	`INT16`	0 = Successful

Example:

```
/* Get the timeout period. */

INT16      ret;
INT32      actualtimo;

ret = WSgetTmo(&actualtimo);
```

WSLcmd

Syntax: `ret = WSLcmd (la, cmd, respflag, response)`

Action: Sends a Longword Serial command or query to a Message-Based device.

Remarks: Input parameters:

`la`	`INT16`	Logical address of the Message-Based device
`cmd`	`UINT32`	Longword Serial command value
`respflag`	`UINT16`	Non-0 = Get a response (query) 0 = Do not get a response

Output parameter:

`response`	`UINT32*`	32-bit location to store response

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device did not recognize the command
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
2	TIMO_RES	Timed out before response received
1	TIMO_SEND	Timed out before able to send command
Successful Transfer (Bit 15 = 0)		
0	IODONE	Command transfer successfully completed

Example:

```
/* Send the Longword Serial command 0xfffcfffd to a device at
   Logical Address 5, and get the response. */

INT16      ret;
INT16      la;
UINT32     cmd;
UINT16     respflag;
UINT32     response;

la = 5;
cmd = 0xfffcfffdL;
respflag = 1;
ret = WSLcmd (la, cmd, respflag, &response);
if ( ret < 0)
   /* An error occurred during command transfer. */;
```

WSLresp

Syntax: `ret = WSLresp (la, response)`

Action: Retrieves a response to a previously sent Longword Serial Protocol query from a VXI Message-Based device. `WSLcmd` can send a query and automatically read a response. However, if it is necessary to break up the sending of the query and the reading of the response, you can use `WSLcmd` to send the query without reading the response and use `WSLresp` to read the response.

Note: ***This function is intended for debugging use only.***

Remarks: Input parameter:

`la`	`INT16`	Logical address of the Message-Based device

Output parameter:

`response`	`UINT32*`	32-bit location to store response

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device did not recognize the command
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
2	TIMO_RES	Timed out before response received
Successful Transfer (Bit 15 = 0)		
0	IODONE	Command transfer successfully completed

Example:

```
/* Retrieve a response for a previously sent Longword Serial query
   from Logical Address 5. */

INT16      ret;
INT16      la;
UINT32     response;

la = 5;
ret = WSLresp (la, &response);
if (ret < 0)
   /* An error occurred during response retrieval. */;
```

WSrd

Syntax: `ret = WSrd (la, buf, count, mode, retcount)`

Action: Transfers the specified number of data bytes from a Message-Based device into a specified local memory buffer, using the VXIbus Byte Transfer Protocol.

Remarks: Input parameters:

`la`	`INT16`	Logical address to read buffer from
`count`	`UINT32`	Maximum number of bytes to transfer
`mode`	`UINT16`	Transfer mode bit vector

Bit	Description
0	Not DOR 0 = Abort if not DOR 1 = Poll till DOR
1	END bit termination suppression 0 = Terminate transfer on END bit 1 = Do not terminate transfer on END
2	LF character termination 1 = Terminate transfer on LF bit 0 = Do not terminate transfer on LF
3	CR character termination 1 = Terminate transfer on CR bit 0 = Do not terminate transfer on CR
4	EOS character termination 1 = Terminate transfer on EOS bit 0 = Do not terminate transfer on EOS
8 to 15	EOS character (valid if EOS termination)

Output parameters:

`buf`	`UINT8*`	Read buffer
`retcount`	`UINT32*`	Number of bytes actually transferred

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device does not support the command
8	TIMO	Timeout
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
4	ForcedAbort	User abort occurred during I/O

Successful Transfer (Bit 15 = 0)

3	DirDorAbort	Transfer aborted–Device not DOR
2	TC	All bytes received
1	END	Any one of the termination received
0	IODONE	Successful transfer

Example:

```
/* Read up to 30 bytes from a device at Logical Address 5. Poll
   until device is DOR. Terminate transfer on END bit only. */

INT16      ret;
INT16      la;
UINT8      buf[100];
UINT32     count;
UINT16     mode;
UINT32     retcount;

la = 5;
count = 30L;
mode = 0x0001;   /* Poll until DOR, terminate transfer on END. */
ret = WSrd (la, buf, count, mode, &retcount);
if (ret < 0)
   /* An error occurred during the buffer read. */;
```

WSrdf

Syntax: `ret = WSrdf (la, filename, count, mode, retcount)`

Action: Reads the specified number of data bytes from a Message-Based device and writes them to the specified file, using the VXIbus Byte Transfer Protocol and standard file I/O.

Remarks: Input parameters:

`la`	`INT16`	Logical address to read buffer from
`filename`	`INT8*`	Name of the file to read data into
`count`	`UINT32`	Maximum number of bytes to transfer
`mode`	`UINT16`	Transfer mode bit vector

Bit	Description
0	Not DOR 0 = Abort if not DOR 1 = Poll till DOR
1	END bit termination suppression 0 = Terminate transfer on END bit 1 = Do not terminate transfer on END
2	LF character termination 1 = Terminate transfer on LF bit 0 = Do not terminate transfer on LF
3	CR character termination 1 = Terminate transfer on CR bit 0 = Do not terminate transfer on CR
4	EOS character termination 1 = Terminate transfer on EOS bit 0 = Do not terminate transfer on EOS
8 to 15	EOS character (valid if EOS termination)

Output parameter:

`retcount`	`UINT32*`	Number of bytes actually transferred

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device does not support the command
8	TIMO	Timeout
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
4	ForcedAbort	User abort occurred during I/O
1	FIOerr	Error reading or writing file
0	FOPENerr	Error opening file

Successful Transfer (Bit 15 = 0)

3	DirDorAbort	Transfer aborted–Device not DOR
2	TC	All bytes received
1	END	Any one of the termination received
0	IODONE	Successful transfer

Example:

```
/* Read 16 kilobytes (0x4000) from a device at Logical Address 5
   into a file called "rdfile.dat." Poll until device is DOR.
   Terminate the transfer on END bit or line feed (LF). */

INT16      ret;
INT8       *filename;
INT16      la;
UINT32     count;
UINT16     mode;
UINT32     retcount;

la = 5;
filename = "rdfile.dat";
count = 0x4000L;
mode = 0x0005;   /* Poll until DOR, terminate on END or LF. */
ret = WSrdf (la, filename, count, mode, &retcount);
if (ret < 0)
   /* An error occurred during the buffer read into the file. */
```

WSresp

Syntax: `ret = WSresp (la, response)`

Action: Retrieves a response to a previously sent Word Serial Protocol query from a VXI Message-Based device. `WScmd` can send a query and automatically read a response. However, if it is necessary to break up the sending of the query and the reading of the response, you can use `WScmd` to send the query without reading the response and use `WSresp` to read the response.

Note: ***This function is intended for debugging use only.***

Remarks: Input parameter:

`la`	`INT16`	Logical address of the Message-Based device

Output parameter:

`response`	`UINT16*`	16-bit location to store response

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device does not support the command
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
2	TIMO_RES	Timed out before response received
Successful Transfer (Bit 15 = 0)		
0	IODONE	Command transfer successfully completed

Example:

```
/* Send Read STB as a command and retrieve the response later. */

INT16      ret;
INT16      la;
UINT16     cmd;
UINT16     respflag;
UINT16     response;

la = 5;
cmd = 0xcfff;
respflag = 0;        /* Do NOT read response. */
ret = WScmd (la, cmd, respflag, &response);
if ( ret < 0)
   /* An error occurred during WS command transfer. */;
else (
   ret = WSresp (la, &response);
   if (ret < 0)
      /* An error occurred during response retrieval. */;
}
```

WSsetTmo

Syntax:

```
ret = WSsetTmo (timo, actualtimo)
```

Action: Sets the time period to wait before aborting a Word Serial, Longword Serial, or Extended Longword Serial Protocol transfer. It returns the actual timeout value set (the nearest timeout period possible greater than or equal to the timeout period specified).

Remarks: Input parameter:

`timo`	`INT32`	Timeout period in milliseconds

Output parameter:

`actualtimo`	`INT32*`	Actual timeout period set in milliseconds

Return value:

`ret`	`INT16`	0 = Successful

Example:

```
/* Set the timeout period to 2 seconds. */

INT16      ret;
INT32      timo;
INT32      actualtimo;

timeout = 2000L;
ret = WSsetTmo (timo, &actualtimo);
```

WStrg

Syntax: `ret = WStrg (la)`

Action: Sends the Word Serial *Trigger* command to a Message-Based device.

Remarks: Input parameter:

`la`	`INT16`	Logical address of the Message-Based device.

Output parameters:

none

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device did not recognize the command
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
4	ForcedAbort	User abort occurred during I/O
1	TIMO_SEND	Timed out before able to send command
Successful Transfer (Bit 15 = 0)		
0	IODONE	Command transfer successfully completed

Example:

```
/* Send Trigger command to Logical Address 5. */

INT16      ret;
INT16      la;

la = 5;
ret = WStrg (la);
if (ret < 0)
   /* An error occurred during the command transfer. */;
```

WSwrt

Syntax: `ret = WSwrt (la, buf, count, mode, retcount)`

Action: Transfers the specified number of data bytes from a specified local memory buffer to a Message-Based device, using the VXIbus Byte Transfer Protocol.

Remarks: Input parameters:

`la`	`INT16`	VXI logical address to write buffer to
`buf`	`UINT8*`	Write buffer
`count`	`UINT32`	Maximum number of bytes to transfer
`mode`	`UINT16`	Mode of transfer (bit vector)

Bit	Description
0	0 = Abort if device is not DIR 1 = Poll until device is DIR
1	1 = Set END bit on the last byte of transfer 0 = Clear END bit on the last byte of transfer

Output parameter:

`retcount`	`UINT32*`	Number of bytes actually transferred

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device does not support the command
8	TIMO	Timeout
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
4	ForcedAbort	User abort occurred during I/O
Successful Transfer (Bit 15 = 0)		
3	DirDorAbort	Transfer aborted–Device not DIR
2	TC	All bytes sent
1	END	The END bit was sent
0	IODONE	Successful transfer

Example:

```
/* Write the 14-byte ASCII command "VXI:CONF:NUMB?" to a device at
   Logical Address 5. Poll until device is DIR, and send END with
   the last byte. */

INT16      ret;
INT16      la;
UINT8      *buf;
UINT32     count;
UINT16     mode;
UINT32     retcount;

la = 5;
buf = "VXI:CONF:NUMB?";
count = strlen(buf);
mode = 0x0003;      /* Poll until DIR; send END with last byte. */
ret = WSwrt (la, buf, count, mode, &retcount);
if (ret < 0)
   /* An error occurred during the buffer write. */;
```

WSwrtf

Syntax: `ret = WSwrtf (la, filename, count, mode, retcount)`

Action: Transfers up to the specified number of data bytes from the specified file to a Message-Based device, using the VXIbus Byte Transfer Protocol and standard file I/O.

Remarks: Input parameters:

`la`	`INT16`	VXI logical address to write buffer to
`filename`	`INT8*`	Name of the file to write data from
`count`	`UINT32`	Maximum number of bytes to transfer
`mode`	`UINT16`	Mode of transfer (bit vector)

Bit	Description
0	0 = Abort if device is not DIR 1 = Poll until device is DIR
1	1 = Set END bit on the last byte of transfer 0 = Clear END bit on the last byte of transfer

Output parameter:

`retcount`	`UINT32*`	Number of bytes actually transferred

Return value:

`ret`	`INT16`	Return status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
10	RdProtErr	Read protocol error
9	UnSupCom	Device does not support the command
8	TIMO	Timeout
7	BERR	Bus error occurred during transfer
6	MQE	Multiple query error occurred during transfer
5	InvalidLA	Invalid `la` specified
4	ForcedAbort	User abort occurred during I/O
1	FIOerr	Error reading or writing file
0	FOPENerr	Error opening file
Successful Transfer (Bit 15 = 0)		
3	DirDorAbort	Transfer aborted–Device not DIR
2	TC	All bytes sent
1	END	The END bit was sent
0	IODONE	Successful transfer

Example:

```
/* Write 16 kilobytes (0x4000) to a device at Logical Address 5
   from the file, "wrtfile.dat." Poll until device is DIR, and
   send END with the last byte. */

INT16      ret;
INT8       *filename;
INT16      la;
UINT32     count;
UINT16     mode;
UINT32     retcount;

la = 5;
filename = "wrtfile.dat";
count = 0x4000L;
mode = 0x0003;       /* Send END, wait until DIR if not
                        already DIR. */
ret = WSwrtf (la, filename, count, mode, &retcount);
if (ret < 0)
   /* An error occurred during the buffer write. */;
```

Chapter 5
Servant Word Serial Protocol Functions

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

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

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

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

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

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

Programming Considerations

Most of the Servant Word Serial functions require an interrupt handler. The commands must be parsed (and responded to) within the appropriate interrupt handler. Word Serial commands *Byte Available* (BAV) and *Byte Request* (BREQ) are handled as a special case for reads and writes. For reads and writes, a user-supplied handler is notified only that the transfer is complete. Asserting and unasserting of all Response register bits (DIR, DOR, WR, RR, and ERR*) are done automatically within the functions as required. Figure 5-1 provides a graphical overview of the Servant Word Serial functions.

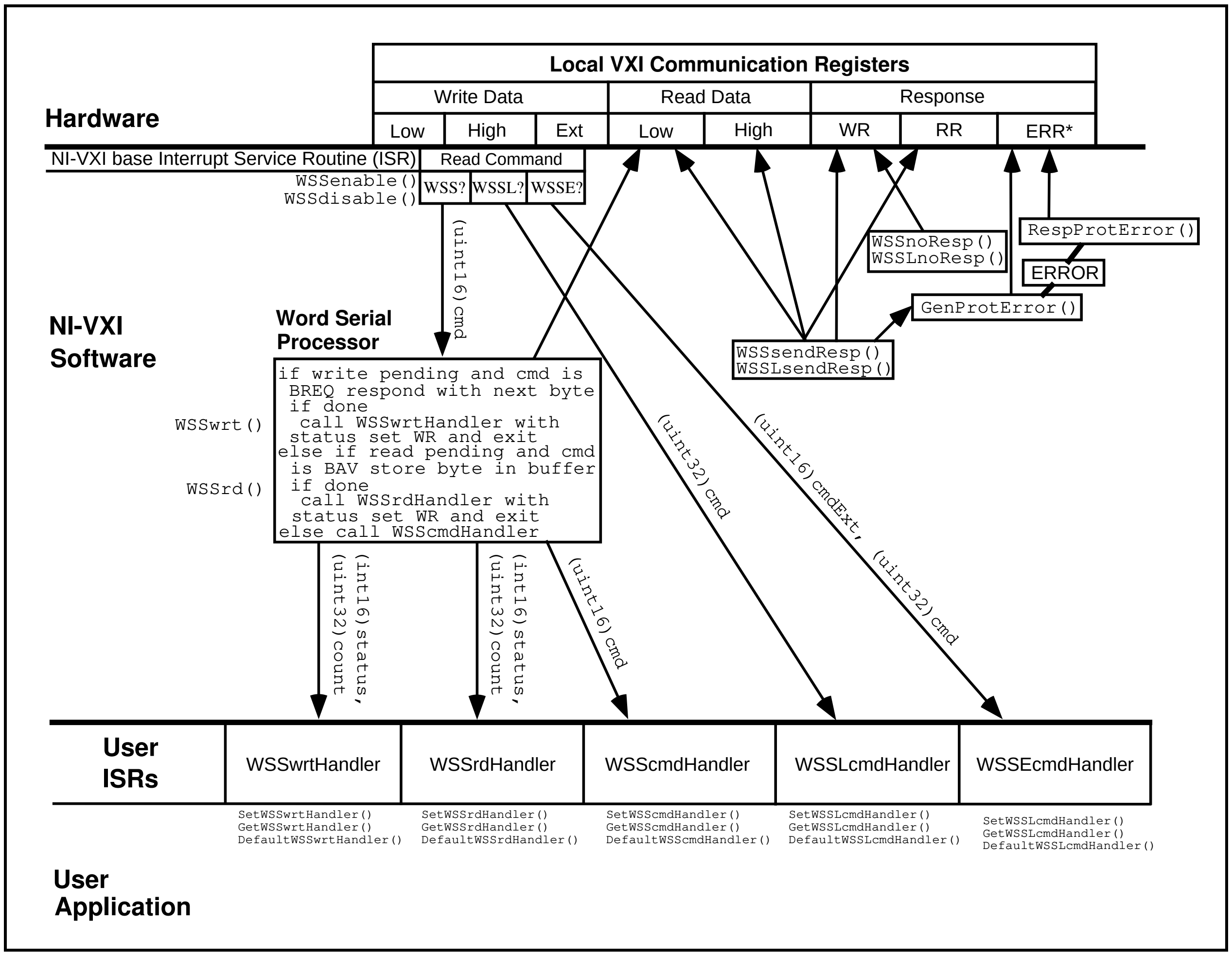

Figure 5-1. NI-VXI Servant Word Serial Model

Functional Overview

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

WSSenable ()

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

WSSdisable ()

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

WSSrd (buf, count, mode)

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

SetWSSrdHandler (func)

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

GetWSSrdHandler ()

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

DefaultWSSrdHandler (status, count)

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

WSSwrt (buf, count, mode)

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

SetWSSwrtHandler (func)

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

GetWSSwrtHandler ()

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

DefaultWSSwrtHandler (status, count)

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

SetWSScmdHandler (func)

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

GetWSScmdHandler ()

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

DefaultWSScmdHandler (cmd)

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

WSSsendResp (response)

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

WSSnoResp ()

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

SetWSSLcmdHandler (func)

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

GetWSSLcmdHandler ()

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

DefaultWSSLcmdHandler (cmd)

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

WSSLsendResp (response)

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

WSSLnoResp ()

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

SetWSSEcmdHandler (func)

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

GetWSSEcmdHandler ()

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

DefaultWSSEcmdHandler (cmdExt, cmd)

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

WSSabort (abortop)

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

GenProtError (proterr)

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

RespProtError ()

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

Function Descriptions

The following paragraphs describe the Servant Word Serial, Longword Serial, and Extended Longword Serial Protocol functions. The descriptions are explained at the C syntax level and are listed in alphabetical order.

GenProtError

Syntax: `ret = GenProtError (proterr)`

Action: Generates a Word Serial protocol error if one is not already pending. It asserts the Response register bit ERR* if the value of the protocol error, `proterr`, is not -1. If `proterr` is -1, it deasserts the ERR* bit. If no previous error existed, it saves the `proterr` value for response to a future *Read Protocol Error* query via the function `RespProtError`. If a previous error does exist, the ERR* bit remains asserted but the protocol error specified by `proterr` is ignored.

Remarks: Input parameter:

`proterr`	`UINT16`	Protocol error to generate

Value	Protocol Error Description
ffffh	Clear any protocol error condition
fffdh	Multiple Query Error (MQE)
fffch	Unsupported Command (UnSupCom)
fffbh	Data In Ready violation (DIRviol)
fffah	Data Out Ready violation (DORviol)
fff9h	Read Ready violation (RRviol)
fff8h	Write Ready violation (WRviol)
others	Reserved

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 1 = Successful, but error will be ignored because a previous error is pending 0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Generate a protocol error of DORviol. */

INT16      ret;
UINT16     proterr;

proterr = 0xfffa;
ret = GenProtError (proterr);
if (ret < 0)
   /* An error occurred in GenProtError. */;
```

GetWSScmdHandler

Syntax: `func = GetWSScmdHandler()`

Action: Returns the address of the current `WSScmd` interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`NIVXI_HWSSCMD*`	Pointer to the current `WSScmd` interrupt handler

Example:

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

NIVXI_HWSSCMD     *func;

func = GetWSScmdHandler();
```

GetWSSEcmdHandler

Syntax: `func = GetWSSEcmdHandler()`

Action: Returns the address of the current `WSSEcmd` interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`NIVXI_HWSSECMD*`	Pointer to the current `WSSEcmd` interrupt handler

Example:

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

NIVXI_HWSSECMD    *func;

func = GetWSSEcmdHandler();
```

GetWSSLcmdHandler

Syntax: `func = GetWSSLcmdHandler()`

Action: Returns the address of the current `WSSLcmd` interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`NIVXI_HWSSLCMD*`	Pointer to the current `WSSLcmd` interrupt handler

Example:

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

NIVXI_HWSSLCMD    *func;

func = GetWSSLcmdHandler();
```

GetWSSrdHandler

Syntax: `func = GetWSSrdHandler()`

Action: Returns the address of the current `WSSrd` done notification interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`NIVXI_HWSSRD*`	Pointer to the current `WSSrd` done notification interrupt handler

Example:

```
/* Get the address of the WSSrd done notification handler. */

NIVXI_HWSSRD *func;

func = GetWSSrdHandler();
```

GetWSSwrtHandler

Syntax: `func = GetWSSwrtHandler()`

Action: Returns the address of the current `WSSwrt` done notification interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`NIVXI_HWSSWRT*`	Pointer to the current `WSSwrt` done notification interrupt handler

Example:

```
/* Get the address of the WSSwrt done notification handler. */

NIVXI_HWSSWRT      *func;

func = GetWSSwrtHandler();
```

RespProtError

Syntax: `ret = RespProtError ()`

Action: Responds to the Word Serial *Read Protocol Error* query with the last protocol error generated via the `GenProtError` function, and then unasserts the ERR* bit.

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported -2 = Response is still pending and a multiple query error is generated

Example:

```
/* Respond to the Word Serial Read Protocol Error query. */

INT16      ret;

ret = RespProtError ();
if (ret < 0)
   /* An error occurred in RespProtError. */;
```

SetWSScmdHandler

Syntax: `ret = SetWSScmdHandler (func)`

Action: Replaces the current `WSScmd` interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`NIVXI_HWSSCMD*`	Pointer to the new `WSScmd` interrupt handler (NULL = `DefaultWSScmdHandler`)

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Set the WSScmd interrupt handler. */

NIVXI_HWSSCMD      func;
INT16    ret;

ret = SetWSScmdHandler(func);
if (ret < 0)
   /* An error occurred in SetWSScmdHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (UINT16 cmd)
{
}
```

SetWSSEcmdHandler

Syntax: `ret = SetWSSEcmdHandler (func)`

Action: Replaces the current `WSSEcmd` interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`NIVXI_HWSSECMD*`	Pointer to the new `WSSEcmd` interrupt handler (`NULL` = `DefaultWSSEcmdHandler`)

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Set the WSSEcmd interrupt handler. */

NIVXI_HWSSECMD    func;
INT16    ret;

ret = SetWSSEcmdHandler(func);
if (ret < 0)
   /* An error occurred in SetWSSEcmdHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (UINT16 cmdExt, UINT32 cmd)
{
}
```

SetWSSLcmdHandler

Syntax: `ret = SetWSSLcmdHandler (func)`

Action: Replaces the current `WSSLcmd` interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`NIVXI_HWSSLCMD*`	Pointer to the new `WSSLcmd` interrupt handler (`NULL` = `DefaultWSSLcmdHandler`)

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Set the WSSLcmd interrupt handler. */

NIVXI_HWSSLCMD    func;
INT16    ret;

ret = SetWSSLcmdHandler(func);
if (ret < 0)
   /* An error occurred in SetWSSLcmdHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (UINT32 cmd)
{
}
```

SetWSSrdHandler

Syntax: `ret = SetWSSrdHandler (func)`

Action: Replaces the current `WSSrd` done notification interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`NIVXI_HWSSRD*`	Pointer to the new `WSSrd` done notification handler (`NULL` = `DefaultWSSrdHandler`)

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Set the WSSrd done notification interrupt handler. */

NIVXI_HWSSRD       func;
INT16    ret;

ret = SetWSSrdHandler(func);
if (ret < 0)
   /* An error occurred in SetWSSrdHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (INT16 status, UINT32 count)
{
}
```

SetWSSwrtHandler

Syntax: `ret = SetWSSwrtHandler (func)`

Action: Replaces the current `WSSwrt` done notification interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`NIVXI_HWSSWRT*`	Pointer to the new `WSSwrt` done notification handler (NULL = `DefaultWSSwrtHandler`)

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Set the WSSwrt done notification interrupt handler. */

NIVXI_HWSSWRT      func;
INT16     ret;

ret = SetWSSwrtHandler(func);
if (ret < 0)
   /* An error occurred in SetWSSwrtHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (INT16 status, UINT32 count)
{
}
```

WSSabort

Syntax: `ret = WSSabort (abortop)`

Action: Aborts the Servant Word Serial operation(s) in progress.

Remarks: Input parameter:

`abortop`	`UINT16`	The operation to abort, bit vector

Bit	Description
0	Abort `WSSwrt`
1	Abort `WSSrd`
2	Abort `WSSsendResp`
15	Initialize Word Serial Servant hardware. This includes aborting all Word Serial operations, clearing out errors, removing all pending Word Serial Servant interrupts, and disabling the interrupts.

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successfully aborted
-1 = Servant Word Serial functions not supported
-2 = Unable to abort

Example:

```
/* Abort WSSwrt. */

INT16      ret;
UINT16     abortop;

abortop = (1<<0);
ret = WSSabort (abortop);
if (ret < 0)
   /* An error occurred during WSSabort. */;
```

WSSdisable

Syntax: `ret = WSSdisable ()`

Action: Desensitizes the local CPU to interrupts generated when a Word Serial command is written to the Data Low register or when a response is read from the Data Low register.

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status
		0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Disable all the Servant Word Serial functions. */

INT16      ret;

ret = WSSdisable();
if (ret < 0)
   /* An error occurred during WSSdisable. */;
```

WSSenable

Syntax: `ret = WSSenable ()`

Action: Sensitizes the local CPU to interrupts generated when a Word Serial command is written to the Data Low register or when a response is read from the Data Low register.

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Enable all the Servant Word Serial functions. */

INT16      ret;

ret = WSSenable();
if (ret < 0)
   /* An error occurred during WSSenable. */;
```

WSSLnoResp

Syntax: `ret = WSSLnoResp ()`

Action: Acknowledges a received Longword Serial Protocol command that has no response and asserts the Write Ready (WR) bit in the local CPU Response register. This function must be called after the processing of a Longword Serial Protocol command (queries are responded to with `WSSLsendResp`).

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status
		0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Acknowledge the reception of a Longword Serial Protocol command
   that has no response. */

INT16      ret;

ret = WSSLnoResp ();
if (ret < 0)
   /* An error occurred during WSSLnoResp. */;
```

WSSLsendResp

Syntax: `ret = WSSLsendResp (response)`

Action: Responds to a received Longword Serial Protocol query with a response and asserts the WR bit (in addition to the RR bit) in the local CPU Response register. This function must be called after processing a Longword Serial Protocol query (commands are acknowledged with `WSSLnoResp`).

Remarks: Input parameter:

`response`	`UINT32`	32-bit response

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported -2 = Response still pending (MQE generated)

Example:

```
/* Respond to a received Longword Serial Protocol query. */

INT16      ret;
UINT32     response;

response = 0xfffcfffdL;
ret = WSSLsendResp (response);
if (ret < 0)
   /* An error occurred during WSSLsendResp. */;
```

WSSnoResp

Syntax: `ret = WSSnoResp ()`

Action: Acknowledges a received Word Serial Protocol command that has no response and asserts the WR bit in the local CPU Response register. This function must be called after the processing of a Word Serial Protocol command (queries are responded to with `WSSsendResp`).

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

Example:

```
/* Acknowledge the reception of a Word Serial Protocol command
   that has no response. */

INT16      ret;

ret = WSSnoResp ();
if (ret < 0)
   /* An error occurred during WSSnoResp. */;
```

WSSrd

Syntax: `ret = WSSrd (buf, count, mode)`

Action: Posts a read operation to begin receiving the specified number of data bytes from a Message-Based Commander into a specified memory buffer, using the VXIbus Byte Transfer Protocol.

Remarks: Input parameters:

`count`	`UINT32`	Maximum number of bytes to transfer
`mode`	`UINT16`	Transfer mode bit vector
	Bit	Description
	0	DIR signal mode to Commander 0 = Do not send DIR signal to Commander 1 = Send DIR signal to Commander
	15 to 1	Reserved (0)

Output parameter:

`buf`	`UINT8*`	Read buffer

Return value:

`ret`	`INT16`	Return Status

1 = Posted successfully; will begin after a `WSSenable()`
0 = Posted successfully
-1 = Servant Word Serial functions not supported
-2 = `WSSrd` already in progress

Example:

```
/* Read 10 bytes from the Commander. */

INT16      ret;
UINT8      buf[100];
UINT32     count;
UINT16     mode;

count = 10L;
mode = 0x0000;       /* Do not send DIR signal to Commander. */
ret = WSSrd (buf, count, mode);
if (ret < 0)
   /* An error occurred during WSSrd. */;
```

WSSsendResp

Syntax: `ret = WSSsendResp (response)`

Action: Responds to a received Word Serial Protocol query with a response and asserts the WR bit (in addition to the RR bit) in the local CPU Response register. This function must be called after processing a Word Serial Protocol query (commands are acknowledged with `WSSnoResp`).

Remarks: Input parameter:

`response`	`UINT16`	16-bit response

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Servant Word Serial functions not supported
-2 = Response still pending (MQE generated)

Example:

```
/*  Respond with 0x1234 to a received Word Serial Protocol
    query. */

INT16      ret;
UINT16     response;

response = 0x1234L;
ret = WSSsendResp (response);
if (ret < 0)
   /* An error occurred during WSSsendResp. */;
```

WSSwrt

Syntax: `ret = WSSwrt (buf, count, mode)`

Action: Posts the write operation to transfer the specified number of data bytes from a specified memory buffer to the Message-Based Commander, using the VXIbus Byte Transfer Protocol.

Remarks: Input parameters:

`buf`	`UINT8*`	Write buffer
`count`	`UINT32`	Maximum number of bytes to transfer
`mode`	`UINT16`	Mode of transfer (bit vector)

Bit	Description
0	DOR signal mode to Commander (if enabled) 0 = Do not send DOR signal to Commander 1 = Send DOR signal to Commander
1	END bit termination with last byte 0 = Do not send END with the last byte 1 = Send END with the last byte

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

1 = Posted successfully; will begin after a `WSSenable()`
0 = Posted successfully
-1 = Servant Word Serial functions not supported
-2 = `WSSwrt` already in progress

Example:

```
/*  Write 6 bytes to the Commander. */

INT16      ret;
UINT8      *buf;
UINT32     count;
UINT16     mode;

buf = "1.0422";
count = 6L;
mode = 0x0002;        /* Send END with the last byte. */
ret = WSSwrt (buf, count, mode);
if (ret < 0)
   /* An error occurred during WSSwrt. */;
```

Default Handlers for the Servant Word Serial Functions

The NI-VXI software provides the following default handlers for the Servant Word Serial 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 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.

- `DefaultWSScmdHandler`
- `DefaultWSSEcmdHandler`
- `DefaultWSSLcmdHandler`
- `DefaultWSSrdHandler`
- `DefaultWSSwrtHandler`

DefaultWSScmdHandler

Syntax: `DefaultWSScmdHandler (cmd)`

Action: Handles any Word Serial Protocol command or query received from a VXI Message-Based Commander. Uses global variables to handle many of the Word Serial commands. Implements all commands required for Servant operation.

Remarks: Input parameter:

`cmd`	`UINT16`	16-bit Word Serial command received

Output parameters:

none

Return value:

none

DefaultWSSEcmdHandler

Syntax: `DefaultWSSEcmdHandler (cmdExt, cmd)`

Action: Handles Extended Longword Serial Protocol commands or queries received from a VXI Message-Based Commander. Returns an Unsupported Command protocol error for all commands and queries because the VXI specification does not define any Extended Longword Serial commands.

Remarks: Input parameters:

`cmdExt`	`UINT16`	Upper 16 bits of 48-bit Extended Longword Serial command received
`cmd`	`UINT32`	Lower 32 bits of 48-bit Extended Longword Serial command received

Output parameters:

none

Return value:

none

DefaultWSSLcmdHandler

Syntax: `DefaultWSSLcmdHandler (cmd)`

Action: Handles Longword Serial Protocol commands or queries received from a VXI Message-Based Commander. Returns an Unsupported Command protocol error for all commands and queries because the VXI specification does not define any Longword Serial commands.

Remarks: Input parameter:

`cmd`	`UINT32`	32-bit Longword Serial command received

Output parameters:

none

Return value:

none

DefaultWSSrdHandler

Syntax: `DefaultWSSrdHandler (status, count)`

Action: Handles the termination of a Servant Word Serial read operation started with `WSSrd`. Sets the global variable `WSSrdDone` to 1, the `WSSrdDoneStatus` variable to `status`, and the `WSSrdDoneCount` global variable to `count`.

Remarks: Input parameters:

`status`	`INT16`	Status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
4	ForcedAbort	`WSSabort` called to force abort
Successful Transfer (Bit 15 = 0)		
2	TC	All bytes received
1	END	END received with last byte
0	IODONE	Transfer successfully completed

`count`	`UINT32`	Actual number of bytes received

Output parameters:

none

Return value:

none

DefaultWSSwrtHandler

Syntax: `DefaultWSSwrtHandler (status, count)`

Action: Handles the termination of a Servant Word Serial write operation started with `WSSwrt`. Sets the global variable `WSSwrtDone` to 1, the `WSSwrtDoneStatus` variable to `status`, and the `WSSwrtDoneCount` variable to `count`.

Remarks: Input parameters:

`status`	`INT16`	Status bit vector

The following table gives the meaning of each bit that is set to one (1).

Bit	Name	Description
Error Conditions (Bit 15 = 1)		
14	WRviol	Write Ready protocol violation during transfer
13	RRviol	Read Ready protocol violation during transfer
12	DORviol	Data Out Ready protocol violation
11	DIRviol	Data In Ready protocol violation
4	ForcedAbort	`WSSabort` called to force abort
Successful Transfer (Bit 15 = 0)		
2	TC	All bytes sent
1	END	END sent with last byte
0	IODONE	Transfer successfully completed

`count`	`UINT32`	Actual number of bytes sent

Output parameters:

none

Return value:

none

Chapter 6
Low-Level VXIbus Access Functions

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

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

Low-level and high-level access to the VXIbus, 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 VXIbus 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 VXIbus access is the fastest access method (in terms of overall throughput to the device) for directly reading or writing to/from any of the VXIbus 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 VXIbus address with a particular VXI access privilege and byte ordering. You need to consider a number of issues when using the direct pointers:

- You need to determine bounds for the pointers.
- Based on the methods in which a particular hardware platform sets up access to VXI 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 VXIbus access functions need not take into account any of the considerations that are required by the low-level VXIbus access functions. The high-level VXIbus access functions have all necessary information for accessing a particular VXIbus 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 VXIbus access functions automatically trap bus errors and return an appropriate error status. Using the high-level VXIbus 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 VXIbus access functions act as an easy-to-use interface that can do any VXIbus accesses necessary for an application. For more information, refer to Chapter 7, *High-Level VXIbus Access Functions*.

Programming Considerations

All accesses to the VXIbus 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 VXIbus (using a complex hardware interface). The areas where the address space of the local CPU is mapped onto the VXIbus 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 VXIbus 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 VXIbus 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: ***It is strongly recommended that all your devices have the same access privileges and byte orders. The VXIbus specification, for example, 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 VXIbus address spaces.***

NI-VXI uses a term within this chapter called the hardware (or window) *context*. The hardware context for window to VXI consists of the VXI address space being accessed, the base offset into the address space, the access privilege, and the byte order for the accesses through the window. Before accessing a particular address, you must set up the window with the appropriate hardware context. You can use the `MapVXIAddress` function for this purpose. This function returns a pointer that you can use for subsequent accesses to the window with the `VXIpeek` and `VXIpoke` functions. On most systems, `VXIpeek` and `VXIpoke` are really C macros (`#defines`) that 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 `windowId` output parameter returned from the `MapVXIAddress` call associates the C pointer returned from the function with a particular window and also signifies Owner Privilege to that window. Owner Privilege access is complete and exclusive. The caller can use `SetPrivilege`, `SetByteOrder`, and `SetContext` with this `windowId` to dynamically change the access privileges. Notice that if the call to `MapVXIAddress` succeeds for either Owner Privilege or Access Only Privilege, the pointer remains valid in both cases until an explicit `UnMapVXIAddress` call is made for the corresponding window. The pointer is guaranteed to be a valid pointer in either multitasking systems or nonmultitasking systems. The advantage with Owner Privilege is that it gives complete and exclusive access for that window to the caller, so you can dynamically change the access privileges. Because no other callers can succeed, there is no problem with either destroying another caller's access state or having an inconsistent pointer environment.

Access Only Privilege

A process can obtain Access Only Privilege by requesting access only privileges in the `MapVXIAddress` call. With this privilege mode, you can have multiple pointers in the same process or over multiple processes to access a particular window simultaneously, while still guaranteeing that the hardware context does not change between accesses. The call succeeds under either of the following conditions:

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 `windowId` 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 `windowId` for Access Only Privilege.

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

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

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, `SaveContext` and `RestoreContext` can be used safely in interrupt service routines, but not outside interrupt service routines.

Functional Overview

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

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

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

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

Note: ***Take into account that the Resource Manager assigns all VXI devices VXI addresses based on a power of two, and that all windows are 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` reads a single byte, word, or longword from a particular address obtained by `MapVXIAddress`. On most embedded CPU systems using C language interfaces, `VXIpeek` is simply a macro to de-reference a C pointer. It is recommended, 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` writes a single byte, word, or longword to a particular address obtained by `MapVXIAddress`. On most embedded CPU systems using C language interfaces, `VXIpoke` is simply a macro to de-reference a C pointer. It is recommended, 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 VXI 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 VXI 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). Use `SaveContext` and `RestoreContext` only in interrupt service routines or code segments that are not pre-emptible. Otherwise you risk inconsistent and indeterminate results.

RestoreContext (contextlist)

`RestoreContext` restores the hardware interface settings (context) for all VXI 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 VXI window. Use `SaveContext` and `RestoreContext` only in interrupt service routines or code segments that are not pre-emptible. Otherwise you risk inconsistent and indeterminate results.

SetContext (window, context)

`SetContext` sets all of the hardware interface settings (context) for a particular VXI window. The application must have Owner Access Privilege to the applicable window for this function to execute successfully. Any application can use `GetContext` along with `SetContext` to save and restore the VXI interface hardware state (context) for a particular window. As a result, the application can set the hardware context associated with a particular pointer into VXI 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 VXI window. The application can have either Owner Access Privilege or Access Only Privilege to the applicable window for this function to execute successfully. Any application can use `GetContext` along with `SetContext` to save and restore the VXI interface hardware state (context) for a particular window.

SetPrivilege (window, priv)

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

GetPrivilege (window, priv)

`GetPrivilege` retrieves the current windowing hardware VXIbus 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 VXIbus 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 VXIbus.

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

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

GetVXIbusStatusInd (controller, field, status)

`GetVXIbusStatusInd` retrieves information about the current state of the VXIbus.

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

The information that can be returned includes the state of the Sysfail, ACfail, VXI interrupt, TTL trigger, or ECL trigger lines as well as the number of VXI signals on the global signal queue. The specified information returns in a single integer value. The `field` parameter specifies the particular VXIbus 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 VXIbus 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 1 = Byte order returned successfully; same for all 0 = Successful -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 VXI 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 VXI/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)
   /* An error occurred in GetPrivilege. */;
```

GetVXIbusStatus

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

Action: Gets information about the state of the VXIbus 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`	`BusStat`	Structure containing VXIbus status

Structure is as follows:

```
typedef struct BusStat {
   INT16  BusError;  /* 1 = Last access BERRed          */
   INT16  Sysfail;   /* 1 = SYSFAIL* asserted           */
   INT16  ACfail;    /* 1 = ACFAIL* asserted            */
   INT16  SignalIn;  /* Number of signals queued        */
   INT16  VXIints;   /* Bit vector 1 = interrupt asserted */
   INT16  ECLtrigs;  /* Bit vector 1 = trigger asserted */
   INT16  TTLtrigs;  /* Bit vector 1 = trigger asserted */
   } BusStat;
```

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

Return value:

`ret`	`INT16`	Return Status

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

Example:

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

INT16      ret;
INT16      controller;
BusStat    status;

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

GetVXIbusStatusInd

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

Action: Gets information about the state of the VXIbus 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          */
5   VXIints;    /* Bit vector 1 = interrupt asserted */
6   ECLtrigs;   /* Bit vector 1 = trigger asserted   */
7   TTLtrigs;   /* Bit vector 1 = trigger asserted   */
```

Output parameter:

`status`	`INT16*`	VXIbus 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 VXIbus 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)
   /* An error occurred 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 VXIbus address space.

Remarks: Input parameter:

`window`	`UINT32`	Window number obtained from `MapVXIAddress`

Output parameters:

`windowbase`	`UINT32*`	Base VXI Address
`windowend`	`UINT32*`	End VXI 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 VXI address spaces according to the access parameters specified, and returns a pointer to a local CPU address that accesses the specified VXI address. This function also returns the window ID associated with the window, which is used with all other low-level VXIbus access functions.

Remarks: Input parameters:

`accessparms`	`UINT16`	(Bits 0 to 1) VXI 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 VXI 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 (base of Logical Address 4's VXI registers) 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 VXI 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 VXI windows. The `contextlist` parameter will be filled with a list of the contexts for all of the VXI windows. This function is recommended for use only within interrupt service routines to guarantee access to a particular VXI 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 1 = Successful; byte order set for all windows 0 = Successful; byte order set for specific window only -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 <0)
    /* An error occurred in SetByteOrder. */;
```

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`	VXI 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
-10 = Base address change is not supported

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);
if (ret <0)
    /* An error occurred in SetContext. */;
```

SetPrivilege

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

Action: Sets the VXI/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)
   /* An 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 VXI 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 the Offset register of the device at
   Logical Address 4. */

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 VXI 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 the Offset register of the device at
   Logical Address 4. */

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 7
High-Level VXIbus Access Functions

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

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

Low-level and high-level access to the VXIbus, 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 VXIbus 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 VXIbus access is the fastest access method (in terms of overall throughput to the device) for directly reading or writing to/from any of the VXIbus 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 VXIbus address with a particular VXI 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 VXI 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 VXIbus Access Functions*.

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

Programming Considerations for High-Level VXIbus Access Functions

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

Functional Overview

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

VXIin (accessparms, address, width, value)

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

Remarks: Input parameters:

`accessparms`	`UINT16`	(Bits 0, 1) VXI 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`	VXI 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 ID register of the device at Logical Address 4. */

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)
   /* An 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;
INT16      la;
UINT16     reg;
UINT16     value;

la = 4;
reg = 0;
ret = VXIinReg (la, reg, &value);
if (ret < 0)
   /* An 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 VXI 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 VXI 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 = 0x200L;         /* 512 elements. */
width = 2;               /* Transfer as words. */
destaddr = (UINT32)malloc(length * width);/* Allocate local
                                                       buffer. */
ret = VXImove (srcparms, srcaddr, destparms, destaddr, length,
width);
if (ret < 0)
   /* An error occurred during VXImove. */;
```

VXIout

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

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

Remarks: Input parameters:

`accessparms`	`UINT16`	(Bits 0, 1) VXI 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`	VXI 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 value 0x2000 to the Offset register of the device at
   Logical Address 4. */

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)
   /* An 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)
   /* An error occurred during write. */;
```

Chapter 8
Local Resource Access Functions

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

Access to the local CPU logical address is required for sending correct VXI signal values to other devices. Reading local VXI registers is required for retrieving configuration information. Writing to the A24 and A32 pointer registers is required for use under the Shared Memory Protocol of the VXIbus specification, Revision 1.2. Exercising the local CPU MODID capabilities (if the local CPU is a VXI Slot 0 device) can be helpful for debugging a prototype VXI device's slot association (MODID) capability.

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. This function can be used to write application specific registers (A24 pointer register, A32 pointer register, and so on) for the local CPU.

SetMODID (enable, modid)

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

ReadMODID (modid)

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

VXImemAlloc (size, useraddr, vxiaddr)

`VXImemAlloc` allocates physical RAM from the operating system's dynamic memory pool. This RAM will reside in the VXI Shared RAM region of the local CPU. `VXImemAlloc` returns not only the user address that the application uses, but also the VXI address that a remote device would use to access this RAM. 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 should make a call to `VXImemFree` to return the memory to the operating system's dynamic memory pool.

VXImemCopy (useraddr, bufaddr, size, dir)

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

VXImemFree (useraddr)

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

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();
```

ReadMODID

Syntax: `ret = ReadMODID (modid)`

Action: Senses the MODID lines of the VXIbus backplane. This function only applies to the local device, which must be a Slot 0 device.

Remarks: Input parameters:

none

Output parameter:

`modid`	`UINT16*`	Bit vector as follows:

Bits	Description
12-0	MODID lines 12 to 0, respectively
13	MODID enable bit

Return value:

`ret`	`INT16`	Return Status
		0 = Successfully read MODID lines
		-1 = Not a Slot 0 device

Example:

```
/*  Read all the MODID lines 0 to 12. */

INT16      ret;
UINT16     modid;

ret = ReadMODID (&modid);
if (ret < 0)
   /* An error occurred in ReadMODID. */;
```

SetMODID

Syntax: `ret = SetMODID (enable, modid)`

Action: Controls the assertion of the MODID lines of the VXIbus backplane. This function only applies to the local device, which must be a Slot 0 device.

Remarks: Input parameters:

`enable`	`UINT16`	1 = Set MODID enable bit 0 = Clear MODID enable bit
`modid`	`UINT16`	Bit vector for Bits 0 to 12, corresponding to Slots 0 to 12

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successfully set MODID lines -1 = Not a Slot 0 device

Example:

```
/*  Set all the MODID lines 0 to 12. */

INT16      ret;
UINT16     enable;
UINT16     modid;

enable = 1;
modid = 0x1fff;    /* Bit vector (Bits 0 to 12). */

ret = SetMODID (enable, modid);
if (ret < 0)
   /* An error occurred in SetMODID. */;
```

VXIinLR

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

Action: Reads a single byte, word, or longword from a particular VXI register on the local VXI 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)
   /* An error occurred in VXIinLR. */;
```

VXImemAlloc

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

Action: Allocates dynamic system RAM from the VXI Shared RAM area of the local CPU and returns both the local and remote VXI addresses. The VXI 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 VXI 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 VXI Shared system
    RAM. */

UINT32     size;
void       *useraddr;
UINT32     vxiaddr;
INT16      ret;

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

/*
   Use buffer.
*/

ret = VXImemFree (useraddr);
if (ret < 0)
   /* An error occurred 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)
   /* An error occurred 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)
   /* An error occurred in VXImemFree. */;
```

VXImemFree

Syntax: `ret = VXImemFree (useraddr)`

Action: Deallocates dynamic system RAM from the VXI 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 VXI Shared system
    RAM. */

UINT32     size;
void       *useraddr;
UINT32     vxiaddr;
INT16      ret;

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

/*
   Use buffer.
*/

ret = VXImemFree (useraddr);
if (ret < 0)
   /* An error occurred 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 VXI 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`	`UINT32`	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;
UINT32     value;

reg = 8;              /* Register offset for Signal register. */
width = 2;            /* Word register. */
value = 0xfd00L;
ret = VXIoutLR (reg, width, value);
if (ret < 0)
   /* An error occurred in VXIoutLR. */;
```

Chapter 9
VXI Signal Functions

This chapter describes the C syntax and use of the VXI signal functions and default handler. With these functions, VXI bus master devices can interrupt another device. VXI signal functions can specify the signal routing, manipulate the global signal queue, and wait for a particular signal value (or set of values) to be received.

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

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

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

In another method for handling signals (and VXI interrupts routed to signals) other than the two previous methods, you can use the function `WaitForSignal`. This function can suspend a process/function until a particular signal (or one of a set of signals) arrives. A multitasking operating system lets you have any number of `WaitForSignal` calls pending, even for the same logical address. A non-multitasking operating system permits only one pending `WaitForSignal` call.

Programming Considerations for Signal Queuing

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

In addition to `DisableSignalInt`, the `DisableVXItoSignalInt` function is also called to disable VXI interrupts from occurring on levels that are routed to the signal Processor. When `SignalDeq` is called to remove a signal from the global signal queue, the interrupts for the Signal register and the VXI interrupt levels routed to the signal handler are automatically re-enabled. If signals received never get dequeued, the global signal queue eventually becomes full and the interrupts will be disabled forever. If the signals were routed to the `DefaultSignalHandler`, all except *Unrecognized Command* Events from Message-Based devices perform no operation. *Unrecognized Command* Events must call the function `WSabort` to abort the current Word Serial operation in progress.

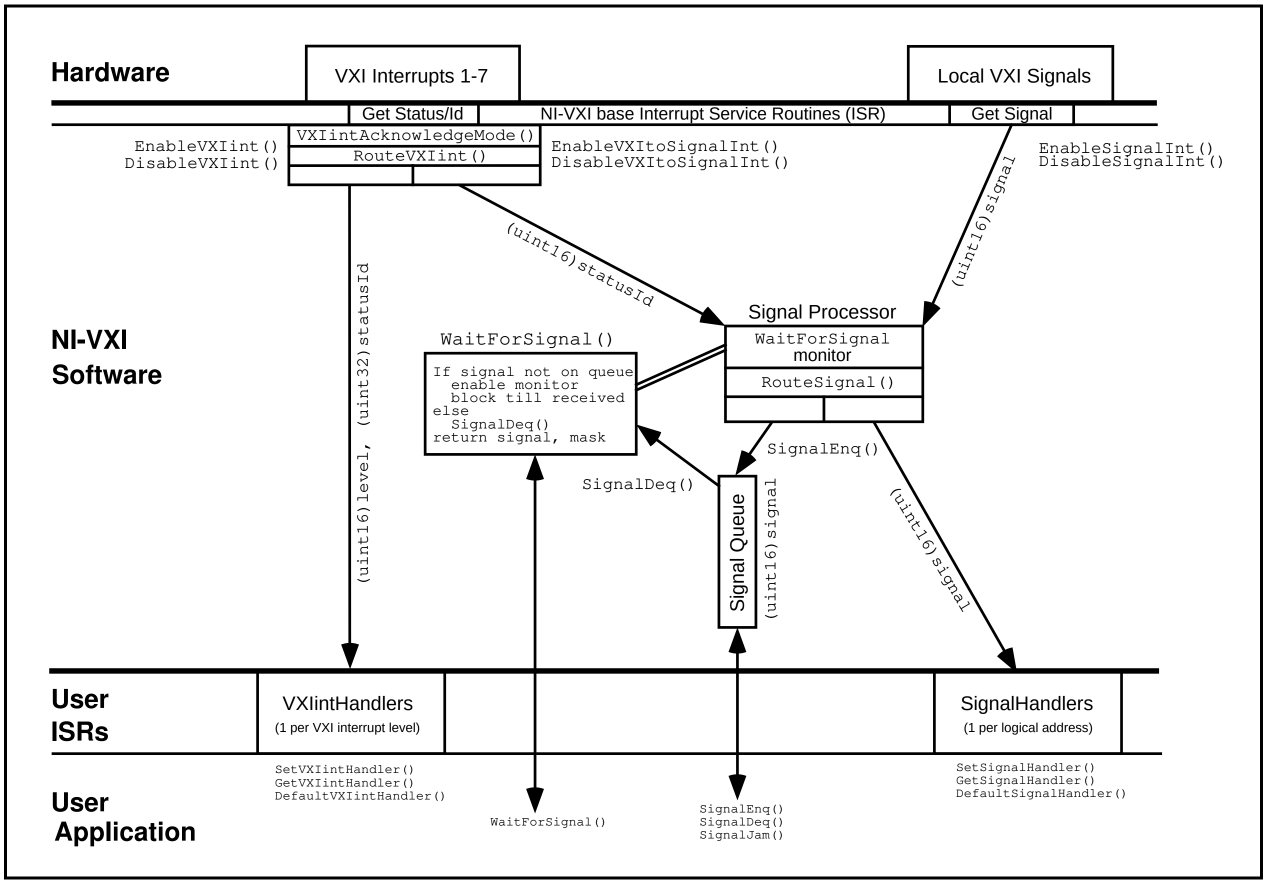

Figure 9-1. NI-VXI VXI Interrupt and Signal Model

WaitForSignal Considerations

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

Functional Overview

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

RouteSignal (la, modemask)

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

EnableSignalInt ()

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

DisableSignalInt ()

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

SetSignalHandler (la, func)

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

GetSignalHandler (la)

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

DefaultSignalHandler (signal)

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

SignalDeq (la, signalmask, signal)

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

SignalEnq (signal)

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

SignalJam (signal)

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

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

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

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

Function Descriptions

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

DisableSignalInt

Syntax: `ret = DisableSignalInt ()`

Action: Desensitizes the local CPU to interrupts generated by writes to the local VXI Signal register. While disabled, no VXI signals are processed. If the local VXI hardware Signal register is implemented as a FIFO, signals are held in the FIFO until the signal interrupt is enabled via the `EnableSignalInt` function. When the FIFO is full, the remote VXI device will get a Bus Error in response to a write to the Signal register.

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status
		0 = Signal interrupts successfully disabled

Example:

```
/* Disable the signal interrupt. */

INT16      ret;

ret = DisableSignalInt ();
```

EnableSignalInt

Syntax: `ret = EnableSignalInt ()`

Action: Sensitizes the local CPU to interrupts generated by writes to the local VXI Signal register.

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status 1 = Signal queue full; will enable after dequeuing a signal 0 = Signal interrupts successfully enabled

Example:

```
/* Enable the signal interrupt. */

INT16      ret;

ret = EnableSignalInt ();
```

GetSignalHandler

Syntax: `func = GetSignalHandler (la)`

Action: Returns the address of the current signal handler for a specified logical address.

Remarks: Input parameter:

`la`	`INT16`	Logical address for which to find address of signal handler -2 = Unknown `la` (miscellaneous)

Output parameters:

none

Return value:

`func`	`NIVXI_HSIGNAL*`	Pointer to the current signal handler for the specified logical address (`NULL` = invalid `la`)

Example:

```
/* Get the address of the signal handler for Logical Address
   5. */

NIVXI_HSIGNAL      *func;
INT16              la;

la = 5;
func = GetSignalHandler (la);
```

RouteSignal

Syntax: `ret = RouteSignal (la, modemask)`

Action: Specifies how each type of signal is to be processed for each logical address. A signal can be enqueued on a global signal queue (for later dequeuing via `SignalDeq`) or handled by an installed signal handler for the specified logical address.

Remarks: Input parameters:

`la` `INT16` Logical address to set handler for (-1 = any known `la`)

`modemask` `UINT32` A bit vector that specifies whether each type of signal is enqueued or handled by the signal handler. A zero in any bit position causes signals of the associated type to be queued on the global signal queue. All other signals are handled by the signal handler.

If `la` is a Message-Based device:

Bit	Event Signal
14	User-Defined events
13	VXI Reserved events
12	Shared Memory events
11	Unrecognized Command events
10	Request False (REQF) events
9	Request True (REQT) events
8	No Cause Given events

Bit	Response Signal
7	Unused
6	B14 (Reserved for future definition)
5	Data Out Ready (DOR)
4	Data In Ready (DIR)
3	Protocol Error (ERR)
2	Read Ready (RR)
1	Write Ready (WR)
0	Fast Handshake (FHS)

If `la` is *not* a Message-Based device:

Bit	Type of Signal (status/ID) values
15 to 8	Active high bit (if 1 in bits 15 to 8, respectively)
7 to 0	Active low bit (if 0 in bits 15 to 8, respectively)

Output parameters:

none

Return value:

`ret` `INT16` Return Status

0 = Successful
-1 = Invalid `la`

Example 1:

```
/* Route signals for Logical Address 4 so that only REQT and REQF
   signals are enqueued on the signal queue, and the rest of the
   signals are handled by the signal handler. */

INT16      la;
UINT32     modemask;
INT16      ret;

la = 4;
modemask = 0xf9ffL;
ret = RouteSignal (la, modemask);
```

Example 2:

```
/* Route Register-Based status/ID values for Logical Address 7 so
   that all status/IDs with a 0 in bits 15 to 12 are queued and
   all status/IDs with a 1 in bits 11 to 8 are handled by the
   signal handler. */

INT16      la;
UINT32     modemask;
INT16      ret;

la = 7;
modemask = 0x0ff0L;
ret = RouteSignal (la, modemask);
```

SetSignalHandler

Syntax: `ret = SetSignalHandler (la, func)`

Action: Replaces the current signal handler for a logical address with a specified handler.

Remarks: Input parameters:

`la`	`INT16`	Logical address to set the handler -1 = All known `la`'s -2 = Unknown `la` (miscellaneous) signal handler
`func`	`NIVXI_HSIGNAL*`	Pointer to the new signal handler NULL = `DefaultSignalHandler`

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Invalid `la`

Example:

```
/* Set the signal handler for Logical Address 5. */

NIVXI_HSIGNAL    func;
INT16       la;
INT16       ret;

la = 5;
ret = SetSignalHandler (la, func);
if (ret < 0)
   /* An error occurred in SetSignalHandler . */;


   /* This is a sample VXI signal handler. */
NIVXI_HQUAL void NIVXI_HSPEC func (UINT16 signal)
{
}
```

SignalDeq

Syntax: `ret = SignalDeq (la, signalmask, signal)`

Action: Gets a signal specified by the signalmask from the signal queue for the specified logical address.

Remarks: Input parameters:

`la`	`INT16`	Logical address to dequeue signal from (255 = VME; -1 = any known `la`)
`signalmask`	`UINT32`	A bit vector indicating the type of signal to dequeue; a one in any bit position causes the subroutine to dequeue signals of the associated type, as follows:

If `la` is a Message-Based device:

Bit	Event Signal
14	User-Defined events
13	VXI Reserved events
12	Shared Memory events
11	Unrecognized Command events
10	Request False (REQF) events
9	Request True (REQT) events
8	No Cause Given events

Bit	Response Signal
7	Unused
6	B14 (Reserved for future definition)
5	Data Out Ready (DOR)
4	Data In Ready (DIR)
3	Protocol error (ERR)
2	Read Ready (RR)
1	Write Ready (WR)
0	Fast Handshake (FHS)

If `la` is *not* a Message-Based device, or if `la` = 255 (VME status/ID):

Bit	Type of Signal (status/ID) values
15 to 8	Active high bit (if 1 in bits 15 to 8, respectively)
7 to 0	Active low bit (if 0 in bits 15 to 8, respectively)

Output parameter:

`signal`	`UINT16*`	Signal value dequeued from the signal queue

Return value:

`ret`	`INT16`	Return Status

0 = A signal was returned in `signal`
-1 = The signal queue is empty or no match

Example:

```
/* Dequeue any type of signal from the signal queue for Logical
   Address 10. */

INT16      ret;
INT16      la;
UINT16     signal;
UINT32     signalmask;

la = 10;
signalmask = 0xffffL;
ret = SignalDeq (la, signalmask, &signal);
if (ret != 0)
   /* Empty signal queue for Logical Address 10. */;
```

SignalEnq

Syntax: `ret = SignalEnq (signal)`

Action: Puts a signal on the tail of the signal queue.

Remarks: Input parameter:

`signal`	`UINT16`	Value to enqueue at the tail of the signal queue

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = `signal` was queued -1 = `signal` was not queued because the signal queue is full -2 = `signal` was not queued because the logical address is invalid

Example:

```
/*  Enqueue signal 0xfd02 (REQT for Logical Address 2) at the tail
    of the signal queue. */

INT16      ret;
UINT16     signal;

signal = 0xfd02;
ret = SignalEnq (signal);
if (ret != 0)
   /* Signal queue is full. */;
```

SignalJam

Syntax: `ret = SignalJam (signal)`

Action: Puts a signal on the head of the signal queue.

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

Remarks: Input parameter:

`signal`	`UINT16`	Signal value to put on the head of the queue

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = `signal` was queued -1 = `signal` was not queued because the signal queue is full -2 = `signal` was not queued because the logical address is invalid

Example:

```
/*  Put signal 0xfd02 (REQT for Logical Address 2) on the head of
    the signal queue. */

INT16      ret;
UINT16     signal;

signal = 0xfd02;
ret = SignalJam (signal);
if (ret != 0)
   /* Signal queue is full. */;
```

WaitForSignal

Syntax: `ret = WaitForSignal (la, signalmask, timeout, retsignal, retsignalmask)`

Action: Waits for a specified type(s) of signal or status/ID to be received from a specified logical address.

Remarks: Input parameters:

`la`	`INT16`	Logical address of device sourcing the signal (255 = VME; -1 = any known `la`)
`signalmask`	`UINT32`	A bit vector indicating the type(s) of signals that the application will wait for; a one in any bit position will cause the subroutine to detect signals of the associated type, as follows:

If `la` is a Message-Based device:

Bit	Event Signal
14	User-Defined events
13	VXI Reserved events
12	Shared Memory events
11	Unrecognized Command events
10	Request False (REQF) events
9	Request True (REQT) events
8	No Cause Given events

Bit	Response Signal
7	Unused
6	B14 (Reserved for future definition)
5	Data Out Ready (DOR)
4	Data In Ready (DIR)
3	Protocol Error (ERR)
2	Read Ready (RR)
1	Write Ready (WR)
0	Fast Handshake (FHS)

If `la` is *not* a Message-Based device
or if `la` = 255 (VME status/ID):

Bit	Type of Signal (status/ID) values
15 to 8	Active high bit (if 1 in bits 15 to 8, respectively)
7 to 0	Active low bit (if 0 in bits 15 to 8, respectively)

`timeout`	`INT32`	Time to wait until signal occurs

Output parameters:

`retsignal`	`UINT16*`	Signal received
`retsignalmask`	`UINT32*`	A bit vector indicating the type(s) of signals that the application received. The bits have the same meaning as that of the input `signalmask.`

Return value:

`ret`	`INT16`	Return Status 0 = One of the specified signals was received -1 = Invalid `la` -2 = Timeout occurred while waiting for the specified signal(s)

Example:

```
/*  Wait 2 seconds for REQT signal from Logical Address 5. */

INT16      ret;
INT16      la;
UINT32     signalmask;
INT32      timeout;
UINT16     retsignal;
UINT32     retsignalmask;

la = 5;
signalmask = 0x0200L;
timeout = 2000L;
ret = WaitForSignal (la, signalmask, timeout, &retsignal,
   &retsignalmask);
if (ret == 0)
   /* Signal received within specified waiting period. */;
```

Default Handler for VXI Signal Functions

The NI-VXI software provides the following default handler for the VXI signals. 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 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.

DefaultSignalHandler

Syntax: `DefaultSignalHandler (signal)`

Action: Handles the VXI signals. It does nothing with the signals, with the exception of the VXIbus specification 1.2 Event signal *Unrecognized Command*. It calls `WSabort` if the *Unrecognized Command* Event is received.

Remarks: Input parameter:

`signal`	`UINT16`	Actual 16-bit VXI signal

Output parameters:

none

Return value:

none

Chapter 10
VXI Interrupt Functions

This chapter describes the C syntax and use of the VXI interrupt functions and default handler. VXI interrupts are a basic form of asynchronous communication used by VXI devices with VXI 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 and is the same as for the VXI signal. The lower 8 bits of the status/ID value form the VXI logical address of the interrupting device, while the upper 8 bits specify the reason for interrupting.

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

Because the VXI interrupt status/ID has the same format as the VXI signal, your application can handle VXI interrupts as VXI signals. However, because VME interrupters may be present in a VXI system, the VXI/VME interrupt handler functions are included with the NI-VXI software. The `RouteVXIint` function specifies whether the status/ID value should be handled as a signal or handled by a VXI/VME interrupt handler. Two methods are available to handle VXI signals. Signals can be handled either by calling a signal handler, or by queueing on a global signal queue. The `RouteSignal` function specifies which types of signals are handled by signal handlers, and which are queued onto the global signal queue for each VXI logical address. A separate handler can be installed for each VXI logical address present (refer to the description for `SetSignalHandler`). A default handler, `DefaultSignalHandler`, is automatically installed when `InitVXIlibrary` is called from the application for every VXI logical address. If signals are queued, the application can use the `SignalDeq` function to selectively return a signal off a global signal queue by VXI logical address and/or type of signal.

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

If the `RouteVXIint` has specified that a status/ID value should be handled by the VXI interrupt handler and not by the signal handler, the specified VXI interrupt handler is invoked. The main use for the VXI interrupt handler is to handle VME interrupters and Register-Based VXI interrupters. The VXI interrupt handler for a particular level is called with the VXI interrupt level and the status/ID without any interpretation of the status/ID value. The VXI 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 VXI interrupts routed to the VXI interrupt handlers. `EnableVXItoSignalInt` and `DisableVXItoSignalInt` are used to sensitize and desensitize the application to VXI interrupts routed to be processed as VXI signals.

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

Programming Considerations

The following is a graphical overview of the NI-VXI interrupt and signal model.

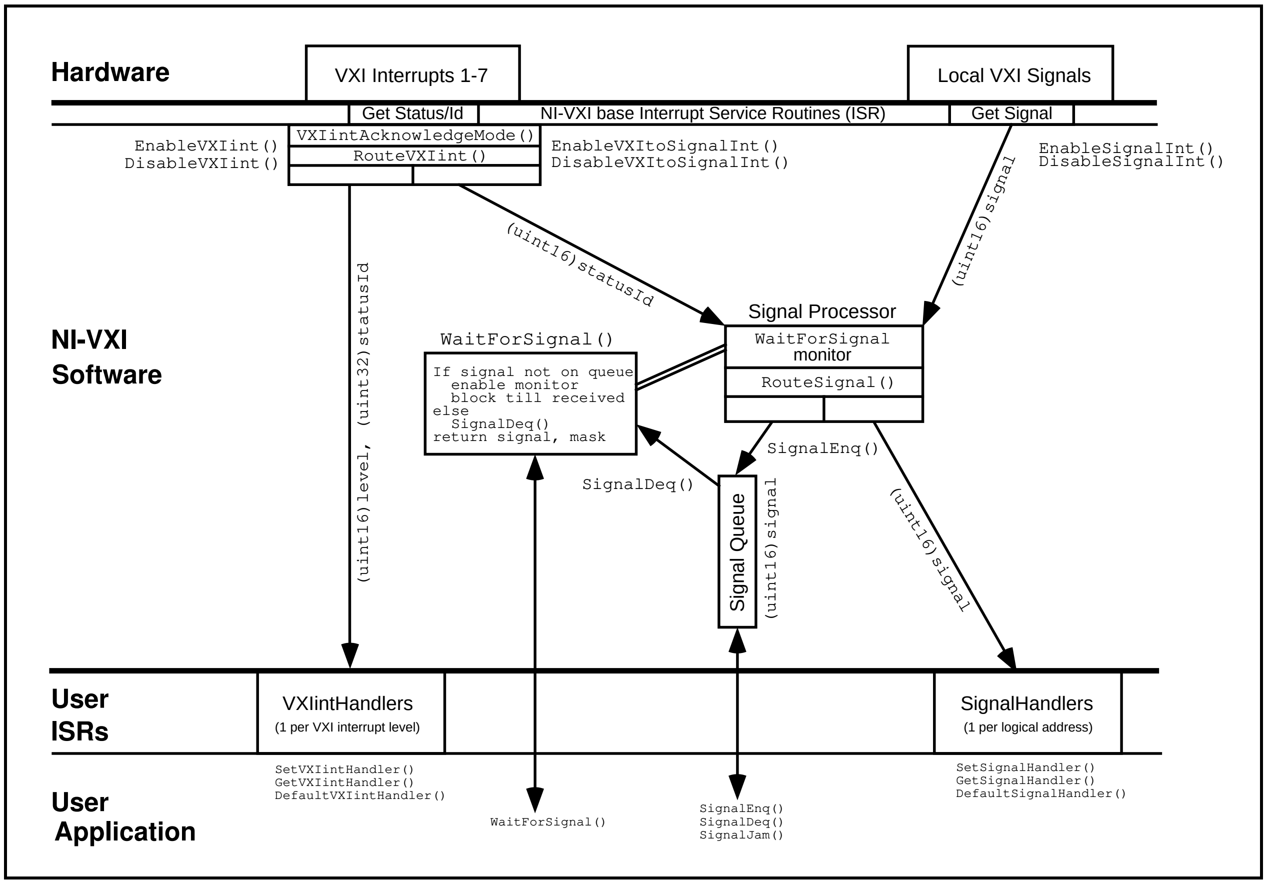

Figure 10-1. NI-VXI VXI Interrupt and Signal Model

ROAK Versus RORA VXI Interrupters

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

Functional Overview

The following paragraphs describe the VXI 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 status/ID values returned from a VXI/VME interrupt acknowledge cycle are routed to a VXI interrupt handler or to the VXI signal processing routine. Because VME interrupters may be present in a VXI system, the VXI interrupt handler functions can be used to directly handle VME interrupts. The function `RouteVXIint` specifies whether the status/ID value should be handled as a signal or handled locally by a VXI interrupt handler. Two methods are available to handle VXI signals. Signals can be handled either by signal handlers (as signals) or by queueing on a global signal queue. The `RouteSignal` function specifies which types of signals should be handled by signal handlers, and which should be queued on the global signal queue for each VXI logical address. If the VXI interrupt status/IDs are specified to be handled by a VXI interrupt handler, the level and status/ID value is sent to the appropriate VXI interrupt handler when a VXI interrupt occurs. An individual handler can be installed for each of the seven VXI interrupt levels. `EnableVXIint` and `EnableVXItoSignalInt` must be used to sensitize the local CPU to interrupts generated by VXI interrupts. Only the levels routed to the appropriate handlers (VXI/VME interrupts or VXI signals) via the `RouteVXIint` function are enabled.

EnableVXItoSignalInt (controller, levels)

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

DisableVXItoSignalInt (controller, levels)

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

EnableVXIint (controller, levels)

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

DisableVXIint (controller, levels)

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

VXIintAcknowledgeMode (controller, modes)

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

SetVXIintHandler (levels, func)

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

GetVXIintHandler (level)

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

DefaultVXIintHandler (controller, level, statusId)

`DefaultVXIintHandler` is the sample handler for VXI interrupts, which is installed when the function `InitVXIlibrary` is called. If VXI interrupts are enabled (via `EnableVXIint`), the VXI interrupt handler for a specific logical address is called. You must first call `RouteVXIint` to route VXI interrupts to the VXI interrupt handler (as opposed to the signal processing routine). `DefaultVXIintHandler` sets the global variables `VXIintController`, `VXIintLevel`, and `VXIintStatusId`. You can leave this default handler installed or install a completely new VXI interrupt handler using `SetVXIintHandler`.

AssertVXIint (controller, level, statusId)

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

DeAssertVXIint (controller, level)

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

Note: ***Unasserting a VXI 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 VXI interrupt acknowledge (IACK cycle) on the VXIbus backplane in the specified controller and VXI interrupt level.

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

Normally, VXI interrupts are automatically acknowledged when enabled via the function `EnableVXIint`. However, if the VXI 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 frame extender (if present) to acknowledge the specified interrupt.

Function Descriptions

The following paragraphs describe the VXI 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 VXIbus on the specified controller (either an embedded CPU or an extended controller) for a particular VXI interrupt level. VXI interrupts are automatically acknowledged when enabled by `EnableVXItoSignalInt` and `EnableVXIint`. Use this function to manually acknowledge VXI interrupts that the local device is not enabled to receive.

Note: ***This function is for debugging 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 VXI interrupt line on the specified controller (either an embedded CPU or an extended controller). When the VXI interrupt is acknowledged (a VXI IACK cycle occurs), the specified status/ID is passed to the device that acknowledges the VXI 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 VXI interrupter) -2 = Invalid `controller` -3 = Invalid `level` -5 = VXI 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 unasserts a VXI 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 debugging purposes only. Unasserting a VXI 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:

```
/* Unassert 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 VXI interrupts generated in the specified controller, which the `RouteVXIint` function routed to be handled as VXI interrupts (not as VXI signals). The RM assigns the interrupt levels automatically. `GetDevInfo` can be used to retrieve the assigned levels.

Remarks: Input parameters:

`controller`	`INT16`	Controller (embedded or extended) to disable interrupts
`levels`	`UINT16`	Vector of VXI interrupt levels to disable. Bits 6 to 0 correspond to VXI 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 = VXI interrupt disabled -1 = No hardware support -2 = Invalid `controller`

Example:

```
/* Disable VXI 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 VXI 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 VXI interrupt levels to disable. Bits 6 to 0 correspond to VXI 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 = VXI interrupt disabled -1 = No hardware support -2 = Invalid `controller`

Example:

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

Remarks: Input parameters:

`controller`	`INT16`	Controller (embedded or extended) to enable interrupts
`levels`	`UINT16`	Vector of VXI interrupt levels to enable. Bits 6 to 0 correspond to VXI 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 = VXI interrupt enabled -1 = No hardware support -2 = Invalid `controller`

Example:

```
/* Enable VXI 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 VXI 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 VXI interrupt is physically enabled only if the `RouteVXIint` function has specified that the VXI 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 VXI interrupt levels to enable. Bits 6 to 0 correspond to VXI 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 = VXI interrupt enabled -1 = No hardware support -2 = Invalid `controller`

Example:

```
/* Enable VXI 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 VXI interrupt handler for a specified VXIbus interrupt level.

Remarks: Input parameter:

`level`	`UINT16`	VXI interrupt level associated with the handler

Output parameters:

none

Return value:

`func`	`NIVXI_HVXIINT*`	Pointer to the current interrupt handler for a specified VXIbus interrupt level (`NULL` = invalid `level` or no hardware support)

Example:

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

NIVXI_HVXIINT      *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 VXI interrupt acknowledge cycle to the VXI interrupt handler or to the signal processing routine. `RouteVXIint` dynamically enables and disables the appropriate VXI interrupts based on the current settings from calls to `EnableVXIint` and `EnableVXItoSignalInt`.

Remarks: Input parameters:

`controller`	`INT16`	Controller (embedded or extended) to specify route for
`Sroute`	`UINT16`	A bit vector that specifies whether to handle a VXI/VME interrupt as a signal or route it to the VXI/VME interrupt handler routine. Bits 6 to 0 correspond to VXI interrupt levels 7 to 1, respectively. 1 = Handle VXI interrupt for this level as a signal 0 = Handle VXI interrupt as a VXI interrupt Bits 14 to 8 correspond to VXI interrupt levels 7 to 1, respectively. 1 = Route as 8-bit VME status/ID 0 = Route as 16-bit VXI status/ID

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = No hardware support -2 = Invalid `controller`

Example:

```
/*  Route VXI interrupts for level 4 (on the local controller) to the VXI
    signal processor. */

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 VXI interrupt handler for the specified VXIbus interrupt levels with a specified VXI interrupt handler.

Remarks: Input parameters:

`levels`	`UINT16`	Bit vector of VXI interrupt levels. Bits 6 to 0 correspond to VXI interrupt levels 7 to 1, respectively. 1 = Set 0 = Do not set handler
`func`	`NIVXI_HVXIINT*`	Pointer to the new VXI interrupt handler (NULL = `DefaultVXIintHandler`)

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = No hardware support

Example:

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

NIVXI_HVXIINT    func;
UINT16   levels;
INT16    ret;

levels = (UINT16)(1<<3);
ret = SetVXIintHandler (levels, func);
if (ret < 0)
   /* An error occurred in SetVXIintHandler. */;


   /* This is a sample VXI interrupt handler. */
NIVXI_HQUAL void NIVXI_HSPEC func (INT16 controller, UINT16 level,
UINT32 statusId)
{
}
```

VXIintAcknowledgeMode

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

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

Remarks: Input parameters:

`controller`	`INT16`	Controller (embedded or extended) for which to specify interrupt acknowledge
`modes`	`UINT16`	Vector of VXI interrupt levels to set to RORA/ROAK interrupt acknowledge mode. Bits 6 to 0 correspond to VXI interrupt levels 7 to 1, respectively. 0 = Set to ROAK VXI interrupt for corresponding level 1 = Set to RORA VXI interrupt for corresponding level

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = No hardware support -2 = Invalid `controller`

Example:

```
/* Set VXI 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 VXI Interrupt Functions

The NI-VXI software provides the following default handler for the VXI 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 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.

DefaultVXIintHandler

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

Action: Handles the VXI 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 VXI interrupt level
`statusId`	`UINT32`	Status/ID obtained during IACK cycle (if it is a 16-bit VXI IACK value, it may be equivalent to a VXI signal)

Output parameters:

none

Return value:

none

Chapter 11
VXI Trigger Functions

This chapter describes the C syntax and use of the VXI trigger functions. VXI triggers are a backplane feature that VXI added to the VME standard. Tight timing and signaling is important between many types of controllers and/or instruments. In the past, clumsy cables of specified length had to be connected between controllers and/or instruments to get the required timing. For many systems, phase shifting and propagation delays had to be calculated precisely, based on the instrument connection scheme. This limited the architecture of many systems. In VXI however, every VXI board with a P2 connector has access to eight 10 MHz TTL trigger lines. If the VXI board has a P3 connector, it has access to six 100 MHz ECL trigger lines. The phase shifting and propagation delays can be held to a known maximum, based on the VXIbus specification's rigid requirement on backplanes. The VXIbus specification does not currently prescribe an allocation method for TTL or ECL trigger lines. The application must decide how to allocate any use any of the trigger lines it requires. The VXIbus specification specifies several trigger protocols that can be supported, thereby promoting compatibility among the various VXI devices. The following is a description of the four basic protocols.

- **SYNC**
  The most basic protocol is SYNC protocol. SYNC protocol is simply a pulse of a minimum time (30 ns on TTL, 8 ns on ECL) on any one of the trigger lines.

- **ASYNC**
  ASYNC is a two-device, two-line handshake protocol. ASYNC uses two consecutive even/odd trigger lines (a source/acceptor line and an acknowledge line, respectively). The sourcing device sources a trigger pulse (30 ns TTL, 8 ns ECL minimum) on the even trigger line (TTL0, TTL2, TTL4, TTL6, ECL0, ECL2, or ECL4) and waits for the acknowledge pulse on the next highest odd trigger line (TTL1, TTL3, TTL5, TTL7, ECL1, ECL3, or ECL5). The acceptor waits for the sourced pulse on the even trigger line. Sometime after the source pulse is sensed (no maximum time is specified), the acceptor sends an acknowledge pulse back on the next highest odd trigger line to complete the handshake.

- **SEMI-SYNC**
  SEMI-SYNC is a one-line, open collector, multiple-device handshake protocol. The sourcing device sources a trigger pulse (50 ns TTL, 20 ns ECL minimum) on any one of the trigger lines. The accepting device(s) must begin to assert the same trigger line upon reception (within 40 ns TTL, 15 ns ECL maximum time from source assertion edge). The accepting device(s) can later unassert the trigger line (no maximum time is specified) to complete the handshake.

- **START/STOP**
  START/STOP is a one-line, multiple-device protocol. START/STOP can be sourced only by the VXI Slot 0 device and sensed by any other devices on the VXI backplane. The START/STOP protocol is synchronized with the backplane clock (CLK10 for TTL, CLK100 and SYNC100 for ECL) onto any one of the trigger lines. A START condition is generated on the assertion edge on the trigger line, and a STOP condition is generated on the unassertion edge of the trigger line.

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

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

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

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

Capabilities of the National Instruments Triggering Hardware

The NI-VXI trigger functions are a general purpose interface designed to accommodate most uses of VXI triggers. The actual capabilities of a particular platform will always be a subset of these capabilities. In general, however, National Instruments hardware has only four current hardware capability categories:

- Trigger control used on a VXI-MXI frame extender when used as an extended controller (under direct control of a root-level MXI controller interface) that *does not* have the National Instruments Trigger Interface Chip (TIC) on it
- An embedded controller *without* the National Instruments TIC
- Trigger control used on a VXI-MXI frame extender when used as an extended controller (under direct control of a root-level MXI controller interface) that *does* have the National Instruments TIC on it
- An embedded controller *with* the National Instruments TIC

External Controller/VXI-MXI Trigger Capabilities (without TIC Chip)

All National Instruments external controllers connected to VXI-MXI frame extenders without the TIC chip have the same basic trigger capabilities:

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

The following capabilities are not supported:

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

Embedded Controller Trigger Capabilities (without TIC Chip)

All National Instruments embedded controllers without the TIC chip have the same basic trigger capabilities:

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

The following capabilities are not supported:

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

Embedded and External Controller Trigger Capabilities (with TIC Chip)

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

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

Figure 11-1 is a block diagram showing the general capabilities of the TIC chip. Figures 11-2 and 11-3 are block diagrams of the trigger module and GPIO module, respectively.

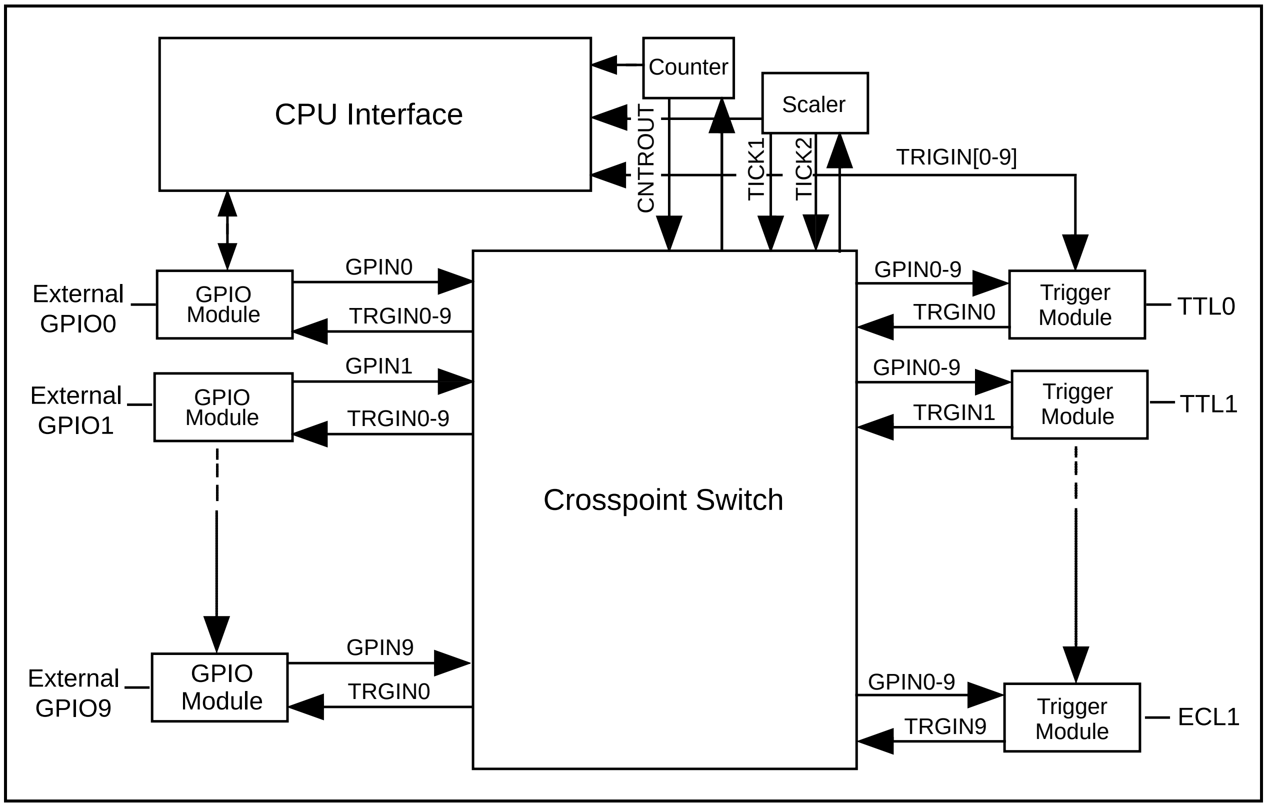

Figure 11-1. TIC Chip Block Diagram

Figure 11-2 is a second-level block diagram of the trigger module within the TIC chip.

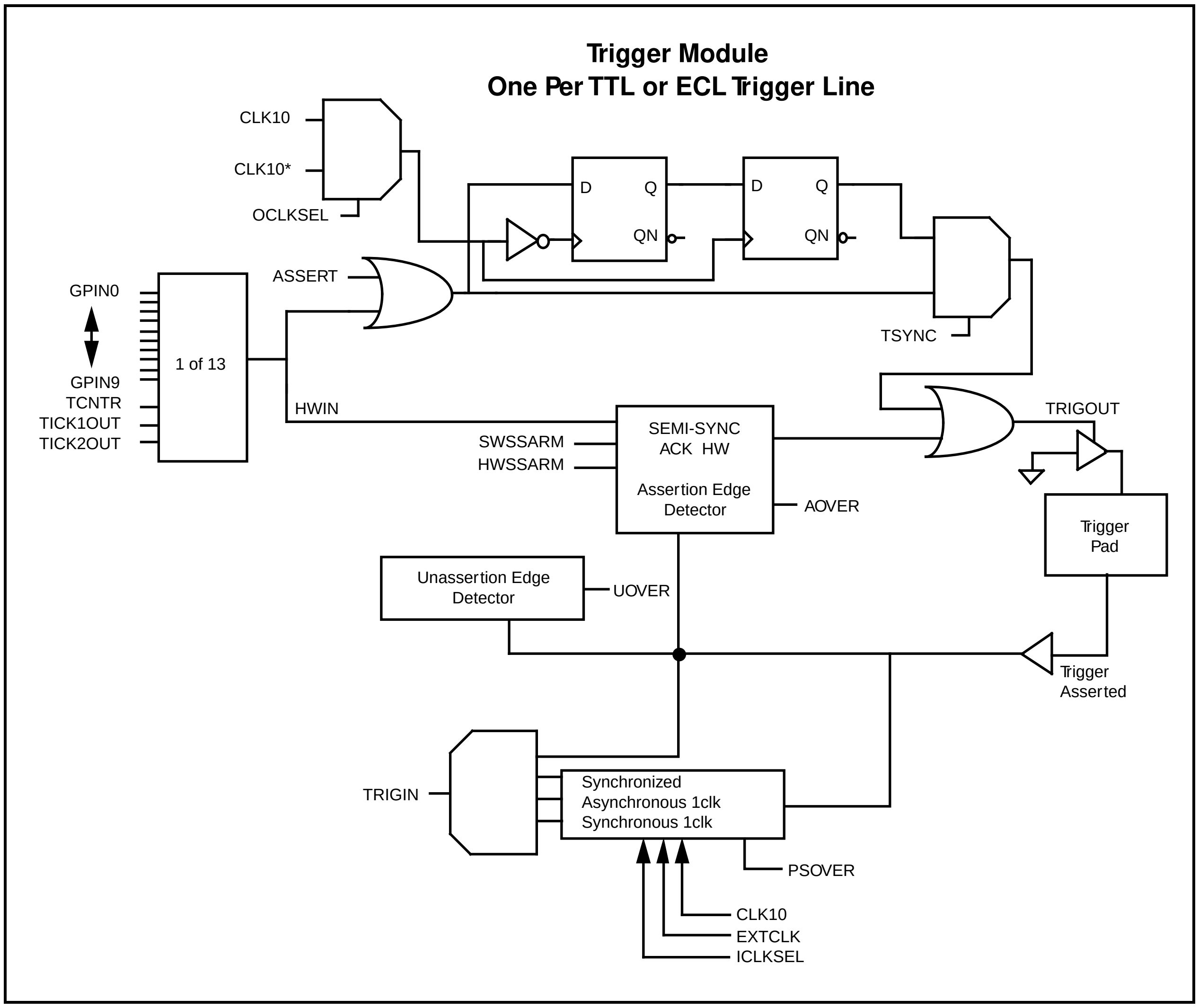

Figure 11-2. Trigger Module Block Diagram

Figure 11-3 is a second-level block diagram of the GPIO module within the TIC chip.

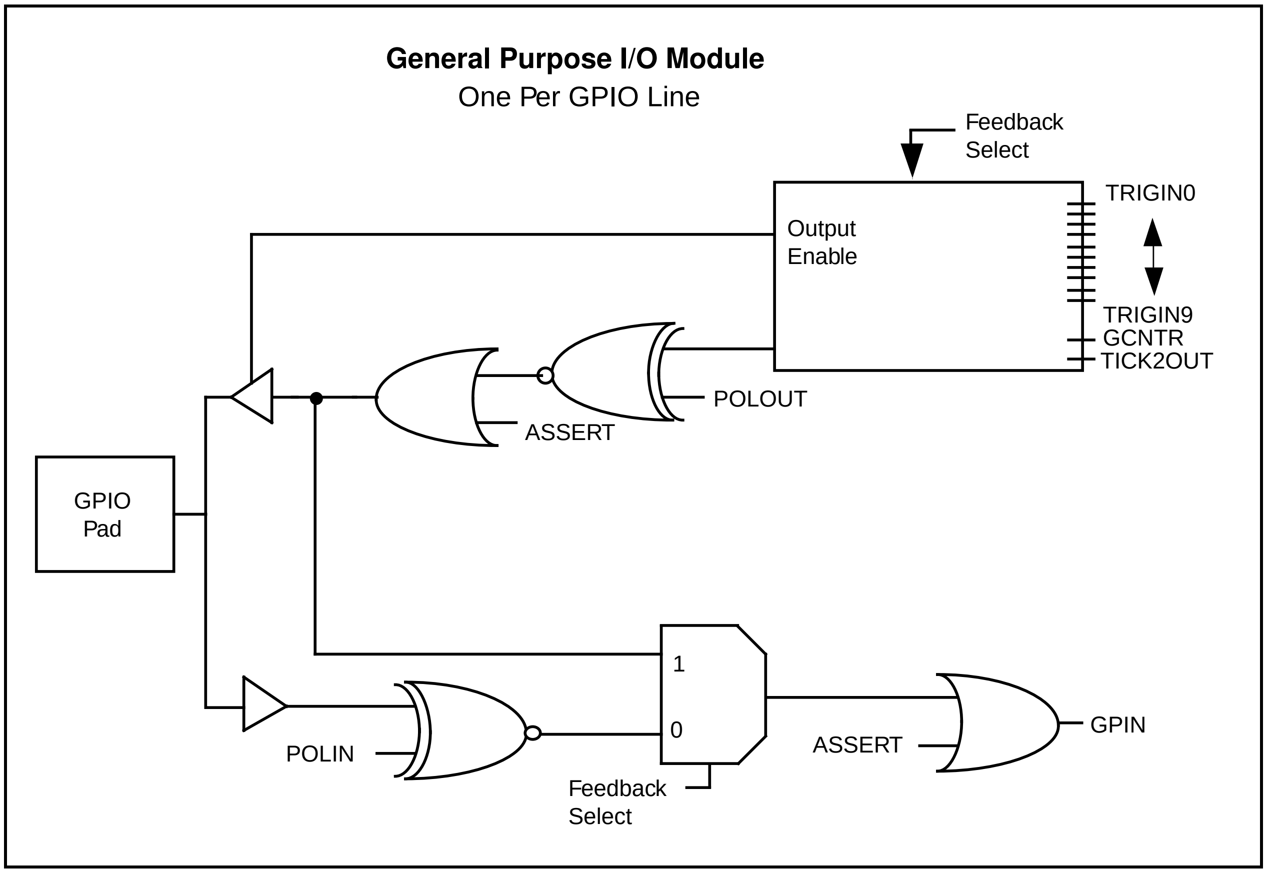

Figure 11-3. GPIO Module Block Diagram

All National Instruments embedded controllers with the TIC chip have the same basic trigger capabilities. Minor differences reside only with the external or GPIO connections. The following sections list these capabilities.

Sourcing

- TTL/ECL lines (*any* line, all lines at the same time)
  - START (continuous ON)
  - STOP (continuous OFF)
  - SYNC
  - SEMI-SYNC (with interrupt when acknowledged)
  - SEMI-SYNC (wait to be acknowledged)
    - Automatic detection of unassertion edge overrun errors

- TTL/ECL lines (*even/odd* pairs of lines, all pairs at the same time)
  - ASYNC (with interrupt when acknowledged)
    - Automatic acknowledge overrun detection provided (will detect if more than one pulse received on acknowledge line)
  - ASYNC (wait to be acknowledged)
- GPIO lines
  - START (continuous ON)
  - STOP (continuous OFF)
  - SYNC
- Counter (one 16-bit counter only)
  - Interrupt when counter has counted down regardless of mapping or protocol
  - Map counter finished signal (GCNTR) to any GPIO line
  - Multiple SYNC sourcing
    - 100 ns pulse sourced 0 to 65535 times at a particular frequency with a minimum gap of 200 ns between pulses (3.3 MHz)
    - Frequency source may be:

CLK10:	3.33 MHz time period (100 ns pulse, 200 ns gap)
EXTCLK:	Configured frequency (100 ns pulse on each clock edge)
TTL/ECL trigger:	Configured frequency (100 ns pulse on each clock edge)

  - Multiple SEMI-SYNC sourcing 0 to 65535 times
    - Uses trigger line as source to counter to wait for acknowledge (unassertion) and generates next pulse (100 ns delay before next pulse)
- TICK timers (count source to power of 2 only from 2^0 to 2^{32})
  - Interrupt when TICK1 has counted down regardless of configuration.
  - TICK2 may be interrupted on only if routed to a trigger line. (refer to the *Mapping/Conditioning* section later in this chapter)
  - Use as a square wave (not as 100 ns pulse) counter: no tick rollover selected.
    - Source may be any GPIO, CLK10, or EXTCLK.
    - TICK1 specifies length total count.
    - TICK2 specifies frequency to tick for TICK1 period of time.

    Note: ***This works up until the last TICK2 count (when TICK1 is going to zero). A 0 to 20 ns glitch on TICK2 happens on the last tick.***
  - Use as a continuous tick timer and/or scaled trigger output: rollover mode
    - Source can be any GPIO, CLK10, or EXTCLK.
    - TICK1 specifies timer interrupt/trigger pulse duration.
    - TICK2 specifies scaled length trigger output of source.
      - TICK2 output may be mapped to any GPIO; if a GPIO is externally (off chip) routed to EXTCLK, any EXTCLK may be configured based on TICK2 output. You can use this EXTCLK for signal conditioning (refer to the *Mapping/Conditioning* section later in this chapter).

Accepting

- TTL/ECL lines (*any* line, all lines at the same time)
  - START (continuous ON)
  - STOP (continuous OFF)
  - SYNC
  - SEMI-SYNC (with function call acknowledge)
  - All protocols can call a handler when trigger occurs, or they can wait until the trigger occurs. In addition, automatic detection of assertion and unassertion overruns is provided.
- TTL/ECL lines (*even/odd* pairs of lines, all pairs at the same time)
  - ASYNC with function call acknowledge
  - ASYNC can either have a handler called when trigger occurs, or wait until the trigger occurs. In addition, automatic detection of assertion edge overruns is provided.
- Counter (one 16-bit counter only)
  - Interrupt when counter has counted down regardless of mapping or protocol
  - Multiple SYNC accepting
    - Detect assertion edges 0 to 65535 times.
    - Detection source can be *any number* of the following: CLK10, EXTCLK (configured frequency), or a TTL/ECL trigger line. Normal operation would use one source (simultaneous sourcing on two sources will be detected as one clock.
  - Multiple SEMI-SYNC accepting
    - Detect assertion edges 0 to 65535 times participating in the SEMI-SYNC acknowledge protocol.
    - Uses trigger line as source to counter to wait for assertion of each trigger and generates proper acknowledge; last count is left unacknowledged so that software can take action before acknowledging.
    - Detection source can be *any number* of the following: CLK10, EXTCLK (configured frequency), or a TTL/ECL trigger line. Normal operation would use one source (simultaneous sourcing on two sources will be detected as one clock.
- TICK timers (count source to power of 2 only from 2^0 to 2^{32}) as described previously in the *Sourcing* section earlier in this chapter

Mapping/Conditioning

- To TTL/ECL lines (*any* line, all lines at the same time)
  - Map any one external input (GPIO) to trigger line.
  - Map CNTR pulse output to trigger line.
  - Map TICK1 square wave output to trigger line.
  - Map TICK2 square wave output to trigger line.
- To external (GPIO) lines (*any* line, all lines at the same time)
  - Map any one TTL/ECL trigger line to GPIO.
  - Map CNTR terminated continuous output to GPIO line.

  - Map TICK2 square wave output to GPIO line.
  - Signal condition any of the above sources before entering the GPIO.
    - Invert the polarity of the source.
    - Synchronize the pulse to the next clock edge.
    - Pulse stretch for one clock period (synchronous or nonsynchronous).
      - Pulse stretch overrun errors (new pulse received before stretching completed) automatically enabled if sensing source. Can call `EnableTrigSense` with a special protocol to monitor only pulse stretch overrun errors.
    - Select clock source (CLK10 or EXTCLK) to synchronize/pulse stretch from.

Setup/Configuration Options

- Configure external (GPIO) lines (*any* line, all lines at the same time).
  - Make mapping to GPIO go out the chip or feed it back for crosspoint.
  - Can invert the polarity of the source mapped to a GPIO (regardless of whether routed out the chip or fed back).
  - If external (off chip) input, can configure to invert polarity.
  - If I/O or fed back, can configure to be high, low, or tristated (unconfigured). Must be tristated to be used in sourcing START/STOP/SYNC out a GPIO. If configured to be high or low, source cannot have another source mapped as an input.
- Configure trigger assertion method (*any* line, all lines at the same time).
  - Have the output driver for a particular trigger line (re-)synchronize the trigger to CLK10. (You can globally select for all trigger lines whether to synchronize to the rising or falling edge of CLK10.)
  - Specify that an automatic hardware SEMI-SYNC acknowledge assertion happen on any trigger input assertion. This acknowledgment can be released either by an external line (GPIO) mapped to the trigger line or from a software acknowledgment call.

Combination Options

- Sourcing and accepting
  - You can source and accept on all TTL/ECL trigger lines for all protocols at the same time. You can also enable both acknowledge and sensing interrupts at the same time. This should help you debug and test your code as well as part of a possible configuration (you could use `SrcTrig` to abort a pending `WaitForTrig`).
  - You can source and accept on all protocols regardless of the crosspoint switch mapping. (For example, you could have a trigger line mapped to a GPIO and back to a different trigger line and still assert the GPIO or assert/sense both trigger lines.) This gives you maximum flexibility for system configuration and debugging purposes.

Functional Overview

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

Source Trigger Functions

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

SrcTrig (controller, line, prot, timeout)

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

SetTrigHandler (lines, func)

`SetTrigHandler` replaces the current trigger handler for the specified VXI trigger lines with an alternate handler. When waiting for an acknowledgment of the ASYNC or SEMI-SYNC protocols after a `SrcTrig` call, the trigger handler for a specific trigger line is called when the accepting device(s) returns an acknowledgment. A default handler, `DefaultTrigHandler`, is automatically installed for every applicable trigger line when the `InitVXIlibrary` function is called.

GetTrigHandler (line)

`GetTrigHandler` returns the address of the current trigger handler for the specified VXI trigger line. When waiting for an acknowledgment of the ASYNC or SEMI-SYNC protocols after a `SrcTrig` call, the trigger handler for a specific trigger line is called when the accepting device(s) returns an acknowledgment. A default handler, `DefaultTrigHandler`, is automatically installed for every applicable trigger line when the `InitVXIlibrary` function is called.

DefaultTrigHandler (controller, line, type)

`DefaultTrigHandler` is the sample handler for the receiving acknowledges and sensing triggers, and is automatically installed when the `InitVXIlibrary` function is called. When waiting for an acknowledgment of the ASYNC or SEMI-SYNC protocols after a `SrcTrig` call, the trigger handler for a specific trigger line is called when the accepting device(s) returns an acknowledgment. `DefaultTrigHandler` calls the `AcknowledgeTrig` function if the `type` parameter specifies that an acknowledge interrupt occurred. Otherwise, `DefaultTrigHandler` performs no operations.

DefaultTrigHandler2 (controller, line, type)

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

Acceptor Trigger Functions

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

EnableTrigSense (controller, line, prot)

`EnableTrigSense` configures and sensitizes the triggering hardware to generate interrupts when the specified VXI-defined trigger protocol is sensed on the specified trigger line. When `EnableTrigSense` has configured and enabled the triggering hardware to generate interrupts, and the specified trigger protocol is sensed, a local CPU interrupt is generated. The trigger handler installed is automatically called when a trigger interrupt occurs. If the trigger protocol requires an acknowledgment (either ASYNC or SEMI-SYNC), you should call `AcknowledgeTrig` when it is appropriate to acknowledge the interrupt. `AcknowledgeTrig` will acknowledge the trigger protocol accordingly. A default handler, `DefaultTrigHandler`, is automatically installed when the `InitVXIlibrary` function is called. You can use `SetTrigHandler` to install a new handler.

DisableTrigSense (controller, line)

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

SetTrigHandler (lines, func)

`SetTrigHandler` replaces the current trigger handler for the specified VXI trigger lines with an alternate handler. After a call to `EnableTrigSense` for a particular VXI trigger line protocol, the trigger handler for a specific trigger line is called when the sourced trigger is sensed from the sourcing device. If the configured VXI trigger protocol requires an acknowledgment (either ASYNC or SEMI-SYNC), you can call the `AcknowledgeTrig` function to perform the acknowledgment. A default handler, `DefaultTrigHandler`, is automatically installed when the `InitVXIlibrary` function is called for every applicable trigger line.

GetTrigHandler (line)

`GetTrigHandler` returns the address of the current trigger handler for the specified VXI trigger line. After a call to `EnableTrigSense` for a particular VXI trigger line protocol, the trigger handler for a specific trigger line is called when the sourced trigger is sensed from the sourcing device. If the configured VXI trigger protocol requires an acknowledgment (either ASYNC or SEMI-SYNC), you can call the `AcknowledgeTrig` function to perform the acknowledgment. A default handler, `DefaultTrigHandler`, is automatically installed when the `InitVXIlibrary` function is called for every applicable trigger line.

DefaultTrigHandler (controller, line, type)

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

DefaultTrigHandler2 (controller, line, type)

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

AcknowledgeTrig (controller, line)

`AcknowledgeTrig` performs the required trigger acknowledgments for the ASYNC or SEMI-SYNC VXI-defined protocol, as configured via the `EnableTrigSense` function. After a call to `EnableTrigSense` for a particular VXI trigger line protocol, the trigger handler for a specific trigger line is called when the sourced trigger is sensed from the sourcing device. If the configured VXI trigger protocol requires an acknowledgment (either ASYNC or SEMI-SYNC), you must call the `AcknowledgeTrig` function to perform the acknowledgment. A default handler, `DefaultTrigHandler`, is automatically installed for every applicable trigger line when the `InitVXIlibrary` function is called. You can use `SetTrigHandler` to install a new handler.

WaitForTrig (controller, line, timeout)

You can use the `WaitForTrig` function to suspend operation until it receives a trigger configured by the `EnableTrigSense` function. After a call to `EnableTrigSense` for a particular VXI trigger line protocol, the trigger handler for a specific trigger line is called when the sourced trigger is sensed from the sourcing device. You can use `WaitForTrig` as an alternate method for receiving sensed triggers by having the caller wait until the trigger occurs instead of installing an interrupt handler. The current trigger interrupt handler is invoked regardless of whether a `WaitForTrig` call is pending. If the configured VXI trigger protocol requires an acknowledgment (either ASYNC or SEMI-SYNC), you can call the `AcknowledgeTrig` to perform the acknowledgment.

Map Trigger Functions

You can use the NI-VXI map trigger functions as configuration tools for multiframe and local support for VXI triggers. You can configure the triggering hardware to route specified source trigger locations to destination trigger locations by using the `MapTrigToTrig` and `UnMapTrigToTrig` functions. The possible values for source or destination locations are the TTL trigger lines, ECL trigger lines, Star X lines, Star Y lines, or miscellaneous external sources. Miscellaneous external sources include front panel trigger ins, front panel trigger outs, local clocks, and crosspoint switch locations. The external source locations are dependent on the particular hardware platforms capabilities. In this way, you can use `MapTrigToTrig` as a simple map from an external source to a trigger line, or as a complex crosspoint switch configurator (depending on the hardware capabilities of the specified device).

MapTrigToTrig (controller, srcTrig, destTrig, mode)

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

UnMapTrigToTrig (controller, srcTrig, destTrig)

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

Trigger Configuration Functions

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

TrigAssertConfig (controller, trigline, mode)

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

TrigExtConfig (controller, extline, mode)

`TrigExtConfig` configures the way the external trigger sources (General Purpose Inputs and Outputs, or *GPIOs*) are configured. The TIC chip has 10 GPIO lines. GPIO0 is connected to the front panel In connector. GPIO 1 is connected to the front panel Out connector. GPIO 2 is connected to a direct ECL bypass from the front panel. GPIO 3 is fed back in as the EXTCLK signal used for signal conditioning modes with `MapTrigToTrig`. The six remaining GPIOs are dependent upon the hardware platform. Consult the documentation for your specific platform for further information. Regardless of the sources connected to the GPIOs, `TrigExtConfig` configures several aspects of the connection. You can disconnect and feed back the connection for use as a crosspoint switch. You can also choose whether to invert the external input. In addition, you can configure the GPIO to be asserted high or low continuously. In this configuration, no input mapping is possible (that is, no trigger line can be mapped to the GPIO).

TrigCntrConfig (controller, mode, source, count)

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

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

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

Function Descriptions

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

AcknowledgeTrig

Note: ***This function call may not exist on some platforms that do not have the TIC chip. If this is the case, you can achieve the same functionality by using the name*** *AcknowledgeTTLtrig* ***or*** *AcknowledgeECLtrig* ***with the same parameters as described below.***

Syntax: `ret = AcknowledgeTrig (controller, line)`

Action: Acknowledges the specified TTL/ECL or external (GPIO) trigger on the specified controller. The TTL/ECL trigger interrupt handler is called after an TTL/ECL trigger is sensed. If the sensed protocol requires an acknowledge (ASYNC or SEMI-SYNC protocols), the application should call `AcknowledgeTrig` after performing any device-dependent operations. If you configured a trigger line using the `TrigAssertConfig` function to participate in external (GPIO) SEMI-SYNC acknowledging, you can use `AcknowledgeTrig` to manually acknowledge a pending external SEMI-SYNC trigger.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to acknowledge trigger interrupt
`line`	`UINT16`	TTL, ECL, or external trigger line to acknowledge

Value	Trigger Line
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
40 to 49	External source/destination (GPIOs 0 to 9)

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

1 = Successful, protocol has no need to acknowledge
0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `line`
-4 = `line` not supported
-12 = `line` not configured for sensing
-17 = No trigger sensed
-18 = `line` not configured for external SEMI-SYNC

Example:

```
/* Acknowledge a trigger interrupt for TTL line 4 on the local CPU
   (or the first extended controller). */

INT16      controller;
UINT16     line;
INT16      ret;

controller = -1;
line = 4;
ret = AcknowledgeTrig (controller, line);
```

DisableTrigSense

Note: ***This function call may not exist on some platforms that do not have the TIC chip. If this is the case, you can achieve the same functionality by using the name*** *DisableTTLsense* ***or*** *DisableECLsense* ***with the same parameters as described below.***

Syntax: `ret = DisableTrigSense (controller, line)`

Action: Disables the sensing of the specified TTL/ECL trigger line, counter, or tick timer that was enabled by `EnableTrigSense`.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to disable sensing
`line`	`UINT16`	Trigger line to disable sensing

Value	Trigger Line
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
50	TIC counter
60	TIC tick timers

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `line`
-4 = `line` not supported
-12 = `line` not configured for sensing

Example:

```
/* Disable sensing of TTL line 4 on the local CPU (or the first
   extended controller). */

INT16      ret;
INT16      controller;
UINT16     line;

controller = -1;
line = 4;
ret = DisableTrigSense (controller, line);
```

EnableTrigSense

Note: ***This function call may not exist on some platforms that do not have the TIC chip. If this is the case, you can achieve the same functionality by using the name*** *EnableTTLsense* ***or*** *EnableECLsense* ***with the same parameters as described below.***

Syntax: `ret = EnableTrigSense (controller, line, prot)`

Action: Enables the sensing of the specified TTL/ECL trigger line or starts up the counter or tick timer for the specified protocol. When the protocol is sensed, the corresponding trigger interrupt handler is invoked. In order to start up the counter or tick timers, you must first call either the `TrigCntrConfig` or the `TrigTickConfig` function, respectively.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to enable sensing
`line`	`UINT16`	Trigger line to enable sensing

Value	Trigger Line
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
50	TIC counter
60	TIC tick timers

`prot`	`UINT16`	Protocol to use

0 = ON
1 = OFF
2 = START
3 = STOP
4 = SYNC
5 = SEMI-SYNC
6 = ASYNC

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `line` or `prot`
-4 = `line` not supported
-5 = `prot` not supported
-7 = `line` already in use
-12 = `line` not configured for use in sensing
-15 = Previous operation incomplete

Example:

```
/* Enable sensing of TTL line 4 on the local CPU (or the first
   extended controller) for SEMI-SYNC protocol. */

INT16      ret;
INT16      controller;
UINT16     line;
UINT16     prot;

controller = -1;
line = 4;
prot = 5;
ret = EnableTrigSense (controller, line, prot);
```

GetTrigHandler

Note: ***This function call may not exist on some platforms that do not have the TIC chip. If this is the case, you can achieve the same functionality by using the name*** *GetTTLtrigHandler* ***or*** *GetECLtrigHandler* ***with the same parameters as described below.***

Syntax:

```
func = GetTrigHandler (line)
```

Action: Returns the address of the current TTL/ECL trigger, counter, or tick timer interrupt handler for a specified trigger source.

Remarks: Input parameter:

`line`	`UINT16`	TTL, ECL trigger line or counter/tick

Value	Trigger Line
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
50	TIC counter
60	TIC TICK1 tick timer

Output parameters:

none

Return value:

`func`	`NIVXI_HTRIG*`	Pointer to the current trigger interrupt handler for a specified trigger line `NULL` = Invalid `line` or no hardware support

Example:

```
/* Get the address of the trigger interrupt handler for TTL
   trigger line 4. */

NIVXI_HTRIG    *func;
UINT16         line;

line = 4;
func = GetTrigHandler (line);
```

MapTrigToTrig

Syntax: `ret = MapTrigToTrig (controller, srcTrig, destTrig, mode)`

Action: Maps the specified TTL, ECL, Star X, Star Y, external connection (GPIO), or miscellaneous signal line to another. The support actually present is completely hardware dependent and is reflected in the error status and in hardware-specific documentation.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to map signal lines
`srcTrig`	`UINT16`	Source line to map to destination
`destTrig`	`UINT16`	Destination line to map from source

Value	Source or Destination Line
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
14 to 26	Star X lines 0 to 12 *
27 to 39	Star Y lines 0 to 12 *
40 to 49	External source/destination (GPIOs 0 to 9)
40	Front panel In (connector 1)
41	Front panel Out (connector 2)
42	ECL bypass from front panel
43	Connection to EXTCLK input pin
44 to 49	Hardware-dependent GPIOs 4 to 9
50	TIC counter pulse output (TCNTR)
51	TIC counter finished output (GCNTR)
60	TIC TICK1 tick timer output
61	TIC TICK2 tick timer output

`mode`	`UINT16`	Signal conditioning mode (0 = no conditioning)

Bit	Conditioning Effect
0	Synchronize with next CLK edge
1	Invert signal polarity
2	Pulse stretch to one CLK minimum
3	Use EXTCLK (not CLK10) for conditioning

All other values are reserved for future expansion.

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Unsupported function, no mapping capability
-2 = Invalid `controller`
-8 = Unsupported `srcTrig`
-9 = Unsupported `destTrig`
-10 = Unsupported `mode`
-11 = Already mapped, must use `UnMapTrigToTrig`

Note**: ***Star X and Star Y are not currently supported lines.

Example:

```
/* Map TTL line 4 on the local CPU (or first extended controller)
   to go out of the front panel with no signal conditioning. */

INT16      controller;
UINT16     srcTrig;
UINT16     destTrig;
UINT16     mode;
INT16      ret;

controller = -1;  /* Local CPU */
src = 4;          /* TTL line 4. **/
dest = 41;        /* Front panel out connector. **/
mode = 0;         /* No conditioning. */
ret = MapTrigToTrig (controller, srcTrig, destTrig, mode);
```

SetTrigHandler

Note: ***This function call may not exist on some platforms that do not have the TIC chip. If this is the case, you can achieve the same functionality by using the name*** *SetTTLtrigHandler* ***or*** *SetECLtrigHandler* ***with the same parameters as described below.***

Syntax: `ret = SetTrigHandler (lines, func)`

Action: Replaces the current TTL/ECL trigger, counter, or tick timer interrupt handler for a specified trigger source with the specified function, `func`.

Remarks: Input parameters:

`lines`	`UINT16`	Bit vector of trigger lines (1 = set, 0 = do not set)

Value	Trigger Line(s) to Set
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
14	TIC counter
15	TIC tick timers

`func`	`NIVXI_HTRIG*`	Pointer to the new trigger interrupt handler

0	= `DefaultTrigHandler`
1	= `DefaultTrigHandler2`
Other	= Address of new trigger interrupt handler

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = No hardware support

Example:

```
/* Set a trigger interrupt handler for TTL trigger line 4. */

NIVXI_HTRIG       func;
UINT16   lines;
INT16    ret;

lines = (UINT16)(1<<4);
ret = SetTrigHandler (lines, func);
if (ret < 0)
   /* An error occurred in SetTrigHandler . */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (INT16 controller, UINT16 line,
UINT16 type)
{
}
```

SrcTrig

Note: ***This function call may not exist on some platforms that do not have the TIC chip. If this is the case, you can achieve the same functionality by using the name** SrcTTLtrig **or** SrcECLtrig **with the same parameters as described below.***

Syntax: `ret = SrcTrig (controller, line, prot, timeout)`

Action: Sources the specified protocol on the specified TTL, ECL, or external trigger line on the specified controller.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to source trigger line
`line`	`UINT16`	Trigger line to source
	<u>Value</u>	<u>Trigger Line</u>
	0 to 7	TTL trigger lines 0 to 7
	8 to 13	ECL trigger lines 0 to 5
	40 to 49	External source/destination (GPIOs 0 to 9) *
	50	TIC counter **
	60	TIC TICK timers **
`prot`	`UINT16`	Protocol to use
		0 = ON
		1 = OFF
		2 = START
		3 = STOP
		4 = SYNC
		5 = SEMI-SYNC
		6 = ASYNC
		7 = SEMI-SYNC and wait for acknowledge
		8 = ASYNC and wait for acknowledge
		ffffh = Abort previous acknowledge pending (5 and 6)
`timeout`	`INT32`	Timeout value in milliseconds

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status
		0 = Successful
		-1 = Unsupportable function (no hardware support)
		-2 = Invalid `controller`
		-3 = Invalid `line` or `prot`
		-4 = `line` not supported
		-5 = `prot` not supported
		-6 = Timeout occurred waiting for acknowledge
		-7 = `line` already in use
		-12 = `line` not configured for use in sourcing
		-15 = Previous operation incomplete
		-16 = Previous acknowledge still pending

* Supports ON, OFF, START, STOP, and SYNC protocols only
** Supports SYNC and SEMI-SYNC protocols only

Example:

```
/* Source TTL line 4 on the local CPU (or the first extended
   controller) for SEMI-SYNC protocol. */

INT16      ret;
INT16      controller;
UINT16     line;
UINT16     prot;
INT32      timeout;

controller = -1;
line = 4;
prot = 5;
timeout = 0L;
ret = SrcTrig (controller, line, prot, timeout);
```

TrigAssertConfig

Syntax: `ret = TrigAssertConfig (controller, line, mode)`

Action: Configures the specified TTL/ECL trigger line assertion method. You can (re-)synchronize TTL/ECL triggers to CLK10 on a per-line basis. You can globally select on all TTL/ECL trigger lines whether to synchronize to the rising or falling edge of CLK10. In addition, you can specify a trigger line to partake in SEMI-SYNC accepting with external acknowledge.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to configure assertion mode
`line`	`UINT16`	Trigger line to configure

Value	Trigger Line
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
ffffh	General assertion configuration (all lines)

`mode`	`UINT16`	Configuration mode

Bit	Specific Line Configuration Modes
0	1 = Synchronize falling edge of CLK10 0 = Synchronize rising edge of CLK10

Bit	General Configuration Modes
1	1 = Pass trigger through asynchronously 0 = Synchronize with next CLK10 edge
2	1 = Participate in SEMI-SYNC with external trigger acknowledge protocol 0 = Do not participate

All other values are reserved for future expansion.

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `line`
-4 = `line` not supported
-10 = Invalid configuration mode

Example 1:

```
/* Configure all TTL/ECL trigger lines generally to synchronize to
   the falling edge of CLK10 (as opposed to the rising edge). */

INT16      ret;
INT16      controller;
UINT16     line;
UINT16     mode;

controller = -1;
line = 0xFFFF;
mode = (1<<0);
ret = TrigAssertConfig (controller, line, mode);
```

Example 2:

```
/* Configure TTL trigger line 4 to synchronize to CLK10 for any
   assertion method and do not participate in SEMI-SYNC. */

INT16      ret;
INT16      controller;
UINT16     line;
UINT16     mode;

controller = -1;
line = 4;
mode = 0;
ret = TrigAssertConfig (controller, line, mode);
```

TrigCntrConfig

Syntax: `ret = TrigCntrConfig (controller, mode, source, count)`

Action: Configures TIC chip internal 16-bit counter. Call `SrcTrig` or `EnableTrigSense` to actually start the counter. The input can be any trigger line, CLK10, or the EXTCLK connection. The counter has two outputs: TCNTR (one 100 ns pulse per input edge) and GCNTR (unasserted until count goes from 1 to 0, then asserted until counter reloaded or reset). You can use `MapTrigToTrig` to map TCNTR to any number of the TTL or ECL trigger lines, and to map GCNTR to any number of the external (GPIO) lines.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to configure the TIC counter
`mode`	`UINT16`	Configuration mode
	<u>Value</u>	<u>Configuration Mode</u>
	0	Initialize the counter
	2	Reload the counter leaving enabled
	3	Disable/abort any count in progress
`source`	`UINT16`	Trigger line to configure as input to counter
	<u>Value</u>	<u>Trigger Line</u>
	0 to 7	TTL trigger lines 0 to 7
	8 to 13	ECL trigger lines 0 to 5
	70	CLK10
	71	EXTCLK connection
`count`	`UINT16`	Number of input pulses to count before terminating

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `source` line
-10 = Invalid configuration mode
-12 = Counter not initialized
-15 = Previous count incomplete

Example:

```
/* Configure the counter to count 25 assertions on TTL trigger
   line 5 (the prot parameter when calling EnableTrigSense will
   determine whether the counter accepts SYNC or SEMI-SYNC
   assertions). */

INT16      ret;
INT16      controller;
UINT16     mode;
UINT16     source;
UINT16     count;

controller = -1;
mode = 0;              /* Initialize the counter */
source = 5;
count = 25;
ret = TrigCntrConfig (controller, mode, source, count);
```

TrigExtConfig

Syntax: `ret = TrigExtConfig (controller, extline, mode)`

Action: Configures the external trigger (GPIO) lines. You can feed back the external trigger lines for use in the crosspoint switch output. You can assert the external trigger lines high or low or leave them unconfigured (tristated) for use as a crosspoint switch input. If you do not feed the external input back, you can invert it before mapping it to a trigger line.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to configure the external connection
`extline`	`UINT16`	Trigger line to configure

Value	Trigger Line
40 to 49	External source/destination (GPIOs 0 to 9)
40	Front panel In (connector 1)
41	Front panel Out (connector 2)
42	ECL bypass from front panel
43	EXTCLK
44 to 49	Hardware-dependent GPIOs 4 to 9

`mode`	`UINT16`	Configuration mode

Bit	Configuration Modes
0	1 = Feed back any line mapped as input into the crosspoint switch 0 = Drive input to external (GPIO) pin
1	1 = Assert input (regardless of feedback) 0 = Leave input unconfigured
2	1 = If assertion selected, assert low 0 = If assertion selected, assert high
3	1 = Invert external input (not feedback) 0 = Pass external input unchanged

All other values are reserved for future expansion.

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `extline`
-10 = Invalid configuration mode

Example 1:

```
/* Configure external line 41 (front panel Out) to not be fed back
   and left tristated for use as a mapped output via
   MapTrigToTrig. */

INT16      ret;
INT16      controller;
UINT16     extline;
UINT16     mode;

controller = -1;
extline = 41;
mode = 0;
ret = TrigExtConfig (controller, extline, mode);
```

Example 2:

```
/* Configure external line 40 (front panel In) to not be fed back
   and left tristated for use as a mapped input via MapTrigToTrig.
   Invert the front panel In signal. */

INT16      ret;
INT16      controller;
UINT16     extline;
UINT16     mode;

controller = -1;
extline = 40;
mode = (1<<3);
ret = TrigExtConfig (controller, line, mode);
```

Example 3:

```
/* Configure external line 48 (GPIO 8) to be fed back for use as a
   crosspoint switch input and output via MapTrigToTrig. */

INT16      ret;
INT16      controller;
UINT16     extline;
UINT16     mode;

controller = -1;
extline = 48;
mode = (1<<0);
ret = TrigExtConfig (controller, line, mode);
```

TrigTickConfig

Syntax: `ret = TrigTickConfig (controller, mode, source, tcount1, tcount2)`

Action: Configures TIC chip internal dual 5-bit tick timers. Call `SrcTrig` or `EnableTrigSense` to actually start the tick timers. `SrcTrig` inhibits the TICK1 output from generating tick timer interrupts. `EnableTrigSense` enables the TICK1 output to generate tick timer interrupts. The input can be any external (GPIO) line, CLK10, or the EXTCLK connection. You can map the two tick timer outputs TICK1 and TICK2 to any number of TTL/ECL trigger lines. In addition, you can map the TICK2 output to any number of external (GPIO) lines.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to configure the TIC chip dual 5-bit tick timers
`mode`	`UINT16`	Configuration mode
	<u>Value</u>	<u>Configuration Mode</u>
	0	Initialize the tick timers (rollover mode)
	1	Initialize the tick timers (non-rollover mode)
	2	Reload the tick timers leaving enabled
	3	Disable/abort any count in progress
`source`	`UINT16`	Trigger line to configure as input to counter
	<u>Value</u>	<u>Trigger Line</u>
	40 to 49	External source/destination (GPIOs 0 to 9)
	70	CLK10
	71	EXTCLK connection
`tcount1`	`UINT16`	Number of input pulses (as a power of two) to count before asserting TICK1 output (and terminating the tick timer if configured for non-rollover mode)
`tcount2`	`UINT16`	Number of input pulses (as a power of two) to count before asserting TICK2 output

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

3 = Successful disable of the tick timers
2 = Successful reload of the tick timers
1 = Successful initialization of non-rollover mode
0 = Successfu1 initialization of rollover mode
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `source` line
-10 = Invalid `mode`
-13 = Invalid `tcount1` or `tcount2`
-15 = Previous tick configured and enabled

Example 1:

```
/* Configure the tick timers to interrupt every 6.55 milliseconds
   by dividing down CLK10 as an input. Call EnableTrigSense to
   start the tick timers and enable interrupts. */

INT16      ret;
INT16      controller;
UINT16     mode;
UINT16     source;
UINT16     tcount1, tcount2;

controller = -1;
mode = 0;               /* Initialize with rollover */
source = 70;      /* CLK10 */
tcount1 = 16;           /* Divide down by 65536 (2^16) */
tcount2 = 0;      /* Does not matter */
ret = TrigTickConfig (controller, mode, source, tcount1, tcount2);
```

Example 2:

```
/* Configure the tick timers to output a continuous 9.765 kHz
   square wave on TICK1 output and a 1.25 MHz clock on TICK2
   output by dividing down CLK10 as an input.  Call SrcTrig to
   start the tick timers. */

INT16      ret;
INT16      controller;
UINT16     mode;
UINT16     source;
UINT16     tcount1, tcount2;

controller = -1;
mode = 0;               /* Initialize with rollover */
source = 70;      /* CLK10 */
tcount1 = 10;           /* Divide down by 1024 (2^10) */
tcount2 = 3;      /* Divide down by 8 (2^3)*/
ret = TrigTickConfig (controller, mode, source, tcount1, tcount2);
```

UnMapTrigToTrig

Syntax: `ret = UnMapTrigToTrig (controller, srcTrig, destTrig)`

Action: Unmaps the specified TTL, ECL, Star X, Star Y, external connection (GPIO), or miscellaneous signal line that was mapped to another line using the `MapTrigToTrig` function.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to unmap signal lines
`srcTrig`	`UINT16`	Source line to unmap from destination
`destTrig`	`UINT16`	Destination line mapped from source

Value	Source or Destination
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
14 to 26	Star X lines 0 to 12 *
27 to 39	Star Y lines 0 to 12 *
40 to 49	External source/destination (GPIOs 0 to 9)
40	Front panel In (connector 1)
41	Front panel Out (connector 2)
42	ECL bypass from front panel
43	Connection to EXTCLK input pin
44 to 49	Hardware-dependent GPIOs 4 to 9
50	TIC counter pulse output (TCNTR)
51	TIC counter finished output (GCNTR)
60	TIC TICK1 tick timer output
61	TIC TICK2 tick timer output

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Unsupported function, no mapping capability
-2 = Invalid `controller`
-12 = Not previously mapped

Note**: ***Star X and Star Y are not currently supported lines.

Example:

```
/* Unmap route of TTL line 4 on the local CPU (or first extended
   controller) to go out of the front panel as mapped by
   MapTrigToTrig(). */

INT16      controller;
UINT16     srcTrig;
UINT16     destTrig;
INT16      ret;

controller = -1;   /* Local CPU. */
src = 4;           /* TTL line 4. */
dest = 49;         /* Front panel out connector. */
ret = UnMapTrigToTrig (controller, srcTrig, destTrig);
```

WaitForTrig

Note: ***This function call may not exist on some platforms that do not have the TIC chip. If this is the case, you can achieve the same functionality by using the name*** *WaitForTTLtrig* ***or*** *WaitForECLtrig* ***with the same parameters as described below.***

Syntax: `ret = WaitForTrig (controller, line, timeout)`

Action: Waits for the specified trigger line to be sensed on the specified controller for the specified time. `EnableTrigSense` must be called to sensitize the hardware to the particular trigger protocol to be sensed.

Remarks: Input parameters:

`controller`	`INT16`	Controller on which to wait for trigger
`line`	`UINT16`	Trigger line to wait on
	Value	Trigger Line
	0 to 7	TTL trigger lines 0 to 7
	8 to 13	ECL trigger lines 0 to 5
	50	TIC counter
	60	TIC TICK1 tick timer
`timeout`	`INT32`	Timeout value in milliseconds

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status

0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `line`
-4 = `line` not supported
-6 = Timeout occurred
-12 = `line` not configured for sensing

Example:

```
/* Wait up to 10 seconds for TTL line 4 on the local CPU (or the
   first extended controller) to be encountered. */

INT16      ret;
INT16      controller;
UINT16     line;
INT32      timeout;

controller = -1;
line = 4;
timeout = 10000L;
ret = WaitForTrig (controller, line, timeout);
```

Default Handlers for VXI Trigger Functions

The NI-VXI software provides the following default handlers for the VXI trigger 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 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.

DefaultTrigHandler

Note: ***This function call may not exist on some platforms that do not have the TIC chip. If this is the case, you can achieve the same functionality by using the name** *DefaultTTLtrigHandler* **or** *DefaultECLtrigHandler* **with the same parameters as described below.***

Syntax: `DefaultTrigHandler (controller, line, type)`

Action: Handles the VXI triggers on specified trigger lines. Calls the `AcknowledgeTrig` function to acknowledge the trigger interrupt if the `type` parameter specifies trigger sensed. Otherwise, the interrupt is ignored.

Remarks: Input parameters:

`controller`	`INT16`	Controller from which the trigger interrupt is received
`line`	`UINT16`	Trigger line interrupt received on
	<u>Value</u>	<u>Trigger Line</u>
	0 to 7	TTL trigger lines 0 to 7
	8 to 13	ECL trigger lines 0 to 5
	50	TIC counter
	60	TIC TICK1 tick timer
`type`	`UINT16`	Conditioning Effect
	<u>Bit</u>	<u>Conditioning Effect</u>
	0	0 = Sourced trigger acknowledged 1 = Trigger sensed
	2	1 = Assertion edge overrun occurred
	3	1 = Unassertion edge overrun occurred
	4	1 = Pulse stretch overrun occurred
	15	1 = Error summary (2, 3, 4 = 1)

Output parameters:

none

Return value:

none

DefaultTrigHandler2

Syntax: `DefaultTrigHandler2 (controller, line, type)`

Action: Handles the VXI triggers on specified trigger lines. This trigger interrupt handler performs no operations. Any triggers that require acknowledgments must be acknowledged at the application level.

Remarks: Input parameters:

`controller`	`INT16`	Controller from which the trigger interrupt is received
`line`	`UINT16`	Trigger line interrupt received on

Value	Trigger Line
0 to 7	TTL trigger lines 0 to 7
8 to 13	ECL trigger lines 0 to 5
50	TIC counter
60	TIC TICK1 tick timer

`type`	`UINT16`	Conditioning Effect

Bit	Conditioning Effect
0	0 = Sourced trigger acknowledged 1 = Trigger sensed
2	1 = Assertion edge overrun occurred
3	1 = Unassertion edge overrun occurred
4	1 = Pulse stretch overrun occurred
15	1 = Error summary (2, 3, 4 = 1)

Output parameters:

none

Return value:

none

Chapter 12
System Interrupt Handler Functions

This chapter describes the C syntax and use of the VXI system interrupt handler functions and default handlers. With these functions, you can handle miscellaneous system conditions that can occur in the VXI environment, such as Sysfail, ACfail, Sysreset, Bus Error, and/or Soft Reset interrupts. The NI-VXI software interface can handle all of these system conditions for the application through the use of 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). The corresponding enable function for each handler must be called in order 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). The VXIbus specification requires that all VXI Commanders monitor the PASSed or FAILed state of their VXI Servants. When a VXIbus device is in the FAILed state, the failed device clears its PASS bit (in its Status register) and asserts the SYSFAIL* signal on the VXIbus backplane. When a Sysfail condition is detected on the local CPU, an interrupt is generated, and the current Sysfail interrupt handler is called. The failed Servant device must be forced offline or brought back online in an orderly fashion.

DisableSysfail (controller)

`DisableSysfail` desensitizes the application to Sysfail interrupts from embedded controller or extended controller(s) Sysfail conditions (dependent on the hardware platform). The VXIbus specification requires that all VXI Commanders monitor the PASSed or FAILed state of their VXI Servants. When a VXIbus device is in the FAILed state, the failed device clears its PASS bit (in its Status register) and asserts the SYSFAIL* signal on the VXIbus backplane.

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

EnableACfail (controller)

`EnableACfail` sensitizes the application to ACfail interrupts from embedded controller or extended controller(s) ACfail conditions (dependent on the hardware platform). The VXIbus specification allows for a minimum amount of time after a power failure condition occurs for the system to remain operational. The detection of the power failure asserts the VXIbus 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 VXIbus specification allows for a minimum amount of time after a power failure condition occurs for the system to remain operational. The detection of the power failure asserts the VXIbus 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 VXIbus specification allows for a minimum amount of time after a power failure condition occurs for the system to remain operational. The detection of a power failure in a 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.

EnableSoftReset ()

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

DisableSoftReset ()

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

SetSoftResetHandler (func)

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

GetSoftResetHandler ()

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

DefaultSoftResetHandler ()

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

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

GetBusErrorHandler ()

`GetBusErrorHandler` returns the address of the current Bus Error interrupt handler. During an access to the VXIbus, the BERR* signal (Bus Error) is asserted to end the bus cycle if the address or mode of access is determined to be invalid. The Bus Error exception condition generates an exception on the local CPU, which can be trapped by the Bus Error handler. Your application should include a retry mechanism if it is possible for a particular access to generate Bus Errors at times and valid results at other times. The `InitVXIlibrary` function automatically installs a default handler, `DefaultBusErrorHandler`, when it initializes the NI-VXI software. It 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 `InitVXIlibrary` initializes the NI-VXI software. During an access to the VXIbus, 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 VXIbus backplane or from the specified extended controller VXI 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);
```

DisableSoftReset

Syntax: `ret = DisableSoftReset ()`

Action: Disables the local Soft Reset interrupt being generated from a write to the Reset bit of the local CPU Control register.

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status
		0 = Soft Reset interrupt successfully disabled -1 = Soft Reset interrupts not supported

Example:

```
/* Disable the Soft Reset interrupt. */

INT16      ret;

ret = DisableSoftReset ();
```

DisableSysfail

Syntax: `ret = DisableSysfail(controller)`

Action: Desensitizes the local CPU from interrupts generated from Sysfail conditions on the embedded CPU VXIbus backplane or from the specified extended controller VXI 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 VXIbus backplane or from the specified extended controller VXI 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 VXIbus backplane or from the specified extended controller VXI 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);
```

EnableSoftReset

Syntax: `ret = EnableSoftReset ()`

Action: Enables the local Soft Reset interrupt being generated from a write to the Reset bit of the local CPU Control register.

Remarks: Parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Soft Reset interrupt successfully enabled -1 = Soft Reset interrupts not supported

Example:

```
/* Enable the Soft Reset interrupt. */

INT16      ret;

ret = EnableSoftReset ();
```

EnableSysfail

Syntax: `ret = EnableSysfail (controller)`

Action: Sensitizes the local CPU to interrupts generated from Sysfail conditions on the embedded CPU VXIbus backplane or from the specified extended controller VXI 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 VXIbus backplane or from the specified extended controller's VXI 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`	`NIVXI_HACFAIL*`	Pointer to the current ACfail interrupt handler `NULL` = ACfail interrupt not supported

Example:

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

NIVXI_HACFAIL     *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`	`NIVXI_HBUSERROR*`	Pointer to the current Bus Error interrupt handler

Example:

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

NIVXI_HBUSERROR   *func;

func = GetBusErrorHandler();
```

GetSoftResetHandler

Syntax: `func = GetSoftResetHandler ()`

Action: Returns the address of the current Soft Reset interrupt handler.

Remarks: Parameters:

none

Return value:

`func`	`NIVXI_HSOFTRESET*`	Pointer to the current Soft Reset interrupt handler `NULL` = Soft Reset interrupt not supported

Example:

```
/* Get the address of the Soft Reset handler. */

NIVXI_HSOFTRESET  *func;

func = GetSoftResetHandler();
```

GetSysfailHandler

Syntax: `func = GetSysfailHandler ()`

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

Remarks: Parameters:

none

Return value:

`func`	`NIVXI_HSYSFAIL*`	Pointer to the current Sysfail interrupt handler `NULL` = Sysfail interrupt not supported

Example:

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

NIVXI_HSYSFAIL    *func;

func = GetSysfailHandler ();
```

GetSysresetHandler

Syntax: `func = GetSysresetHandler ()`

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

Remarks: Parameters:

none

Return value:

`func`	`NIVXI_HSYSRESET*`	Pointer to the current SYSRESET* interrupt handler `NULL` = SYSRESET* interrupt not supported

Example:

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

NIVXI_HSYSRESET   *func;

func = GetSysresetHandler();
```

SetACfailHandler

Syntax: `ret = SetACfailHandler (func)`

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

Remarks: Input parameter:

`func`	`NIVXI_HACFAIL*`	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. */

NIVXI_HACFAIL      func;
INT16    ret;

ret = SetACfailHandler (func);
if (ret < 0)
   /* An error occurred in SetACfailHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (INT16 controller)
{
}
```

SetBusErrorHandler

Syntax: `ret = SetBusErrorHandler(func)`

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

Remarks: Input parameter:

`func`	`NIVXI_HBUSERROR*`	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. */

NIVXI_HBUSERROR   func;
INT16    ret;

ret = SetBusErrorHandler(func);
if (ret < 0)
   /* An error occurred in SetBusErrorHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func ()
{
}
```

SetSoftResetHandler

Syntax: `ret = SetSoftResetHandler (func)`

Action: Replaces the current Soft Reset interrupt handler with a specified handler.

Remarks: Input parameter:

`func`	`NIVXI_HSOFTRESET*`	Pointer to the new Soft Reset interrupt handler `NULL` = `DefaultSoftResetHandler`

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Soft Reset interrupt not supported

Example:

```
/* Set the Soft Reset handler. */

NIVXI_HSOFTRESET  func;
INT16    ret;

ret = SetSoftResetHandler (func);
if (ret < 0)
   /* An error occurred in SetSoftResetHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func ()
{
}
```

SetSysfailHandler

Syntax: `ret = SetSysfailHandler (func)`

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

Remarks: Input parameter:

`func`	`NIVXI_HSYSFAIL*`	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. */

NIVXI_HSYSFAIL    func;
INT16    ret;

ret = SetSysfailHandler (func);
if (ret < 0)
   /* An error occurred in SetSysfailHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (INT16 controller)
{
}
```

SetSysresetHandler

Syntax: `ret = SetSysresetHandler (func)`

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

Remarks: Input parameter:

`func`	`NIVXI_HSYSRESET*`	Pointer to the new SYSRESET* interrupt handler NULL = `DefaultSysresetHandler`

Output parame ters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = SYSRESET* interrupt not supported

Example:

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

NIVXI_HSYSRESET   func;
INT16    ret;

ret = SetSysresetHandler (func);
if (ret < 0)
   /* An error occurred in SetSysresetHandler. */;


/* Example handler */
NIVXI_HQUAL void NIVXI_HSPEC func (INT16 controller)
{
}
```

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 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

DefaultSoftResetHandler

Syntax: `DefaultSoftResetHandler ()`

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

Remarks: Parameters:
none

Return value:
none

DefaultSysfailHandler

Syntax: `DefaultSysfailHandler (controller)`

Action: Handles the interrupt generated when the SYSFAIL* signal on the VXI backplane is asserted. If a Servant is detected to have failed (as indicated when its PASS bit is cleared), the default Sysfail handler sets that Servant's Sysfail Inhibit bit and optionally sets its Reset bit. In addition, the global variable `SysfailRecv` is incremented.

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 VXI 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 13
VXIbus Extender Functions

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

Functional Overview

The following paragraphs describe the VXIbus extender 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.

MapECLtrig (extender, lines, directions)

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

MapTTLtrig (extender, lines, directions)

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

MapUtilBus (extender, modes)

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

MapVXIint (extender, levels, directions)

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

Function Descriptions

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

MapECLtrig

Syntax: `ret = MapECLtrig (extender, lines, directions)`

Action: Maps the specified ECL trigger lines for the specified mainframe in the specified direction (into or out of the mainframe).

Remarks: Input parameters:

`extender`	`INT16`	Mainframe extender for which to map ECL lines
`lines`	`UINT16`	Bit vector of ECL trigger lines. Bits 5 to 0 correspond to ECL lines 5 to 0, respectively. 1 = Enable for appropriate line 0 = Disable for appropriate line
`directions`	`UINT16`	Bit vector of directions for ECL lines. Bits 5 to 0 correspond to ECL lines 5 to 0, respectively. 1 = Into the mainframe 0 = Out of the mainframe

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Unsupportable function (no hardware support) -2 = Invalid `extender`

Example:

```
/* Map ECL lines 0 and 1 on the mainframe extender at Logical
   Address 5 to go into the mainframe. */

INT16      extender;
UINT16     lines;
UINT16     directions;
INT16      ret;

extender = 5;
lines = (UINT16)((1<<0) | (1<<1));  /** ECL lines 0 and 1. **/
directions = (UINT16)((1<<0) | (1<<1));
ret = MapECLtrig (extender, lines, directions);
```

MapTTLtrig

Syntax: `ret = MapTTLtrig (extender, lines, directions)`

Action: Maps the specified TTL trigger lines for the specified mainframe in the specified direction (into or out of the mainframe).

Remarks: Input parameters:

`extender`	`INT16`	Mainframe extender for which to map TTL lines
`lines`	`UINT16`	Bit vector of TTL trigger lines. Bits 7 to 0 correspond to TTL lines 7 to 0, respectively. 1 = Enable for appropriate line 0 = Disable for appropriate line
`directions`	`UINT16`	Bit vector of directions for TTL lines. Bits 7 to 0 correspond to TTL lines 7 to 0, respectively. 1 = Into the mainframe 0 = Out of the mainframe

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Unsupportable function (no hardware support) -2 = Invalid `extender`

Example:

```
/* Map TTL lines 4 and 5 on the mainframe extender at Logical
   Address 5 to go out of the mainframe. */

INT16      extender;
UINT16     lines;
UINT16     directions;
INT16      ret;

extender = 5;
lines = (UINT16)((1<<4) | (1<<5));  /** TTL lines 4, 5. **/
directions = (UINT16)0x0000;
ret = MapTTLtrig (extender, lines, directions);
```

MapUtilBus

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

Action: Maps the specified VXI utility bus signal for the specified mainframe into and/or out of the mainframe. 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 Mainframe 5.  Map Sysreset into and out of
   Mainframe 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 VXI interrupt levels for the specified mainframe in the specified direction (into or out of the mainframe).

Remarks: Input parameters:

`extender`	`INT16`	Mainframe extender for which to map VXI interrupt levels
`levels`	`UINT16`	Bit vector of VXI interrupt levels. Bits 6 to 0 correspond to VXI interrupt levels 7 to 1 respectively. 1 = Enable for appropriate level 0 = Disable for appropriate level
`directions`	`UINT16`	Bit vector of directions for VXI interrupt levels. Bits 6 to 0 correspond to VXI interrupt levels 7 to 1, respectively. 1 = Into the mainframe 0 = Out of the mainframe

Output parameters:

none

Return value:

`ret`	`INT16`	Return Status 0 = Successful -1 = Unsupportable function (no hardware support) -2 = Invalid `extender`

Example:

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

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. Filling out a copy of the *Technical Support Form* from your Getting Started manual 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
Mexico	95 800 010 0793	95 800 010 0793
Netherlands	03480-33466	03480-30673
Norway	32-848400	32-848600
Singapore	2265886	2265887
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
Taiwan	02 377 1200	02 737 4644
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™ Software Reference Manual for C**

Edition Date: **October 1994**

Part Number: **371693A-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.
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange. A 7-bit standard code adopted to facilitate the interchange of data among various types of data processing and data communications equipment.
ASIC	Application-Specific Integrated Circuit (a custom chip)
asserted	A signal in its active true state.
asynchronous	Not synchronized; not controlled by periodic time signals, and therefore unpredictable with regard to the timing of execution of commands.

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

B

backplane	An assembly, typically a PCB, with 96-pin connectors and signal paths that bus the connector pins. A C-size VXIbus system will have two sets of bused connectors called the J1 and J2 backplanes. A D-size VXIbus system will have three sets of bused connectors called the J1, J2, and J3 backplane.
base address	A specified address that is combined with a *relative* address (or offset) to determine the *absolute* address of a data location. All VXI address windows have an associated base address for their assigned VXI address spaces.
BAV	Word Serial Byte Available command. Used to transfer 8 bits of data from a Commander to its Servant under the Word Serial Protocol.
BERR*	Bus Error signal. This signal is asserted by either a slave device or the BTO unit when an incorrect transfer is made on the Data Transfer Bus (DTB). The BERR* signal is also used in VXI for certain protocol implementations such as writes to a full Signal register and synchronization under the Fast Handshake Word Serial Protocol.
binary	A numbering system with a base of 2.
bit	Binary digit. The smallest possible unit of data: a two-state, yes/no, 0/1 alternative. The building block of binary coding and numbering systems. Several bits make up a *byte*.
bit vector	A string of related bits in which each bit has a specific meaning.
BREQ	Word Serial Byte Request query. Used to transfer 8 bits of data from a Servant to its Commander under the Word Serial Protocol.
BTO	See *Bus Timeout Unit*.
buffer	Temporary memory/storage location for holding data before it can be transmitted elsewhere.
bus master	A device that is capable of requesting the Data Transfer Bus (DTB) for the purpose of accessing a slave device.
bus timeout unit	A VMEbus functional module that times the duration of each data transfer on the Data Transfer Bus (DTB) and terminates the DTB cycle if the duration is excessive. Without the termination capability of this module, a bus master could attempt to access a nonexistent slave, resulting in an indefinitely long wait for a slave response.
byte	A grouping of adjacent binary digits operated on by the computer as a single unit. 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

clearing	Replacing the information in a register, storage location, or storage unit with zeros or blanks.
CLK10	A 10 MHz, ± 100 ppm, individually buffered (to each module slot), differential ECL system clock that is sourced from Slot 0 and distributed to Slots 1 through 12 on P2. It is distributed to each slot as a single-source, single-destination signal with a matched delay of under 8 ns.
command	A directive to a device. In VXI, three types of commands are as follows: In Word Serial Protocol, a 16-bit imperative to a servant from its Commander (written to the Data Low register); In Shared Memory Protocol, a 16-bit imperative from a client to a server, or vice versa (written to the Signal register); In Instrument devices, an ASCII-coded, multi-byte directive.
Commander	A Message-Based device which is also a bus master and can control one or more Servants.
communications registers	In Message-Based devices, a set of registers that are accessible to the device's Commander and are used for performing Word Serial Protocol communications.
configuration registers	A set of registers through which the system can identify a module device type, model, manufacturer, address space, and memory requirements. In order to support automatic system and memory configuration, the VXIbus specification requires that all VXIbus devices have a set of such registers.
controller	An intelligent device (usually involving a CPU) that is capable of controlling other devices.
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.
ERR	Protocol error
Event signal	A 16-bit value written to a Message-Based device's Signal register in which the most significant bit (bit 15) is a 1, designating an Event (as opposed to a Response signal). The VXI specification reserves half of the Event values for definition by the VXI Consortium. The other half are user defined.
Extended Class device	A class of VXIbus device defined for future expansion of the VXIbus specification. These devices have a subclass register within their configuration space that defines the type of extended device.
Extended 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
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*.
I/O	Input/output; the techniques, media, or devices used to achieve communication between entities.
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.
INTX	Interrupt and Timing Extension; a daughter card option for MXI mainframe extenders that extends interrupt lines and reset signals on VME boards. On VXI boards it also extends trigger lines and the VXIbus CLK10 signal.

K

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

L

LF	Linefeed; the ASCII character 0Ah.

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

M

MB	1,048,576 or 2^{20}
mapping	Establishing a range of address space for a one-to-one correspondence between each address in the window and an address in VXIbus memory.
master	A functional part of a MXI/VME/VXIbus device that initiates data transfers on the backplane. A transfer can be either a read or a write.
megabyte	A million bytes.
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

s	Seconds
SEMI-SYNC Protocol	A one-line, open collector, multiple-device handshake trigger protocol.
Servant	A device controlled by a Commander.
setting	To place a binary cell into the 1 (non-zero) state.
Shared Memory Protocol	A communications protocol for Message-Based devices that uses a block of memory that is accessible to both a client and a server. The memory block acts as the medium for the protocol transmission.
short integer	Data type of 16 bits, same as *word*.
signal	Any communication between Message-Based devices consisting of a write to a Signal register. Sending a signal requires that the sending device have VMEbus master capability.
signed integer	n bit pattern, interpreted such that the range is from $-2^{(n-1)}$ to $+2^{(n-1)} -1$.
slave	A functional part of a MXI/VME/VXIbus device that detects data transfer cycles initiated by a VMEbus master and responds to the transfers when the address specifies one of the device's registers.

SMP	See *Shared Memory Protocol*.
SRQ	Service Request
status/ID	A value returned during an IACK cycle. In VME, usually an 8-bit value which is either a status/data value or a vector/ID value used by the processor to determine the source. In VXI, a 16-bit value used as a data; the lower 8 bits form the VXI logical address of the interrupting device and the upper 8 bits specify the reason for interrupting.
STST	START/STOP trigger protocol; a one-line, multiple-device protocol 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