LabWindows® VXI
Library Reference Manual

Version 2.3

March 1995 Edition

Part Number 320318-01

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (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 65 33 70,
Germany 089 714 50 93, Hong Kong 02 26375019, Italy 02 48301892, Japan 03 3788 1921, Korea 02 596-7456,
Mexico 05 202 2544, Netherlands 01720 45761, Norway 03 846866, Spain 91 640 0085, Sweden 08 730 49 70,
Switzerland 056 27 00 20, U.K. 0635 523545

Limited Warranty

The GPIB-PCII is warranted against defects in materials and workmanship for a period of two years from the date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace equipment that proves to be defective during the warranty period. This warranty includes parts and labor.

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

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

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

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

Copyright

Trademarks

NI-VXI™ and TIC™ are trademarks of National Instruments Corporation.

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

Warning Regarding Medical and Clinical Use of National Instruments Products

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

Contents

Tables

About This Manual

The *LabWindows VXI Library Reference Manual* describes the functions in the LabWindows VXI Library. The *LabWindows VXI Library Reference Manual* is intended for use by VXI users who are familiar with LabWindows and DOS fundamentals. This manual assumes that you are familiar with the material presented in the *LabWindows User Manual*, that LabWindows is already installed on your computer, and that you are familiar with the LabWindows software. Please refer to the *LabWindows User Manual* for specific instructions on operating LabWindows.

Organization of This Manual

The *LabWindows VXI Library Reference Manual* is organized as follows:

- Chapter 1, *VXI Library Overview*, contains information about the VXI Library package, a brief product overview, the procedure for installing the VXI Library, and general information about the VXI Library functions and panels. We recommend that you begin by reading this section before using the VXI Library.

- Chapter 2, *System Configuration Functions*, describes the functions in the LabWindows VXI System Configuration Library. LabWindows uses these functions to 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 3, *Commander Word Serial Protocol Functions*, describes the functions in the LabWindows VXI Commander Word Serial Protocol Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 4, *Servant Word Serial Protocol Functions*, describes the functions in the LabWindows VXI Servant Word Serial Protocol Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 5, *Low-Level VXIbus Access Functions*, describes the functions in the LabWindows VXI Low-Level VXIbus Access Library. Low-level VXIbus access is the fastest access method for directly reading from or writing to any of the VXIbus address spaces. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 6, *High-Level VXIbus Access Functions*, describes the functions in the LabWindows VXI High-Level VXIbus Access Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 7, *Local Resource Access Functions*, describes the functions in the LabWindows VXI Local Resource Access Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 8, *VXI Signal Functions*, describes the functions in the LabWindows VXI Signal Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 9, *VXI Interrupt Functions*, describes the functions in the LabWindows VXI Interrupt Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 10, *VXI Trigger Functions*, describes the functions in the LabWindows VXI Trigger Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 11, *System Interrupt Handler Functions*, describes functions in the LabWindows VXI System Interrupt Handler Library. With these functions, you can handle miscellaneous system conditions that can occur in the VXI environment. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- Chapter 12, *VXIbus Extender Functions*, describes functions in the LabWindows VXIbus Extender Library. These functions can be used to dynamically reconfigure multi-mainframe transparent mapping of the VXI interrupt and trigger lines and utility bus signals. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

- The Appendix, *Customer Communication*, contains forms for you to complete to facilitate communication with National Instruments concerning our products.

- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.

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

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`	Lowercase text in this font denotes text or characters that are to be literally input from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, device names, VXI function names, variables, parameters, filenames, and extensions, and for statements and comments taken from program code.

Numbers in this manual are base 10 unless noted as follows:

- Binary numbers are indicated by a -b suffix (for example, 11010101b).
- 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").

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

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

Related Documentation

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

- *NI-VXI Software Reference Manual for C*, National Instruments part number 320307-01
- *NI-VXI Software Reference Manual for BASIC*, National Instruments part number 320328-01
- *IEEE Standard for a Versatile Backplane Bus: VMEbus*, ANSI/IEEE Standard 1014-1987
- VXI-1, *VXIbus System Specification*, Revision 1.3, VXIbus Consortium
- VXI-6, *VXIbus Mainframe Extender Specification*, Revision 1.0, VXIbus Consortium

Customer Communication

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

Chapter 1
VXI Library Overview

This chapter contains information about the LabWindows for DOS VXI Library package, a brief product overview, the procedure for installing the VXI Library, and general information about the VXI Library functions and panels.

You should use this manual in conjunction with the NI-VXI software reference manual, either for C or for BASIC, that was shipped with your VXI hardware. Before you use the VXI Library, National Instruments recommends that you read this chapter, the `VXI.DOC` readme file in the LabWindows directory, and both Chapter 1, *Introduction to VXI*, and Chapter 2, *Introduction to NI-VXI Functions*, in your NI-VXI C or BASIC software reference manual.

The LabWindows VXI Library Package

The LabWindows VXI Library software package consists of one 5.25 in. or one 3.5 in. diskette and one manual, the *LabWindows VXI Library Reference Manual*, part number 320318-01.

Please review the contents of the package and contact National Instruments if anything is missing.

The LabWindows VXI Library package contains a Customer Registration Form. Please fill out this form and return it to National Instruments. This will entitle you to receive product upgrades and technical support.

Installing the VXI Library

Begin by making a backup copy of the LabWindows VXI distribution disk. Copy the disk onto a backup disk and store the distribution disk in a safe place.

To install LabWindows on your hard disk, insert the LabWindows VXI Library Program Disk into your computer and enter the following command at the DOS prompt:

```
x:\setup
```

where you replace x with the letter to indicate the disk drive you used.

The `SETUP` program prompts you for information, including the drive letter and directory in which you have installed the standard LabWindows package. It also verifies that your disk has enough space to hold the LabWindows VXI Library files.

As the `SETUP` program executes, it copies LabWindows files into the existing LabWindows directories. Table 1-1 shows the directories affected by `SETUP`.

Table 1-1. LabWindows Directories

Directory Name	Contents
`\LW`	System files
`\LW\INCLUDE`	Include files associated with libraries
`\LW\LIBRARY`	Library files for linking Microsoft C and BASIC programs
`\LW\BORLAND`	Library files for linking Borland C programs

After `SETUP` has successfully executed, follow the software configuration steps outlined in the *Getting Started* manual that came with your hardware.

LabWindows VXI Library Overview

The LabWindows VXI Library is an interface to VXI instruments from LabWindows. The VXI Library includes functions for Commander and Servant Word Serial Protocol, low-level and high-level VXIbus access, local resource access, VXI signals, interrupts, and triggers, system interrupt handlers, and system configuration.

The VXI Library Functions

The VXI Library functions are grouped in a tree structure according to the types of operations performed. Table 1-2 shows the VXI Library function tree. Table 1-3 lists C functions that you can use in standalone C Programs and `.obj` instrument programs inside the environment. See the LabWindows Instrument Library Developer's Guide (Part No. 320315-01) for more information on `.obj` instrument programs. Table 1-4 lists the names of older VXI trigger functions that are available for backward compatibility.

Table 1-2. The VXI Library Function Tree

VXI	
System Configuration Functions	
Initialize VXI Library	***InitVXIlibrary***
Close VXI Library	***CloseVXIlibrary***
Find Device's Logical Address	***FindDevLA***
Get Device Information, Long Integer Fields	***GetDevInfoLong***
Get Device Information, Short Integer Fields	***GetDevInfoShort***
Get Device Information, String fields	***GetDevInfoStr***
Set Device Information, Long Integer Fields	***SetDevInfoLong***
Set Device Information, Short Integer Fields	***SetDevInfoShort***
Set Device Information, String Fields	***SetDevInfoStr***
Create Device Information Entry	***CreateDevInfo***
Commander Word Serial Protocol Functions	
Read Series of Bytes/Characters	***WSrd***
Read Series of Short Integers	***WSrdi***
Read Series of Long Integers	***WSrdl***
Read into a File	***WSrdf***
Write Series of Bytes/Characters	***WSwrt***
Write Series of Short Integers	***WSwrti***
Write Series of Long Integers	***WSwrtl***
Write from a File	***WSwrtf***
Send Command	***WScmd***
Retrieve Query Response	***WSresp***
Send Trigger Command	***WStrg***
Send Clear Command	***WSclr***
Abort Operation	***WSabort***
Send Longword Command	***WSLcmd***
Retrieve Longword Query Response	***WSLresp***
Send Extended Command	***WSEcmd***
Set Timeout Value	***WSsetTmo***
Get Timeout Value	***WSgetTmo***

(continues)

Table 1-2. The VXI Library Function Tree (Continued)

Servant Word Serial Protocol Functions	
Enable Servant-Side Interrupts	***WSSenable***
Disable Servant-Side Interrupts	***WSSdisable***
Accept Series of Bytes	***WSSrd***
Accept Series of Shorts	***WSSrdi***
Accept Series of Longs	***WSSrdl***
Return Series of Bytes	***WSSwrt***
Return Series of Shorts	***WSSwrti***
Return Series of Longs	***WSSwrtl***
No Response to Command	***WSSnoResp***
Response to Query	***WSSsendResp***
No Longword Response to Command	***WSSLnoResp***
Longword Response to Query	***WSSLsendResp***
Abort Servant Operation	***WSSabort***
Generate Protocol Error	***GenProtError***
Respond to Read Protocol Error	***RespProtError***
Low-Level VXIbus Access Functions	
Map VXI Address	***MapVXIAddress***
Unmap VXI Address	***UnMapVXIAddress***
Get Window Range	***GetWindowRange***
Read Value	***VXIpeek***
Write Value	***VXIpoke***
Clear Bus Error	***ClearBusError***
Save Context	***SaveContext***
Restore Context	***RestoreContext***
Set Context	***SetContext***
Get Context	***GetContext***
Set Access Privilege	***SetPrivilege***
Get Access Privilege	***GetPrivilege***
Set Byte/Word Order	***SetByteOrder***
Get Byte/Word Order	***GetByteOrder***
Get VXIbus Status, One Field	***GetVXIbusStatusInd***
High-Level VXIbus Access Functions	
Read Value	***VXIin***
Write Value	***VXIout***
Read VXI Register	***VXIinReg***
Write VXI Register	***VXIoutReg***
Move Buffer	***VXImove***
Local Resource Access Functions	
Get Local Logical Address	***GetMyLA***
Read Local VXI Register	***VXIinLR***
Write Local VXI Register	***VXIoutLR***
Set MODID lines	***SetMODID***
Read MODID lines	***ReadMODID***
Allocate Local VXI Shared Memory	***VXImemAlloc***
Free Local VXI Shared Memory	***VXImemFree***
Update Local VXI Shared Memory	***VXImemCopy***

(continues)

Table 1-2. The VXI Library Function Tree (Continued)

VXI Signal Functions	
Route Signals	***RouteSignal***
Enable Signal Interrupts	***EnableSignalInt***
Disable Signal Interrupts	***DisableSignalInt***
Dequeue Signal	***SignalDeq***
Enqueue Signal	***SignalEnq***
Jam Signal	***SignalJam***
Wait for Signal	***WaitForSignal***
VXI Interrupt Functions	
Route VXI Interrupts	***RouteVXIint***
Enable VXI to Signal Interrupts	***EnableVXItoSignalInt***
Disable VXI to Signal Interrupts	***DisableVXItoSignalInt***
Enable VXI Interrupts	***EnableVXIint***
Disable VXI Interrupts	***DisableVXIint***
Acknowledge VXI Interrupt	***AcknowledgeVXIint***
Assert VXI Interrupt Line	***AssertVXIint***
Deassert VXI Interrupt Line	***DeAssertVXIint***
Set VXI Interrupt Acknowledge Mode	***VXIintAcknowledgeMode***
VXI Trigger Functions	
Source Trigger	***SrcTrig***
Enable Trigger Sensing	***EnableTrigSense***
Disable Trigger Sensing	***DisableTrigSense***
Acknowledge Trigger	***AcknowledgeTrig***
Wait for Trigger	***WaitForTrig***
Trigger Assert Configure	***TrigAssertConfig***
Trigger Control Configure	***TrigCntrConfig***
Trigger External Configure	***TrigExtConfig***
Trigger Timer Configure	***TrigTickConfig***
Map Trigger to Trigger	***MapTrigToTrig***
Unmap Trigger to Trigger	***UnMapTrigToTrig***
System Interrupt Handler Functions	
Enable Sysfail Interrupts	***EnableSysfail***
Disable Sysfail Interrupts	***DisableSysfail***
Enable ACfail Interrupts	***EnableACfail***
Disable ACfail Interrupts	***DisableACfail***
Enable Sysreset Interrupts	***EnableSysreset***
Disable Sysreset Interrupts	***DisableSysreset***
Assert Sysreset	***AssertSysreset***
Enable Soft Reset Interrupts	***EnableSoftReset***
Disable Soft Reset Interrupts	***DisableSoftReset***
VXIbus Extender Functions	
Map VXI Interrupts on Extender	***MapVXIint***
Map TTL Triggers on Extender	***MapTTLtrig***
Map ECL Triggers on Extender	***MapECLtrig***
Map Utility Bus Signals on Extender	***MapUtilBus***

The first-level bold headings in the tree are the names of function classes. Function classes are groups of related function panels. The second-level headings in plain text are the names of individual function panels. Each VXI function panel generates one VXI function call. The names of the corresponding VXI function calls are in bold italics to the right of the function panel names.

The classes in the function tree are described here:

- **System Configuration** is a class of function panels that configure the NI-VXI interface and retrieve information from the Resource Manager table.
- **Commander Word Serial Protocol** is a class of function panels that perform the basic mode of communication between VXI Message-Based devices within the Commander/Servant hierarchy. Specifically, this class of function panels is used by the Commander device to communicate with its Servants. Longword Serial and Extended Longword Serial Protocols are extensions to the Word Serial Protocol.
- **Servant Word Serial Protocol** is a class of function panels that perform the basic mode of communication between VXI Message-Based devices within the Commander/Servant hierarchy. Specifically, this class of function panels is used by Servant devices to communicate with the Commander. Longword Serial and Extended Longword Serial Protocols are extensions to the Word Serial Protocol.
- **Low-Level VXIbus Access** is a class of function panels that perform operations requiring direct access to the VXIbus.
- **High-Level VXIbus Access** is a class of function panels that perform operations requiring protected access to the VXIbus.
- **Local Resource Access** is a class of function panels that control resources under direct control by the device on which the software resides.
- **VXI Signals** is a class of function panels that perform basic asynchronous peer-to-peer communication used by Message-Based devices. VXI signals can be either Response signals or Event signals. Response signals report changes in Word Serial communication between a Servant and its Commander. Event signals inform another device of other asynchronous changes. LabWindows can handle signals either in the interrupt service routine or by using a dequeue function to get signals from a system queue. The signal handling mode can be individually configured for each class of signal.
- **VXI Interrupts** is a class of functions panels that handle interrupts from one or more of the seven VXI backplane interrupts. The usage of these interrupts is virtually the same as for signals.
- **VXI Triggers** is a class of function panels that accommodate all trigger lines and protocols for all TTL and ECL VXI trigger lines.
- **System Interrupt Handlers** is a class of function panels that let you install interrupt service routines for the system interrupt conditions. These conditions include Sysfail, ACfail, Sysreset, Bus Error, and Soft Reset interrupts.
- **VXIbus Extender** is a class of function panels that set up the VXI extenders in a multiple-mainframe system.

The online help with each panel contains specific information about operating each function panel.

Table 1-3 groups the VXI Library functions that are available in standalone C Programs and `.obj` instrument programs inside the environment. See the LabWindows Instrument Library Developer's Guide (Part No. 320315-01) for more information on `.obj` instrument programs.

Table 1-3. Functions for use in C Programs, .obj files, or the LabWindows Run-Time System

System Configuration Functions	
Get Device Information, All Fields	***GetDevInfo***
Set Device Information, All fields	***SetDevInfo***
Servant Word Serial Protocol Functions	
Set Read Handler	***SetWSSrdHandler***
Get Read Handler	***GetWSSrdHandler***
Default Read Handler	***DefaultWSSrdHandler***
Set Write Handler	***SetWSSwrtHandler***
Get Write Handler	***GetWSSwrtHandler***
Default Write Handler	***DefaultWSSwrtHandler***
Set Command Handler	***SetWSScmdHandler***
Get Command Handler	***GetWSScmdHandler***
Default Command Handler	***DefaultWSScmdHandler***
Set Longword Command Handler	***SetWSSLcmdHandler***
Get Longword Command Handler	***GetWSSLcmdHandler***
Default Longword Command Handler	***DefaultWSSLcmdHandler***
Set Extended Longword Command Handler	***SetWSSEcmdHandler***
Get Extended Longword Command Handler	***GetWSSEcmdHandler***
Default Extended Longword Command Handler	***DefaultWSSEcmdHandler***
Low-Level VXIbus Access Functions	
Get VXIbus Status, All Information	***GetVXIbusStatus***
VXI Signal Functions	
Set Signal Handler	***SetSignalHandler***
Get Signal Handler	***GetSignalHandler***
Default Signal Handler	***DefaultSignalHandler***
VXI Interrupt Functions	
Set VXI Interrupt Handler	***SetVXIintHandler***
Get VXI Interrupt Handler	***GetVXIintHandler***
Default VXI Interrupt Handler	***DefaultVXIintHandler***
VXI Trigger Functions	
Set Trigger Handler	***SetTrigHandler***
Get Trigger Handler	***GetTrigHandler***
Default Trigger Handler	***DefaultTrigHandler***
Default Trigger Handler 2	***DefaultTrigHandler2***
System Interrupt Handler Functions	
Set Sysfail Handler	***SetSysfailHandler***
Get Sysfail Handler	***GetSysfailHandler***
Default Sysfail Handler	***DefaultSysfailHandler***
Set ACfail Handler	***SetACfailHandler***
Get ACfail Handler	***GetACfailHandler***
Default ACfail Handler	***DefaultACfailHandler***
Set Soft Reset Handler	***SetSoftResetHandler***
Get Soft Reset Handler	***GetSoftResetHandler***
Default Soft Reset Handler	***DefaultSoftResetHandler***
Set Bus Error Handler	***SetBusErrorHandler***
Get Bus Error Handler	***GetBusErrorHandler***
Default Bus Error Handler	***DefaultBusErrorHandler***
Set Sysreset Handler	***SetSysresetHandler***
Get Sysreset Handler	***GetSysresetHandler***
Default Sysreset Handler	***DefaultSysresetHandler***

Table 1-4 lists older names of VXI trigger functions for backward compatibility. If you are using an older version of the NI-VXI software for LabWindows , you can use the following function names with the same parameters to achieve the same results as the functions given in Chapter 10, *VXI Trigger Functions*. However, you should not use these older function names in new or updated programs. Also keep in mind that the value of the `line` parameter in the older functions is specific to TTL (0 to 7) or ECL (0 to 5). An asterisk (*) following a function name denotes that the function is to be used only in standalone C programs.

Table 1-4. Old VXI Trigger Functions

Source TTL Trigger	***SrcTTLtrig***
Enable TTL Sensing	***EnableTTLsense***
Disable TTL Sensing	***DisableTTLsense***
Set TTL Trigger Handler	***SetTTLtrigHandler ** **
Get TTL Trigger Handler	***GetTTLtrigHandler ** **
Default TTL Trigger Handler	***DefaultTTLtrigHandler ** **
Acknowledge TTL Trigger	***AcknowledgeTTLtrig***
Wait for TTL Trigger	***WaitForTTLtrig***
Source ECL Trigger	***SrcECLtrig***
Enable ECL Sensing	***EnableECLsense***
Disable ECL Sensing	***DisableECLsense***
Set ECL Trigger Handler	***SetECLtrigHandler ** **
Get ECL Trigger Handler	***GetECLtrigHandler ** **
Default ECL Trigger Handler	***DefaultECLtrigHandler ** **
Acknowledge ECL Trigger	***AcknowledgeECLtrig***
Wait for ECL Trigger	***WaitForECLtrig***

Reporting Status Information

The functions in the VXI Library are supported through a set of global variables. These global variables are updated by the VXI Library Device Configuration functions or by the default handlers associated with particular events. Please refer to the entry of these functions in the following chapters for a complete description of the global variables modified by each of them. You can view the global variable within LabWindows using the standard View Variables command.

Chapter 2
System Configuration Functions

This chapter describes the functions in the LabWindows VXI System Configuration Library. LabWindows uses these functions to 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

The following 12 functions are described in this chapter:

- `CloseVXIlibrary`
- `CreateDevInfo`
- `FindDevLA`
- `GetDevInfo`
- `GetDevInfoLong`
- `GetDevInfoShort`
- `GetDevInfoStr`
- `InitVXIlibrary`
- `SetDevInfo`
- `SetDevInfoLong`
- `SetDevInfoShort`
- `SetDevInfoStr`

CloseVXIlibrary

Syntax:

BASIC Syntax	`ret% = CloseVXIlibrary% ()`
C 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`	`integer`	Return Status 0 = NI-VXI library closed successfully 1 = Successful; previous `InitVXIlibrary` calls still pending. -1 = NI-VXI library not open

BASIC Example:

```
' NI-VXI application shell program.

ret% = InitVXIlibrary% ()
IF ret% < 0 THEN
     ' RM table memory allocation or file open failed.
END IF

' Application-specific program.

ret% = CloseVXIlibrary% ()
```

C Example:

```
/* NI-VXI application shell program. */

main()
{
int      ret;

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

   /*
      Application-specific program.
   */

   ret = CloseVXIlibrary();
}
```

CreateDevInfo

Syntax:

BASIC Syntax	`ret% = CreateDevInfo% (la%)`
C 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 parameter:

`la`	`integer`	Logical address of device to create entry for

Return value:

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

BASIC Example:

```
' Create a new entry for pseudo logical address 298.

la% = 298
ret% = CreateDevInfo% (la%)
IF ret% <> 0 THEN
     ' Error creating new entry.
END IF
```

C Example:

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

intret;
intla;

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

FindDevLA

Syntax:

BASIC Syntax	`ret% = FindDevLA% (namepat$, manid%, modelcode%, devclass%, slot%, mainframe%, cmdrla%, la%)`
C 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 a device with the name `GPIB-VXI` it accepts `GPI`). If two or more devices match, it returns the logical address of the first device found.

Remarks:

Input parameters:

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

Output parameter:

`la`	`integer`	Logical address of the device found

Return value:

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

BASIC Example:

```
' Find the logical address of a device with manid = &HFF6
' (National Instruments) and modelcode = &HFF (GPIB-VXI).

DIM namepat AS STRING * 13

namepat$ = ""
manid% = &HFF6
modelcode% = &HFF
devclass% = -1
mainframe% = -1
slot% = -1
cmdrla% = -1
ret% = FindDevLA% (namepat$, manid%, modelcode%, devclass%,
                   mainframe%, slot%, cmdrla%, la%)
IF ret% <> 0 THEN
     ' No device with manid = &HFF6 and modelcode = &HFF was found.
ELSE
     ' Device was found, logical address in la.
END IF
```

C Example:

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

int      ret;
char     *namepat;
int      manid;
int      modelcode;
int      devclass;
int      mainframe;
int      slot;
int      cmdrla;
int      la;

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

GetDevInfo

Syntax:

BASIC Syntax	none
C Syntax	`ret = GetDevInfo (la, field, fieldvalue)`

Action: Gets device information about a specified device.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

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

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

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

Output parameter:

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

Return value:

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

BASIC Example:

none

C Example:

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

int      ret;
int      la;
int      field;
int      fieldvalue;

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

GetDevInfoLong

Syntax:

BASIC Syntax	`ret% = GetDevInfoLong% (la%, field%, longvalue&)`
C Syntax	`ret = GetDevInfoLong (la, field, longvalue)`

Action: Gets information about a specified device from the device information table.

Remarks:

Input parameters:

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

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

Output parameter:

`longvalue`	`long`	Information for that field

Return value:

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

BASIC Example:

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

la% = 4
field% = 12
ret% = GetDevInfoLong% (la%, field%, longvalue&)
IF ret% <> 0 THEN
     ' Invalid logical address or field specified.
END IF
```

C Example:

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

int      ret;
int      la;
int      field;
long     longvalue;

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

GetDevInfoShort

Syntax:

BASIC Syntax	`ret% = GetDevInfoShort% (la%, field%, shortvalue%)`
C Syntax	`ret = GetDevInfoShort (la, field, shortvalue)`

Action: Gets information about a specified device from the device information table.

Remarks:

Input parameters:

`la`	`integer`	Logical address of device to get information about
`field`	`integer`	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

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`	`integer`	Information for that field

Return value:

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

BASIC Example:

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

la% = 4
field% = 7
ret% = GetDevInfoShort% (la%, field%, shortvalue%)
IF ret% <> 0 THEN
    ' Invalid logical address or field specified.
END IF
```

C Example:

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

int      ret;
int      la;
int      field;
int      shortvalue;

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

GetDevInfoStr

Syntax:

BASIC Syntax	`ret% = GetDevInfoStr% (la%, field%, stringvalue$)`
C Syntax	`ret = GetDevInfoStr (la, field, stringvalue)`

Action: Gets information about a specified device from the device information table.

Remarks:

Input parameters:

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

Field	Description
1	Device name
6	Manufacturer name
8	Model name

Output parameter:

`stringvalue`	`string`	Buffer to receive information for that field

Return value:

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

BASIC Example:

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

DIM stringvalue AS STRING * 14

la% = 4
field% = 8
ret% = GetDevInfoStr% (la%, field%, stringvalue$)
IF ret% <> 0 THEN
     ' Invalid logical address or field specified.
ENDIF
```

C Example:

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

int      ret;
int      la;
int      field;
char     stringvalue[14];

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

InitVXIlibrary

Syntax:

BASIC Syntax	`ret% = InitVXIlibrary% ()`
C 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`	`integer`	Return Status 0 = NI-VXI library initialized 1 = NI-VXI library already initialized (repeat call) -1 = NI-VXI library initialization failed

BASIC Example:

```
' NI-VXI application shell program.

ret% = InitVXIlibrary% ()
IF ret% < 0 THEN
     ' RM table memory allocation or file open failed.
END IF

' Application-specific program.

ret% = CloseVXIlibrary% ()
```

C Example:

```
/* NI-VXI application shell program. */

main()
{
    int     ret;

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

    /*
      Application-specific program.
    */

    ret = CloseVXIlibrary();
}
```

SetDevInfo

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetDevInfo (la, field, fieldvalue)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

`la` `integer` Logical address of device to set information for

`field` `integer` Field identification number

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

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

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

Output parameters:
none

Return value:

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

BASIC Example:
none

C Example:

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

int      ret;
int      la;
int      field;
long     fieldvalue;

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

SetDevInfoLong

Syntax:

BASIC Syntax	`ret% = SetDevInfoLong% (la%, field%, longvalue&)`
C Syntax	`ret = SetDevInfoLong (la, field, longvalue)`

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

Remarks:

Input parameters:

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

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

`longvalue`	`long`	Information for that field

Output parameters:

none

Return value:

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

BASIC Example:

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

la% = 4
field% = 12
longvalue& = &H200000&
ret% = SetDevInfoLong% (la%, field%, longvalue&)
IF ret% <> 0 THEN
     ' Invalid logical address or field specified.
END IF
```

C Example:

```
/* Set the A24 base of a device at Logical Address 4. */
int      ret;
int      la;
int      field;
long     longvalue;

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

SetDevInfoShort

Syntax:

BASIC Syntax	`ret% = SetDevInfoShort% (la%, field%, shortvalue%)`
C Syntax	`ret = SetDevInfoShort (la, field, shortvalue)`

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

Remarks:

Input parameters:

`la` `integer` Logical address of device to set information for

`field` `integer` 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

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` `integer` Information for that field

Output parameters:

none

Return value:

`ret` `integer` Return Status

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

BASIC Example:

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

la% = 4
field% = 7
shortvalue% = &HFFFF
ret% = SetDevInfoShort% (la%, field%, shortvalue%)
IF ret% <> 0 THEN
     ' Invalid logical address or field specified.
END IF
```

C Example:

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

int      ret;
int      la;
int      field;
int      shortvalue;

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

SetDevInfoStr

Syntax:

BASIC Syntax	`ret% = SetDevInfoStr% (la%, field%, stringvalue$)`
C Syntax	`ret = SetDevInfoStr (la, field, stringvalue)`

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

Remarks:

Input parameters:

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

Field	Description
1	Device name
6	Manufacturer name
8	Model name

`stringvalue`	`string`	Buffer to set the information for that field

Output parameters:

none

Return value:

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

BASIC Example:

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

la% = 4
field% = 8
stringvalue$ = "DMM0"
ret% = SetDevInfoStr% (la%, field%, stringvalue$)
IF ret% <> 0 THEN
     ' Invalid logical address or field specified.
END IF
```

C Example:

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

   int      ret;
   int      la;
   int      field;
   char     stringvalue[14];

   la = 4;
   field = 8;
   strcpy (stringvalue, "DMM0");
   ret = SetDevInfoStr (la, field, stringvalue);
   if (ret != 0)
      /* Invalid logical address or field specified. */;
```

Chapter 3
Commander Word Serial Protocol Functions

This chapter describes the functions in the LabWindows VXI Commander Word Serial Protocol Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

The following 18 functions are described in this chapter:

- `WSabort`
- `WSclr`
- `WScmd`
- `WSEcmd`
- `WSgetTmo`
- `WSLcmd`
- `WSLresp`
- `WSrd`
- `WSrdf`
- `WSrdi`
- `WSrdl`
- `WSresp`
- `WSsetTmo`
- `WStrg`
- `WSwrt`
- `WSwrtf`
- `WSwrti`
- `WSwrtl`

WSabort

Syntax:

BASIC Syntax	`ret% = WSabort% (la%, abortop%)`
C 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`	`integer`	Logical address of the Message-Based device
`abortop`	`integer`	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`	`integer`	Return Status 0 = Successfully aborted -1 = Invalid `la` -2 = Invalid `abortop`

BASIC Example:

```
' Perform Unsupported Command abort on Logical Address 5.

la% = 5
abortop% = 2
ret% = WSabort% (la%, abortop%)
IF ret% < 0 THEN
   ' An error occurred during WSabort.
END IF
```

C Example:

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

   int      ret;
   int      la;
   int      abortop;

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

WSclr

Syntax:

BASIC Syntax	`ret% = WSclr% (la%)`
C Syntax	`ret = WSclr (la)`

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

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Send Clear command to Logical Address 5.

la% = 5
ret% = WSclr% (la%)
IF ret% < 0 THEN
   ' An error occurred during the command transfer.
END IF
```

C Example:

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

int      ret;
int      la;

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

WScmd

Syntax:

BASIC Syntax	`ret% = WScmd% (la%, cmd%, respflag%, response%)`
C Syntax	`ret = WScmd (la, cmd, respflag, response)`

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

Remarks:

Input parameters:

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

Output parameter:

`response`	`integer`	16-bit location to store response

Return value:

`ret`	`integer`	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

BASIC Example:

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

la% = 5
cmd% = &HCFFF
respflag% = 1
ret% = WScmd% (la%, cmd%, respflag%, response%)
IF ret% < 0 THEN
   ' Error occurred during WS command transfer.
END IF
```

C Example:

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

   int      ret;
   int      la;
   int      cmd;
   int      respflag;
   int      response;

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

WSEcmd

Syntax:

BASIC Syntax	`ret% = WSEcmd% (la%, cmdExt%, cmd&, respflag%, response&)`
C 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`	`integer`	Logical address of the Message-Based device
`cmdExt`	`integer`	Upper 16 bits of 48-bit Extended Longword Serial command value
`cmd`	`long`	Lower 32 bits of 48-bit Extended Longword Serial command value
`respflag`	`integer`	Non-0 = Get a response (query) 0 = Do not get a response

Output parameter:

`response`	`long`	32-bit location to store response

Return value:

`ret`	`integer`	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

BASIC Example:

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

la% = 5
cmdExt% = &HFFFC
cmd& = &HFFFDFFFE&
respflag% = 1
ret% = WSEcmd% (la%, cmdExt%, cmd&, respflag%, response&)
IF ret% < 0 THEN
   ' Error occurred during command transfer.
END IF
```

C Example:

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

int      ret;
int      la;
int      cmdExt;
long     cmd;
int      respflag;
long     response;

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

WSgetTmo

Syntax:

BASIC Syntax	`ret% = WSgetTmo% (actualtimo&)`
C 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`	`long`	Timeout period in milliseconds

Return value:

`ret`	`integer`	0 = Successful

BASIC Example:

```
' Get the timeout period.

ret% = WSgetTmo% (actualtimo&)
```

C Example:

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

int   ret;
long     actualtimo;

ret = WSgetTmo(&actualtimo);
```

WSLcmd

Syntax:

BASIC Syntax	`ret% = WSLcmd% (la%, cmd&, respflag%, response&)`
C Syntax	`ret = WSLcmd (la, cmd, respflag, response)`

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

Remarks:

Input parameters:

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

Output parameter:

`response`	`long`	32-bit location to store response

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Send the Longword Serial command &HFFFCFFFD& to a device at Logical
' Address 5, and get the response.

la% = 5
cmd& = &HFFFCFFFD&
respflag% = 1
ret% = WSLcmd% (la%, cmd&, respflag%, response&)
IF ret% < 0 THEN
   ' Error occurred during command transfer.
END IF
```

C Example:

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

   int      ret;
   int      la;
   long     cmd;
   int      respflag;
   long     response;

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

WSLresp

Syntax:

BASIC Syntax	`ret% = WSLresp% (la%, response&)`
C 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 debug use only.

Remarks:

Input parameter:

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

Output parameter:

`response`	`long`	32-bit location to store response

Return value:

`ret`	`integer`	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

BASIC Example:

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

la% = 5
ret% = WSLresp% (la%, response&)
IF ret% < 0 THEN
   ' Error occurred during transfer.
END IF
```

C Example:

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

   int      ret;
   int      la;
   long     response;

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

WSrd

Syntax:

BASIC Syntax	`ret% = WSrd% (la%, buf$, count&, modevalue%, retcount&)`
C Syntax	`ret = WSrd (la, buf, count, modevalue, 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`	`integer`	Logical address to read buffer from
`count`	`long`	Maximum number of bytes to transfer
`modevalue`	`integer`	Mode of transfer (bit vector)

Bit	Description
0	Not DOR 0 = Abort if not DOR 1 = Poll until 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`	`string`	Read buffer
`retcount`	`long`	Number of bytes actually transferred

Return value:

`ret`	`integer`	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

BASIC Example:

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

DIM buf AS STRING * 100
la% = 5
count& = 30&
modevalue% = &H0001  ' Poll until DOR, terminate transfer on END.
ret% = WSrd% (la%, buf$, count&, modevalue%, retcount&)
IF ret% < 0 THEN
   ' An error occurred during the buffer read.
END IF
```

C Example:

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

int      ret;
int      la;
char     buf[100];
long     count;
int      modevalue;
long     retcount;

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

WSrdf

Syntax:

BASIC Syntax	`ret% = WSrdf% (la%, filename$, count&, modevalue%, retcount&)`
C Syntax	`ret = WSrdf (la, filename, count, modevalue, 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`	`integer`	Logical address to read buffer from
`filename`	`string`	Name of the file to read data into
`count`	`long`	Maximum number of bytes to transfer
`modevalue`	`integer`	Mode of transfer (bit vector)

Bit	Description
0	Not DOR 0 = Abort if not DOR 1 = Poll until 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`	`long`	Number of bytes actually transferred

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Read 16 kilobytes (&H4000) 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).

la% = 5
filename$ = "rdfile.dat"
count& = &H4000&
modevalue% = &H0005       ' Poll until DOR, terminate on END or LF.
ret% = WSrdf% (la%, filename$, count&, modevalue%, retcount&)
IF ret% < 0 THEN
   ' An error occurred during the buffer read into the file.
END IF
```

C 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). */

   int      ret;
   char     *filename;
   int      la;
   long     count;
   int      modevalue;
   long     retcount;

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

WSrdi

Syntax:

BASIC Syntax	`ret% = WSrdi% (la%, buf%(), count&, modevalue%, retcount&)`
C Syntax	`ret = WSrdi (la, buf, count, modevalue, retcount)`

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

Remarks:

Input parameters:

`la`	`integer`	Logical address to read buffer from
`count`	`long`	Maximum number of integers to transfer
`modevalue`	`integer`	Mode of transfer (bit vector)

Bit	Description
0	Not DOR 0 = Abort if not DOR 1 = Poll until 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`	`integer array`	Read buffer
`retcount`	`long`	Number of integers actually transferred

Return value:

`ret`	`integer`	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

BASIC Example:

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

DIM buf%(100)
la% = 5
count& = 30&
modevalue% = &H0001       ' Poll until DOR, terminate transfer on END.
ret% = WSrdi% (la%, buf%(), count&, modevalue%, retcount&)
IF ret% < 0 THEN
   ' An error occurred during the buffer read.
END IF
```

C Example:

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

int      ret;
int      la;
int      buf[100];
long     count;
int      modevalue;
long     retcount;

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

WSrdl

Syntax:

BASIC Syntax	`ret% = WSrdl% (la%, buf&(), count&, modevalue%, retcount&)`
C Syntax	`ret = WSrdl (la, buf, count, modevalue, retcount)`

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

Remarks:

Input parameters:

`la`	`integer`	Logical address to read buffer from
`count`	`long`	Maximum number of long integers to transfer
`modevalue`	`integer`	Mode of transfer (bit vector)

Bit	Description
0	Not DOR 0 = Abort if not DOR 1 = Poll until 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`	`long array`	Read buffer
`retcount`	`long`	Number of long integers actually transferred

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Read up to 30 long integers from a device at Logical Address 5.
' Poll until device is DOR. Terminate transfer on END bit only.

DIM buf&(100)
la% = 5
count& = 30&
modevalue% = &H0001       ' Poll until DOR, terminate transfer on END.
ret% = WSrdl% (la%, buf&(), count&, modevalue%, retcount&)
IF ret% < 0 THEN
   ' An error occurred during the buffer read.
END IF
```

C Example:

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

int      ret;
int      la;
long     buf[100];
long     count;
int      modevalue;
long     retcount;

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

WSresp

Syntax:

BASIC Syntax	`ret% = WSresp% (la%, response%)`
C 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 debug use only.

Remarks:

Input parameter:

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

Output parameter:

`response`	`integer`	16-bit location to store response

Return value:

`ret`	`integer`	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

BASIC Example:

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

la% = 5
cmd% = &HCFFF
respflag% = 0                  ' Do NOT read response.
ret% = WScmd% (la%, cmd%, respflag%, response%)
IF ret% < 0 THEN
   ' Error occurred during WS command transfer.
ELSE
   ret% = WSresp% (la%, response%)
   IF ret% < 0 THEN
       ' Error occurred during response retrieval.
   END IF
END IF
```

C Example:

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

int      ret;
int      la;
int      cmd;
int      respflag;
int      response;

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

WSsetTmo

Syntax:

BASIC Syntax	`ret% = WSsetTmo% (timo&, actualtimo&)`
C 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`	`long`	Timeout period in milliseconds

Output parameter:

`actualtimo`	`long`	Actual timeout period set in milliseconds

Return value:

`ret`	`integer`	0 = Successful

BASIC Example:

```
' Set the timeout period to 2 seconds.

timo& = 2000&
ret% = WSsetTmo% (timo&, actualtimo&)
```

C Example:

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

int      ret;
long     timo;
long     actualtimo;

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

WStrg

Syntax:

BASIC Syntax	`ret% = WStrg% (la%)`
C Syntax	`ret = WStrg (la)`

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

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Send Trigger command to Logical Address 5.

la% = 5
ret% = WStrg% (la%)
IF ret% < 0 THEN
   ' An error occurred during the command transfer.
END IF
```

C Example:

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

   int   ret;
   int   la;

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

WSwrt

Syntax:

BASIC Syntax	`ret% = WSwrt% (la%, buf$, count&, modevalue%, retcount&)`
C Syntax	`ret = WSwrt (la, buf, count, modevalue, 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`	`integer`	VXI logical address to write buffer to
`buf`	`string`	Write buffer
`count`	`long`	Maximum number of bytes to transfer
`modevalue`	`integer`	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`	`long`	Number of bytes actually transferred

Return value:

`ret`	`integer`	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 received
1	END	Any one of the termination received
0	IODONE	Successful transfer

BASIC 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.

la% = 5
buf$ = "VXI:CONF:NUMB?"
count& = StringLength% (buf$)
modevalue% = &H0003          ' Poll until DIR; send END with last byte.
ret% = WSwrt% (la%, buf$, count&, modevalue%, retcount&)
IF ret% < 0 THEN
   ' An error occurred during the buffer write.
END IF
```

C 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. */

int      ret;
int      la;
char     *buf;
long     count;
int      modevalue;
long     retcount;

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

WSwrtf

Syntax:

BASIC Syntax	`ret% = WSwrtf% (la%, filename$, count&, modevalue%, retcount&)`
C Syntax	`ret = WSwrtf (la, filename, count, modevalue, 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`	`integer`	VXI logical address to write buffer to
`filename`	`string`	Name of the file to write data from
`count`	`long`	Maximum number of bytes to transfer
`modevalue`	`integer`	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`	`long`	Number of bytes actually transferred

Return value:

`ret`	`integer`	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 received
1	END	Any one of the termination received
0	IODONE	Successful transfer

BASIC Example:

```
' Write 16 kilobytes (&H4000&) to a device at Logical Address 5 from the
' file "wrtfile.dat." Poll until device is DIR, and send END with the
' last byte.

la% = 5
filename$ = "wrtfile.dat"
count& = &H4000&
modevalue% = &H0003        ' Send END, wait until DIR if not already DIR.
ret% = WSwrtf% (la%, filename$, count&, modevalue%, retcount&)
IF ret% < 0 THEN
   ' An error occurred during the buffer write.
END IF
```

C 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. */

int      ret;
char     *filename;
int      la;
long     count;
int      modevalue;
long     retcount;

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

WSwrti

Syntax:

BASIC Syntax	`ret% = WSwrti% (la%, buf%(), count&, modevalue%, retcount&)`
C Syntax	`ret = WSwrti (la, buf, count, modevalue, retcount)`

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

Remarks:

Input parameters:

`la`	`integer`	VXI logical address to write buffer to
`buf`	`integer array`	Write buffer
`count`	`long`	Maximum number of integers to transfer
`modevalue`	`integer`	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`	`long`	Number of integers actually transferred

Return value:

`ret`	`integer`	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 received
1	END	Any one of the termination received
0	IODONE	Successful transfer

BASIC Example:

```
' Write an array containing binary short integer data to a device at
' Logical Address 5.  Poll until device is DIR, and send END with the last
' byte.

DIM buf%(100)
CALL InitBuf (buf%())         ' Initialize buf with data.
la% = 5
count& = 14&
modevalue% = &H0003           ' Poll until DIR; send END with last byte.
ret% = WSwrti% (la%, buf%(), count&, modevalue%, retcount&)
IF ret% < 0 THEN
   ' An error occurred during the buffer write.
END IF
```

C Example:

```
 /* Write an array containing binary short integer data to a device at
    Logical Address 5.  Poll until device is DIR, and send END with the
    last byte. */

int      ret;
int      la;
int      buf[100];
long     count;
int      modevalue;
long     retcount;

la = 5;
InitBuf(buf);          /* Initialize buf with data. */
count = StringLength(buf);  /* Find the length of buf string. */
modevalue = 0x0003;    /* Poll until DIR; send END with last byte. */
ret = WSwrti (la, buf, count, modevalue, &retcount);
if (ret < 0)
   /* An error occurred during the buffer write. */;
```

WSwrtl

Syntax:

BASIC Syntax	`ret% = WSwrtl% (la%, buf&(), count&, modevalue%, retcount&)`
C Syntax	`ret = WSwrtl (la, buf, count, modevalue, retcount)`

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

Remarks:

Input parameters:

`la`	`integer`	VXI logical address to write buffer to
`buf`	`long array`	Write buffer
`count`	`long`	Maximum number of long integers to transfer
`modevalue`	`integer`	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`	`long`	Number of long integers actually transferred

Return value:

`ret`	`integer`	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 received
1	END	Any one of the termination received
0	IODONE	Successful transfer

BASIC Example:

```
' Write an array containing binary long integer data to a device at Logical
' Address 5.  Poll until device is DIR, and send END with the last byte.

DIM buf&(100)
la% = 5
CALL InitBuf(buf&())          ' Initialize buf with data.
count& = 14
modevalue% = &H0003           ' Poll until DIR; send END with last byte.
ret% = WSwrtl% (la%, buf&(), count&, modevalue%, retcount&)
IF ret% < 0 THEN
   ' An error occurred during the buffer write.
END IF
```

C Example:

```
 /* Write an array containing binary long integer data to a device at
    Logical Address 5.  Poll until device is DIR, and send END with the
    last byte. */

int      ret;
int      la;
long     buf[100];
long     count;
int      modevalue;
long     retcount;

la = 5;
InitBuf(buf);       /* Initialize buf with data. */
count = StringLength(buf);  /* Find the length of buf string. */
modevalue = 0x0003;    /* Poll until DIR; send END with last byte. */
ret = WSwrtl (la, buf, count, modevalue, &retcount);
if (ret < 0)
   /* An error occurred during the buffer write. */;
```

Chapter 4
Servant Word Serial Protocol Functions

This chapter describes the functions in the LabWindows VXI Servant Word Serial Protocol Library. 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. The descriptions are explained in both BASIC and C syntax, and are arranged alphabetically.

The following 25 functions are described in this chapter:

- `GenProtError`
- `GetWSScmdHandler`
- `GetWSSEcmdHandler`
- `GetWSSLcmdHandler`
- `GetWSSrdHandler`
- `GetWSSwrtHandler`
- `RespProtError`
- `SetWSScmdHandler`
- `SetWSSEcmdHandler`
- `SetWSSLcmdHandler`
- `SetWSSrdHandler`
- `SetWSSwrtHandler`
- `WSSabort`
- `WSSdisable`
- `WSSenable`
- `WSSLnoResp`
- `WSSLsendResp`
- `WSSnoResp`
- `WSSrd`
- `WSSrdi`
- `WSSrdl`
- `WSSsendResp`
- `WSSwrt`
- `WSSwrti`
- `WSSwrtl`

GenProtError

Syntax:

BASIC Syntax	`ret% = GenProtError% (proterr%)`
C 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`.

Remarks:

Input parameter:

`proterr`	`integer`	Protocol error to generate

Value	Protocol Error Description
-1	Clear any protocol error condition
-3	Multiple Query Error (MQE)
-4	Unsupported Command (UnSupCom)
-5	Data In Ready violation (DIRviol)
-6	Data Out Ready violation (DORviol)
-7	Read Ready violation (RRviol)
-8	Write Ready violation (WRviol)
others	Reserved

Output parameters:

none

Return value:

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

BASIC Example:

```
' Generate a protocol error of DORviol.

proterr% = &HFFFA
ret% = GenProtError% (proterr%)
IF ret% < 0 THEN
   ' An error occurred in GenProtError.
END IF
```

C Example:

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

int      ret;
int      proterr;

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

GetWSScmdHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetWSScmdHandler()`

Action: Returns the address of the current Servant Word Serial command interrupt handler.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func (*void)()` Pointer to the new Servant Word Serial command interrupt handler

BASIC Example:

none

C Example:

```
 /* Get the address of the Servant Word Serial command interrupt
    handler. */

void     (*func)();

func = GetWSScmdHandler();
```

GetWSSEcmdHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetWSSEcmdHandler()`

Action: Returns the address of the current Servant Extended Longword Serial command interrupt handler.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func (*void)()` Pointer to the new Servant Extended Longword Serial command interrupt handler

BASIC Example:

none

C Example:

```
/*  Get the address of the Servant Extended Longword Serial command
    handler. */

void     (*func)();

func = GetWSSEcmdHandler();
```

GetWSSLcmdHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetWSSLcmdHandler()`

Action: Returns the address of the current Servant Longword Serial command interrupt handler.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func (*void)()` Pointer to the new Servant Longword Serial command interrupt handler

BASIC Example:

none

C Example:

```
/* Get the address of the Servant Longword Serial command interrupt
   handler. */

void     (*func)();

func = GetWSSLcmdHandler();
```

GetWSSrdHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetWSSrdHandler()`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func (*void)()` Pointer to the current `WSSrd` done notification interrupt handler

BASIC Example:

none

C Example:

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

void     (*func)();

func = GetWSSrdHandler();
```

GetWSSwrtHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetWSSwrtHandler()`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func (*void)()` Pointer to the current `WSSwrt` done notification interrupt handler

BASIC Example:

none

C Example:

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

void     (*func)();

func = GetWSSwrtHandler();
```

RespProtError

Syntax:

BASIC Syntax	`ret% = RespProtError% ()`
C 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 deasserts the ERR* bit.

Remarks:

Parameters:

none

Return value:

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

BASIC Example:

```
' Respond to the Word Serial Read Protocol Error query.

ret% = RespProtError% ()
IF ret% < 0 THEN
   ' An error occurred in RespProtError.
END IF
```

C Example:

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

int   ret;

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

SetWSScmdHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetWSScmdHandler (func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     func (int);
int   ret;

ret = SetWSScmdHandler(func);
```

SetWSSEcmdHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetWSSEcmdHandler (func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     func (int, long);
int   ret;

ret = SetWSSEcmdHandler(func);
```

SetWSSLcmdHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetWSSLcmdHandler (func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     func (long);
int   ret;

ret = SetWSSLcmdHandler(func);
```

SetWSSrdHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetWSSrdHandler (func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     func (int, long);
int   ret;

ret = SetWSSrdHandler(func);
```

SetWSSwrtHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetWSSwrtHandler (func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     func (int, long);
int   ret;

ret = SetWSSwrtHandler(func);
```

WSSabort

Syntax:

BASIC Syntax	`ret% = WSSabort% (abortop%)`
C Syntax	`ret = WSSabort (abortop)`

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

Remarks:

Input parameter:

`abortop`	`integer`	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`	`integer`	Return Status 0 = Successfully aborted -1 = Servant Word Serial functions not supported -2 = Unable to abort

BASIC Example:

```
' Abort WSSwrt.

abortop% = &H0001
ret% = WSSabort% (abortop%)
IF ret% < 0 THEN
   ' An error occurred during WSSabort.
END IF
```

C Example:

```
 /* Abort WSSwrt. */

int   ret;
int   abortop;

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

WSSdisable

Syntax:

BASIC Syntax	`ret% = WSSdisable% ()`
C 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`	`integer`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

BASIC Example:

```
' Disable all the Servant Word Serial functions.

ret% = WSSdisable% ()
```

C Example:

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

int   ret;

ret = WSSdisable();
```

WSSenable

Syntax:

BASIC Syntax	`ret% = WSSenable% ()`
C 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`	`integer`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

BASIC Example:

```
' Enable all the Servant Word Serial functions.

ret% = WSSenable% ()
```

C Example:

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

int   ret;

ret = WSSenable();
```

WSSLnoResp

Syntax:

BASIC Syntax	`ret% = WSSLnoResp% ()`
C 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`	`integer`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

BASIC Example:

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

ret% = WSSLnoResp% ()
IF ret% < 0 THEN
   ' An error occurred during WSSLnoResp.
END IF
```

C Example:

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

int   ret;

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

WSSLsendResp

Syntax:

BASIC Syntax	`ret% = WSSLsendResp% (response&)`
C 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 the processing of a Longword Serial Protocol query (commands are acknowledged with `WSSLnoResp`).

Remarks:

Input parameter:

`response`	`long`	32-bit response

Output parameters:

none

Return value:

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

BASIC Example:

```
' Respond to a received Longword Serial Protocol query.

response& = &HFFFCFFFD&
ret% = WSSLsendResp% (response&)
IF ret% < 0 THEN
   ' An error occurred during WSSLsendResp.
END IF
```

C Example:

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

int      ret;
long     response;

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

WSSnoResp

Syntax:

BASIC Syntax	`ret% = WSSnoResp% ()`
C 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`	`integer`	Return Status 0 = Successful -1 = Servant Word Serial functions not supported

BASIC Example:

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

ret% = WSSnoResp% ()
IF ret% < 0 THEN
   ' An error occurred during WSSnoResp.
END IF
```

C Example:

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

int   ret;

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

WSSrd

Syntax:

BASIC Syntax	`ret% = WSSrd% (buf$, count&, modevalue%)`
C Syntax	`ret = WSSrd (buf, count, modevalue )`

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`	`long`	Maximum number of bytes to transfer
`modevalue`	`integer`	Mode of transfer (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`	`string`	Read buffer

Return value:

`ret`	`integer`	Return Status 0 = Posted successfully -1 = Servant Word Serial functions not supported -2 = Word Serial Servant read operation already in progress

BASIC Example:

```
' Read 10 bytes from the Commander.

DIM buf AS STRING * 100
count& = 10&
modevalue% = &H0000          ' Do not send DIR signal to Commander.
ret% = WSSrd% (buf$, count&, modevalue%)
IF ret% < 0 THEN
   ' An error occurred during WSSrd.
END IF
```

C Example:

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

   int   ret;
   char     buf[100];
   long     count;
   int   modevalue;

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

WSSrdi

Syntax:

BASIC Syntax	`ret% = WSSrdi% (buf%(), count&, modevalue%)`
C Syntax	`ret = WSSrdi (buf, count, modevalue)`

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

Remarks:

Input parameters:

`count`	`long`	Maximum number of integers to transfer
`modevalue`	`integer`	Mode of transfer (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`	`integer array`	Read buffer

Return value:

`ret`	`integer`	Return Status 0 = Posted successfully -1 = Servant Word Serial functions not supported -2 = Word Serial Servant read operation already in progress

BASIC Example:

```
' Read 10 integers from the Commander.

DIM buf%(100)
count& = 10&
modevalue% = &H0000          ' Do not send DIR signal to Commander.
ret% = WSSrdi% (buf%(), count&, modevalue%)
IF ret% < 0 THEN
   ' An error occurred during WSSrdi.
END IF
```

C Example:

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

   int   ret;
   int   buf[100];
   long     count;
   int   modevalue;

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

WSSrdl

Syntax:

BASIC Syntax	`ret% = WSSrdl% (buf&(), count&, modevalue%)`
C Syntax	`ret = WSSrdl (buf, count, modevalue)`

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

Remarks:

Input parameters:

`count`	`long`	Maximum number of long integers to transfer
`modevalue`	`integer`	Mode of transfer (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`	`long array`	Read buffer

Return value:

`ret`	`integer`	Return Status 0 = Posted successfully -1 = Servant Word Serial functions not supported -2 = Word Serial Servant read operation already in progress

BASIC Example:

```
' Read 10 long integers from the Commander.

DIM buf&(100)
count& = 10&
modevalue% = &H0000          ' Do not send DIR signal to Commander.
ret% = WSSrdl% (buf&(), count&, modevalue%)
IF ret% < 0 THEN
   ' An error occurred during WSSrdl.
END IF
```

C Example:

```
    /* Read 10 long integers from the Commander. */

   int   ret;
   int   buf[100];
   long     count;
   int   modevalue;

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

WSSsendResp

Syntax:

BASIC Syntax	`ret% = WSSsendResp% (response%)`
C 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 the processing of a Word Serial Protocol query (commands are acknowledged with `WSSnoResp`).

Remarks:

Input parameter:

`response`	`integer`	16-bit response

Output parameters:

none

Return value:

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

BASIC Example:

```
' Respond with &H1234 to a received Word Serial Protocol query.

response% = &H1234
ret% = WSSsendResp% (response%)
IF ret% < 0 THEN
   ' An error occurred during WSSsendResp.
END IF
```

C Example:

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

int   ret;
int   response;

response = 0x1234;
ret = WSSsendResp (response);
if (ret < 0)
   /* An error occurred during WSSsendResp. */;
```

WSSwrt

Syntax:

BASIC Syntax	`ret% = WSSwrt% (buf$, count&, modevalue%)`
C Syntax	`ret = WSSwrt (buf, count, modevalue)`

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`	`string`	Write buffer
`count`	`long`	Maximum number of bytes to transfer
`modevalue`	`integer`	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`	`integer`	Return Status 0 = Posted successfully -1 = Servant Word Serial functions not supported -2 = Word Serial Servant write operation already in progress

BASIC Example:

```
' Write 6 bytes to the Commander.

buf$ = "1.0422"
count& = 6&
modevalue% = &H0002           ' Send END with the last byte.
ret% = WSSwrt% (buf$, count&, modevalue%)
IF ret% < 0 THEN
   ' An error occurred during WSSwrt.
END IF
```

C Example:

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

   int   ret;
   char     *buf;
   long     count;
   int   modevalue;

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

WSSwrti

Syntax:

BASIC Syntax	`ret% = WSSwrti% (buf%(), count&, modevalue%)`
C Syntax	`ret = WSSwrti (buf, count, modevalue)`

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

Remarks:

Input parameters:

`buf`	`integer array`	Write buffer
`count`	`long`	Maximum number of integers to transfer
`modevalue`	`integer`	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`	`integer`	Return Status 0 = Posted successfully -1 = Servant Word Serial functions not supported -2 = Word Serial Servant write operation already in progress

BASIC Example:

```
' Write 6 integers to the Commander.

DIM buf%(100)
count& = 6&
modevalue% = &H0002           ' Send END with the last byte.
ret% = WSSwrti% (buf%(), count&, modevalue%)
IF ret% < 0 THEN
   ' An error occurred during WSSwrti.
END IF
```

C Example:

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

   int   ret;
   int   buf[100];
   long     count;
   int   modevalue;

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

WSSwrtl

Syntax:

BASIC Syntax	`ret% = WSSwrtl% (buf&(), count&, modevalue%)`
C Syntax	`ret = WSSwrtl (buf, count, modevalue)`

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

Remarks:

Input parameters:

`buf`	`long array`	Write buffer
`count`	`long`	Maximum number of long integers to transfer
`modevalue`	`integer`	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`	`integer`	Return Status 0 = Posted successfully -1 = Servant Word Serial functions not supported -2 = Word Serial Servant write operation already in progress

BASIC Example:

```
' Write 6 long integers to the Commander.

DIM buf&(100)
count& = 6&
modevalue% = &H0002           ' Send END with the last byte.
ret% = WSSwrtl% (buf&(100), count&, modevalue%)
IF ret% < 0 THEN
   ' An error occurred during WSSwrtl.
END IF
```

C Example:

```
    /* Write 6 long integers to the Commander. */

   int   ret;
   long     buf[100];
   long     count;
   int   modevalue;

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

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

Syntax:

BASIC Syntax	none
C 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.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

none

DefaultWSSEcmdHandler

Syntax:

BASIC Syntax	none
C 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.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

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

Output parameters:

none

Return value:

none

DefaultWSSLcmdHandler

Syntax:

BASIC Syntax	none
C 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.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

none

DefaultWSSrdHandler

Syntax:

BASIC Syntax	none
C 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` to `count`.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

`status`	`integer`	Status bit vector

The following table gives the meaning of each bit that is set to 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`	`long`	Actual number of bytes received

Output parameters:

none

Return value:

none

DefaultWSSwrtHandler

Syntax:

BASIC Syntax	none
C 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`.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

`status`	`integer`	Status bit vector

The following table gives the meaning of each bit that is set to 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`	`long`	Actual number of bytes sent

Output parameters:

none

Return value:

none

Chapter 5
Low-Level VXIbus Access Functions

This chapter describes the functions in the LabWindows VXI Low-Level VXIbus Access Library. Low-level and high-level VXIbus Access functions are used 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/VME device
- Implementing shared memory protocols

Low-level VXIbus access is the fastest access method for directly reading from or writing to any of the VXIbus address spaces. The functions are explained in both BASIC and C syntax, and are arranged alphabetically. The following 16 functions are described in this chapter:

- `ClearBusError`
- `GetByteOrder`
- `GetContext`
- `GetPrivilege`
- `GetVXIbusStatus`
- `GetVXIbusStatusInd`
- `GetWindowRange`
- `MapVXIAddress`
- `RestoreContext`
- `SaveContext`
- `SetByteOrder`
- `SetContext`
- `SetPrivilege`
- `UnMapVXIAddress`
- `VXIpeek`
- `VXIpoke`

ClearBusError

Syntax:

BASIC Syntax	none
C Syntax	`ret = ClearBusError ()`

Action: Clears Bus Errors generated during low-level VXIbus access functions.

Note: For standalone C programs only.

Remarks:

Parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Bus Error cleared successfully -1 = No hardware support

BASIC Example:

none

C Example:

```
 /* Set the Bus Error handler at the beginning of the program. */

NIVXI_HBUSERROR UserBusErrorHandler;

SetBusErrorHandler (UserBusErrorHandler);

/* The UserBusErrorHandler can be defined as following. */

void NIVXI_HANDLER UserBusErrorHandler (void)
{
    ClearBusError ();

    /*
      User code
    */

}
```

GetByteOrder

Syntax:

BASIC Syntax	`ret% = GetByteOrder% (windownum&, ordermode%)`
C Syntax	`ret = GetByteOrder (windownum, ordermode)`

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

Remarks:

Input parameter:

`windownum`	`long`	Window number as returned from `MapVXIAddress`

Output parameter:

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

Return value:

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

BASIC Example:

```
' Get the byte order for the specified window.

' Window value is set in MapVXIAddress.

ret% = GetByteOrder% (windownum&, ordermode%)
```

C Example:

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

int      ret;
long     windownum;
int      ordermode;

/* Window value is set in MapVXIAddress. */

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

GetContext

Syntax:

BASIC Syntax	`ret% = GetContext% (windownum&, context&)`
C Syntax	`ret = GetContext (windownum, context)`

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

Remarks:

Input parameter:

`windownum`	`long`	Window number as returned from `MapVXIAddress`

Output parameter:

`context`	`long`	Returned VXI hardware access context

Return value:

`ret`	`integer`	Return Status 0 = Successful -1 = Invalid `windownum`

BASIC Example:

```
' Get or set the context for a window.

' Window ID set in MapVXIAddress call.

ret% = GetContext% (windownum&, context&)

' Change window settings as needed.

ret% = SetContext% (windownum&, context&)
```

C Example:

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

int      ret;
long     windownum;
long     context;

   /* Window ID set in MapVXIAddress call. */

ret = GetContext (windownum, &context);

   /* Change window settings as needed. */

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

GetPrivilege

Syntax:

BASIC Syntax	`ret% = GetPrivilege% (windownum&, priv%)`
C Syntax	`ret = GetPrivilege (windownum, priv)`

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

Remarks:

Input parameter:

`windownum`	`long`	Window number as returned from `MapVXIAddress`

Output parameter:

`priv`	`integer`	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`	`integer`	Return Status 0 = Successful -1 = Invalid `windownum`

BASIC Example:

```
' Get the privilege for a window.

' Window value is returned from MapVXIAddress.

ret% = GetPrivilege% (windownum&, priv%)
IF ret% <> 0 THEN
   ' Error occurred in GetPrivilege.
END IF
```

C Example:

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

int      ret;
long     windownum;
int      priv;

   /* Window value is returned from MapVXIAddress. */

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

GetVXIbusStatus

Syntax:

BASIC Syntax	none
C 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).

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameter:

`status` Structure containing VXIbus status

Structure is as follows:

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

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

BASIC Example:

none

C Example:

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

int      ret;
int      controller;
BusStat  status;

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

GetVXIbusStatusInd

Syntax:

BASIC Syntax	`ret% = GetVXIbusStatusInd% (controller%, field%, status%)`
C 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`	`integer`	Controller to get status from (-2 = OR of all)
`field`	`integer`	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`	`integer`	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`	`integer`	Return Status 0 = Status information received successfully -1 = Unsupportable function (no hardware support) -2 = Invalid `controller` -3 = Invalid `field`

BASIC Example:

```
' Get the VXIbus status for Sysfail on local (or first) controller.

controller% = -1
field% = 2
ret% = GetVXIbusStatusInd% (controller%, field%, status%)
IF ret% < 0 THEN
   ' Error in GetVXIbusStatusInd.
END IF
```

C Example:

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

   int      ret;
   int      controller;
   int      field;
   int      status;

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

GetWindowRange

Syntax:

BASIC Syntax	`ret% = GetWindowRange% (windownum&, windowbase&, windowend&)`
C Syntax	`ret = GetWindowRange (windownum, 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:

`windownum`	`long`	Window number obtained from `MapVXIAddress`

Output parameters:

`windowbase`	`long`	Base VXI Address
`windowend`	`long`	End VXI Address

Return value:

`ret`	`integer`	Return Status 0 = Successful -1 = Invalid `windownum`

BASIC Example:

```
' Get the range for the window obtained from MapVXIAddress.

accessparms% = 1
address& = &HC100&
timo& = 0&
addr& = MapVXIAddress& (accessparms%,address&,timo&,windownum&,ret%)
IF ret% < 0 THEN
   ' Map failed; handle error.
END IF

ret% = GetWindowRange% (windownum&, windowbase&, windowend&)
```

C Example:

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

   int      accessparms;
   long     address;
   long     timo;
   long     windownum;
   long     windowbase;
   long     windowend;
   int      ret;
   long     addr;

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

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

MapVXIAddress

Syntax:

BASIC Syntax	`addr& = MapVXIAddress& (accessparms%, address&, timo&, windownum&, ret%)`
C Syntax	`addr = MapVXIAddress (accessparms, address, timo, windownum, 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` `integer`
(Bits 0-1) VXI Address Space
1 = A16
2 = A24
3 = A32
(Bits 2-4) 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 8-15)
0

`address` `long` Address within A16, A24, or A32

`timo long` Timeout (in milliseconds)

Output parameters:

`windownum` `long` Window number for use with other functions

`ret` `integer` 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 = Timeout (window still in use; must use `UnMapVXIAddress`)

Return value:

`addr long`	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`.

BASIC Example:

```
' Get the local address pointer for address &HC100& 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.

accessparms% = 1     ' A16, Motorola, nonprivileged data
address& = &HC100&' Address = &HC000 + &H40 * Logical Address
timo&     = 5000&    ' 5 seconds (5000 milliseconds)
addr& = MapVXIAddress& (accessparms%, address&, timo&, windownum&, ret%)
IF ret% < 0 THEN
   ' Unable to get the pointer.
END IF
```

C 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. */

int      accessparms;
long     address;
long     timo;
long     windownum;
int      ret;
long     addr;

accessparms = 1;     /* A16, Motorola, nonprivileged data */
address = 0xc100L;/* Address = 0xc000 + 0x40 * Logical Address */
timo  = 5000L;       /* 5 seconds (5000 milliseconds) */
addr = MapVXIAddress (accessparms, address, timo, &windownum, &ret);
if (ret < 0)
   /* Unable to get the pointer. */;
```

RestoreContext

Syntax:

BASIC Syntax	none
C 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`.

Note: For standalone C programs only.

Remarks:

Input parameters:

none

Output parameter:

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

Return value:

`ret`	`integer`	Return Status 0 = Successful -2 = NULL `contextlist` pointer

BASIC Example:

none

C Example:

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

int      ret;
ContextStruct  contextlist;

ret = SaveContext (&contextlist);

   /*
      Interrupt service routine code.
   */

ret = RestoreContext (&contextlist);
```

SaveContext

Syntax:

BASIC Syntax	none
C Syntax	`ret = SaveContext (contextlist)`

Action: Saves the hardware context for all of the VXI windows. The `contextlist` parameter is 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.

Note: For standalone C programs only.

Remarks:

Input parameters:

none

Output parameter:

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

Return value:

`ret`	`integer`	Return Status 0 = Successful -2 = NULL `contextlist` pointer

BASIC Example:

none

C Example:

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

int      ret;
ContextStruct  contextlist;

ret = SaveContext (&contextlist);

   /*
      Interrupt service routine code.
   */

ret = RestoreContext (&contextlist);
```

SetByteOrder

Syntax:

BASIC Syntax	`ret% = SetByteOrder% (windownum&, ordermode%)`
C Syntax	`ret = SetByteOrder (windownum, ordermode)`

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

Remarks:

Input parameters:

`windownum`	`long`	Window number as returned from `MapVXIAddress`
`ordermode`	`integer`	Specifies the byte/word ordering 0 = Motorola byte ordering 1 = Intel byte ordering

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Successful; byte order set for specific window only 1 = Successful; byte order set for all windows -1 = Invalid `windownum` -2 = Invalid `ordermode` -5 = `ordermode` not supported -9 = No Owner Access for `windownum`

BASIC Example:

```
' Set the byte order to Motorola for a window.

' Window set in call to MapVXIAddress().
ordermode% = 0
ret% = SetByteOrder% (windownum&, ordermode%)
IF ret% < 0 THEN
   ' Capability not present.
END IF
```

C Example:

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

int      ret;
long     windownum;
int      ordermode;

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

SetContext

Syntax:

BASIC Syntax	`ret% = SetContext% (windownum&, context&)`
C Syntax	`ret = SetContext (windownum, 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:

`windownum`	`long`	Window number as returned from `MapVXIAddress`
`context`	`long`	VXI hardware context to install (context returned from `GetContext`)

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Successful -1 = Invalid `windownum` -2 = Invalid/unsupported `context` -9 = No Owner Access for `windownum`

BASIC Example:

```
' Get or set the context for a window.

' Window ID set in MapVXIAddress call.
ret% = GetContext% (windownum&, context&)

' Change window settings as needed.

ret% = SetContext% (windownum&, context&)
```

C Example:

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

int      ret;
long     windownum;
long     context;

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

   /* Change window settings as needed. */

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

SetPrivilege

Syntax:

BASIC Syntax	`ret% = SetPrivilege% (windownum&, priv%)`
C Syntax	`ret = SetPrivilege (windownum, priv)`

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

Remarks:

Input parameters:

`windownum`	`long`	Window number as returned from `MapVXIAddress`
`priv` `integer`	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`	`integer`	Return Status 0 = Successful -1 = Invalid `windownum` -2 = Invalid `priv` -7 = `priv` not supported -9 = No Owner Access for `windownum`

BASIC Example:

```
' Set nonprivileged data access for a window.

' Window ID set in MapVXIAddress call.
priv% = 0
ret% = SetPrivilege% (windownum&, priv%)
IF ret% < 0 THEN
   ' Error occurred in SetPrivilege.
END IF
```

C Example:

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

int      ret;
long     windownum;
int      priv;

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

UnMapVXIAddress

Syntax:

BASIC Syntax	`ret% = UnMapVXIAddress% (windownum&)`
C Syntax	`ret = UnMapVXIAddress (windownum)`

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

Remarks:

Input parameter:

`windownum`	`long`	Window number obtained from `MapVXIAddress`

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 1 = Access Only released (accessors remain) 0 = Window successfully unmapped -1 = Invalid `windownum`

BASIC Example:

```
' Unmap the window obtained from MapVXIAddress.

accessparms% = 1
address& = &HC100&
timo& = 0&
addr& = MapVXIAddress& (accessparms%, address&, timo&, windownum&, ret%)
IF addr& <> 0& THEN
   ' Use the pointer here.
   ret% = UnMapVXIAddress% (windownum&)
   IF ret% < 0 THEN
         ' Unmap successful.
   END IF
END IF
```

C Example:

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

   int      accessparms;
   long     address;
   long     timo;
   long     windownum;
   int      ret;
   void     *addr;

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

VXIpeek

Syntax:

BASIC Syntax	`CALL VXIpeek (addressptr&, accwidth%, value)`
C Syntax	`VXIpeek (addressptr, accwidth, value)`

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

Remarks:

Input parameters:

`addressptr`	`long`	Address pointer obtained from `MapVXIAddress`
`accwidth`	`integer`	Byte, word or longword 1 = Byte 2 = Word 4 = Longword

Output parameter:

`value`	`any`	Data value read (string, integer, or long)

Return value:

none

BASIC Example:

```
' Read the value from the VXI Status register of the device at Logical
' Address 4 into value, an integer variable.

accessparms% = 1     ' A16, Motorola, nonprivileged data.
addressptr& = MapVXIAddress (accessparms%, &HC106&, &H7FFFFFFF&,
              windownum&, ret%)
IF ret% >= 0 THEN    ' If a valid pointer was returned.
   CALL VXIpeek (addressptr&, 2, value%)
END IF
```

C Example:

```
 /* Read the value from the VXI Status register of the device at Logical
    Address 4. */

int      accessparms;
long     windownum;
int      ret;
long     addressptr;
int      value;

accessparms = 1;  /* A16, Motorola, nonprivileged data. */
addressptr =  MapVXIAddress (accessparms, (long)0xc106,
              (long)0x7fffffff, &windownum, &ret);
if (ret >= 0)  /* If a valid pointer was returned. */
{
   VXIpeek (addressptr, 2, &value);
}
```

VXIpoke

Syntax:

BASIC Syntax	`CALL VXIpoke (addressptr&, accwidth%, value&)`
C Syntax	`VXIpoke (addressptr, accwidth, value)`

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

Remarks:

Input parameters:

`addressptr`	`long`	Address pointer obtained from `MapVXIAddress`
`accwidth`	`integer`	Byte, word or longword 1 = Byte 2 = Word 4 = Longword
`value`	`long`	Data value to write

Output parameters:

none

Return value:

none

BASIC Example:

```
' Write the value &HFD04& (REQT event) to the Signal register of the
' device at Logical Address 0.

accessparms% = 1     ' A16, Motorola, nonprivileged data.
addressptr& = MapVXIAddress (accessparms%, &HC008&, &H7FFFFFFF&,
                             windownum&, ret%)
IF ret% >= 0& THEN   ' If a valid pointer was returned.
   value& = &HFD04&
   CALL VXIpoke (addressptr&, 2, value&)
END IF
```

C Example:

```
    /* Write the value 0xfd04 (REQT event) to the Signal register of the
       device at Logical Address 0. */

   int      accessparms;
   long     windownum;
   int      ret;
   long     addressptr;
   long     value;

   accessparms = 1;  /* A16, Motorola, nonprivileged data. */
   addressptr =  MapVXIAddress (accessparms, (long)0xc008, (long)0x7fffffff,
                 &windownum, &ret);
   if (ret >= 0)  /* If a valid pointer was returned. */
   {
      value = 0xfd04L;
      VXIpoke (addressptr, 2, value);
   }
```

Chapter 6
High-Level VXIbus Access Functions

This chapter describes the functions in the LabWindows VXI High-Level VXIbus Access Library. Low-level and high-level VXIbus Access functions are used 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/VME device
- Implementing shared memory protocols

With high-level 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.

The functions are explained in both BASIC and C syntax, and are arranged alphabetically. The following five functions are described in this chapter:

- `VXIin`
- `VXIinReg`
- `VXImove`
- `VXIout`
- `VXIoutReg`

VXIin

Syntax:

BASIC Syntax	`ret% = VXIin% (accessparms%, address&, accwidth%, value)`
C Syntax	`ret = VXIin (accessparms, address, accwidth, 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`	`integer`	(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`	`long`	VXI address within specified space
`accwidth`	`integer`	Read Width 1 = Byte 2 = Word 4 = Longword

Output parameter:

`value`	`void`	Value read (byte, integer, or long).

Return value:

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

BASIC Example:

```
' Read Protocol register of the device at Logical Address 4.

accessparms% = 1
address& = &HC108&   ' &HC000 + LA * &H40 + Protocol register offset 8.
accwidth% = 2
ret% = VXIin% (accessparms%, address&, accwidth%, value%)
IF ret% < 0 THEN
   ' Error occurred during read.
END IF
```

C Example:

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

int      ret;
int      accessparms;
long     address;
int      accwidth;
int      value;

accessparms = 1;
address = 0xc108L;  /* 0xc000 + LA * 0x40 + Protocol register offset 8. */
accwidth = 2;
ret = VXIin (accessparms, address, accwidth, &value);
if (ret != 0)
   /* Error occurred during read. */;
```

VXIinReg

Syntax:

BASIC Syntax	`ret% = VXIinReg% (la%, reg%, value%)`
C 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`	`integer`	Logical address of the device to read from
`reg`	`integer`	Offset within VXI logical address registers

Output parameter:

`value`	`integer`	Value read from the device VXI register

Return value:

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

BASIC Example:

```
' Read Protocol register of the device at Logical Address 4.

la% = 4
reg% = 8    ' Protocol register offset.
ret% = VXIinReg% (la%, reg%, value%)
IF ret% < 0 THEN
   ' Error occurred during read.
END IF
```

C Example:

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

int      ret;
int      la;
int      reg;
int      value;

la = 4;
reg = 8;    /* Protocol register offset. */
ret = VXIinReg (la, reg, &value);
if (ret != 0)
   /* Error occurred during read. */;
```

VXImove

Syntax:

BASIC Syntax	`ret% = VXImove% (srcparms%, srcaddr, destparms%, destaddr, length%, accwidth%)`
C Syntax	`ret = VXImove (srcparms, srcaddr, destparms, destaddr, length, accwidth)`

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` `integer`

(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` `any`

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` `integer`

(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` `any`

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`	`long`	Number of elements to transfer
`accwidth`	`integer`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword

Output parameters:

none

Return value:

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

BASIC Example:

```
' Move 1 kilobyte from A24 space at &H200000& to a local buffer.

DIM destaddr AS STRING * 1024
srcparms% = 2                 ' A24, nonprivileged data, Motorola
srcaddr& = &H200000&
destparms% = 0                ' Local space.
length& = &H400&              ' 1 kilobyte.
accwidth% = 2                 ' Transfer as words.
ret% = VXImove% (srcparms%, srcaddr&, destparms%, destaddr$, length%,
                 accwidth%)
IF ret% < 0 THEN
   ' Error occurred during VXImove.
END IF
```

C Example:

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

int      ret;
int      srcparms;
long     srcaddr;
int      destparms;
char     destaddr[1024];
long     length;
int      accwidth;

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

VXIout

Syntax:

BASIC Syntax	`ret% = VXIout% (accessparms%, address&, accwidth%, value&)`
C Syntax	`ret = VXIout (accessparms, address, accwidth, 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`	`integer`	(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`	`long`	VXI address within specified address space
`accwidth`	`integer`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword
`value`	`long`	Data value to write

Output parameters:

none

Return value:

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

BASIC Example:

```
' Write the value &HFD04 (the REQT event for Logical Address 4) to the
' Signal register of the device at Logical device at Address 0.

accessparms% = 1
address& = &HC008&   ' address = &HC000 + LA * &H40 + register offset 8
accwidth% = 2
value& = &HFD04&     ' REQT
ret% = VXIout% (accessparms%, address&, accwidth%, value&)
IF ret% < 0 THEN
   ' Error occurred during write.
END IF
```

C Example:

```
 /* Write the value 0xfd04 (the REQT event for Logical Address 4) to the
    Signal register of the device at Logical Address 0. */

int      ret;
int      accessparms;
long     address;
int      accwidth;
long     value;

accessparms = 1;
address = 0xc008L;  /* address = 0xc000 + LA * 0x40 + register offset 8 */
accwidth = 2;
value = 0xfd04L;    /* REQT */
ret = VXIout (accessparms, address, accwidth, value);
if (ret < 0)
   /* Error occurred during write. */;
```

VXIoutReg

Syntax:

BASIC Syntax	`ret% = VXIoutReg% (la%, reg%, value%)`
C 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`	`integer`	Logical address of the device to write to
`reg`	`integer`	Offset within VXI logical address registers
`value`	`integer`	Value written to the device VXI register

Output parameters:

none

Return value:

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

BASIC Example:

```
' Write Signal register of the device at Logical Address 0 with the
' value &HFD0A (REQT for Logical Address 10).

la% = 0
reg% = 8              ' Signal register offset
value% = &HFD0A       ' REQT for Logical Address 10
ret% = VXIoutReg% (la%, reg%, value%)
IF ret% < 0 THEN
   ' Error occurred during write.
END IF
```

C Example:

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

int      ret;
int      la;
int      reg;
int      value;

la = 0;
reg = 8;          /* Signal register offset */
value = 0xfd0a;/* REQT for Logical Address 10 */
ret = VXIoutReg (la, reg, value);
if (ret != 0)
   /* Error occurred during write. */;
```

Chapter 7
Local Resource Access Functions

This chapter describes the functions in the LabWindows VXI Local Resource Access Library. Local resources are hardware and/or software capabilities that are reserved for the local CPU (the CPU on which the NI-VXI interface resides). 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.

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.

The functions are explained in both BASIC and C syntax, and are arranged alphabetically. The following eight functions are described in this chapter:

- `GetMyLA`
- `ReadMODID`
- `SetMODID`
- `VXIinLR`
- `VXImemAlloc`
- `VXImemCopy`
- `VXImemFree`
- `VXIoutLR`

GetMyLA

Syntax:

BASIC Syntax	`la% = GetMyLA% ()`
C 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`	`integer`	Logical address of the local device

BASIC Example:

```
' Get my logical address.

la% = GetMyLA% ()
```

C Example:

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

int      la;

la = GetMyLA();
```

ReadMODID

Syntax:

BASIC Syntax	`ret% = ReadMODID% (modid%)`
C Syntax	`ret = ReadMODID (modid)`

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

Remarks:

Input parameters:

none

Output parameter:

`modid`	`integer`	Bit vector as follows:

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

Return value:

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

BASIC Example:

```
' Read all the MODID lines 0 to 12.

ret% = ReadMODID% (modid%)
IF ret% <> 0 THEN
   ' Error occurred in ReadMODID.
END IF
```

C Example:

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

int      ret;
int      modid;

ret = ReadMODID (&modid);
if (ret != 0)
   /* Error occurred in ReadMODID. */;
```

SetMODID

Syntax:

BASIC Syntax	`ret% = SetMODID% (enable%, modid%)`
C Syntax	`ret = SetMODID (enable, modid)`

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

Remarks:

Input parameters:

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

Output parameters:

none

Return value:

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

BASIC Example:

```
' Set all the MODID lines 0 to 12.

enable% = 1
modid% = &H1FFF  ' Bit vector (Bits 0 to 12).

ret% = SetMODID% (enable%, modid%)
IF ret% <> 0 THEN
   ' Error occurred in SetMODID.
END IF
```

C Example:

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

int      ret;
int      enable;
int      modid;

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

ret = SetMODID (enable, modid);
if (ret != 0)
   /* Error occurred in SetMODID. */;
```

VXIinLR

Syntax:

BASIC Syntax	`ret% = VXIinLR% (reg%, accwidth%, value)`
C Syntax	`ret = VXIinLR (reg, accwidth, 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`	`integer`	Offset within VXI logical address registers
`accwidth`	`integer`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword

Output parameter:

`value`	`any`	Data value read (byte, integer, or long)

Return value:

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

BASIC Example:

```
' Read the value of the local VXI Status register.

reg% = 4              ' VXI Status register offset within registers.
accwidth% = 2         ' Read word register.
ret% = VXIinLR% (reg%, accwidth%, value%)
IF ret% <> 0 THEN
   ' Error in VXIinLR.
END IF
```

C Example:

```
 /* Read the value of the local VXI Status register. */

intret;
intreg;
intaccwidth;
intvalue;

reg = 4;          /* VXI Status register offset within registers. */
accwidth = 2;     /* Read word register. */
ret = VXIinLR (reg, accwidth, &value);
if (ret != 0)
   /* Error in VXIinLR. */;
```

VXImemAlloc

Syntax:

BASIC Syntax	`ret% = VXImemAlloc% (size&, useraddr$, vxiaddr&)`
C 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. This function can be used for setting up shared memory transfers.

Remarks:

Input parameter:

`size`	`long`	Number of bytes to allocate

Output parameters:

`useraddr`	`string`	Returned application memory buffer address (in standalone C, this parameter is type `void*`)
`vxiaddr`	`long`	Returned remote VXI memory buffer address

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Allocate, use, and free 32 kilobytes of VXI Shared system RAM.

size& = &H8000&      ' 32 kilobytes
ret% = VXImemAlloc% (size&, useraddr$, vxiaddr&)
IF ret% < 0 THEN
   ' Error in VXImemAlloc.
END IF

' Use buffer.

ret% = VXImemFree% (useraddr$)
IF ret% <> 0 THEN
   ' Error in VXImemFree.
END IF
```

C Example:

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

long     size;
char*    useraddr;
long     vxiaddr;
int      ret;

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

/*
   Use buffer.
*/

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

VXImemCopy

Syntax:

BASIC Syntax	`ret% = VXImemCopy% (useraddr$, bufaddr&, size&, dir&)`
C 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`	`string`	User address returned by `VXImemAlloc` (in standalone C, this parameter is type `void*`)
`bufaddr`	`long`	User's local buffer address
`size`	`long`	Size of buffer to be copied
`dir`	`integer`	Direction of transfer 1 = Copy from `bufaddr` to `useraddr` 0 = Copy from `useraddr` to `bufaddr`

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Buffer copied successfully -1 = Copy failed -5 = Invalid `dir`

BASIC Example:

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

DIM bufaddr% (16384)
size& = &H8000&      ' 32 kilobytes
ret% = VXImemAlloc% (size&, useraddr$, vxiaddr&)
IF ret% < 0 THEN
   ' Error in VXImemAlloc.
END IF

' Remote Bus Master access.

IF ret% = 1 THEN
   ret% = VXImemCopy% (useraddr$, bufaddr&, size&, 0)
END IF
' Use the buffer.
ret% = VXImemFree% (useraddr$)
```

C Example:

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

   long     size;
   char*    useraddr;
   long     vxiaddr;
   int      ret;
   int      bufaddr[0x4000];

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

         /* Error in VXImemAlloc. */

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

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

   /*
      Use buffer.
   */

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

VXImemFree

Syntax:

BASIC Syntax	`ret% = VXImemFree% (useraddr$)`
C 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`	`string`	User address (returned by `VXImemAlloc`) to be freed (in standalone C, this parameter is type `void*`)

Output parameters:

none

Return value:

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

BASIC Example:

```
' Allocate, use, and free 32 kilobytes of VXI Shared system RAM.

size& = &H8000&
ret% = VXImemAlloc% (size&, useraddr$, vxiaddr&)
IF ret% <> 0 THEN
   ' Error in VXImemAlloc.
END IF

' Use buffer.

ret% = VXImemFree% (useraddr$)
IF ret% <> 0 THEN
   ' Error in VXImemFree.
END IF
```

C Example:

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

long     size;
char*    useraddr;
long     vxiaddr;
int      ret;

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

/*
   Use buffer.
*/

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

VXIoutLR

Syntax:

BASIC Syntax	`ret% = VXIoutLR% (reg%, accwidth%, value&)`
C Syntax	`ret = VXIoutLR (reg, accwidth, 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`	`integer`	Offset within VXI logical address registers
`accwidth`	`integer`	Byte, word, or longword 1 = Byte 2 = Word 4 = Longword
`value`	`long`	Data value to write

Output parameters:

none

Return value:

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

BASIC Example:

```
' Write the value of &HFD00 (REQT) to the local Signal register.

reg% = 8              ' Register offset for Signal register.
accwidth% = 2         ' Word register.
value& = &HFD00       ' REQT.
ret% = VXIoutLR% (reg%, accwidth%, value&)
IF ret% <> 0 THEN
   ' Error in VXIoutLR.
END IF
```

C Example:

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

   intret;
   intreg;
   intaccwidth;
   long     value;

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

Chapter 8
VXI Signal Functions

This chapter describes the functions in the LabWindows VXI Signal Library. 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.

The functions are explained in both BASIC and C syntax, and are arranged alphabetically. The following nine functions are described in this chapter:

- `DisableSignalInt`
- `EnableSignalInt`
- `GetSignalHandler`
- `RouteSignal`
- `SetSignalHandler`
- `SignalDeq`
- `SignalEnq`
- `SignalJam`
- `WaitForSignal`

DisableSignalInt

Syntax:

BASIC Syntax	`ret% = DisableSignalInt% ()`
C 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`	`integer`	Return Status 0 = Signal interrupts successfully disabled

BASIC Example:

```
' Disable the signal interrupt.

ret% = DisableSignalInt% ()
```

C Example:

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

int      ret;

ret = DisableSignalInt ();
```

EnableSignalInt

Syntax:

BASIC Syntax	`ret% = EnableSignalInt% ()`
C 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`	`integer`	Return Status 1 = Signal queue full, will enable after dequeuing a signal 0 = Signal interrupts successfully enabled

BASIC Example:

```
' Enable the signal interrupt.

ret% = EnableSignalInt% ()
```

C Example:

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

int      ret;

ret = EnableSignalInt ();
```

GetSignalHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetSignalHandler (la)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

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

BASIC Example :

none

C Example:

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

void     (*func)();
intla;

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

RouteSignal

Syntax:

BASIC Syntax	`ret% = RouteSignal% (la%, modemask&)`
C 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 at interrupt service routine time by an installed signal handler for the specified logical address.

Remarks:

Input parameters:

`la` `integer` Logical address to set handler for (-1 = all known la's)

`modemask` `long` 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
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	integer	Return Status 0 = Successful -1 = Invalid la

BASIC 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.

la% = 4
modemask& = &HF9FF&
ret% = RouteSignal% (la%, modemask&)
```

C 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. */

intla;
long     modemask;
int      ret;

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

BASIC 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.

la% = 7
modemask& = &H0FF0&
ret% = RouteSignal% (la%, modemask&)
```

C 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. */

intla;
long     modemask;
int      ret;

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

SetSignalHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetSignalHandler (la, func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

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

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     func (int);
int      la;
int      ret;

la = 5;
ret = SetSignalHandler (la, func);


   /* This is a sample VXI signal handler. */
void func (sigval)
int sigval;          /* signal value received. */
{
}
```

SignalDeq

Syntax:

BASIC Syntax	`ret% = SignalDeq% (la%, signalmask&, sigval%)`
C Syntax	`ret = SignalDeq (la, signalmask, sigval)`

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

Remarks:

Input parameters:

`la` `integer` Logical address to dequeue signal from (255=VME intrrupt routed to signal queue;-1=any known la)

`signalmask` `long` 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
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:

`sigval` `integer` Signal value dequeued from the signal queue

Return value:

`ret`	`integer`	Return Status 0 = A signal was returned in `sigval` -1 = The signal queue is empty or no match

BASIC Example:

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

la% = 10
signalmask& = &HFFFF&
ret% = SignalDeq% (la%, signalmask&, sigval%)
IF ret% <> 0 THEN
   ' Empty signal queue for Logical Address 10.
END IF
```

C Example:

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

int      ret;
int      la;
int      sigval;
long     signalmask;

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

SignalEnq

Syntax:

BASIC Syntax	`ret% = SignalEnq% (sigval%)`
C Syntax	`ret = SignalEnq (sigval)`

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

Remarks:

Input parameter:

`sigval`	`integer`	Value to enqueue at the tail of the signal queue

Output parameters:

none

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Enqueue signal &HFD02 (REQT for Logical Address 2) at the tail of the
' signal queue.

sigval% = &HFD02
ret% = SignalEnq% (sigval%)
IF ret% <> 0 THEN
   ' signal queue is full.
END IF
```

C Example:

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

int      ret;
int      sigval;

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

SignalJam

Syntax:

BASIC Syntax	`ret% = SignalJam% (sigval%)`
C Syntax	`ret = SignalJam (sigval)`

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

Remarks:

Input parameter:

`sigval`	`integer`	Signal value to put on the head of the queue

Output parameters:

none

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Put signal &HFD02 (REQT for Logical Address 2) on the head of the signal
' queue.

sigval% = &HFD02
ret% = SignalJam% (sigval%)
IF ret% <> 0 THEN
   ' signal queue is full.
END IF
```

C Example:

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

int      ret;
int      sigval;

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

WaitForSignal

Syntax:

BASIC Syntax	`ret% = WaitForSignal% (la%, signalmask&, timeout&, retsignal%, retsignalmask&)`
C 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` `integer` — Logical address of device sourcing the signal (255=VME intrrupt routed to signal queue;-1=any known la)

`signalmask` `long` — A bit vector indicating the type(s) of signals that the application waits for; a one in any bit position causes 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
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` `long` — Time to wait until signal occurs (0 = forever)

Output parameters:

`retsignal`	`integer`	Signal received
`retsignalmask`	`long`	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`	`integer`	Return Status 0 = One of the specified signals was received -1 = Invalid `la` -2 = Timeout occurred while waiting for the specified signal(s)

BASIC Example:

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

la% = 5
signalmask& = &H0200&
timeout& = 2000&     ' 2000 milliseconds = 2 seconds.
ret% = WaitForSignal% (la%, signalmask&, timeout&, retsignal%,
            retsignalmask&)
IF ret% <> 0 THEN
   ' signal received within specified waiting period.
END IF
```

C Example:

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

int      ret;
int      la;
long     signalmask;
long     timeout;
int      retsignal;
long     retsignalmask;

la = 5;
signalmask = 0x0200L;
timeout = 2000L;     /* 2000 milliseconds = 2 seconds. */
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:

BASIC Syntax	none
C Syntax	`DefaultSignalHandler (sigval)`

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.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

`sigval`	`integer`	Actual 16-bit VXI signal

Output parameters:

none

Return value:

none

Chapter 9
VXI Interrupt Functions

This chapter describes the functions in the LabWindows VXI Interrupt Library. 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. On most 680X0-based VME CPUs, this 8-bit value is used as a local interrupt vector value and routed directly to the 680X0 processor. This value is used to look up which interrupt service routine to invoke. In VXI, 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.

The functions are explained in both BASIC and C syntax, and are arranged alphabetically. The following 11 functions are described in this chapter:

- `AcknowledgeVXIint`
- `AssertVXIint`
- `DeAssertVXIint`
- `DisableVXIint`
- `DisableVXItoSignalInt`
- `EnableVXIint`
- `EnableVXItoSignalInt`
- `GetVXIintHandler`
- `RouteVXIint`
- `SetVXIintHandler`
- `VXIintAcknowledgeMode`

AcknowledgeVXIint

Syntax:

BASIC Syntax	`ret% = AcknowledgeVXIint% (controller%, level%, statusId&)`
C 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 intended for debug purposes only. VXI interrupts are automatically acknowledged when the local CPU is sensitized to interrupts via the `EnableVXIint` or `EnableVXItoSignalInt` functions.

Remarks:

Input parameters:

`controller`	`integer`	Controller on which to acknowledge interrupt
`level`	`integer`	Interrupt level to acknowledge

Output parameter:

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

Return value:

`ret`	`integer`	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

BASIC Example:

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

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

C Example:

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

int      controller;
int      level;
long     statusId;
int      ret;

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

AssertVXIint

Syntax:

BASIC Syntax	`ret% = AssertVXIint% (controller%, level%, statusId&)`
C 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`	`integer`	Controller on which to assert interrupt
`level`	`integer`	Interrupt level to assert
`statusId`	`long`	Status/ID to present during IACK cycle

Output parameters:

none

Return value:

`ret`	`integer`	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`

BASIC Example:

```
' Assert Interrupt 4 on the local CPU (or first extended controller) with
' status/ID of &H1111&.

controller% = -1
level% = 4
statusId& = &H1111&
ret% = AssertVXIint% (controller%, level%, statusId&)
```

C Example:

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

int      ret;
int      controller;
int      level;
long     statusId;

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

DeAssertVXIint

Syntax:

BASIC Syntax	`ret% = DeAssertVXIint% (controller%, level%)`
C Syntax	`ret = DeAssertVXIint (controller, level)`

Action: Asynchronously deasserts 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 debug purposes only. Deasserting a VXI interrupt can cause a violation of the VME and VXIbus specifications.

Remarks:

Input parameters:

`controller`	`integer`	Controller on which to deassert interrupt
`level`	`integer`	Interrupt level to deassert

Output parameters:

none

Return value:

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

BASIC Example:

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

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

C Example:

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

int      controller;
int      level;
int      ret;

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

DisableVXIint

Syntax:

BASIC Syntax	`ret% = DisableVXIint% (controller%, levels%)`
C Syntax	`ret = DisableVXIint (controller, levels)`

Action: Desensitizes the local CPU to specified VXI interrupts generated in the specified controller that 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.

Remarks:

Input parameters:

`controller`	`integer`	Controller (embedded or extended) to disable interrupts
`levels`	`integer`	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`	`integer`	Return Status 0 = VXI interrupt disabled -1 = No hardware support -2 = Invalid `controller`

BASIC Example:

```
' Disable VXI Interrupt 4 on the local CPU (or first extended controller).

controller% = -1 ' Local CPU or first frame.
levels% = &H0008           ' Interrupt level 4.
ret% = DisableVXIint% (controller%, levels%)
```

C Example:

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

int      controller;
int      levels;
int      ret;

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

DisableVXItoSignalInt

Syntax:

BASIC Syntax	`ret% = DisableVXItoSignalInt% (controller%, levels%)`
C Syntax	`ret = DisableVXItoSignalInt (controller, levels)`

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

Remarks:

Input parameters:

`controller`	`integer`	Controller (embedded or extended) to disable interrupts
`levels`	`integer`	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`	`integer`	Return Status 0 = VXI interrupt disabled -1 = No hardware support -2 = Invalid `controller` specified

BASIC Example:

```
' Disable VXI Interrupt 6 on the local CPU (or first extended controller).

controller% = -1 ' Local CPU or first frame.
levels% = &H0020           ' Interrupt level 6.
ret% = DisableVXItoSignalInt% (controller%, levels%)
```

C Example:

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

int      controller;
int      levels;
int      ret;

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

EnableVXIint

Syntax:

BASIC Syntax	`ret% = EnableVXIint% (controller%, levels%)`
C Syntax	`ret = EnableVXIint (controller, levels)`

Action: Sensitizes the local CPU to specified VXI interrupts generated in the specified controller that 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 VXI/VME interrupt.

Remarks:

Input parameters:

`controller`	`integer`	Controller (embedded or extended) to enable interrupts
`levels`	`integer`	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`	`integer`	Return Status 0 = VXI interrupt enabled -1 = No hardware support -2 = Invalid `controller` specified

BASIC Example:

```
' Enable VXI Interrupt 4 on the local CPU (or first extended controller).

controller% = -1 ' Local CPU or first frame.
levels% = &H0008           ' Interrupt level 4.
ret% = EnableVXIint% (controller%, levels%)
```

C Example:

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

int      controller;
int      levels;
int      ret;

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

EnableVXItoSignalInt

Syntax:

BASIC Syntax	`ret% = EnableVXItoSignalInt% (controller%, levels%)`
C Syntax	`ret = EnableVXItoSignalInt (controller, levels)`

Action: Sensitizes the local CPU to specified VXI interrupts generated in the specified controller that 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 VXI signal.

Remarks:

Input parameters:

`controller`	`integer`	Controller (embedded or extended) to enable interrupts
`levels`	`integer`	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`	`integer`	Return Status 0 = VXI interrupt enabled -1 = No hardware support -2 = Invalid `controller` specified

BASIC Example:

```
' Enable VXI Interrupt 6 on the local CPU (or first extended controller).

controller% = -1 ' Local CPU or first frame.
levels% = &H0020           ' Interrupt level 6.
ret% = EnableVXItoSignalInt% (controller%, levels%)
```

C Example:

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

int      controller;
int      levels;
int      ret;

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

GetVXIintHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetVXIintHandler (level)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     (*func)();
intlevel;

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

RouteVXIint

Syntax:

BASIC Syntax	`ret% = RouteVXIint% (controller%, Sroute%)`
C 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`	`integer`	Controller (embedded or extended) to specify route for
`Sroute`	`integer`	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`	`integer`	Return Status 0 = Successful -1 = No hardware support

BASIC Example:

```
' Route VXI interrupts for level 4 (on the local controller) to the VXI
' interrupt handler and the rest of the levels to the signal processor.

controller% = -1
Sroute% = &HFFF7
ret% = RouteVXIint% (controller%, Sroute%)
```

C Example:

```
    /* Route VXI interrupts for level 4 (on the local controller) to the
       VXI interrupt handler and the rest of the levels to the signal
       processor. */

   int      controller;
   int      Sroute;
   int      ret;

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

SetVXIintHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetVXIintHandler (levels, func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

`levels`	`integer`	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`	`void (*)()`	Pointer to the new VXI interrupt handler (NULL = `DefaultVXIintHandler`)

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     func (int, int, long) ;
intlevels;
intret;

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



   /* This is a sample VXI interrupt handler. */
void func (controller, level, statusId)
int controller;         /* Controller VXI interrupt received from. */
int level;                 /* VXI interrupt level. */
long statusId;          /* 32-bit VXI interrupt acknowledge (IACK)
                        status/ID. */
{
}
```

VXIintAcknowledgeMode

Syntax:

BASIC Syntax	`ret% = VXIintAcknowledgeMode% (controller%, modes%)`
C Syntax	`ret = VXIintAcknowledgeMode (controller, modes)`

Action: Specifies whether to handle the VXI interrupt acknowledge cycle for the specified controller (embedded or extended) for the specified levels as Release On AcKnowledge (ROAK) interrupts or as Release On Register Access (RORA) interrupts. If the VXI interrupt level is handled as a RORA VXI interrupt, further local interrupt generation is automatically inhibited while 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` `integer` Controller (embedded or extended) to specify route for

`modes` `integer` 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` `integer` Return Status

0 = VXI interrupt enabled
-1 = No hardware support
-2 = Invalid `controller` specified
-5 = Invalid `modes` specified

BASIC Example:

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

controller% = -1 ' Local CPU or first frame.
modes% = &H0006  ' Levels 2 and 3 are RORA mode.
ret% = VXIintAcknowledgeMode% (controller%, modes%)
```

C Example:

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

int      controller;
int      modes;
int      ret;

controller = -1;          /** Local CPU or first frame. **/
                          /** Levels 2 and 3 are RORA mode. **/
modes = (int)((1<<1) | (1<<2));
ret = VXIintAcknowledgeMode (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:

BASIC Syntax	none
C 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`.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

`controller`	`integer`	Controller (embedded or extended) that interrupted
`level`	`integer`	The received VXI interrupt level
`statusId`	`long`	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 10
VXI Trigger Functions

This chapter describes the functions in the LabWindows VXI Trigger Library. The trigger functions fall into three categories:

- Source trigger functions act as a standard interface for asserting (sourcing) triggers, as well as for detecting acknowledgments from accepting devices.
- Acceptor trigger functions act as a standard interface for sensing (accepting) triggers, as well as for sending acknowledgments back to the sourcing device.
- Map trigger functions act as configuration tools for multiframe and local support for VXI triggers.

The actual capabilities of specific systems are based on the triggering capabilities of the hardware devices involved (both the sourcing and accepting devices).

The functions are explained in both BASIC and C syntax, and are arranged alphabetically. The following 13 functions are described in this chapter:

- `AcknowledgeTrig`
- `DisableTrigSense`
- `EnableTrigSense`
- `GetTrigHandler`
- `MapTrigToTrig`
- `SetTrigHandler`
- `SrcTrig`
- `TrigAssertConfig`
- `TrigCntrConfig`
- `TrigExtConfig`
- `TrigTickConfig`
- `UnMapTrigToTrig`
- `WaitForTrig`

AcknowledgeTrig

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 *AcknowledgeTTLtrig* or *AcknowledgeECLtrig* with the same parameters as described below.

Syntax:

BASIC Syntax	`ret% = AcknowledgeTrig% (controller%, line%)`
C Syntax	`ret = AcknowledgeTrig (controller, line)`

Action: Acknowledges the specified trigger on the specified controller. The trigger interrupt handler is called after a 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.

Remarks:

Input parameters:

`controller`	`integer`	Controller on which to acknowledge trigger interrupt
`line`	`integer`	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 (GPIO 0 to 9)

Output parameters:

none

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Acknowledge the ECL trigger interrupt for line 1 on the local CPU
' (or the first extended controller).

controller% = -1
line% = 9
ret% = AcknowledgeTrig% (controller%, line%)
```

C Example:

```
    /* Acknowledge the ECL trigger interrupt for line 1 on the local CPU
       (or the first extended controller). */

   int      controller;
   int      line;
   int      ret;

   controller = -1;
   line = 9;  /* ECL line + 8 */
   ret = AcknowledgeTrig (controller, line);
```

DisableTrigSense

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 *DisableTTLsense* or *DisableECLsense* with the same parameters as described below.

Syntax:

BASIC Syntax	`ret% = DisableTrigSense% (controller%, line%)`
C Syntax	`ret = DisableTrigSense (controller, line)`

Action: Disables the sensing of the specified trigger line that was enabled by `EnableTrigSense`.

Remarks:

Input parameters:

`controller`	`integer`	Controller on which to disable sensing
`line`	`integer`	TTL, ECL, or external 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`	`integer`	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

BASIC Example:

```
' Disable sensing of ECL line 1 on the local CPU
' (or the first extended controller).

controller% = -1
line% = 9
ret% = DisableTrigSense% (controller%, line%)
```

C Example:

```
 /* Disable sensing of ECL line 1 on the local CPU
    (or the first extended controller). */

int      ret;
int      controller;
int      line;

controller = -1;
line = 9;  /* ECL line + 8 */
ret = DisableTrigSense (controller, line);
```

EnableTrigSense

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 *EnableTTLsense* or *EnableECLsense* with the same parameters as described below.

Syntax:

BASIC Syntax	`ret% = EnableTrigSense% (controller%, line%, prot%)`
C Syntax	`ret = EnableTrigSense (controller, line, prot)`

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

Remarks:

Input parameters:

`controller`	`integer`	Controller on which to enable sensing
`line`	`integer`	TTL, ECL, or external 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`	`integer`	Protocol to use 2 = START 3 = STOP 4 = SYNC 5 = SEMI-SYNC 6 = ASYNC

Output parameters:

none

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Enable sensing of ECL line 1 on the local CPU
' (or the first extended controller) for SEMI-SYNC protocol.

controller% = -1
line% = 9
prot% = 5
ret% = EnableTrigSense% (controller%, line%, prot%)
```

C Example:

```
/* Enable sensing of ECL line 1 on the local CPU (or the first extended
   controller) for SEMI-SYNC protocol. */

int      ret;
int      controller;
int      line;
int      prot;

controller = -1;
line = 9;  /* ECL line + 8 */
prot = 5;
ret = EnableTrigSense (controller, line, prot);
```

GetTrigHandler

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 *GetTTLtrigHandler* or *GetECLtrigHandler* with the same parameters as described below.

Syntax:

BASIC Syntax	none
C Syntax	`func = GetTrigHandler (line)`

Action: Returns the address of the current trigger interrupt handler for a specified trigger line, counter, or timer.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

`line`	`integer`	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 tick timers

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

```
 /* Get the address of the ECL trigger interrupt handler for
    ECL trigger line 1. */

void     (*func)();
int      line;

line = 9;  /* ECL line + 8 */
func = GetTrigHandler (line);
```

MapTrigToTrig

Syntax:

BASIC Syntax	`ret% = MapTrigToTrig% (controller%, srcTrig%, destTrig%, mapmode%)`
C Syntax	`ret = MapTrigToTrig (controller, srcTrig, destTrig, mapmode)`

Action: Maps the specified TTL, ECL, Star X, Star Y, external connection (GPIO), or miscellaneous external 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` `integer` Controller on which to map signal lines

`srcTrig` `integer` Source line to map to destination

`destTrig` `integer` Destination line to map 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 (GPIO 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

`mapmode` `integer` Signal conditioning mode (0 = no conditioning)

Bit	Conditioning Effect
0	Synchronize with next CLK10 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` `integer` Return Status

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

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

BASIC Example:

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

controller% = -1           ' Local CPU
srcTrig% = 4' TTL line 4.
destTrig% = 41             ' Front panel out connector.
mapmode% = 0' No conditioning.
ret% = MapTrigToTrig% (controller%, srcTrig%, destTrig%, mapmode%)
```

C Example:

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

int      controller;
int      srcTrig;
int      destTrig;
int      mapmode;
int      ret;

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

SetTrigHandler

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 *SetTTLtrigHandler* or *SetECLtrigHandler* with the same parameters as described below.

Syntax:

BASIC Syntax	none
C 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 function `func`.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

`lines` `integer` 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` `void (*)()` Pointer to the new trigger interrupt handler
0 = `DefaultTrigHandler`
1 = `DefaultTrigHandler2`
Other = Address of new trigger interrupt handler

Output parameters:

none

Return value:

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

BASIC Example:

none

C Example:

```
 /* Set the trigger interrupt handler for ECL trigger line 1. */

void     func (int, int, int);
int      lines;
int      ret;

lines = (int)(1<<(3+8));  /* ECL line + 8 */
ret = SetTrigHandler (lines, func);
```

SrcTrig

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 *SrcTTLtrig* or *SrcECLtrig* with the same parameters as described below.

Syntax:

BASIC Syntax	`ret% = SrcTrig% (controller%, line%, prot%, timeout&)`
C Syntax	`ret = SrcTrig (controller, line, prot, timeout)`

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

Remarks:

Input parameters:

`controller` `integer` Controller on which to source trigger line

`line` `integer` Trigger line to source

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 (GPIO 0 to 9) *
50	TIC counter **
60	TIC tick timers **

`prot` `integer` 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` `long` Timeout value in milliseconds

Output parameters:

none

Return value:

`ret` `integer` 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

BASIC Example:

```
' Source ECL line 1 on the local CPU (or the first extended controller)
' for SEMI-SYNC protocol.

controller% = -1
line% = 9
prot% = 5
timeout& = 0&
ret% = SrcTrig% (controller%, line%, prot%, timeout&)
```

C Example:

```
/* Source ECL line 1 on the local CPU (or the first extended controller)
   for SEMI-SYNC protocol. */

int      ret;
int      controller;
int      line;
int      prot;
long     timeout;

controller = -1;
line = 9;  /* ECL line + 8 */
prot = 5;
timeout = 0L;
ret = SrcTrig (controller, line, prot, timeout);
```

TrigAssertConfig

Syntax:

BASIC Syntax	`ret% = TrigAssertConfig% (controller%, line%, configmode%)`
C Syntax	`ret = TrigAssertConfig (controller, line, configmode)`

Action: Configures the specified TTL/ECL trigger line assertion method. TTL/ECL triggers can be (re-) synchronized 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` `integer` Controller on which to configure assertion mode

`line` `integer` 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)

`configmode` `integer` Configuration mode

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

Bit	General Configuration Modes
0	1 = Pass trigger through asynchronously 0 = Synchronize with next CLK10 edge
1	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` `integer` Return Status

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

BASIC Example:

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

controller% = -1
line% = -1
configmode% = 1
ret% = TrigAssertConfig% (controller%, line%, configmode%)
```

C Example:

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

int      ret;
int      controller;
int      line;
int      configmode;

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

TrigCntrConfig

Syntax:

BASIC Syntax	`ret% = TrigCntrConfig% (controller%, configmode%, source%, count%)`
C Syntax	`ret = TrigCntrConfig (controller, configmode, 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-nsec 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`	`integer`	Controller on which to configure the TIC counter
`configmode`	`integer`	Configuration mode Value — Configuration Mode 0 — Initialize the counter 2 — Reload the counter leaving enabled 3 — Disable/abort any count in progress
`source`	`integer`	Trigger line to configure as input to counter Value — Trigger Line 0 to 7 — TTL trigger lines 0 to 7 8 to 13 — ECL trigger lines 0 to 5 70 — CLK10 71 — EXTCLK connection
`count`	`integer`	Number of input pulses to count before terminating

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Successful -1 = Unsupportable function (no hardware support) -2 = Invalid `controller` -3 = Invalid `source` line -10 = Invalid `configmode` -12 = Counter not initialized -15 = Previous count incomplete

BASIC Example:

```
' Configure counter to count 25 assertions on TTL trigger line 5.
' (Prot parameter in EnableTrigSense determines whether counter
'  accepts SYNC or SEMI-SYNC assertions.)

controller% = -1
configmode% = 0            ' initialize the counter.
source% = 5
count% = 25
ret% = TrigCntrConfig% (controller%, configmode%, source%, count%)
```

C Example:

```
/* Configure counter to count 25 assertions on TTL trigger line 5.
  (Prot parameter in EnableTrigSense determines whether counter
   accepts SYNC or SEMI-SYNC assertions.) */

int controller;
int configmode;
int source;
int count;
int ret;

controller = -1;
configmode = 0;   /* initialize the counter. */
source = 5;
count = 25;
ret = TrigCntrConfig (controller, configmode, source, count);
```

TrigExtConfig

Syntax:

BASIC Syntax	`ret% = TrigExtConfig% (controller%, extline%, configmode%)`
C Syntax	`ret = TrigExtConfig (controller, extline, configmode)`

Action: Configures the external trigger (GPIO) lines. The external trigger lines can be fed back for use in the crosspoint switch output. The external trigger lines can be asserted high or low, or left unconfigured (tri-stated) for use as a crosspoint switch input. If not fed back, the external input can be inverted before mapped to a trigger line.

Remarks:

Input parameters:

`controller` `integer` Controller on which to configure external connection

`extline` `integer` Trigger line to configure

Value	Trigger Line
40 to 49	External source/destination (GPIO 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

`configmode` `integer` 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` `integer` Return Status

0 = Successful
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `extline`
-10 = Invalid `configmode`

BASIC Example:

```
' Configure external line 40 (front panel In) to not be fed back,
' and left tri-stated for use as a mapped input via MapTrigToTrig.
' Invert the front panel In signal.

controller% = -1
extline% = 40
configmode% = 8
ret% = TrigExtConfig% (controller%, extline%, configmode%)
```

C Example:

```
/* Configure external line 40 (front panel In) to not be fed back,
   and left tri-stated for use as a mapped input via MapTrigToTrig.
   Invert the front panel In signal. */

int controller;
int extline;
int configmode;
int ret;

controller = -1;
extline = 40;
configmode = (1<<3);  /* turn on bit 3 */
ret = TrigExtConfig (controller, extline, configmode);
```

TrigTickConfig

Syntax:

BASIC Syntax	`ret% = TrigTickConfig% (controller%, configmode%, source%, tcount1%, tcount2%)`
C Syntax	`ret = TrigTickConfig (controller, configmode, 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` `integer` Controller on which to configure the TIC chip dual 5-bit tick timers

`configmode` `integer` Configuration mode

Value	Configuration Mode
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` `integer` Trigger line to configure as input to counter

Value	Trigger Line
40 to 49	External source/destination (GPIO 0 to 9)
70	CLK10
71	EXTCLK connection

`tcount1` `integer` 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` `integer` Number of input pulses (as a power of two) to count before asserting TICK2 output

Output parameters:

none

Return value:

`ret` `integer` Return Status

3 = Successful disable of the tick timers
2 = Successful reload of the tick timers
1 = Successful initialization of non-rollover mode
0 = Successful initialization of rollover mode
-1 = Unsupportable function (no hardware support)
-2 = Invalid `controller`
-3 = Invalid `source` line
-10 = Invalid `configmode`
-15 = Previous tick configured and enabled

BASIC Example:

```
' 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.

controller% = -1
configmode% = 0             ' Initialize with rollover
source% = 70     ' CLK10
tcount1% = 16               ' Divide down by 65536 (2^16)
tcount2% = 0     ' Does not matter
ret% = TrigTickConfig% (controller%, configmode%, source%, tcount1&,
           tcount2&)
```

C Example:

```
 /* 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. */

int      ret;
int      controller;
int      configmode;
int      source;
int      tcount1, tcount2;

controller = -1;
configmode = 0;/* Initialize with rollover */
source = 70;   /* CLK10 */
tcount1 = 16;     /* Divide down by 65536 (2^16) */
tcount2 = 0;   /* Does not matter */
ret = TrigTickConfig (controller, configmode, source, tcount1, tcount2);
```

UnMapTrigToTrig

Syntax:

BASIC Syntax	`ret% = UnMapTrigToTrig% (controller%, srcTrig%, destTrig%)`
C Syntax	`ret = UnMapTrigToTrig (controller, srcTrig, destTrig)`

Action: Unmaps the specified TTL, ECL, Star X, Star Y, external connection (GPIO), or miscellaneous external signal line that was mapped to another line using the `MapTrigToTrig` function.

Remarks:

Input parameters:

`controller`	`integer`	Controller on which to unmap signal lines
`srcTrig`	`integer`	Source line to unmap from destination
`destTrig`	`integer`	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 (GPIO 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`	`integer`	Return Status 0 = Successful -1 = Unsupported function, no mapping capability -2 = Invalid `controller` -12 = Not previously mapped

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

BASIC 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.

controller% = -1           ' Local CPU
srcTrig% = 4' TTL line 4
destTrig% = 49             ' Front panel out connector
ret% = UnMapTrigToTrig% (controller%, srcTrig%, destTrig%)
```

C 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(). */

int      controller;
int      srcTrig;
int      destTrig;
int      ret;

controller = -1;        /* Local CPU */
src = 4;                /* TTL line 4 */
dest = 49;              /* Front panel out connector */
ret =  UnMapTrigToTrig (controller, srcTrig, destTrig);
```

WaitForTrig

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 *WaitForTTLtrig* or *WaitForECLtrig* with the same parameters as described below.

Syntax:

BASIC Syntax	`ret% = WaitForTrig% (controller%, line%, timeout&)`
C 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`	`integer`	Controller on which to wait for trigger
`line`	`integer`	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`	`long`	Timeout value in milliseconds

Output parameters:

none

Return value:

`ret`	`integer`	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

BASIC Example:

```
' Wait for ECL line 1 on the local CPU (or the first extended controller)
' to be encountered.

controller% = -1
line% = 9
timeout& = 10000&
ret% = WaitForTrig% (controller%, line%, timeout&)
```

C Example:

```
/* Wait for ECL line 1 on the local CPU (or the first extended controller)
   to be encountered. */

int      ret;
int      controller;
int      line;
long     timeout;

controller = -1;
line = 9;  /* ECL line + 8 */
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

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 *DefaultTTLtrigHandler* or *DefaultECLtrigHandler* with the same parameters as described below.

Syntax:

BASIC Syntax	none
C 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.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

`controller` `integer` Controller from which the trigger interrupt is received

`line` `integer` 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` `integer` Conditioning effect

Bit	Conditioning Effect
0	1 = Trigger sensed 0 = Sourced trigger acknowledged
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:

BASIC Syntax	none
C 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.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameters:

`controller` `integer` Controller from which the trigger interrupt is received

`line` `integer` 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` `integer` Conditioning effect

Bit	Conditioning Effect
0	1 = Trigger sensed 0 = Sourced trigger acknowledged
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 11
System Interrupt Handler Functions

This chapter describes the LabWindows VXI System Interrupt Handler Library. With these functions, you can handle miscellaneous system conditions that can occur in the VXI environment, such as Sysfail, ACfail, Bus Error, Soft Reset, and/or Sysreset interrupts. The NI-VXI software interface can handle all of these system conditions for the application through the use of default 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 the `InitVXIlibrary` function is called. 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 handler.

The functions are explained in both BASIC and C syntax, and are arranged alphabetically. The following 19 functions are described in this chapter:

- `AssertSysreset`
- `DisableACfail`
- `DisableSoftReset`
- `DisableSysfail`
- `DisableSysreset`
- `EnableACfail`
- `EnableSoftReset`
- `EnableSysfail`
- `EnableSysreset`
- `GetACfailHandler`
- `GetBusErrorHandler`
- `GetSoftResetHandler`
- `GetSysfailHandler`
- `GetSysresetHandler`
- `SetACfailHandler`
- `SetBusErrorHandler`
- `SetSoftResetHandler`
- `SetSysfailHandler`
- `SetSysresetHandler`

AssertSysreset

Syntax:

BASIC Syntax	`ret% = AssertSysreset% (controller%, resetmode%)`
C Syntax	`ret = AssertSysreset (controller, resetmode)`

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

Remarks:

Input parameter:

`controller`	`integer`	Logical address of mainframe extender on which to assert SYSRESET* -1 = From the local CPU or first extended controller -2 = All controllers
`resetmode`	`integer`	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`	`integer`	Return Status 0 = SYSRESET* signal successfully asserted -1 = No hardware support for this function -2 = Invalid `controller`

BASIC Example:

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

controller% = -1
resetmode% = 0
ret% = AssertSysreset% (controller%, resetmode%)
```

C Example:

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

int      controller;
int      resetmode;
int      ret;

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

DisableACfail

Syntax:

BASIC Syntax	`ret% = DisableACfail% (controller%)`
C 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`	`integer`	Logical address of mainframe extender to disable

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = ACfail interrupt successfully disabled -1 = No hardware support for this function -2 = Invalid `controller`

BASIC Example:

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

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

C Example:

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

int      controller;
int      ret;

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

DisableSoftReset

Syntax:

BASIC Syntax	`ret% = DisableSoftReset% ()`
C 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`	`integer`	Return Status 0 = Soft Reset interrupt successfully disabled -1 = No hardware support for this function

BASIC Example:

```
' Disable the Soft Reset interrupt.

ret% = DisableSoftReset% ()
```

C Example:

```
/* Disable the Soft Reset interrupt. */

int      ret;

ret = DisableSoftReset ();
```

DisableSysfail

Syntax:

BASIC Syntax	`ret% = DisableSysfail% (controller%)`
C 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`	`integer`	Logical address of mainframe extender to disable

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Sysfail interrupt successfully disabled -1 = No hardware support for this function -2 = Invalid `controller`

BASIC Example:

```
' Disable the Sysfail interrupt.

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

C Example:

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

int      controller;
int      ret;

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

DisableSysreset

Syntax:

BASIC Syntax	`ret% = DisableSysreset% (controller%)`
C Syntax	`ret = DisableSysreset (controller)`

Action: Desensitizes the local CPU from Sysreset interrupts from the embedded CPU VXIbus backplane or from the specified extended controller VXI backplane (if external CPU).

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Sysreset interrupt successfully disabled -1 = No hardware support for this function -2 = Invalid `controller`

BASIC Example:

```
' Disable the Sysreset interrupt.

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

C Example:

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

int      controller;
int      ret;

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

EnableACfail

Syntax:

BASIC Syntax	`ret% = EnableACfail% (controller%)`
C 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`	`integer`	Logical address of mainframe extender to enable

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = ACfail interrupt successfully enabled -1 = No hardware support for this function -2 = Invalid `controller`

BASIC Example:

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

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

C Example:

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

int      controller;
int      ret;

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

EnableSoftReset

Syntax:

BASIC Syntax	`ret% = EnableSoftReset% ()`
C 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`	`integer`	Return Status 0 = Soft Reset interrupt successfully enabled -1 = No hardware support for this function

BASIC Example:

```
' Enable the Soft Reset interrupt.

ret% = EnableSoftReset% ()
```

C Example:

```
/* Enable the Soft Reset interrupt. */

int      ret;

ret = EnableSoftReset ();
```

EnableSysfail

Syntax:

BASIC Syntax	`ret% = EnableSysfail% (controller%)`
C 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`	`integer`	Logical address of mainframe extender to enable

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Sysfail interrupt successfully enabled -1 = No hardware support for this function -2 = Invalid `controller`

BASIC Example:

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

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

C Example:

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

int      controller;
int      ret;

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

EnableSysreset

Syntax:

BASIC Syntax	`ret% = EnableSysreset% (controller%)`
C Syntax	`ret = EnableSysreset (controller)`

Action: Sensitizes the local CPU to Sysreset interrupts from the embedded CPU VXIbus backplane or from the specified extended controller VXI backplane (if external CPU).

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Sysreset interrupt successfully enabled -1 = No hardware support for this function -2 = Invalid `controller`

BASIC Example:

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

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

C Example:

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

int      controller;
int      ret;

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

GetACfailHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetACfailHandler ()`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func`	`void (*)()`	Pointer to the current ACfail interrupt handler NULL = No hardware support for this function

BASIC Example:

none

C Example:

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

void     (*func)();

func = GetACfailHandler();
```

GetBusErrorHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetBusErrorHandler()`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

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

BASIC Example:

none

C Example:

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

void     (*func)();

func = GetBusErrorHandler ();
```

GetSoftResetHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetSoftResetHandler ()`

Action: Returns the address of the current Soft Reset interrupt handler.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func`	`void (*)()`	Pointer to the current Soft Reset interrupt handler NULL = No hardware support for this function

BASIC Example:

none

C Example:

```
/* Get the address of the Soft Reset handler. */

void     (*func)();

func = GetSoftResetHandler();
```

GetSysfailHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetSysfailHandler ()`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func`	`void (*)()`	Pointer to the current Sysfail interrupt handler NULL = No hardware support for this function

BASIC Example:

none

C Example:

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

void     (*func)();

func = GetSysfailHandler ();
```

GetSysresetHandler

Syntax:

BASIC Syntax	none
C Syntax	`func = GetSysresetHandler ()`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

`func`	`void (*)()`	Pointer to the current Sysreset interrupt handler NULL = No hardware support for this function

BASIC Example:

none

C Example:

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

void     (*func)();

func = GetSysresetHandler ();
```

SetACfailHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetACfailHandler (func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Successful -1 = No hardware support for this function

BASIC Example:

none

C Example:

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

void     func (int);
int      ret;

ret = SetACfailHandler (func);
```

SetBusErrorHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetBusErrorHandler(func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Successful

BASIC Example:

none

C Example:

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

void     func ();
int      ret;

ret = SetBusErrorHandler(func);
```

SetSoftResetHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetSoftResetHandler (func)`

Action: Replaces the current Soft Reset interrupt handler with a specified handler.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

`func`	`void (*)()`	Pointer to the new Soft Reset interrupt handler NULL = `DefaultSoftResetHandler`

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Successful -1 = No hardware support for this function

BASIC Example:

none

C Example:

```
/* Set the Soft Reset handler. */

void     func ();
int      ret;

ret = SetSoftResetHandler (func);
```

SetSysfailHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetSysfailHandler (func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Successful -1 = No hardware support for this function

BASIC Example:

none

C Example:

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

void     func (int);
int      ret;

ret = SetSysfailHandler (func);
```

SetSysresetHandler

Syntax:

BASIC Syntax	none
C Syntax	`ret = SetSysresetHandler (func)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

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

Output parameters:

none

Return value:

`ret`	`integer`	Return Status 0 = Successful -1 = No hardware support for this function

BASIC Example:

none

C Example:

```
 /* Set the Sysreset handler. */

void     func (int);
int      ret;

ret = SetSysresetHandler (func);
```

Default Handlers for the System Interrupt Handler Functions

The NI-VXI software provides the following default handlers for the system interrupt handler functions. These are sample handlers that `InitVXIlibrary` installs when it initializes the software at the beginning of the application program. Default handlers give you the minimal and most common functionality required for a 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:

BASIC Syntax	none
C Syntax	`DefaultACfailHandler (controller)`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

`controller`	`integer`	Logical address of controller interrupting

Output parameters:

none

Return value:

none

DefaultBusErrorHandler

Syntax:

BASIC Syntax	none
C Syntax	`DefaultBusErrorHandler ()`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

none

DefaultSoftResetHandler

Syntax:

BASIC Syntax	none
C Syntax	`DefaultSoftResetHandler ()`

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

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Parameters:

none

Return value:

none

DefaultSysfailHandler

Syntax:

BASIC Syntax	none
C 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.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

`controller`	`integer`	Logical address of controller interrupting

Output parameters:

none

Return value:

none

DefaultSysresetHandler

Syntax:

BASIC Syntax	none
C 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`.

Note: You can only use this function in standalone C programs or loadable object modules.

Remarks:

Input parameter:

`controller`	`integer`	Logical address of controller interrupting

Output parameters:

none

Return value:

none

Chapter 12
VXIbus Extender Functions

This chapter describes the LabWindows VXIbus Extender Library. 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.

The functions are explained in both BASIC and C syntax, and are arranged alphabetically. The following four functions are described in this chapter:

- `MapECLtrig`
- `MapTTLtrig`
- `MapUtilBus`
- `MapVXIint`

MapECLtrig

Syntax:

BASIC Syntax	`ret% = MapECLtrig% (extender%, lines%, directions%)`
C 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`	`integer`	Mainframe extender for which to map ECL lines
`lines`	`integer`	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`	`integer`	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`	`integer`	Return Status 0 = Successful -1 = Unsupportable function (no hardware support) -2 = Invalid `extender`

BASIC Example:

```
' Map ECL lines 0 and 1 on the mainframe extender at Logical Address 5
' to go into the mainframe.

extender% = 5
lines% = &H003                ' ECL lines 0 and 1.
directions% = &H0003
ret% = MapECLtrig% (extender%, lines%, directions%)
```

C Example:

```
 /* Map ECL lines 0 and 1 on the mainframe extender at Logical Address 5
    to go into the mainframe. */

int      extender;
int      lines;
int      directions;
int      ret;

extender = 5;
lines = (int)((1<<0) | (1<<1));  /* ECL lines 0 and 1. */
directions = (int)((1<<0) | (1<<1));
ret = MapECLtrig (extender, lines, directions);
```

MapTTLtrig

Syntax:

BASIC Syntax	`ret% = MapTTLtrig% (extender%, lines%, directions%)`
C 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`	`integer`	Mainframe extender for which to map TTL lines
`lines`	`integer`	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`	`integer`	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`	`integer`	Return Status 0 = Successful -1 = Unsupportable function (no hardware support) -2 = Invalid `extender`

BASIC Example:

```
' Map TTL lines 4 and 5 on the mainframe extender at Logical Address 5
' to go out of the mainframe.

extender% = 5
lines% = &H0030               ' TTL lines 4, 5.
directions% = &H0
ret% = MapTTLtrig% (extender%, lines%, directions%)
```

C Example:

```
 /* Map TTL lines 4 and 5 on the mainframe extender at Logical Address 5
    to go out of the mainframe. */

int      extender;
int      lines;
int      directions;
int      ret;

extender = 5;
lines = (int)((1<<4) | (1<<5));  /* TTL lines 4, 5. */
directions = (int)0x0000;
ret = MapTTLtrig (extender, lines, directions);
```

MapUtilBus

Syntax:

BASIC Syntax	`ret% = MapUtilBus% (extender%, modes%)`
C 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`	`integer`	Mainframe extender for which to map utility bus signals
`modes`	`integer`	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 mainframe
4	ACfail out of the mainframe
3	Sysfail into the mainframe
2	Sysfail out of the mainframe
1	Sysreset into the mainframe
0	Sysreset out of the mainframe

Output parameters:

none

Return value:

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

BASIC Example:

```
' Map Sysfail into Mainframe 5.  Map Sysreset into and out of Mainframe 5.
' Do not map ACfail at all.

extender% = 5
modes% = &H000B
ret% = MapUtilBus% (extender%, modes%)
```

C Example:

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

int      extender;
int      modes;
int      ret;

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

MapVXIint

Syntax:

BASIC Syntax	`ret% = MapVXIint% (extender%, levels%, directions%)`
C 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`	`integer`	Mainframe extender for which to map VXI interrupt levels
`levels`	`integer`	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`	`integer`	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`	`integer`	Return Status 0 = Successful -1 = Unsupportable function (no hardware support) -2 = Invalid `extender`

BASIC 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.

extender% = 5
levels% = &H0049           ' Levels 1, 4, 7.
directions% = &H0001       ' Level 1 only one in.
ret% = MapVXIint% (extender%, levels%, directions%)
```

C 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. */

   int      extender;
   int      levels;
   int      directions;
   int      ret;

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

Appendix
Customer Communication

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

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

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 24	(1) 48 14 24 14
Germany	089/7 41 31 30	089/7 14 60 35
Hong Kong	(02) 26375019	(02) 226868505
Italy	02/48301892	02/48301915
Japan	(03) 3788-1921	(03) 3788-1923
Korea	(02) 596-7456	(02) 596-7455
Mexico	05/2022544	05/2022544
Netherlands	03480-33466	03480-30673
Norway	32-848400	32-848600
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 27 00 20	056 27 00 25
U.K.	0635 523545	0635 523154
	or 0800 289877 (in U.K. only)	

LabWindows® Technical Support Form

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

Be sure to fax copies of your `AUTOEXEC.BAT` and `CONFIG.SYS` files as well. If one or more National Instruments hardware products are involved in this problem, include the Hardware Configuration form from each hardware product's user manual. Include additional pages as necessary.

Name _____

Company _____

Address _____

_____

Fax (_____)_____ Phone (_____)_____

Computer brand _____ Model _____ Processor _____ Coprocessor _____

Operating system _____ Version _____ Bus (XT/AT/ISA, Micro Channel, or EISA) _____

Speed (MHz) CPU_____BUS_ RAM _____ (Extended) _____ (Expanded) _____

Video Board _____ Mouse (Yes/No) Mouse Type _____ Mouse Driver Version _____

Other adapters installed _____

Base I/O Address Level of Other Boards _____ Interrupt Level of Other Boards _____

Hard disk capacity _____ Brand _____

Instruments used _____

National Instruments hardware product models _____ Version _____

Configuration _____

Base I/O Address of Board(s) _____ Interrupt Level of Board(s) _____

LabWindows Version Number _____ Size and date of LW.EXE file _____

LabWindows Run-Time System Version Number _____ Size and date of LWRTS.EXE file _____

Other National Instruments software product _____ Version _____

Programming Language and Version _____

The problem is _____

_____

_____

List any error messages _____

_____

_____

The following steps will reproduce the problem _____

_____

_____

Documentation Comment Form

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

Title: **LabWindows® VXI Library Reference Manual**

Edition Date: **March 1995**

Part Number: **320318-01**

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

If you find errors in this 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: (512) 794-5678
Technical Publications
National Instruments Corporation
MS 53-02

Glossary

Prefix	Meaning	Value
n-	nano-	10^{-9}
m-	milli-	10^{-3}
k-	kilo-	10^{3}

A

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

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

A32 space — One of the VXIbus address spaces. Equivalent to the VME 4 Gigabyte *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.
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, true/false, 1/0 alternative. The building block of binary coding and numbering systems. Eight bits make up a *byte*.
bit vector	A string of related bits in which each bit has a specific meaning.
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.
byte	A grouping of adjacent binary digits operated on by the computer as a single unit. In VXI systems, a byte consists of 8 bits.
byte order	How bytes are arranged within a word or how words are arranged within a longword. Motorola ordering stores the most significant byte (MSB) or word first, followed by the least significant byte (LSB) or word. Intel ordering stores the LSB or word first, followed by the MSB or word.

C

CLK10	A 10-MHz, ± 100-ppm, individually buffered (to each module slot), differential ECL system clock that is sourced from Slot 0 and distributed to Slots 1 through 12 on P2. It is distributed to each slot as a single-source, single-destination signal with a matched delay of under 8 nsec.
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.
controller	An intelligent device (usually involving a CPU) that is capable of controlling other devices.
CR	Carriage Return; the ASCII character 0Dh.

D

deasserted	A signal in its inactive false state.
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
DirDorAbort	Transfer aborted; device not DIR or not DOR
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.

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.

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.
FIOerr	Error reading or writing file
ForcedAbort	User abort occurred during I/O.

G

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.
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.

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.
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.
InvalidLA	Invalid logical address
IODONE	Successful data transfer

K

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

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.
Memory Class device	A VXIbus device that, in addition to configuration registers, has memory in VME A24 or A32 space that is accessible through addresses on the VME/VXI data transfer bus.
Message-Based device	An intelligent device that implements the defined VXIbus registers and communication protocols. These devices are able to use Word Serial Protocol to communicate with one another through communication registers.
MODID	Module Identification lines; 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.
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.

P

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*.
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-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.
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.

S

sec	seconds
SEMI-SYNC Protocol	A one-line, open collector, multiple-device handshake trigger protocol.
servant	A device controlled by a Commander; there are Message-Based and Register-Based Servants.
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*.
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 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

TC	All bytes received
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.
TIMO_RES	Timed out before response received
TIMO_SEND	Timed out before able to send command
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

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
void	In the C language, a generic data type that can be cast to any specific data type.
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.
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.
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.
write	Copying data to a storage device.
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.

Index

A

B

C

D

E

F

G

H

I

L

M

P

R

S

T

U

V

W