HP EPC-7 486

Embedded Computer, 100 MHz Processor

In Stock

Used and in Excellent Condition

Open Web Page

https://www.artisantg.com/52812-50

EPConnect/VXI for Windows Manual Set

VOL 2 of 3

RadiSys® Corporation

15025 S.W. Koll Parkway

Beaverton, OR 97006

Phone: (503) 646-1800

FAX: (503) 646-1850

http://www.radisys.com

07-0231-03 January 1995

This manual part number is 07-0231-03.
It contains manuals 07-0222-01 and 07-0223-02.

May 1994

Bus Management for Windows Programmer's Reference Guide

RadiSys® Corporation

15025 S.W. Koll Parkway

Beaverton, OR 97006

Phone: (503) 646-1800

FAX: (503) 646-1850

07-0222-01 December 1994

May 1994

Software License and Warranty

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THE DISKETTE OR DISK UNIT PACKAGE. BY OPENING THE PACKAGE, YOU INDICATE THAT YOU ACCEPT THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, YOU SHOULD PROMPTLY RETURN THE UNOPENED PACKAGE, AND YOU WILL BE REFUNDED.

LICENSE

You may:

1. Use the product on a single computer;
2. Copy the product into any machine-readable or printed form for backup or modification purposes in support of your use of the product on a single computer;
3. Modify the product or merge it into another program for your use on the single computer—any portion of this product merged into another program will continue to be subject to the terms and conditions of this agreement;
4. Transfer the product and license to another party if the other party agrees to accept the terms and conditions of this agreement—if you transfer the product, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copy not transferred, including all modified versions and portions of the product contained in or merged into other programs.

You must reproduce and include the copyright notice on any copy, modification, or portion merged into another program.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PRODUCT OR ANY COPY, MODIFICATION, OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION, OR MERGED PORTION OF THE PRODUCT TO ANOTHER PARTY, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

TERM

The license is effective until terminated. You may terminate it at any time by destroying the product and all copies, modifications, and merged portions in any form. The license will also terminate upon conditions set forth elsewhere in this agreement or if you fail to comply with any of the terms or conditions of this agreement. You agree upon such termination to destroy the product and all copies, modifications, and merged portions in any form.

LIMITED WARRANTY

RadiSys Corporation ("RadiSys") warrants that the product will perform in substantial compliance with the documentation provided. However, RadiSys does not warrant that the functions contained in the product will meet your requirements or that the operation of the product will be uninterrupted or error-free.

RadiSys warrants the diskette(s) on which the product is furnished to be free of defects in materials and workmanship under normal use for a period of ninety (90) days from the date of shipment to you.

LIMITATIONS OF REMEDIES

RadiSys' entire liability shall be the replacement of any diskette that does not meet RadiSys' limited warranty (above) and that is returned to RadiSys.

IN NO EVENT WILL RADISYS BE LIABLE FOR ANY DAMAGES, INCLUDING LOST PROFITS OR SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THE PRODUCT EVEN IF RADISYS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

GENERAL

You may not sublicense the product or assign or transfer the license, except as expressly provided for in this agreement. Any attempt to otherwise sublicense, assign, or transfer any of the rights, duties, or obligations hereunder is void.

This agreement will be governed by the laws of the state of Oregon.

If you have any questions regarding this agreement, please contact RadiSys by writing to RadiSys Corporation, 15025 SW Koll Parkway, Beaverton, Oregon 97006.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATION BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

NOTES

Table of Contents

NOTES

1. Introducing Bus Management for Windows

This manual is intended for programmers using the Bus Management for Windows programming interface to develop enhanced mode Windows applications that control I/O modules via the VXI expansion interface on an EPC or a VXLink card. You are expected to have read the *EPConnect/VXI for DOS & Windows User's Guide* for an understanding of what is in EPConnect/VXI, how to install it, and how to use the Start-Up Resource Manager (SURM). You are not expected to have in-depth knowledge of Windows.

Bus Management for Windows is designed to execute under enhanced mode Windows only. It will not execute properly under Windows standard mode. It is also designed to execute on EPC-7 hardware or better. It will not execute properly on an EPC-2.

The Bus Management for Windows API provides a powerful interface for interacting with the VXI expansion interface on an EPC or a VXLink card. The Bus Management API offers considerable flexibility by providing a C/C++ dynamic link library (DLL) interface that obeys the MS Pascal binding conventions. By observing the same conventions, you can use EPConnect with other languages, such as Visual Basic.

This chapter introduces you to the RadiSys® Bus Management for Windows product. In it you will find the following:

- What is in this manual and how to use it
- What is Bus Management for Windows?
- Programming, Compiling and Linking
- What to do next

1.1 How This Manual is Organized

This manual has five chapters:

Chapter 1, *Introduction*, introduces Bus Management for Windows and this manual.

Chapter 2, *Function Descriptions*, describes the major categories of functions and gives complete descriptions of each function. The function descriptions also contain supporting examples or references to an example that demonstrates use of the function. Function descriptions are alphabetic by function name.

Chapter 3, *Advanced Topics*, provides information about byte swapping, interrupts, and the backward-compatibility library.

Chapter 4, *Support and Service*, describes how to contact RadiSys Technical Support for support and service.

1.2 What is Bus Management for Windows?

Bus Management for Windows consists of those portions of the EPConnect software package that are required by C/C++ and Visual BASIC programmers developing VXI control applications that execute in enhanced mode Windows. Figure 1-1 is a diagram of the Bus Management for Windows software architecture that shows how the architecture relates to the VXIbus hardware.

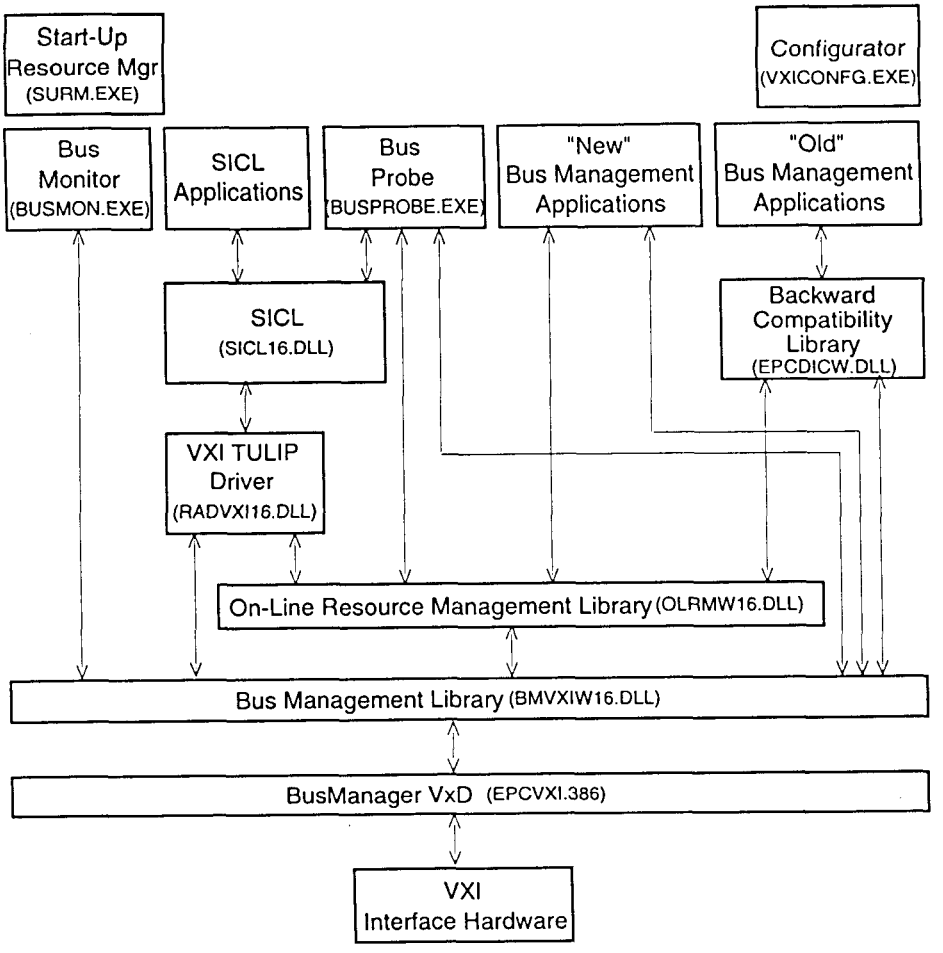

Figure 1-1. EPConnect/VXI for Windows Software Architecture

1.2.1 Bus Management Library and BusManager VxD

The Bus Management library and BusManager VxD are at the foundation of EPConnect. They provide the lowest level interface to the VXIbus hardware through their function libraries. These functions allow you to:

- Control VXIbus word serial registers.
- Send word serial commands of all sizes.
- Transfer blocks of data to and from VXIbus devices, with BERR detection.
- Control EPC Slave memory
- Query EPC driver, firmware, and hardware version or type.

The Bus Management DLL supports ANSI "C" compilers such as Microsoft C/C++ and Borland C/C++, as well as Visual Basic.

The Bus Management Library is fully re-entrant.

1.2.2 OLRM

The On-Line Resource Management library (**OLRMW16.DLL**) provides user applications with access to results of the resource management process, as well as retrieving status information from devices over the VXI bus. A C/C++ language interface is provided to access OLRM functions. OLRM accesses the VXIbus through the Bus Management library and the BusManager VxD.

1.2.3 Backward-Compatibility Library

The Backward-Compatibility Library (**EPCDICW.DLL**) maintains compatibility for applications that were developed using previous versions of EPConnect. It provides a C/C++ interface that is compatible with the Bus Management for DOS API.

1.2.4 SURM

The Start-Up Resource Manager (SURM) determines the physical content of the system and configures the devices. It is typically the first program to run after DOS boots. The SURM is the EPConnect implementation of the resource manager defined in the VXIbus specification. However, SURM extends the specification definition to include non-VXIbus devices, such as GPIB instruments. The SURM uses the **DEVICES** file to obtain device information not directly available from the devices. SURM accesses VXIbus devices in the system directly.

1.3 Programming, Compiling and Linking

This section contains information about programming with Bus Management for Windows. Included is a list of the header files provided, the programming interfaces, and compiling and linking hints.

1.3.1 Header Files

Bus Management for Windows provides the following header files:

BMVXI.BAS A Microsoft Visual Basic (Professional Edition) header file containing constant definitions and function declarations.

BUSMGR.H A "C" header file containing the constant definitions, macro definitions, and function prototypes required to compile EPConnect applications using any Microsoft or Borland "C" or C++ compiler.

BUSMGR.INC A copy of **BUSMGR.H** that's been converted so that it is suitable for inclusion into an assembly language source file.

EPC_OBM.H A "C" header file containing the constant definitions, macro definitions, structure definitions, and function prototypes required to compile Bus Management applications for DOS. This header file also provides backward compatibility for Bus Management for Windows applications written for releases preceding revision 4.0.

EPC_OBM.H should never be included in a source file directly. **BUSMGR.H** includes **EPC_OBM.H**.

EPC_OBM.INC A copy of **EPC_OBM.H** that has been converted so that it is suitable for inclusion into an assembly language source file.

EPC_OBM.INC should never be included in a source file directly. **BUSMGR.INC** includes **EPC_OBM.INC.**

EPCSTD.H A "C" header file containing macro definitions to standardize non-ANSI, compiler-dependent keywords. By using the macros defined here, an application can compile successfully using any revision of Microsoft or Borland "C" or C++ compiler without modifying the source file.

EPCSTD.H should never be included in a source file directly. **BUSMGR.H** includes **EPC_OBM.H.**

EPCSTD.INC A copy of **EPCSTD.H** that has been converted so that it is suitable for inclusion into an assembly language source file.

EPCSTD.INC should never be included in a source file directly. **BUSMGR.INC** includes **EPCSTD.INC.**

OLRM.H A "C" header file containing the constant definitions, macro definitions, and function prototypes required to compile OLRM applications using any Microsoft or Borland C/C++ compiler.

OLRM.INC A copy of **OLRM.H** that has been converted so that it is suitable for inclusion into an assembly language source file.

OBS_OLRM.H A "C" header file containing the constant definitions, macro definitions, and function prototypes required to compile OLRM applications for DOS. This header file also provides backward-compatibility for Bus Management for Windows applications written for releases preceding revision 4.0.

OBS_OLRM.H should never be included in a source file directly. **OLRM.H** includes **OBS_OLRM.H.**

OBS_OLRM.INC
A copy of **OBS_OLRM.H** that has been converted so that it is suitable for inclusion into an assembly language source file.

VMEREGS.H A "C" header file containing constant and macro definitions for accessing the EPC VMEbus control registers.

VMEREGS.INC
A copy of **VMEREGS.H** that has been converted so that it is suitable for inclusion into an assembly language source file.

All Bus Management for Windows header files contain an **#if/#endif** pair surrounding the contents of the header file so that the file can be included multiple times without causing compiler errors.

All Bus Management for Windows "C" header files also contain **extern "C"{}** bracketing for C++ compilers. Because **extern "C"** is strictly a C++ keyword, it is also bracketed and only visible when compiling under C++ and not standard "C."

1.3.2 Programming Interface

Bus Management for Windows functions are accessible through interfaces for "C" and Visual BASIC languages. The following table shows the interface libraries and header files for each of the language interfaces.

Language	*Library files*	*Header files*
MS "C"	**BMVXIW16.LIB** **OLRMW16.LIB**	**BUSMGR.H** **OLRM.H**
Borland "C"	**BMVXIW16.LIB** **OLRMW16.LIB**	**BUSMGR.H** **OLRM.H**
Visual Basic	**BMVXIW16.LIB** **OLRMW16.LIB**	**BMVXI.BAS** **OLRM.BAS**

The use of these files is discussed in the following sections.

Calling Bus Management for Windows from MS "C" and QuickC

The "C" language interface is designed to work with Version 5.1 and later versions of the Microsoft "C" compiler and libraries. The libraries are created for the large memory model (far code and far data). This is sufficient for linking programs of any model size, due to the prototyping of all library functions in the header files. The include files provide strong type checking and convert near code and data to far code and data for programs using the small (near code and near data), compact (near code and far data), or medium (far code and near data) memory models.

Calling Bus Management for Windows from Borland C

Bus Management for Windows was designed to work with "C" compilers adhering to the Microsoft "C" calling conventions. Both Microsoft and Borland "C" compilers work equally well.

Calling Bus Management for Windows from Visual Basic

Calling Bus Management for Windows functions from Visual Basic requires using the **BMVXI.BAS** and **OLRM.BAS** header files in your project.

To compile and link a program once your project is built, choose "Make .EXE File" from the File Menu.

For more information about calling "C" DLLs from Visual Basic, refer to the *Microsoft Visual Basic Programming System for Windows Programmer's Guide*.

1.3.3 Compiling and Linking Applications

NOTE: For specific compiler and/or linker options, refer to your vendor's documentation.

The following examples assume that EPConnect software has been installed in the **C:\EPCONNEC** directory.

When compiling applications, ensure that the Bus Management for Windows header files are in the compiler search path by doing one of the following:

1. Specify the entire header file pathname when including the header file in the source file.

2. Specify **C:\EPCONNEC\INCLUDE** as part of the header file search path parameter on the compiler invocation line.

3. Specify **C:\EPCONNEC\INCLUDE** as part of the header file search path environment variable.

Also, ensure that Bus Management for Windows libraries are in the linker search path by doing one of the following:

1. Specify the entire library pathname when linking object files.

2. Specify **C:\EPCONNEC\LIB** as part of the linker library search path.

1.4 What to do Next

To begin using Bus Management for Windows:

1. If Bus Management for Windows is not pre-installed on your system, install and configure it using the procedures in Chapter 2 of the *EPConnect/VXI for DOS & Windows User's Guide*.

2. Refer to the function descriptions in Chapter 2 of this manual for details about a function and/or its parameters to develop applications.

3. Refer to the sample code included with the Bus Management for Windows software under the **C:\EPCONNEC\SAMPLES\BUSMGR.W31** directory.

NOTES

2. Function Descriptions

2

This chapter lists the Bus Management for Windows library functions by category and by name. It is for the programmer who needs a particular fact, such as what function performs a specific task or what a function's arguments are.

The first section lists the functions categorically by the task each performs. It also gives you a brief description of what each function does. The second section lists the functions alphabetically and describes each function in detail.

2.1 Functions by Category

The categorical listing provides an overview of the operations performed by the EPConnect functions. Included with each category is a description of the operations performed, a listing of the functions in the category, and a brief description of each function.

The categories of the Bus Management for Windows library functions include:

- Environment
- Bus Sessions
- Locking
- Memory Mapping
- Byte Order
- Events
- EPC Configuration
- Bus Control Lines
- Watchdog Timer
- Commander Support
- Servant Support

2.1.1 Environment

Bus Management for Windows provides support that allows an application to query and verify the state of its environment.

The Bus Management for Windows library supplies two functions for environment support:

Function	Description
EpcGetErrorString	Queries a null-terminated string corresponding to an EPConnect return value.
EpcVerifyEnvironment	Verifies and queries the EPConnect environment.

2.1.2 Sessions

Bus Management for Windows provides support for multiple simultaneous *sessions*. A session encapsulates shared operating system and interface hardware state in an environment where multiple applications may be accessing the interface hardware simultaneously.

Each session contains a set of attributes that define how resources are managed. Session attributes include:

- A locking timeout. A locking timeout defines how long the session will wait for shared EPC hardware to become unlocked. See the **Locking** section for more information.

- A list of memory mappings. A memory mapping defines where in the EPC's address space an access takes place and how data is accessed. See the **Memory Mapping** section for more information.

- An enabled event mask attribute. The enabled event mask attribute defines the set of events that the session can receive when each of the events' corresponding interrupt or error occurs. See the **Events** section for more information.

- An event handler attribute array. The event handler attribute array defines the functions that are called when the session receives events. The session maintains one entry in the event handler attribute array for each possible event. See the **Events** section for more information.

The Bus Management for Windows library supplies the following functions in support of sessions:

Function	Description
EpcCloseSession	Destroys an open session.
EpcGetSessionData	Queries a session's application-specified data.
EpcOpenSession	Creates a session
EpcSetSessionData	Defines a session's application-specified data.

To use session functionality, an application must first call **EpcOpenSession** to create a session. Once a session exists, an application can access and manage the bus using any of the remaining Bus Management for Windows library functions. The application can define and query application-specific data using **EpcSetSessionData** and **EpcGetSessionData**. When the application is finished with a session, it should call **EpcCloseSession** to destroy the session. Failing to destroy an existing session before an application terminates may result in the loss of both virtual and physical resources.

2.1.3 Locking

Bus Management for Windows provides support for *locking*. Locking gives a session exclusive access to shared interface hardware. Locking is used in multithreaded environments to prevent simultaneous, potentially conflicting hardware manipulation.

Locks can be nested. Bus Management for Windows maintains a global lock counter. At most one session may "own" the lock counter. Initially, the lock counter is zero, indicating that no session has locked the shared interface hardware. Locking acquires and increments the lock counter for a session. Unlocking decrements the lock counter for the same session. A non-zero lock counter indicates that shared interface hardware is locked.

When an application calls a Bus Management for Windows library function that obeys the locking paradigm, the function checks for an existing lock. If no lock exists or the specified session "owns" the lock, the function proceeds. Otherwise, the function suspends execution until the lock is released or the specified session's locking timeout expires. If the existing lock is not released before the specified session's locking timeout expires, the function returns **EPC_LOCKED**.

The Bus Management for Windows library supplies the following functions in support of locking:

Function	Description
EpcGetLockingTimeout	Queries a session's locking timeout.
EpcLockSession	Locks shared interface hardware for a session.
EpcSetLockingTimeout	Defines a session's locking timeout.
EpcUnlockSession	Unlocks shared interface hardware for a session.

To use locking functionality, an application must first call **EpcOpenSession** to create a session. Once a session exists, an application can lock and unlock shared interface hardware for the session using **EpcLockSession** and **EpcUnlockSession**, respectively. When the application completes executing locked operations, it should unlock the session. Failing to unlock a session before an application terminates, either explicitly using **EpcUnlockSession** or implicitly using **EpcCloseSession**, may prevent other applications from accessing shared interface hardware.

An application can also query and define the session's locking timeout using **EpcGetLockingTimeout** and **EpcSetLockingTimeout.**

Only Bus Management for Windows library functions that may make conflicting hardware accesses obey the locking paradigm:

EpcAssertInterrupt	**EpcSetEpcLines**
EpcCmdReceiveWSBuffer	**EpcSetEpcMODID**
EpcCmdSendWSBuffer	**EpcSetEpcTriggers**
EpcCmdSendWSCommand	**EpcSetMiscAttributes**
EpcDeassertInterrupt	**EpcSetSlaveMapping**
EpcLockSession	**EpcSetULA**
EpcMapBusMemory	**EpcSrvEnableWsCommand**
EpcMapEpcTriggers	**EpcSrvReceiveWSCommand**
EpcMapSharedMemory	**EpcSrvSendProtocolEvent**
EpcPulseEpcLines	**EpcSrvSendWSProtocolError**
EpcPulseEpcTriggers	**EpcSrvSendWSResponse**
EpcSetBusAttributes	**EpcValidateBusMapping**

Note that the ability to directly map bus memory allows an application to circumvent the locking protections provided in Bus Management for Windows for VXIbus word serial, byte transfer, and event protocols. Each application is responsible for ensuring that it obeys all bus protocols when accessing bus memory directly.

Locking is not a substitute for a sound shared memory protocol. Locking does not protect against multiple processors making simultaneous accesses to the same memory.

2.1.4 Memory Mapping

EPConnect provides support for *memory mappings.* A memory mapping defines where in the interface's physical address space a mapped access takes place and how data is accessed. Each session contains a list of memory mappings.

A memory mapping can map either bus memory or shared memory. Bus memory is VMEbus memory accessed using the interface's VMEbus hardware. Shared memory is an area of local memory that has a fixed size and a fixed physical location and is accessible via the VMEbus, thereby making it suitable for implementing shared memory communication protocols in a multiple processor system.

Memory Mapping Attributes

Each memory mapping contains a set of attributes that define where and how memory is accessed. Memory mapping attributes include:

- An address modifier attribute. The address modifier attribute defines whether the memory mapping maps to bus memory or shared memory. If the memory mapping maps to bus memory, the address modifier attribute also defines the mapping's VMEbus address space and VMEbus access mode.

- A byte ordering attribute. The byte ordering attribute defines whether Motorola or Intel byte ordering is assumed when the memory mapping is used to access data in widths greater than 8 bits. For a bus memory mapping, the byte ordering attribute specifies either Motorola or Intel byte ordering. For a shared memory mapping, the byte ordering attribute always specifies Intel byte ordering.

- A base address attribute. The base address attribute defines where the memory mapping begins. For a bus memory mapping, the base address attribute is an address in one of the VMEbus address spaces. For a shared memory mapping, the base address attribute is an address in the local address space.

- A size attribute. The size attribute defines the extent of a memory mapping, in bytes.

- A type attribute. The type attribute defines whether a mapping is a shared memory mapping, a bus memory mapping that uses statically configured bus window hardware, or a bus memory mapping that uses dynamically configured bus window hardware.

  Statically configured bus window hardware corresponds to a fixed address modifier, byte ordering, and bus address range. A bus memory mapping that uses statically configured bus window hardware can access mapped bus memory at will.

  Dynamically configured bus window hardware corresponds to a variable address modifier, byte ordering, and bus address ramge. A bus memory mapping that uses dynamically configured bus window hardware must configure its bus window hardware before accessing mapped bus memory.

The EPConnect Bus Management Library supplies the following functions in support of memory mappings:

Function	Description
EpcCopyData	Copy a block of data from consecutive memory locations to consecutive memory locations.
EpcGetMappingAttributes	Query a memory mapping's attributes.
EpcMapBusMemory	Create a bus memory mapping using a statically configured bus window.
EpcMapBusMemoryExt	Create a bus memory mapping using a dynamically configured bus window.
EpcMapSharedMemory	Create a shared memory mapping.
EpcPopData	Pop a block of data from a single memory location to consecutive memory locations.
EpcPushData	Push a block of data from consecutive memory locations to a single memory location.
EpcUnmapBusMemory	Destroy a bus memory mapping.
EpcUnmapSharedMemory	Destroy a shared memory mapping.
EpcValidateBusMapping	Validate a bus memory mapping that uses a dynamically configured bus window.

2.1.4 Memory Mapping

To use memory mapping functionality, an application must first call **EpcOpenSession** to open a bus session and either **EpcMapBusMemory, EpcMapBusMemoryExt,** or **EpcMapSharedMemory** to create a memory mapping. Once a memory mapping exists, an application can access the mapped memory either directly or by using **EpcCopyData**, **EpcPopData**, or **EpcPushData**. When the application is finished with a memory mapping, it should call either **EpcUnmapBusMemory** or **EpcUnmapSharedMemory** to destroy the mapping. Failing to destroy an existing memory mapping before an application terminates, either explicitly using **EpcUnmapBusMemory** or **EpcUnmapSharedMemory** or implicitly using **EpcCloseSession**, may result in the loss of both virtual and physical resources.

Direct access of mapped memory provides the maximum possible data transfer performance, but it does not automatically detect and handle misaligned data, potential bus errors, or hardware restrictions. Direct access requires that the application guarantee data alignment and bus error avoidance. Direct access using a bus memory mapping created with **EpcMapBusMemoryExt** requires using **EpcValidateBusMapping** before an access to insure that the dynamically configured bus window references the desired bus memory. Finally, direct access using a bus memory mapping created with **EpcMapBusMemoryExt** in a preemptively scheduled environment requires using **EpcLockSession** before **EpcValidateBusMapping** and **EpcUnlockSession** after the direct access to ensure that the dynamically configured bus window is not reconfigured by another thread during the direct access. Note that **EpcCopyData**, **EpcPopData**, and **EpcPushData** copy blocks of data while taking hardware restrictions, data alignment, and potential bus error considerations into account.

An application can use **EpcGetMappingAttributes** to query an existing memory mapping's attributes.

2.1.5 Byte Order

The Bus Management for Windows library provides support for converting data between Intel and Motorola byte order through byte-swapping.

The Bus Management for Windows library supplies the following functions in support of byte order conversion:

Function	Description
EpcSwapBuffer	Byte-swaps a buffer of values.
EpcSwap16	Byte-swaps a 16-bit value.
EpcSwap32	Byte-swaps a 32-bit value.
EpcSwap48	Byte-swaps a 48-bit value.
EpcSwap64	Byte-swaps a 64-bit value.
EpcSwap80	Byte-swaps an 80-bit value.

2.1.6 Events

EPConnect provides support for *events*. An event is an interrupt or error that occurs asynchronously with respect to normal program execution.

Each session contains a set of event attributes that define how the session handles events. Event attributes include:

- An enabled event mask attribute. The enabled event mask attribute defines the set of events that the session can receive when each of the events' corresponding interrupt or error occurs.

- An array of event handlers. The event handler array defines the functions that are called when the session receives events. The session maintains one entry in the event handler array for each possible event.

2.1.6 Events

The EPConnect Bus Management Library supplies the following functions in support of events:

Function	Description
EpcGetEventEnableMask	Queries a session's enabled event mask attribute.
EpcGetEventHandler	Queries an entry in a session's event handler array.
EpcSetEventEnableMask	Defines a session's enabled event mask attribute.
EpcSetEventHandler	Defines an entry in a session's event handler array.
EpcWaitForEvent	Waits for an event to occur.

To use event functionality, an application must first call **EpcOpenSession** to create a session. Once a session exists, an application can either wait for the desired events to occur using **EpcWaitForEvent** or it can define handlers for the desired events using **EpcSetEventHandler**. In either case, the application must enable reception of the events using **EpcSetEventEnableMask** to receive them.

When the application is finished receiving events, it should disable reception of the events. Failing to disable reception of events before an application terminates, either explicitly using **EpcSetEventEnableMask** or implicitly using **EpcCloseSession**, may result in a system crash the next time an event occurs.

An application can use **EpcGetEventEnableMask** and **EpcGetEventHandler** to query a session's current event attributes.

The Bus Management for Windows library supports all possible VXIbus events. In practice, however, event support is limited by the underlying interface hardware.

The table below describes the events:

Event	Description
EPC_MSG_INT	Message interrupt (EPC-7 and EPC-8 only)
EPC_VME1_INT	VMEbus interrupt 1
...	
EPC_VME7_INT	VMEbus interrupt 7
EPC_SIGNAL_INT	VXIbus signal FIFO interrupt
EPC_TTL_TRIG0_INT	VXIbus TTL Trigger 0 interrupt (EPC-7 only)
...	
EPC_TTL_TRIG7_INT	VXIbus TTL Trigger 7 interrupt (EPC-7 only)
EPC_SYSRESET_ERR	VMEbus SYSRESET error
EPC_ACFAIL_ERR	VMEbus power failure error
EPC_BERR_ERR	VMEbus access error
EPC_SYSFAIL_ERR	VMEbus SYSFAIL error
EPC_WATCHDOG_ERR	Watchdog timer expiration error (EPC-7 and EPC-8 only)
EPC_EXT_TRIG0_INT	External Trigger 0 interrupt (VXLink only)
EPC_EXT_TRIG1_INT	External Trigger 1 interrupt (VXLink only)

2.1.7 EPC Configuration

Bus Management for Windows provides support for maintaining global interface configuration attributes. The values of global interface configuration attributes affect all the behavior of the interface hardware for all sessions.

The Bus Management for Windows library supplies the following functions in support of interface configuration:

Function	Description
EpcGetBusAttributes	Queries the interface's bus management attributes.
EpcGetMiscAttributes	Queries the interface's miscellaneous configuration attributes.
EpcGetSlaveMapping	Queries the interface's slave memory mapping.
EpcGetULA	Queries the interface's unique logical address.
EpcSetBusAttributes	Defines the interface's bus management attributes.
EpcSetMiscAttributes	Defines the interface's miscellaneous configuration attributes.
EpcSetSlaveMapping	Defines the interface's slave memory mapping.
EpcSetULA	Defines the interface's unique logical address.

To use interface configuration functionality, an application must first call **EpcOpenSession** to create a session. Once a session exists, an application can define the global interface configuration attributes using **EpcSetBusAttributes**, **EpcSetSlaveMapping**, and **EpcSetULA**. An application can query the global interface configuration attributes using **EpcGetBusAttributes**, **EpcGetSlaveMapping**, and **EpcGetULA**.

2.1.8 Bus Lines

Bus Management for Windows provides support for defining, querying, and pulsing the interface line state. It also provides support for monitoring actual bus line state.

In general, interface line state reflects the state of bits in the interface's line drive registers, while actual bus line state is an OR'd combination of the states of all devices on the bus. If the interface asserts a line, the actual bus line transitions from deasserted to asserted only if all other devices on the bus have previously deasserted the line. Likewise, if the interface deasserts a line, the actual bus line transitions from asserted to deasserted only if all devices on the bus have previously deasserted the line.

The Bus Management for Windows library supplies the following functions in support of bus lines:

Function	Description
EpcAssertInterrupt	Asserts a VME interrupt.
EpcDeassertInterrupt	Deasserts a VME interrupt.
EpcGetBusInterrupts	Queries actual bus VME interrupt line state.
EpcGetBusLines	Queries actual bus control line state.
EpcGetBusMODID	Queries actual bus MODID line state.
EpcGetBusTriggers	Queries actual bus trigger line state.
EpcGetEpcInterrupt	Queries interface VME interrupt assertion state.
EpcGetEpcLines	Queries interface control line state.
EpcGetEpcMODID	Queries interface MODID line state.
EpcGetEpcTriggers	Queries interface trigger line state.
EpcGetEpcTriggerMapping	Queries an interface trigger line mapping.
EpcMapEpcTriggers	Maps one interface trigger line to another.
EpcPulseEpcLines	Pulses interface control lines.
EpcPulseEpcTriggers	Pulses interface trigger lines.
EpcSetEpcLines	Defines the interface control line state.

EpcSetEpcMODID	Defines the interface MODID line state.
EpcSetEpcTriggers	Defines the interface trigger line state.

To use bus control line functionality, an application must first call **EpcOpenSession** to create a session. Once a session exists, an application can define interface line state using **EpcAssertInterrupt**, **EpcDeassertInterrupt**, **EpcSetEpcLines**, **EpcSetEpcMODID**, or **EpcSetEpcTriggers**. An application can pulse interface lines using **EpcPulseEpcLines** or **EpcPulseEpcTriggers**. To query the interface line state, an application can use **EpcGetEpcInterrupt**, **EpcGetEpcLines**, **EpcGetEpcMODID**, **EpcGetEpcTriggers**, or **EpcGetEpcTriggerMapping**. To query actual bus line state, the application can use **EpcGetBusInterrupts**, **EpcGetBusLines**, **EpcGetBusMODID**, or **EpcGetBusTriggers**.

2.1.9 Watchdog Timer

Bus Management for Windows provides watchdog timer services that allow an application to prevent interface lock-up under extraordinary circumstances.

If an EPC's watchdog timer is not reset within the current watchdog timer period, either a system reset occurs or a watchdog timer error event occurs. In the latter case, an application can enable the event and install an event handler to gracefully handle the error.

The EPC's watchdog timer is typically reset in sections of code that execute frequently and/or execute at regular time intervals.

The Bus Management for Windows library supplies a single function in support of the watchdog timer:

Function	Description
EpcWatchdogTimer	Modifies EPC watchdog timer configuration.

To use watchdog timer functionality, an application must first call **EpcOpenSession** to create a session. Once a session exists, an application can configure the EPC's watchdog timer using **EpcWatchdogTimer**.

An EPC's watchdog timer is a shared EPC hardware resource. However, Bus Management for Windows provides no functionality for controlling shared access to the watchdog timer. Multiple applications may simultaneously use watchdog timer functionality. However, EPConnect software cannot guarantee the result.

The purpose of the watchdog timer hardware is to allow an application to prevent EPC lock-up under extraordinary circumstances. Placing additional layers of software between an application and the watchdog timer hardware to control sharing of the resource would necessarily restrict an application's access to the watchdog timer, thereby violating its original purpose.

By default, an EPC is configured to use a long watchdog timer period and to generate a watchdog error event upon expiration. Assuming that no application attempts to modify these default watchdog timer settings, any number of applications may use the watchdog timer simultaneously.

VXLink does not contain a watchdog timer. This function is valid only on an EPC-7 and EPC-8.

2.1.10 Commander Support

EPConnect provides support for using an EPC or VXlink card as a commander device in a VXIbus system.

The Bus Management for Windows library supplies the following functions in support of using an EPC as a commander device:

Function	Description
EpcCmdReceiveWSBuffer	Receives a buffer of data from a servant device.
EpcCmdSendWSBuffer	Sends a buffer of data to a servant device.
EpcCmdSendWSCommand	Sends a word serial command to a servant device.

To use commander functionality, an application must first call **EpcOpenSession** to create a session. Once a session exists, an application can send 16-bit, 32-bit, or 48-bit word serial commands and receive responses using **EpcCmdSendWSCommand**. To quickly send multiple data bytes to a servant device, an application should use **EpcCmdSendWSBuffer**. To quickly receive multiple data bytes from a servant device, an application should use **EpcCmdReceiveWSBuffer**.

2.1.11 Servant Support

Bus Management for Windows provides support for using an EPC as a servant device in a VMEbus system.

The Bus Management for Windows library supplies the following functions in support of using an EPC as a servant device:

Function	Description
EpcSrvEnableWSCommand	Enables word serial command reception.
EpcSrvReceiveWSCommand	Receives a word serial command from the commander device.
EpcSrvSendProtocolEvent	Sends a protocol event to the commander device.
EpcSrvSendWSProtocolError	Sends a word serial protocol error to the commander device.
EpcSrvSendWSResponse	Sends a word serial command response to the commander device.

To use servant functionality, an application must first call **EpcOpenSession** to create a session. Once a session exists, an application can receive 16-bit or 32-bit word serial commands using **EpcSrvEnableWSCommand** and **EpcSrvReceiveWSCommand** and send responses to received word serial commands using **EpcSrvSendWSResponse**. An application can use **EpcSrvSendProtocolEvent** to send events and/or responses to a commander device via the commander device's signal register.

Servant functionality is supported on an EPC-7 and EPC-8 only. A VXLink interface does not support servant functionality.

2.2 Functions By Name

This section contains an alphabetical listing of the Bus Management for Windows library functions. Each listing describes the function, gives its invocation sequence and arguments, discusses its operation, and lists its returned values. Where usage of the function may not be clear, an example with comments is given.

EpcAssertInterrupt

Description Asserts a VME interrupt.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcAssertInterrupt(unsigned long Session_ID, unsigned long
Event_Mask);
```

Session_ID	*Session_ID* specifies a session.
Event_Mask	*Event_Mask* specifies a VME interrupt.

Visual Basic Synopsis

```
Declare Function
EpcAssertInterrupt% Lib "bmvxiw16.dll" (ByVal Session_ID&,
ByVal Event_Mask&)
```

Remarks **EpcAssertInterrupt** causes the interface to assert a VME interrupt.

Event_Mask specifies the VME interrupt to assert. Valid values are:

Event_Mask	Description
EPC_VME1_INT	VMEbus interrupt 1.
...	
EPC_VME7_INT	VMEbus interrupt 7.

An interface acts as both a D08(O) and a D16 interrupter. For D08 interrupt acknowledge cycles, the interface uses its unique logical address as the status/ID value. For D16 interrupt acknowledge cycles, the interface uses the upper 8 bits of its response register for the upper 8 bits of the status/ID value and its unique logical address as the lower 8 bits of the status/ID value.

Return Value The function returns a Bus Management return value:

EPC_INV_ASSERT	The interface is already asserting a VMEbus interrupt.
EPC_INV_MASK	The parameter *Event_Mask* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present or there is a revision mismatch between the Bus Management Library and the BusManager VxD.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcDeassertInterrupt, EpcGetEpcInterrupt, EpcSrvSendProtocolEvent.**

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * buslines.c -- Bus Management Library interface bus line functions sample code.
 */

#include "busmgr.h"

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
BusLinesSample(void);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * CODE...
 */

short FAR
BusLinesSample(void)
{
    char                err_string[ERROR_STRING_SZ];
    short               err_num;
    unsigned long       bus_int_mask;
    unsigned long       bus_line_mask;
    unsigned long       epc_int_mask;
```

```
    unsigned long         epc_line_mask;
    unsigned long         session_id;
    struct EpcEnvironment environment;

    /*
     * Verify the interfaceonnect environment.
     */

    if ((err_num = EpcVerifyEnvironment(&environment)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcVerifyEnvironment() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Open a session.
     */

    if ((err_num = EpcOpenSession(&session_id)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcOpenSession() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Assert VMEbus interrupt #1, query bus and interface VMEbus interrupt
       * assertions, deassert VMEbus interrupt #1, and query bus and interface
VMEbus
     * interrupt assertions again.
     *
     */

    EpcAssertInterrupt(session_id, EPC_VME1_INT);
    EpcGetBusInterrupts(session_id, &bus_int_mask);
    EpcGetEpcInterrupt(session_id, &epc_int_mask);
    WinPrintf("VMEbus interrupt 1 asserted.\n");
        WinPrintf("Bus  interrupt  mask  =  0x%08lX,  interface  interrupt  mask  =
0x%08lX\n",
                bus_int_mask,
                epc_int_mask);
    EpcDeassertInterrupt(session_id);
    EpcGetBusInterrupts(session_id, &bus_int_mask);
    EpcGetEpcInterrupt(session_id, &epc_int_mask);
    WinPrintf("VMEbus interrupt 1 deasserted.\n");
        WinPrintf("Bus  interrupt  mask  =  0x%08lX,  interface  interrupt  mask  =
0x%08lX\n",
                bus_int_mask,
                epc_int_mask);

    /*
     * Assert SYSFAIL, query bus and interface line assertions, pulse SYSFAIL, and
     * query bus and interface line assertions again.
     */

    EpcSetEpcLines(session_id, EPC_SYSFAIL);
    EpcGetBusLines(session_id, &bus_line_mask);
    EpcGetEpcLines(session_id, &epc_line_mask);
    WinPrintf("SYSFAIL asserted.\n");
    WinPrintf("Bus line mask = 0x%08lX, interface line mask = 0x%08lX\n",
```

```
            bus_line_mask,
            epc_line_mask);
    EpcPulseEpcLines(session_id, EPC_SYSFAIL);
    WinPrintf("SYSFAIL pulsed.\n");
    WinPrintf("Bus line mask = 0x%08lX, interface line mask = 0x%08lX\n",
            bus_line_mask,
            epc_line_mask);

    /*
     * Close the session and return.
     */

    EpcCloseSession(session_id);
    WinPrintf("SUCCESS:  BusLinesSample() complete.\n");
    return (EPC_SUCCESS);
}
```

EpcCloseSession

Description Destroys an open session.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcCloseSession(unsigned long *Session_ID*);

Session_ID *Session_ID* specifies an open session.

Visual Basic Synopsis

Declare Function
EpcCloseSession% Lib "bmvxiw16.dll" (ByVal *Session_ID*&)

Remarks **EpcCloseSession** closes an open session.

If the specified session has locked shared interface hardware, the hardware is unlocked before the function destroys the session.

If the specified session has one or more enabled events, the events are disabled before the function destroys the session. Also, all of the session's defined event handlers are removed.

If the specified session contains one or more memory mappings, the mappings are destroyed before the function destroys the session.

Return Value The function returns a Bus Management return value:

EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present or there is a revision mismatch between the Bus Management Library and the BusManager VxD.
EPC_INV_USAGE	The session specified by *Session_ID* is currently in use by another thread.
EPC_SUCCESS	The function completed successfully.

See Also **EpcOpenSession, EpcSetEventEnableMask, EpcUnlockSession, EpcUnmapBusMemory, EpcUnmapSharedMemory.**

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * sessions.c -- Bus Management Library session functions sample code.
 */

#include "busmgr.h"

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
SessionsSample(void);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * CODE...
 */

short FAR
SessionsSample(void)
{
    char                err_string[ERROR_STRING_SZ];
    short               err_num;
    unsigned long       session_data;
    unsigned long       session_id;
    struct EpcEnvironment environment;

    /*
     * Verify the EPConnect environment.
     */

    if ((err_num = EpcVerifyEnvironment(&environment)) != EPC_SUCCESS)
```

```
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcVerifyEnvironment() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /**
     **Open a session.
     */

    if ((err_num = EpcOpenSession(&session_id)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcOpenSession() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     **Define the session's application-specific data.
     */

    EpcSetSessionData(session_id, session_id);

    /**
     **Query the session's application-specific data.
     */

    EpcGetSessionData(session_id, &session_data);

    /*
     **Close the session and return.
     */

    EpcCloseSession(session_id);
    WinPrintf("SUCCESS:  SessionsSample() complete.\n");
    return (EPC_SUCCESS);
}
```

EpcCmdReceiveWSBuffer

Description Receives a buffer of data from a servant device.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcCmdReceiveWSBuffer
        (unsigned long          Session_ID,
         unsigned short         ULA,
         unsigned char FAR *    Buffer_Ptr,
         unsigned long          Buffer_Size,
         short                  Term_Character,
         unsigned long          Timeout,
         unsigned long FAR *    Term_Reason_Ptr,
         unsigned long FAR *    Receive_Size_Ptr);
```

Session_ID	*Session_ID* specifies a session.
ULA	*ULA* specifies a servant device's unique logical address.
Buffer_Ptr	*Buffer_Ptr* specifies the location of a buffer where the received data will be placed.
Buffer_Size	*Buffer_Size* specifies the size of the buffer where the received data will be placed.
Term_Character	*Term_Character* specifies a termination character for the receive operation.
Timeout	*Timeout* specifies the number of milliseconds to wait while receiving a buffer of data.

Term_Reason_Ptr	*Term_Reason_Ptr* specifies a location where a bit mask defining the reason(s) for terminating the receive operation will be placed.
Receive_Size_Ptr	*Receive_Size_Ptr* specifies a location where the actual number of bytes received will be placed.

Visual Basic Synopsis

Declare Function
EpcCmdReceiveWSBuffer% Lib "bmvxiw16.dll"

(ByVal *Session_ID&*,
ByVal *ULA%*,
ByVal *Buffer_Ptr$*,
ByVal *Buffer_Size&*,
ByVal *Term_Character%*,
ByVal *Timeout&*,
Term_Reason_Ptr&,
Receive_Size_Ptr&)

Remarks

EpcCmdReceiveWSBuffer receives up to *Buffer_Size* bytes of data from the servant device specified by *ULA* and places them in the buffer pointed to by *Buffer_Ptr*.

Term_Character specifies an optional termination character for the receive operation. Valid termination character values are -1 and 0 through 255. A termination character value of -1 specifies that no termination character is defined. A termination character value of 0 through 255 specifies a termination character. If the function detects a termination character while it's receiving data, it places the termination character in the buffer and returns **EPC_SUCCESS**.

If *Term_Reason_Ptr* is non-null and the function returns **EPC_SUCCESS**, the location pointed by *Term_Reason_Ptr* contains a bit mask defining the reason(s) for terminating the receive operation. The bit mask is an OR'd combination of the following constants:

Constant	Description
EPC_TERM_CHAR	The function detected the specified termination character.
EPC_TERM_EOI	The function received a data byte with the EOI indicator set.
EPC_TERM_FULL	The specified buffer is full.

The value of the location pointed to by *Term_Reason_Ptr* is undefined when the function does not return **EPC_SUCCESS**.

If *Receive_Size_Ptr* is non-null, the location it points to always contains the number of data bytes actually received.

If the function detects a word serial protocol error while receiving data, it returns **EPC_WS_PROTOCOL**. To determine the protocol error, use **EpcCmdSendWSCommand** to send a READ PROTOCOL ERROR word serial command to the servant device and receive its response.

EpcCmdReceiveWSBuffer is intended for use by a commander device to quickly receive multiple data bytes from one of its servants via word serial commands. A servant device should use **EpcSrvReceiveWSCommand** to receive a word serial command from its commander and **EpcSrvSendWSResponse** to send a word serial command response to its commander.

Return Value

The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Buffer_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present or there is a revision mismatch between the Bus Management Library and the BusManager VxD.
EPC_INV_TERMCHR	The parameter *Term_Character* is invalid.

EPC_INV_ULA	The parameter *ULA* is invalid.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_RECV_BERR	A bus error occurred receiving a word serial command response.
EPC_RECV_TIMEOUT	A timeout occurred receiving a word serial command response.
EPC_SEND_BERR	A bus error occurred sending a word serial command.
EPC_SEND_TIMEOUT	A timeout occurred sending a word serial command.
EPC_SUCCESS	The function completed successfully.
EPC_WS_PROTOCOL	A word serial protocol error occurred.

See Also **EpcCmdSendWSBuffer, EpcCmdSendWSCommand, EpcOpenSession, EpcSrvReceiveWSCommand, EpcSrvSendWSResponse.**

EpcCmdSendWSBuffer

Description Sends a buffer of data to a servant device.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcCmdSendWSBuffer( unsigned long          Session_ID,
                    unsigned short         ULA,
                    unsigned char  FAR *   Buffer_Ptr,
                    unsigned long          Buffer_Size,
                    unsigned short         EOI_Flag,
                    unsigned long          Timeout,
                    unsigned long  FAR *   Send_Size_Ptr);
```

Session_ID	*Session_ID* specifies a session.
ULA	*ULA* specifies a servant device's unique logical address.
Buffer_Ptr	*Buffer_Ptr* specifies the location of a buffer containing the data to be sent.
Buffer_Size	*Buffer_Size* specifies the number of bytes to be sent.
EOI_Flag	*EOI_Flag* specifies whether the EOI indicator should be set when the function sends the last byte from the specified buffer.
Timeout	*Timeout* specifies the number of milliseconds to wait while sending the buffer of data.
Send_Size_Ptr	*Send_Size_Ptr* specifies a location where the actual number of bytes sent will be placed.

Visual Basic Synopsis

```
Declare Function
EpcCmdSendWSBuffer% Lib "bmvxiw16.dll" (  ByVal
Session_ID&,
                                        ByVal ULA%,
                                        ByVal Buffer_Ptr$,
                                        ByVal Buffer_Size&,
                                        ByVal EOI_Flag%,
                                        ByVal Timeout&,
                                        Send_Size_Ptr&)
```

Remarks

EpcCmdSendWSBuffer sends up to *Buffer_Size* bytes of data from the buffer pointed to by *Buffer_Ptr* to the servant device specified by *ULA*.

EOI_Flag specifies whether the EOI indicator should be set when the function sends the last byte from the specified buffer. A non-zero *EOI_Flag* value causes the function to set the EOI indicator when it sends the last byte from the buffer. A zero *EOI_Flag* value causes the function to not set the EOI indicator when it sends the last byte from the specified buffer.

If *Send_Size_Ptr* is non-null, the location it points to always contains the number of data bytes actually sent.

If the function detects a word serial protocol error while sending data, it returns **EPC_WS_PROTOCOL**. To determine the protocol error, use **EpcCmdSendWSCommand** to send a READ PROTOCOL ERROR word serial command to the servant device and receive its response.

EpcCmdSendWSBuffer is intended for use by a commander device to quickly send multiple data bytes to one of its servants via word serial commands. A servant device should use **EpcSrvReceiveWSCommand** to receive a word serial command from its commander and **EpcSrvSendWSResponse** to send a word serial command response to its commander.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Buffer_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present or there is a revision mismatch between the Bus Management Library and the BusManager VxD.
EPC_INV_ULA	The parameter *ULA* is invalid.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SEND_BERR	A bus error occurred sending a word serial command.
EPC_SEND_TIMEOUT	A timeout occurred sending a word serial command.
EPC_SUCCESS	The function completed successfully.
EPC_WS_PROTOCOL	A word serial protocol error occurred.

See Also **EpcCmdReceiveWSBuffer, EpcCmdSendWSCommand, EpcOpenSession, EpcSrvReceiveWSCommand, EpcSrvSendWSResponse.**

EpcCmdSendWSCommand

Description Sends a word serial command to a servant device.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcCmdSendWSCommand( unsigned long  Session_ID,
                     unsigned short ULA,
                     void   FAR *   Command_Ptr,
                     void   FAR *   Response_Ptr,
                     unsigned short Size,
                     unsigned long  Timeout);
```

Session_ID	*Session_ID* specifies a session.
ULA	*ULA* specifies a servant device's unique logical address.
Command_Ptr	*Command_Ptr* specifies the location of a word serial command.
Response_Ptr	*Response_Ptr* specifies a location where the response to the word serial command will be placed.
Size	*Size* specifies the size of both the word serial command and the optional word serial command response.
Timeout	*Timeout* specifies the number of milliseconds to wait while sending the word serial command and receiving the word serial command response.

Visual Basic Synopsis

```
Declare Function
EpcCmdSendWSCommand% Lib "bmvxiw16.dll" (ByVal
Session_ID&,
                                         ByVal ULA%,
                                         Command_Ptr As Any,
                                         Response_Ptr As Any,
                                         ByVal Size%,
                                         ByVal Timeout&)
```

Remarks

EpcCmdSendWSCommand optionally sends a word serial command, then optionally receives a word serial command response. If *Command_Ptr* is not null, the function sends the word serial command at the location pointed to by *Command_Ptr* to the servant device specified by *ULA*. Otherwise, the function skips sending a command. If *Response_Ptr* is not null, the function then receives a word serial command response from the servant device specified by *ULA* and places it in the location pointed to by *Response_Ptr*. Otherwise, the function returns without attempting to receive a response.

Size specifies the size of both the word serial command and the word serial command response:

Size	Description
EPC_16_BIT	Send a 16-bit word serial command and receive a 16-bit word serial command response.
EPC_32_BIT	Send a 32-bit long word serial command and receive a 32-bit long word serial command response.
EPC_48_BIT	Send a 48-bit extended long word serial command and receive a 32-bit long word serial command response.

If the function detects a word serial protocol error while sending a command or receiving a response, it returns EPC_WS_PROTOCOL. To determine the protocol error, use **EpcCmdSendWSCommand** to send a READ PROTOCOL ERROR word serial command to the servant device and receive its response.

EpcCmdSendWSCommand is intended for use by a commander device to send a word serial command to one of its servants and/or to receive a word serial command response from one of its servants. A servant device should use **EpcSrvReceiveWSCommand** to receive a word serial command from its commander and **EpcSrvSendWSResponse** to send a word serial command response to its commander.

Return Value

The function returns a Bus Management return value:

EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SIZE	The parameter *Size* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_INV_ULA	The parameter *ULA* is invalid.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_RECV_BERR	A bus error occurred receiving the word serial command response.
EPC_RECV_TIMEOUT	A timeout occurred receiving the word serial command response.
EPC_SEND_BERR	A bus error occurred sending the word serial command.
EPC_SEND_TIMEOUT	A timeout occurred sending the word serial command.
EPC_SUCCESS	The function completed successfully.
EPC_WS_PROTOCOL	A word serial protocol error occurred.

See Also EpcOpenSession, EpcSrvReceiveWSCommand, EpcSrvSendWSResponse.

EpcCopyData

Description Copies a block of data.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcCopyData(  unsigned long          Session_ID,
              void HUGE *            Source_Ptr,
              void HUGE *            Dest_Ptr,
              unsigned long          Size,
              unsigned short         Data_Width,
              unsigned long FAR *    Actual_Size_Ptr);
```

Session_ID	*Session_ID* specifies a bus session.
Source_Ptr	*Source_Ptr* specifies the address of a data buffer from which data will be copied.
Dest_Ptr	*Dest_Ptr* specifies the address of a data buffer into which data will be copied.
Size	*Size* specifies the number of data bytes to copy.
Data_Width	*Data_Width* specifies the number of data bits to copy per bus access.
Actual_Size_Ptr	*Actual_Size_Ptr* specifies a location where the actual number of bytes copied will be placed.

Visual Basic Synopses

Declare Function
BasicCopyEpcToVME% Lib "bmvxiw16.dll" (ByVal *Session_ID&*,
Source_Ptr As Any,
ByVal *Dest_Ptr* As Any,
ByVal *Size&*,
ByVal *Data_Width%*,
Actual_Size_Ptr&)

Declare Function
BasicCopyVMEToEpc% Lib "bmvxiw16.dll" (ByVal *Session_ID&*,
ByVal *Source_Ptr* As Any,
Dest_Ptr As Any,
ByVal *Size&*,
ByVal *Data_Width%*,
Actual_Size_Ptr&)

Declare Function
BasicCopyVMEToVME% Lib "bmvxiw16.dll"

(ByVal *Session_ID&*,
ByVal *Source_Ptr* As Any,
ByVal *Dest_Ptr* As Any,
ByVal *Size&*,
ByVal *Data_Width%*,
Actual_Size_Ptr&)

Remarks

EpcCopyData efficiently copies blocks of data from consecutive memory locations to consecutive memory locations using the attributes of pointers *Source_Ptr* and *Dest_Ptr*. The intended use of the function is copying large blocks of data to or from consecutive bus locations.

The *Size* parameter should always express the number of bytes to be copied, regardless of the specified *Data_Width* parameter. Passing a zero *Size* parameter results in no data being copied.

The following constants define valid values for the *Data_Width* parameter:

Constant	Description
EPC_8_BIT	8-bit data width
EPC_8_BIT_ODD	8-bit data width, odd bytes only
EPC_16_BIT	16-bit data width
EPC_32_BIT	32-bit data width
EPC_FASTCOPY	To increase copy performance, don't check for intermediate bus errors. This constant cannot be used alone; it must be OR'd with one of the preceding constants.

The function returns the actual number of bytes copied in the location pointed to by *Actual_Size_Ptr*.

The function operates correctly using both unmapped pointers and memory mapped pointers for *Source_Ptr* and *Dest_Ptr*. EPC-to-EPC, EPC-to-VME, VME-to-EPC, and VME-to-VME copies all execute properly.

For a 16-bit or 32-bit copy to complete, no individual data element may span a segment boundary. Otherwise, the function returns an **EPC_INV_ALIGN** error. For example, if *Data_Width* is **EPC_16_BIT** and *Size* is greater than 64 Kbytes, both *Source_Ptr* and *Dest_Ptr* must be aligned on a 16-bit boundary for the copy operation to complete successfully

For a VME-to-VME copy to complete, both *Source_Ptr* and *Dest_Ptr* must correspond to VMEbus addresses aligned on an address boundary equivalent to the specified *Data_Width*. Otherwise, the function returns an **EPC_INV_ALIGN** error. For example, if both *Source_Ptr* and *Dest_Ptr* correspond to VMEbus memory and *Data_Width* is **EPC_16_BIT**, then both *Source_Ptr* and *Dest_Ptr* must correspond to VMEbus addresses aligned on a 16-bit boundary for the copy to complete successfully.

For EPC-to-VME, VME-to-EPC, and VME-to-VME copies to complete when hardware byte-swapping occurs, *Size* must be a multiple of the specified *Data_Width* and all VMEbus addresses must be aligned on an address boundary equivalent to the specified *Data_Width*. Otherwise, the function returns an **EPC_INV_SWAP** error. For example, if *Source_Ptr* corresponds to EPC memory, *Dest_Ptr* corresponds to VMEbus memory, and *Data_Width* is **EPC_16_BIT**, *Size* must be a multiple of two and *Dest_Ptr* must correspond to a VMEbus address aligned on a 16-bit boundary for the copy to complete successfully.

To ensure that all accesses are the specified *Data_Width*, the function handles non-aligned leading and trailing bytes as a special case. When transferring data from a non-aligned address, the function reads the nearest aligned chunk and extracts the non-aligned bytes. When transferring data to a non-aligned address, the function reads the nearest aligned chunk, copies the non-aligned bytes into the chunk, and replaces the chunk. Note that, for VMEbus transfers, this read-modify-write algorithm is executed in software -- it is not a read-modify-write bus cycle.

Return Value The function returns a Bus Management return value:

EPC_BERR	A bus error occurred during the copy.
EPC_INV_ALIGN	A 16-bit or 32-bit data element spans a segment boundary or both *Source_Ptr* and *Dest_Ptr* are mapped to VMEbus addresses and they are not aligned on equivalent VMEbus address boundaries.
EPC_INV_PTR	One or more of *Source_Ptr*, *Dest_Ptr*, or *Actual_Size_Ptr* is invalid.
EPC_INV_RANGE	The address range defined by *Source_Ptr* and *Size* and/or the address range defined by *Dest_Ptr* and *Size* contains bus addresses that are not currently mapped.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_INV_SWAP	*Source_Ptr* and/or *Dest_Ptr* are mapped to the VMEbus so that hardware byte-swapping will occur, but *Size* is not a multiple of *Data_Width* and/or a VMEbus address is misaligned.
EPC_INV_WIDTH	The *Data_Width* parameter is invalid.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetMappingAttributes, EpcMapBusMemory, EpcMapSharedMemory, EpcOpenSession, EpcUnmapBusMemory, EpcUnmapSharedMemory.**

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * mapping.c -- Bus Management Library mapping functions sample code.
 */

#include "busmgr.h"

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
MappingSample(void);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * CODE...
 */

short FAR
MappingSample(void)
{
    char                err_string[ERROR_STRING_SZ];
    short               err_num;
    unsigned short      bus_add_mod;
    unsigned short      bus_byte_order;
    unsigned short      ula;
    unsigned long       actual_size;
    unsigned long       bus_base;
    unsigned long       bus_size;
    unsigned long       session_id;
    unsigned long       shared_base;
    unsigned long       shared_size;
    volatile void   HUGE *bus_ptr;
    volatile void   HUGE *shared_ptr;
    struct EpcEnvironment environment;

    /*
     * Verify the EPConnect environment.
     */

    if ((err_num = EpcVerifyEnvironment(&environment)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcVerifyEnvironment() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Open a session.
     */

    if ((err_num = EpcOpenSession(&session_id)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcOpenSession() error == %s (%d).\n",
```

```
                err_string,
                err_num);
    return (err_num);
}

/*
 * Map all of A16 space using Motorola byte ordering.
 */

if ((err_num = EpcMapBusMemory(session_id,
                               EPC_A16S,
                               EPC_MBO,
                               0x00000000,
                               0x00010000,
                               &bus_ptr)) != EPC_SUCCESS)
{
    EpcCloseSession(session_id);
    EpcGetErrorString(err_num, err_string);
    WinPrintf("FAILURE:  EpcMapBusMemory() error == %s (%d).\n",
                err_string,
                err_num);
    return (err_num);
}

/*
 * Query the bus mapping's attributes.
 */

EpcGetMappingAttributes(session_id,
                        bus_ptr,
                        &bus_add_mod,
                        &bus_byte_order,
                        &bus_base,
                        &bus_size);

/*
 * Map the EPC's shared memory buffer.
 */

if ((err_num = EpcMapSharedMemory(session_id,
                                  &shared_base,
                                  &shared_size,
                                  &shared_ptr)) != EPC_SUCCESS)
{
    EpcCloseSession(session_id);
    EpcGetErrorString(err_num, err_string);
    WinPrintf("FAILURE:  EpcMapSharedMemory() error == %s (%d).\n",
                err_string,
                err_num);
    return (err_num);
}

/*
 * Copy the EPC's A16 registers to the shared memory buffer in Motorola
 * byte order.
 */

EpcGetULA(session_id, &ula);
EpcCopyData(session_id,
            (void HUGE *) ((char HUGE *) bus_ptr + 0xC000 + (ula << 6)),
            (void HUGE *) shared_ptr,
            0x00000040,
            EPC_16_BIT,
            &actual_size);
```

```
    /*
     * Unmap A16 space.
     */

    EpcUnmapBusMemory(session_id, bus_ptr);

    /*
     * Unmap the shared memory buffer.
     */

    EpcUnmapSharedMemory(session_id, shared_ptr);

    /*
     * Close the session and return.
     */

    EpcCloseSession(session_id);
    WinPrintf("SUCCESS:  MappingSample() complete.\n");
    return (EPC_SUCCESS);
}
```

EpcDeassertInterrupt

Description Deasserts a VME interrupt.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcDeassertInterrupt(unsigned long *Session_ID*);

Session_ID *Session_ID* specifies a session.

Visual Basic Synopsis

Declare Function
EpcDeassertInterrupt% Lib "bmvxiw16.dll" (ByVal *Session_ID&*)

Remarks **EpcDeassertInterrupt** deasserts a currently asserted VME interrupt. If the interface is not currently asserting a VME interrupt, the function has no effect.

When an asserted VME interrupt is acknowledged by a device on the bus, it is automatically deasserted. No call to **EpcDeassertInterrupt** is necessary to deassert the VME interrupt.

Warning:

Deasserting a VME interrupt without waiting for interrupt acknowledgment may cause certain hardware configurations to "lock up." Deasserting a VME interrupt should be executed with extreme care.

Return Value The function returns a Bus Management return value:

EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcAssertInterrupt, EpcGetEpcInterrupt.**

Example See **EpcAssertInterrupt.**

EpcGetBusAttributes

Description Queries the interface's bus management attributes.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetBusAttributes
        (unsigned long          Session_ID,
         unsigned short FAR *   Bus_Enable_Ptr,
         unsigned short FAR *   Bus_Arb_Mode_Ptr,
         unsigned short FAR *   Bus_Arb_Priority_Ptr,
         unsigned short FAR *   Bus_Release_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Bus_Enable_Ptr	*Bus_Enable_Ptr* specifies a location where the interface's bus enable attribute will be placed.
Bus_Arb_Mode_Ptr	*Bus_Arb_Mode_Ptr* specifies a location where the interface's bus arbitration mode attribute will be placed.
Bus_Arb_Priority_Ptr	*Bus_Arb_Priority_Ptr* specifies a location where the interface's bus arbitration priority attribute will be placed.
Bus_Release_Ptr	*Bus_Release_Ptr* specifies a location where the interface's bus release mode attribute will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetBusAttributes% Lib "bmvxiw16.dll"
        (ByVal Session_ID&,
         Bus_Enable_Ptr%,
         Bus_Arb_Mode_Ptr%,
         Bus_Arb_Priority_Ptr%,
         Bus_Release_Ptr%)
```

Remarks

EpcGetBusAttributes queries the interface's bus management attributes and places them in the locations pointed to by *Bus_Enable_Ptr*, *Bus_Arb_Mode_Ptr*, *Bus_Arb_Priority_Ptr*, and *Bus_Release_Ptr*.

The interface's bus enable attribute defines whether accesses made by the interface reach the bus. Possible values placed at *Bus_Enable_Ptr* are:

**Bus_Enable_Ptr*	Description
EPC_DISABLE_BUS	Disable bus accesses for the interface. (EPC-7 and EPC-8 only)
EPC_ENABLE_BUS	Enable bus accesses for the interface.

The interface's bus arbitration mode defines how the interface arbitrates bus collisions. The value placed at *Bus_Arb_Mode_Ptr* only has meaning if the interface has been designated the VXIbus slot-0 controller. Possible values placed at *Bus_Arb_Mode_Ptr* are:

**Bus_Arb_Mode_Ptr*	Description
EPC_PRIORITY	Priority bus arbitration.
EPC_ROUND_ROBIN	Round-robin bus arbitration.

The interface's bus arbitration priority defines the priority level at which the interface arbitrates for the bus. Possible values placed at *Bus_Arb_Priority_Ptr* are:

**Bus_Arb_Priority_Ptr*	Description
EPC_PRIORITY0	Bus arbitration priority 0.
EPC_PRIORITY1	Bus arbitration priority 1.
EPC_PRIORITY2	Bus arbitration priority 2.
EPC_PRIORITY3	Bus arbitration priority 3.

The interface's bus release mode determines when the interface requests and/or releases the bus. Possible values placed at *Bus_Release_Ptr* are:

**Bus_Release_Ptr*	Description
EPC_ROR	"Release On Request" bus release mode.
EPC_RONR	"Release On No Request" bus release mode.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	One or more of the parameters *Bus_Enable_Ptr*, *Bus_Arb_Mode_Ptr*, *Bus_Arb_Priority_Ptr*, and *Bus_Release_Ptr* is invalid.
EPC_INV_SESSION	The parameter *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcOpenSession, EpcSetBusAttributes.**

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * epccfg.c -- Bus Management Library interface configuration functions sample
code.
 */

#include "busmgr.h"

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
EpcCfgSample(void);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * CODE...
```

```
 */

short FAR
EpcCfgSample(void)
{
    char                    err_string[ERROR_STRING_SZ];
    short                   err_num;
    unsigned short          bus_enable;
    unsigned short          bus_arb_mode;
    unsigned short          bus_arb_priority;
    unsigned short          bus_release;
    unsigned short          slave_space;
    unsigned short          ula;
    unsigned long           misc_mask;
    unsigned long           session_id;
    unsigned long           slave_base;
    struct EpcEnvironment environment;

    /*
     * Verify the EPConnect environment.
     */

    if ((err_num = EpcVerifyEnvironment(&environment)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcVerifyEnvironment() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Open a session.
     */

    if ((err_num = EpcOpenSession(&session_id)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcOpenSession() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Query the interface's current configuration settings.
     */

    EpcGetBusAttributes(session_id,
                        &bus_enable,
                        &bus_arb_mode,
                        &bus_arb_priority,
                        &bus_release);
    EpcGetSlaveMapping(session_id, &slave_space, &slave_base);
    EpcGetULA(session_id, &ula);
    EpcGetMiscAttributes(session_id, &misc_mask);

    /*
     * Define the interface's configuration settings.
     */

    EpcSetBusAttributes(session_id,
                        EPC_ENABLE_BUS,
                        EPC_PRIORITY,
                        EPC_PRIORITY0,
```

```
                        EPC_ROR);
    EpcSetSlaveMapping(session_id, EPC_A24, 0x00400000);
    EpcSetULA(session_id, 0xF8);
    EpcSetMiscAttributes(session_id, EPC_PASS | EPC_READY);

    /*
     * Restore the interface's original configuration settings.
     */

    EpcSetBusAttributes(session_id,
                        bus_enable,
                        bus_arb_mode,
                        bus_arb_priority,
                        bus_release);
    EpcSetSlaveMapping(session_id, slave_space, slave_base);
    EpcSetULA(session_id, ula);
    EpcSetMiscAttributes(session_id, misc_mask);

    /*
     * Close the session and return.
     */

    EpcCloseSession(session_id);
    WinPrintf("SUCCESS:  EpcCfgSample() complete.\n");
    return (EPC_SUCCESS);
}
```

EpcGetBusInterrupts

Description Queries actual bus VME interrupt line state.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetBusInterrupts( unsigned long          Session_ID,
                     unsigned long FAR *    Event_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Event_Mask_Ptr	*Event_Mask_Ptr* specifies a location where the actual bus VME interrupt line state will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetBusAttributes% Lib "bmvxiw16.dll"
        (ByVal Session_ID&,
        Event_Mask_Ptr&)
```

Remarks **EpcGetBusInterrupts** queries the actual bus VME interrupt line state and places it in the location pointed to by *Event_Mask_Ptr*.

The value pointed to by *Event_Mask_Ptr* is either zero or an OR'd mask of the following constants. A set bit indicates that the corresponding actual bus VME interrupt line is asserted. A clear bit indicates that the corresponding actual bus VME interrupt line is deasserted:

Constant	Description
EPC_VME1_INT	VME interrupt 1.
...	
EPC_VME7_INT	VME interrupt 7.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Event_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcAssertInterrupt, EpcDeassertInterrupt, EpcGetEpcInterrupt, EpcOpenSession.**

Example See **EpcAssertInterrupt.**

EpcGetBusLines

Description Queries actual bus control line state.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetBusLines( unsigned long          Session_ID,
                unsigned long FAR *    Line_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Line_Mask_Ptr	*Line_Mask_Ptr* specifies a location where the actual bus control line state will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetBusLines% Lib "bmvxiw16.dll"
                (ByVal Session_ID&, Line_Mask_Ptr&)
```

Remarks **EpcGetBusLines** queries the actual bus control line state and places it in the location pointed to by *Line_Mask_Ptr*.

The value pointed to by *Line_Mask_Ptr* is either zero or an OR'd mask of the following constants. A set bit indicates that the corresponding actual bus control line is asserted. A clear bit indicates that the corresponding actual bus control line is deasserted:

Constant	Description
EPC_ACFAIL	ACFAIL.
EPC_SYSFAIL	SYSFAIL.
EPC_SYSRESET	SYSRESET. Supported on EPC-7 and VXLink only.

The value pointed to by *Line_Mask_Ptr* reflects the actual bus control line state, not the interface control line state. Use **EpcGetEpcLines** to query the interface control line state.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Line_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetEpcLines, EpcOpenSession, EpcPulseEpcLines, EpcSetEpcLines.**

Example See **EpcAssertInterrupt.**

EpcGetBusMODID

Description Queries the actual bus MODID line state.

C Synopsis

```
#include "busmgr.h"

short
EpcGetBusMODID( unsigned long      Session_ID,
                unsigned long FAR * MODID_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
MODID_Mask_Ptr	*MODID_Mask_Ptr* specifies a location where the actual bus MODID line state will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetBusMODID%Lib"bmvxiw16.dll" (ByVal
Session_ID&,MODID_Mask_Ptr&)
```

Remarks **EpcGetBusMODID** queries the actual bus MODID line state and places it in the location pointed to by *MODID_Mask_Ptr*.

The value pointed to by *MODID_Mask_Ptr* is either zero or an OR'd mask of the following constants. A set bit indicates that the corresponding actual bus MODID line is asserted.

A clear bit indicates that the corresponding actual bus MODID line is deasserted:

Constant	Description
EPC_SLOT_MODID	MODID line for the interface's bus slot.

The value pointed to by *MODID_Mask_Ptr* reflects the actual bus MODID line state, not the interface MODID line state. Use **EpcGetEpcMODID** to query the interface MODID line state.

A device can always query the state of the MODID bus control line corresponding to its bus slot. The **MODID_Mask_Ptr* state bit **EPC_SLOT_MODID** always contains valid data.

Return Value The function returns a EPConnect return value:

EPC_INV_PTR	The parameter *MODID_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetEpcMODID, EpcOpenSession, EpcSetEpcMODID.**

EpcGetBusTriggers

Description Queries the actual bus trigger line state.

C Synopsis

```
#include "busmgr.h"

short
EpcGetBusTriggers( unsigned long          Session_ID,
                   unsigned long FAR *    Trigger_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Trigger_Mask_Ptr	*Trigger_Mask_Ptr* specifies a location where the actual bus trigger line state will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetBusTriggers% Lib "bmvxiw16.dll" (ByVal
Session_ID&,Trigger_Mask_Ptr&)
```

Remarks **EpcGetBusTriggers** queries the actual bus trigger line state and places it in the location pointed to by *Trigger_Mask_Ptr*.

The value pointed to by *Trigger_Mask_Ptr* is either zero or an OR'd mask of the following constants. A set bit indicates that the corresponding actual bus trigger line is asserted.

A clear bit indicates that the corresponding actual bus trigger line is deasserted:

Constant	Description
EPC_ECL_TRIG0	ECL trigger 0 (EPC-7 only).
EPC_ECL_TRIG1	ECL trigger 1 (EPC-7 only).
EPC_TTL_TRIG0	TTL trigger 0 (EPC-7 and VXLink only).
...	
EPC_TTL_TRIG7	TTL trigger 7 (EPC-7 and VXLink only).

The value pointed to by *Trigger_Mask_Ptr* reflects the actual bus trigger line state, not the interface trigger line state. Use **EpcGetEpcTriggers** to query the interface trigger line state.

Return Value

The function returns a EPConnect return value:

EPC_INV_PTR	The parameter *Trigger_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also

EpcGetEpcTriggers, EpcOpenSession, EpcPulseEpcTriggers, EpcSetEpcTriggers.

EpcGetEpcInterrupt

Description Queries the interface VME interrupt assertion state.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetEpcInterrupt( unsigned long          Session_ID,
                    unsigned long FAR *    Event_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Event_Mask_Ptr	*Event_Mask_Ptr* specifies a location where the currently asserted VME interrupt will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetEpcInterrupt% Lib "bmvxiw16.dll" (ByVal Session_ID&,
Event_Mask_Ptr&)
```

Remarks **EpcGetEpcInterrupt** queries the VME interrupt currently asserted by the interface and places a it in the location pointed to by *Event_Mask_Ptr*.

The function places a constant at *Event_Mask_Ptr* specifying the VME interrupt currently asserted by the interface. Possible values are:

Constant	Description
EPC_NO_INT	The interface is not currently asserting a VME interrupt.
EPC_VME1_INT	The interface is currently asserting VME interrupt 1.
...	

EPC_VME7_INT	The interface is currently asserting VME interrupt 7.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Event_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcAssertInterrupt, EpcDeassertInterrupt, EpcGetBusInterrupts.**

Example See **EpcAssertInterrupt.**

EpcGetEpcLines

Description Queries the interface control line state.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetEpcLines(unsigned long          Session_ID,
               unsigned long FAR *    Line_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Line_Mask_Ptr	*Line_Mask_Ptr* specifies a location where the interface control line state will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetEpcLines% Lib "bmvxiw16.dll" (ByVal Session_ID&,
Line_Mask_Ptr&)
```

Remarks **EpcGetEpcLines** queries the interface control line state and places it in the location pointed to by *Line_Mask_Ptr*.

The value pointed to by *Line_Mask_Ptr* is either zero or an OR'd mask of the following constants. A set bit indicates that the corresponding interface control line is asserted. A clear bit indicates that the corresponding interface control line is deasserted:

Constant	Description
EPC_SYSFAIL	SYSFAIL.
EPC_SYSRESET	SYSRESET.

The value pointed to by *Line_Mask_Ptr* reflects the interface control line state, not the actual bus control line state. Use **EpcGetBusLines** to query the actual bus control line state.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Line_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetBusLines, EpcOpenSession, EpcPulseEpcLines, EpcSetEpcLines.**

Example See **EpcAssertInterrupt.**

EpcGetEpcMODID

Description Queries the interface MODID line state.

C Synopsis

```
#include "busmgr.h"

short
EpcGetEpcMODID( unsigned long        Session_ID,
                unsigned long FAR *  MODID_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
MODID_Mask_Ptr	*MODID_Mask_Ptr* specifies a location where the interface MODID line state will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetEpcMODID% Lib "bmvxiw16.dll" (ByVal
Session_ID&,MODID_Mask_Ptr&)
```

Remarks **EpcGetEpcMODID** queries the interface MODID line state and places it in the location pointed to by *MODID_Mask_Ptr*.

The value pointed to by *MODID_Mask_Ptr* is either zero or an OR'd mask of the following constants. A set bit indicates that the corresponding interface MODID line is asserted. A clear bit indicates that the corresponding interface MODID line is deasserted:

Constant	Description
EPC_MODID0	MODID line 0 (EPC-7 and VXLink only).
...	
EPC_MODID12	MODID line 12 (EPC-7 and VXLink only).

The value pointed to by *MODID_Mask_Ptr* reflects the interface MODID line state, not the actual bus MODID line state. Use **EpcGetBusMODID** to query the actual bus MODID line state.

Return Value The function returns a EPConnect return value:

EPC_INV_PTR	The parameter *MODID_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetBusMODID, EpcOpenSession, EpcSetEpcMODID.**

EpcGetEpcTriggerMapping

Description Queries an interface trigger line mapping.

C Synopsis

```
#include "busmgr.h"

short
EpcGetEpcTriggerMapping
                (unsigned long Session_ID,
                unsigned long In_Trigger_Mask,
                unsigned long FAR * Out_Trigger_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
In_Trigger_Mask	*In_Trigger_Mask* specifies an interface trigger line.
Out_Trigger_Mask_Ptr	*Out_Trigger_Mask_Ptr* specifies a location where a mask of interface trigger lines will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetEpcTriggerMapping% Lib "bmvxiw16.dll" (ByVal
        Session_ID&, ByVal In_Trigger_Mask&,
        Out_Trigger_Mask_Ptr&)
```

Remarks **EpcGetEpcTriggerMapping** queries the interface trigger lines mapped to the specified *In_Trigger_Mask* and places a mask identifying them in the location pointed to by *Out_Trigger_Mask_Ptr*.

The parameter *In_Trigger_Mask* is a constant specifying a single interface trigger line. The value placed at the location pointed to by *Out_Trigger_Mask_Ptr* is an OR'd mask of constants identifying the interface trigger lines mapped to the specified input interface trigger line. The table below enumerates possible trigger mapping combinations for an EPC-7 interface:

In_Trigger_Mask	**Out_Trigger_Mask_Ptr*	Description
EPC_EXT_TRIG0	EPC_TTL_TRIG0 ... EPC_TTL_TRIG7	External trigger 0 mapped as input to a single TTL trigger line.
EPC_TTL_TRIG0 ... EPC_TTL_TRIG7	EPC_EXT_TRIG0	A single TTL trigger line mapped as input to external trigger 0.

The table below enumerates possible trigger mapping combinations for a VXLink interface:

In_Trigger_Mask	**Out_Trigger_Mask_Ptr*	Description
EPC_EXT_TRIG0	0x00000000	External trigger 0 unmapped.
EPC_EXT_TRIG0	EPC_TTL_TRIG0 ... EPC_TTL_TRIG7	External trigger 0 mapped as input to a single TTL trigger line.
EPC_TTL_TRIG0 ... EPC_TTL_TRIG7	EPC_EXT_TRIG1	A single TTL trigger line mapped as input to external trigger 1.

When an external trigger line is mapped as input to one or more interface trigger lines, asserting the external trigger line asserts all of the mapped interface trigger lines. Likewise, deasserting the external trigger line deasserts all of the mapped interface trigger lines.

When one or more interface trigger lines are mapped as input to an external trigger line, asserting one of the interface trigger lines asserts the mapped external trigger line. Likewise, deasserting one of the interface trigger lines deasserts the mapped external trigger line.

An EPC-7 interface provides a single bi-directional external trigger. The external trigger is always mapped; it cannot be unmapped. Specifying a mapping for external trigger 0 overrides the previous mapping. By default, TTL trigger 1 is mapped as an output to external trigger 0.

A VXLink interface provides two unidirectional external triggers. External trigger 0 is an input-only trigger and external trigger 1 is an output-only trigger. The external triggers can be independently mapped or unmapped. By default, both external triggers are unmapped.

Return Value The function returns a EPConnect return value:

EPC_INV_MASK	The parameter *In_Trigger_Mask* is invalid.
EPC_INV_PTR	The parameter *Out_Trigger_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcMapEpcTriggers, EpcOpenSession.**

EpcGetEpcTriggers

Description Query the interface trigger line state.

C Synopsis

```
#include "busmgr.h"

short
EpcGetEpcTriggers   (unsigned long      Session_ID,
                     unsigned long FAR * Trigger_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Trigger_Mask_Ptr	*Trigger_Mask_Ptr* specifies a location where the EPC trigger line state will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetEpcTriggers% Lib "bmvxiw16.dll" (ByVal
Session_ID&,Trigger_Mask_Ptr&)
```

Remarks **EpcGetEpcTriggers** queries the interface trigger line state and places it in the location pointed to by *Trigger_Mask_Ptr.*

The value pointed to by *Trigger_Mask_Ptr* is either zero or an OR'd mask of the following constants. A set bit indicates that the corresponding interface trigger line is asserted.

A clear bit indicates that the corresponding interface trigger line is deasserted:

Constant	Description
EPC_ECL_TRIG0	ECL trigger 0 (EPC-7 only)
EPC_ECL_TRIG1	ECL trigger 1 (EPC-7 only)
EPC_TTL_TRIG0	TTL trigger 0 (EPC-7 and VXLink only)
...	
EPC_TTL_TRIG7	TTL trigger 7 (EPC-7 and VXLink only)

The value pointed to by *Trigger_Mask_Ptr* reflects the interface trigger line state, not the actual bus trigger line state. Use **EpcGetBusTriggers** to query the actual bus trigger line state.

Return Value

The function returns a EPConnect return value:

EPC_INV_PTR	The parameter *Trigger_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also

EpcGetBusTriggers, **EpcOpenSession**, **EpcPulseEpcTriggers**, **EpcSetEpcTriggers**.

EpcGetErrorString

Description Queries a null-terminated string corresponding to a Bus Management return value.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetErrorString( short       Return_Value
                   char FAR*   Buffer_Ptr);
```

Return_Value — *Return_Value* specifies a Bus Management return value.

Buffer_Ptr — *Buffer_Ptr* specifies the location of a buffer where the null-terminated string will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetErrorString% Lib "bmvxiw16.dll" (ByVal
Return_Value&, Buffer_Ptr&)
```

Remarks **EpcGetErrorString** places a null-terminated ASCII character string describing a Bus Management return value in the buffer pointed to by *Buffer_Ptr*.

Return_Value specifies a Bus Management return value. Specifying an invalid value results in the function returning a pointer to the string "Unknown EPConnect Return Value".

The buffer pointed to by *Buffer_Ptr* must be at least **ERROR_STRING_SZ** bytes long.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Buffer_Ptr* is invalid.
EPC_SUCCESS	The function completed successfully.

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * environ.c -- Bus Management Library environment functions sample code.
 */

#include "busmgr.h"

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
EnvironmentSample(void);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * CODE...
 */

short FAR
EnvironmentSample(void)
{
    char                  err_string[ERROR_STRING_SZ];
    short                 err_num;
    struct EpcEnvironment environment;

    /*
     * Verify the EPConnect environment.
     */

    if ((err_num = EpcVerifyEnvironment(&environment)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcVerifyEnvironment() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }
    WinPrintf("SUCCESS:  EnvironmentSample() complete.\n");
    return (EPC_SUCCESS);
}
```

EpcGetEventEnableMask

Description Queries a session's enabled event mask attribute.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetEventEnableMask
        (unsigned long          Session_ID,
        unsigned long FAR *  Event_Mask_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Event_Mask_Ptr	*Event_Mask_Ptr* specifies a location where the enabled event mask attribute of the specified session will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetEventEnableMask% Lib "bmvxiw16.dll" (ByVal
Session_ID&, Event_Mask_Ptr&)
```

Remarks **EpcGetEventEnableMask** places the specified session's enabled event mask attribute in the location pointed to by *Event_Mask_Ptr*.

An enabled event mask attribute is a bit mask where each bit corresponds to an event. A zero in a bit position specifies that the corresponding event's reception is disabled. A one in a bit position specifies that the corresponding event's reception is enabled.

The mask is either zero or an OR'd combination of the following constants:

Event	Description
EPC_MSG_INT	Message interrupt (EPC-7 and EPC-8 only)
EPC_VME1_INT	VMEbus interrupt 1
...	
EPC_VME7_INT	VMEbus interrupt 7
EPC_SIGNAL_INT	VXIbus signal FIFO interrupt
EPC_TTL_TRIG0_INT	VXIbus TTL Trigger 0 interrupt (EPC-7 only)
...	
EPC_TTL_TRIG7_INT	VXIbus TTL Trigger 7 interrupt (EPC-7 only)
EPC_SYSRESET_ERR	VMEbus SYSRESET error
EPC_ACFAIL_ERR	VMEbus power failure error
EPC_BERR_ERR	VMEbus access error
EPC_SYSFAIL_ERR	VMEbus SYSFAIL error
EPC_WATCHDOG_ERR	Watchdog timer expiration error (EPC-7 and EPC-8 only)
EPC_EXT_TRIG0_INT	External trigger 0 interrupt (VXLink only)
EPC_EXT_TRIG1_INT	External trigger 1 interrupt (VXLink only)

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Event_Mask_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_SUCCESS	The function completed successfully.
EPC_INV_SW	The BusManager device driver is not present.

See Also **EpcSetEventEnableMask, EpcOpenSession.**

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * events.c -- Bus Management Library events functions sample code.
 */

#include "busmgr.h"

/*
 * CONSTANTS...
 */

#define STACK_SIZE                4096

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
EventsSample(void);

void FAR LOADDS
EventHandler(unsigned long Session_ID,
             unsigned long Event_Mask,
             unsigned long Event_Data);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * GLOBAL DATA...
 */

unsigned char EventStack[STACK_SIZE] = { 0 };

/*
 * CODE...
 */

short FAR
EventsSample(void)
{
    char                  err_string[ERROR_STRING_SZ];
    short                 err_num;
    unsigned long         event_data;
    unsigned long         event_mask;
    unsigned long         session_id;
    void             FAR *event_stack;
    void            (FAR *event_handler)(unsigned long,
                                         unsigned long,
                                         unsigned long);
    struct EpcEnvironment environment;

    /*
     * Verify the EPConnect environment.
     */

    if ((err_num = EpcVerifyEnvironment(&environment)) != EPC_SUCCESS)
    {
```

```
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcVerifyEnvironment() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Open a session.
     */

    if ((err_num = EpcOpenSession(&session_id)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcOpenSession() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Define the session's event handler for VMEbus interrupt 1.
     */

    EpcSetEventHandler(session_id,
                       EPC_VME1_INT,
                       (void (FAR *)(unsigned long,
                                     unsigned long,
                                     unsigned long)) EventHandler,
                       (void FAR *) &EventStack[STACK_SIZE]);

    /*
     * Define the session's event enable mask to enable VMEbus interrupt 1.
     */

    EpcSetEventEnableMask(session_id, EPC_VME1_INT);

    /*
     * Query the session's event handler for VMEbus interrupt 1.
     */

    EpcGetEventHandler(session_id, EPC_VME1_INT, &event_handler, &event_stack);

    /*
     * Query the session's event enable mask.
     */

    EpcGetEventEnableMask(session_id, &event_mask);

    /*
     * Wait up to one second (1000 ms) for VMEbus interrupt 1 to occur.
     */

    EpcWaitForEvent(session_id, 1000, EPC_VME1_INT, &event_mask, &event_data);

    /*
     * Close the session and return.
     */

    EpcCloseSession(session_id);
    WinPrintf("SUCCESS:  EventsSample() complete.\n");
    return (EPC_SUCCESS);
}

void FAR LOADDS
```

```
EventHandler(unsigned long Session_ID,
             unsigned long Event_Mask,
             unsigned long Event_Data)
{
    /*
     * Avoid compiler warnings.
     */

    Session_ID = Session_ID;
    Event_Mask = Event_Mask;
    Event_Data = Event_Data;
}
```

EpcGetEventHandler

Description Queries an entry in a session's event handler array.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetEventHandler

(unsigned long      Session_ID,
unsigned long       Event_Mask,
void (FAR * FAR *   Event_Handler_Ptr)(unsigned long,
                      unsigned long, unsigned long),
void FAR * FAR *    Stack_Ptr_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Event_Mask	*Event_Mask* specifies an event.
Event_Handler_Ptr	*Event_Handler_Ptr* specifies a location where the specified session's specified event handler will be placed.
Stack_Ptr_Ptr	*Stack_Ptr_Ptr* specifies a location where the specified session's specified event handler stack will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetEventHandler% Lib "bmvxiw16.dll"
                    (ByVal Session_ID&,
                    ByVal Event_Mask&,
                    Event_Handler_Ptr As Any,
                    Stack_Ptr_Ptr As Any)
```

Remarks **EpcGetEventHandler** places the specified session's specified event handler address and event handler stack pointer in the locations pointed to by *Event_Handler_Ptr* and *Stack_Ptr_Ptr*.

The *Event_Mask* parameter is a bit mask where each bit corresponds to an event. The *Event_Mask* parameter should be one of the following constants:

Event	Description
EPC_MSG_INT	Message interrupt (EPC-7 and EPC-8 only)
EPC_VME1_INT	VMEbus interrupt 1
...	
EPC_VME7_INT	VMEbus interrupt 7
EPC_SIGNAL_INT	VXIbus signal FIFO interrupt
EPC_TTL_TRIG0_INT	VXIbus TTL Trigger 0 interrupt (EPC-7 only)
...	
EPC_TTL_TRIG7_INT	VXIbus TTL Trigger 7 interrupt (EPC-7 only)
EPC_SYSRESET_ERR	VMEbus SYSRESET error
EPC_ACFAIL_ERR	VMEbus power failure error
EPC_BERR_ERR	VMEbus access error
EPC_SYSFAIL_ERR	VMEbus SYSFAIL error
EPC_WATCHDOG_ERR	Watchdog timer expiration error (EPC-7 and EPC-8 only)
EPC_EXT_TRIG0_INT	External trigger 0 interrupt (VXLink only)
EPC_EXT_TRIG1_INT	External trigger 1 interrupt (VXLink only)

If the session has no event handler defined for the specified event, the function places NULL in the locations pointed to by *Event_Handler_Ptr* and *Stack_Ptr_Ptr*.

Return Value The function returns a Bus Management return value:

EPC_INV_MASK	*Event_Mask* contains more than one event or contains an event that is not valid for this EPC.
EPC_INV_PTR	One or both of the parameters *Event_Handler_Ptr* and *Stack_Ptr_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcSetEventHandler, EpcOpenSession.**

Example See **EpcGetEventEnableMask.**

EpcGetLockingTimeout

Description Queries a session's locking timeout.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetLockingTimeout( unsigned long          Session_ID,
                      unsigned long FAR *    Timeout_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Timeout_Ptr	*Timeout_Ptr* specifies a location where the specified session's locking timeout will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetLockingTimeout% Lib "bmvxiw16.dll" (ByVal
Session_ID&, Timeout_Ptr&)
```

Remarks **EpcGetLockingTimeout** queries the specified session's locking timeout and places it in the location pointed to by *Timeout_Ptr*.

Upon successful function completion, *Timeout_Ptr* contains the session's locking timeout, in milliseconds.

By default, a session has a locking timeout of zero milliseconds. When the session encounters a locking conflict, an **EPC_LOCKED** error is returned immediately.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Timeout_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcLockSession, EpcOpenSession, EpcSetLockingTimeout.**

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * locking.c -- Bus Management Library locking functions sample code.
 */

#include "busmgr.h"

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
LockingSample(void);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * CODE...
 */

short FAR
LockingSample(void)
{
    char                  err_string[ERROR_STRING_SZ];
    short                 err_num;
    unsigned long         session_id1;
    unsigned long         session_id2;
    unsigned long         timeout;
    struct EpcEnvironment environment;

    /*
     * Verify the EPConnect environment.
     */

    if ((err_num = EpcVerifyEnvironment(&environment)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcVerifyEnvironment() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Open two sessions.
     */

    if ((err_num = EpcOpenSession(&session_id1)) != EPC_SUCCESS ||
        (err_num = EpcOpenSession(&session_id2)) != EPC_SUCCESS  )
    {
        EpcCloseSession(session_id1);
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcOpenSession() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
```

```
     * Define the second session's locking timeout to be one second (1000 ms).
     */

    EpcSetLockingTimeout(session_id2, 1000);

    /*
     * Query the second session's locking timeout.
     */

    EpcGetLockingTimeout(session_id2, &timeout);

    /*
     * Lock shared interface hardware.
     *
     * NOTES:
     *      1.  The EpcLockSession() call for the second session fails after a
     *          one second (1000 ms) timeout, since shared interface hardware is
     *          already locked by the first session.
     */

    EpcLockSession(session_id1);
    EpcLockSession(session_id2);

    /*
     * Unlock shared interface hardware with both sessions.
     */

    EpcUnlockSession(session_id1);

    /*
     * Close the sessions and return.
     */

    EpcCloseSession(session_id1);
    EpcCloseSession(session_id2);
    WinPrintf("SUCCESS:  LockingSample() complete.\n");
    return (EPC_SUCCESS);
}
```

EpcGetMappingAttributes

Description Queries a memory mapping's attributes.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetMappingAttributes
    (unsigned long          Session_ID,
    volatile void   HUGE *  Mapped_Ptr,
    unsigned short  FAR *   Address_Mod_Ptr,
    unsigned short  FAR *   Byte_Ordering_Ptr,
    unsigned long   FAR *   Base_Address_Ptr,
    unsigned long   FAR *   Size_Ptr);
```

Session_ID — *Session_ID* specifies a bus session.

Mapped_Ptr — *Mapped_Ptr* specifies a pointer to the base of a memory mapping.

Address_Mod_Ptr — *Address_Mod_Ptr* specifies a location where the address modifier attribute of the specified memory mapping will be placed.

Byte_Ordering_Ptr — *Byte_Ordering_Ptr* specifies a location where the byte ordering attribute of the specified memory mapping will be placed.

Base_Address_Ptr — *Base_Address_Ptr* specifies a location where the base address attribute of the specified memory mapping will be placed.

Size_Ptr — *Size_Ptr* specifies a location where the size attribute of the specified memory mapping, in bytes, will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetMappingAttributes% Lib "bmvxiw16.dll"
                         (ByVal Session_ID&,
                         ByVal Mapped_Ptr As Any,
                         Address_Mod_Ptr%,
                         Byte_Ordering_Ptr%,
                         Base_Address_Ptr&,
                         Size_Ptr&)
```

Remarks

EpcGetMappingAttributes places the specified memory mapping's attributes in the locations pointed to by *Address_Mod_Ptr*, *Byte_Ordering_Ptr*, *Base_Address_Ptr*, and *Size_Ptr*, respectively.

The location pointed to by *Address_Mod_Ptr* can contain the following values:

Constant	Description
EPC_A16N	VMEbus A16 non-supervisory address modifier.
EPC_A16S	VMEbus A16 supervisory address modifier.
EPC_A24ND	VMEbus A24 non-supervisory data address modifier.
EPC_A24SD	VMEbus A24 supervisory data address modifier.
EPC_A24NP	VMEbus A24 non-supervisory program address modifier.
EPC_A24SP	VMEbus A24 supervisory program address modifier.
EPC_A32ND	VMEbus A32 non-supervisory data address modifier.
EPC_A32SD	VMEbus A32 supervisory data address modifier.
EPC_A32NP	VMEbus A32 non-supervisory program address modifier.
EPC_A32SP	VMEbus A32 supervisory program address modifier.
EPC_SHARED	Shared memory address modifier.

The location pointed to by *Byte_Ordering_Ptr* can have the following values:

Constant	Description
EPC_IBO	Intel (80X86) byte ordering.
EPC_MBO	Motorola (68XXX) byte ordering.

For shared memory mappings, the value in the location pointed to by *Byte_Ordering_Ptr* is always **EPC_IBO**.

The values in the locations pointed to by *Base_Address_Ptr* and *Size_Ptr* define a range of addresses α, where:

**Base_Address_Ptr* <= α <= **Base_Address_Ptr* + **Size_Ptr* - 1;

For bus memory mappings, the value in the location pointed to by *Base_Address_Ptr* specifies a physical VMEbus address.

For shared memory mappings, the value in the location pointed to by *Base_Address_Ptr* specifies a physical PC address. To determine the corresponding physical VMEbus address, the value should be added to the base address of the interface's slave memory. Use **EpcGetSlaveMapping** to determine the base address of the interface's slave memory.

Return Value

The function returns a Bus Management return value:

EPC_INV_MAP	The specified *Mapped_Ptr* is invalid.
EPC_INV_PTR	One or more of the parameters *Address_Mod_Ptr*, *Byte_Ordering_Ptr*, *Base_Address_Ptr*, or *Size_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also

EpcGetSlaveMapping, **EpcMapBusMemory**, **EpcMapSharedMemory**, **EpcOpenSession**.

Example

See **EpcCopyData**.

EpcGetMiscAttributes

2

Description Queries the interface's miscellaneous configuration attributes.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetMiscAttributes( unsigned long          Session_ID,
                      unsigned long FAR *    Misc_Mask_Ptr);
```

Session_ID — *Session_ID* specifies a session.

Misc_Mask_Ptr — *Misc_Mask_Ptr* specifies a location where the interface's miscellaneous configuration attributes will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetMiscAttributes% Lib "bmvxiw16.dll" (ByVal
Session_ID&, Misc_Mask_Ptr&)
```

Remarks **EpcGetMiscAttributes** queries the interface's miscellaneous configuration attributes and places them in the location pointed to by *Misc_Mask_Ptr*.

The location pointed to by *Misc_Mask_Ptr* contains either a zero or an OR'd bit mask of the following constants, where a set bit indicates that the corresponding miscellaneous interface attribute is asserted.

A clear bit indicates that the corresponding miscellaneous interface attribute is deasserted:

Constant	Description
EPC_DIR	Word serial byte transfer protocol DIR bit. Asserting the bit indicates that the interface is ready to receive data from its commander device. Supported on EPC-7 and EPC-8 only.

EPC_DOR	Word serial byte transfer protocol DOR bit. Asserting the bit indicates that the interface is ready to send data to its commander device. Supported on EPC-7 and EPC-8 only.
EPC_ERR	Word serial protocol ERR* bit. Asserting the bit indicates to the commander device that the interface has detected a word serial protocol error. Supported on EPC-7 and EPC-8 only.
EPC_LOCK	VXIbus message-based device LOCKED* bit. Asserting the bit indicates that the commander has locked access to the interface from local sources (IEEE-488 local lockout). Supported on EPC-7 and EPC-8 only.
EPC_MULTIPLE_ LOCK	Word serial protocol extension multiple commander lock bit. When asserted, the first commander to read the asserted bit from interface's Response register can safely send a word serial command. Supported on EPC-7 and EPC-8 only.
EPC_PASS	Device initialization PASSED bit. Asserting the bit indicates that the interface has passed self-test.
EPC_PIPELINE_BUSY	Bus hardware pipeline busy bit. When asserted, the bit indicates that the interface is executing a pipelined write to the bus.
EPC_READY	Device initialization READY bit. Asserting the bit indicates that the interface is ready to begin normal operation.

EPC_RRDY	Word serial protocol Read Ready bit. Asserting the bit indicates to a commander device that the interface has a word serial response in its message register. Supported on EPC-7 and EPC-8 only.
EPC_RSRC_MGR	Resource manager execution bit. Asserting the bit indicates that resource manager execution is complete.
EPC_STICKY_BERR	"Sticky" bus error bit. When asserted, the bit indicates that a bus error has occurred since the bit was last deasserted.
EPC_SYSFAIL_OUT	SYSFAIL output enable bit. When asserted, the interface can assert SYSFAIL. When deasserted, the interface cannot assert SYSFAIL.
EPC_SYSRESET_IN	SYSRESET input enable bit. When asserted, asserting SYSRESET resets the interface. When deasserted, asserting SYSRESET does not reset the interface.
EPC_TTL_LATCH0 ... EPC_TTL_LATCH7	TTL trigger latch bits. When asserted, a bit indicates that the interface has latched the corresponding TTL trigger interrupt. Supported on EPC-7 only.
EPC_WATCHDOG	Watchdog timer expiration bit. When asserted, the bit indicates that a watchdog timeout error has occurred since the watchdog timer was last reset. Supported on EPC-7 and EPC-8 only.
EPC_WRDY	Word serial protocol Write Ready bit. Asserting the bit indicates to a commander device that the interface is ready to receive a word serial command. Supported on EPC-7 and EPC-8 only.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Misc_Mask_Ptr* is invalid.
EPC_INV_SESSION	The parameter *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcOpenSession, EpcSetMiscAttributes.**

Example See **EpcGetBusAttributes.**

EpcGetSessionData

Description Queries a session's application-specified data.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetSessionData( unsigned long          Session_ID,
                   unsigned long FAR *    Session_Data_Ptr);
```

Session_ID	*Session_ID* specifies an open session.
Session_Data_Ptr	*Session_Data_Ptr* specifies a location where the session's application-specified data will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetSessionData% Lib "bmvxiw16.dll" (ByVal Session_ID&,
Session_Data_Ptr&)
```

Remarks **EpcGetSessionData** queries the specified session's application-specified data and places it in the location pointed to by *Session_Data_Ptr*.

The application-specified data is a 4-byte quantity.

Typically, an application uses **EpcSetSessionData** to store a pointer to one of its data structures. Later, the application uses **EpcGetSessionData** to quickly retrieve the pointer during performance-critical operations (like event handling).

Return Value The function returns a Bus Management return value:

EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_PTR	The *Session_Data_Ptr* parameter is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcOpenSession, EpcSetSessionData.**

Example See **EpcCloseSession.**

EpcGetSlaveMapping

Description Queries the interface's slave memory mapping.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcGetSlaveMapping
        (unsigned long          Session_ID,
        unsigned short  FAR *   Addess_Space_Ptr,
        unsigned long   FAR *   Base_Addess_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Addess_Space_Ptr	*Addess_Space_Ptr* specifies a location where the interface's slave memory address space will be placed.
Base_Addess_Ptr	*Base_Addess_Ptr* specifies a location where the interface's slave memory base address will be placed.

Visual Basic Synopsis

```
Declare Function
EpcGetSlaveMapping% Lib "bmvxiw16.dll"
                        ( ByVal Session_ID&,
                        Address_Space_Ptr%,
                        Base_Address_Ptr&)
```

Remarks **EpcGetSlaveMapping** queries the mapping of the interface's slave memory and places the result in the locations pointed to by *Addess_Space_Ptr* and *Base_Addess_Ptr*.

Possible values at *Addess_Space_Ptr* and *Base_Addess_Ptr* are dependent on the interface type:

Interface Type	*Addess_Space_Ptr	*Base_Addess_Ptr
EPC-4	**EPC_DISABLED**	N/A
	EPC_A24	0x00000000, 0x00400000, ..., 0x00C00000
	EPC_A32	0x18000000, 0x19000000, ..., 0x1F000000
EPC-5	**EPC_DISABLED**	N/A
	EPC_A24	0x00000000, 0x00400000, ..., 0x00C00000
	EPC_A32	0x18000000, 0x19000000, ..., 0x1F000000
EPC-7	**EPC_DISABLED**	N/A
	EPC_A24	0x00000000, 0x00400000, ..., 0x00C00000
	EPC_A32	0x00000000, 0x01000000, ..., 0xFF000000
EPC-8	**EPC_DISABLED**	N/A
VXLink	**EPC_DISABLED**	N/A

A24 base addresses are aligned on a 4 Mbyte boundary, and only the first 4 Mbytes of the interface's slave memory is mapped to the bus. A32 base addresses are aligned on a 16 Mbyte boundary, and only the first 16 Mbytes of the interface's slave memory is mapped to the bus.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	One or more of the parameters *Addess_Space_Ptr* and *Base_Addess_Ptr* is invalid.
EPC_INV_SESSION	The parameter *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcOpenSession**, **EpcSetSlaveMapping**.

Example See **EpcGetBusAttributes**.

EpcGetULA

Description Queries the interface's unique logical address.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcGetULA(unsigned long *Session_ID*, unsigned short **FAR** **ULA_Ptr*);

Session_ID	*Session_ID* specifies a session.
ULA_Ptr	*ULA_Ptr* specifies a location where the interface's unique logical address will be placed.

Visual Basic Synopsis

Declare Function
EpcGetULA% Lib "bmvxiw16.dll" (ByVal *Session_ID&*, *ULA_Ptr%*)

Remarks **EpcGetULA** queries the interface's unique logical address and places the result in the locations pointed to by *ULA_Ptr*. Possible unique logical addresses are 0x00 through 0xFF.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameters *ULA_Ptr* is invalid.
EPC_INV_SESSION	The parameter *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcOpenSession, EpcSetULA.**

Example See **EpcGetBusAttributes.**

EpcLockSession

Description Locks shared interface hardware for a session.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcLockSession(unsigned long *Session_ID*);

Session_ID *Session_ID* specifies a session.

Visual Basic Synopsis

Declare Function
EpcLockSession% Lib "bmvxiw16.dll" (ByVal *Session_ID*&)

Remarks **EpcLockSession** locks shared interface hardware for the specified *Session_ID*.

Locking gives a session exclusive access to shared interface hardware. Locking is used in multithreaded environments to prevent simultaneous, potentially conflicting hardware accesses.

Locks can be nested. EPConnect maintains a global lock counter. The global lock counter can be "owned" by at most one session. Initially, the lock counter is zero, indicating that no session has locked shared interface hardware. **EpcLockSession** acquires and increments the lock counter for a session. **EpcUnlockSession** decrements the lock counter for the same session. A non-zero lock counter indicates that shared interface hardware is locked.

When an application calls a Bus Management Bus Management Library function that obeys the locking paradigm, the function checks for an existing lock. If no lock exists or the specified session "owns" the lock, the function proceeds. Otherwise, the function suspends execution until the lock is released or the specified session's locking timeout expires. If the existing lock is not released before the specified session's locking timeout expires, the function returns **EPC_LOCKED**.

Use **EpcGetLockingTimeout** and **EpcSetLockingTimeout** to query and define a session's locking timeout.

The following EPConnect Bus Management Library functions obey locks:

EpcAssertInterrupt	**EpcSetEpcLines**
EpcCmdReceiveWSBuffer	**EpcSetEpcMODID**
EpcCmdSendWSBuffer	**EpcSetEpcTriggers**
EpcCmdSendWSCommand	**EpcSetMiscAttributes**
EpcDeassertInterrupt	**EpcSetSlaveMapping**
EpcLockSession	**EpcSetULA**
EpcMapBusMemory	**EpcSrvEnableWsCommand**
EpcMapEpcTriggers	**EpcSrvReceiveWSCommand**
EpcMapSharedMemory	**EpcSrvSendProtocolEvent**
EpcPulseEpcLines	**EpcSrvSendWSProtocolError**
EpcPulseEpcTriggers	**EpcSrvSendWSResponse**
EpcSetBusAttributes	**EpcValidateBusMapping**

Return Value

The function returns a Bus Management return value:

EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also

EpcGetLockingTimeout, EpcOpenSession, EpcSetLockingTimeout, EpcUnlockSession.

Example

See **EpcGetLockingTimeout.**

EpcMapBusMemory

Description Creates a bus memory mapping using statically configured bus window hardware.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcMapBusMemory( unsigned long          Session_ID,
                 unsigned short         Address_Mod,
                 unsigned short         Byte_Ordering,
                 unsigned long          Base_Address,
                 unsigned long          Size,
                 void HUGE * FAR *      Mapped_Ptr_Ptr);
```

Session_ID	*Session_ID* specifies a bus session.
Address_Mod	*Address_Mod* specifies the address modifier attribute of the desired memory mapping.
Byte_Ordering	*Byte_Ordering* specifies the byte ordering attribute of the desired memory mapping.
Base_Address	*Base_Address* specifies base address attribute of the desired memory mapping.
Size	*Size* specifies the size attribute of the desired memory mapping, in bytes.
Mapped_Ptr_Ptr	*Mapped_Ptr_Ptr* points to a location where a pointer to the base of the desired memory will be placed.

Visual Basic Synopsis

```
Declare Function
EpcMapBusMemory% Lib "bmvxiw16.dll"
                 (ByVal Session_ID&,
                 ByVal Address_Mod%,
                 ByVal Byte_Ordering%,
                 ByVal Base_Address&,
                 ByVal Size&,
                 Mapped_Ptr_Ptr As Any)
```

Remarks

EpcMapBusMemory creates a memory mapping with the specified attributes using statically configured bus window hardware and places a pointer to the base of the memory in the location pointed to by *Mapped_Ptr_Ptr*.

The following constants define valid values for the *Address_Mod* parameter:

Constant	Description
EPC_A16N	VMEbus A16 non-supervisory address modifier.
EPC_A16S	VMEbus A16 supervisory address modifier.
EPC_A24ND	VMEbus A24 non-supervisory data address modifier.
EPC_A24SD	VMEbus A24 supervisory data address modifier.
EPC_A24NP	VMEbus A24 non-supervisory program address modifier.
EPC_A24SP	VMEbus A24 supervisory program address modifier.
EPC_A32ND	VMEbus A32 non-supervisory data address modifier.
EPC_A32SD	VMEbus A32 supervisory data address modifier.
EPC_A32NP	VMEbus A32 non-supervisory program address modifier.
EPC_A32SP	VMEbus A32 supervisory program address modifier.

The following constants define valid values for the *Byte_Ordering* parameter:

Constant	Description
EPC_IBO	Intel (80X86) byte ordering.
EPC_MBO	Motorola (68000) byte ordering.

EPC hardware provides a number of statically configured bus windows. The table below enumerates the bus memory mapping attributes supported by an EPC's statically configured bus window hardware:

Address_Mod	*Byte_Odrering*	*Base_Address* Range	*Size* Range
EPC_A16S	EPC_MBO EPC_IBO	0x00 to 0x0000FFFF	0x00010000 to 0x01
EPC_A24SD	EPC_MBO EPC_IBO	0x00 to 0x00FFFFFF	0x01000000 to 0x01
EPC_A32SD	EPC_MBO EPC_IBO	0x00 to 0x3FFFFFFF	0x40000000 to 0x01

The *Base_Address* and *Size* parameters define a range of addresses α, where:

$$Base\_Address <= \alpha <= Base\_Address + Size - 1;$$

The function rounds the specified *Base_Address* down to the nearest 4-byte boundary. The function also limits the size of the mapping according to the specified *Base_Address* and the bus window's maximum accessible bus address.

EPC and VXLink hardware provides one or more dynamically configured bus memory windows. Use **EpcMapBusMemoryExt** to map bus memory using dynamically configured bus memory window hardware.

Return Value The function returns a Bus Management return value:

EPC_INV_ADDMOD	The specified *Address_Mod* parameter is invalid.
EPC_INV_BORDER	The specified *Byte_Ordering* parameter is invalid.
EPC_INV_PTR	The specified *Mapped_Ptr_Ptr* parameter is invalid.
EPC_INV_RANGE	The specified *Base_Address* and *Size* Parameters define a bus address range that contains invalid addresses for the specified *Address_Mod* parameter and/or this interface.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_OS_ERROR	An operating system error occurred.
EPC_OUT_OF_RSRCS	The underlying operating system currently contains insufficient resources to create the specified mapping.
EPC_SUCCESS	The function completed successfully.

See Also **EpcCopyData, EpcGetMappingAttributes, EpcMapBusMemoryExt, EpcOpenSession, EpcPopData, EpcPushData, EpcUnmapBusMemory.**

Example See **EpcCopyData**.

EpcMapBusMemoryExt

Description Creates a bus memory mapping using dynamically configured bus window hardware.

C Synopsis

```
short FAR PASCAL
EpcMapBusMemoryExt(unsigned long Session_ID,
                   unsigned short Address_Mod,
                   unsigned short Byte_Ordering,
                   unsigned long Base_Address,
                   volatile void HUGE * FAR *
        Mapped_Ptr_Ptr);
```

Session_ID	*Session_ID* specifies a bus session.
Address_Mod	*Address_Mod* specifies the address modifier attribute of the desired memory mapping.
Byte_Ordering	*Byte_Ordering* specifies the byte ordering attribute of the desired memory mapping.
Base_Address	*Base_Address* specifies base address attribute of the desired memory mapping.
Mapped_Ptr_Ptr	*Mapped_Ptr_Ptr* points to a location where a pointer to the base of the desired memory will be placed.

Visual Basic Synopsis

```
Declare Function
EpcMapBusMemoryExt% Lib "bmvxiw16.dll"
                    (ByVal Session_ID&,
                    ByVal Address_Mod%,
                    ByVal Byte_Ordering%,
                    ByVal Base_Address&,
                    ByVal Size&,
                    Mapped_Ptr_Ptr As Any)
```

Remarks

EpcMapBusMemoryExt creates a memory mapping with the specified attributes using statically configured bus window hardware and places a pointer to the base of the memory in the location pointed to by *Mapped_Ptr_Ptr*.

The following constants define valid values for the *Address_Mod* parameter:

Constant	Description
EPC_A16N	VMEbus A16 non-supervisory address modifier.
EPC_A16S	VMEbus A16 supervisory address modifier.
EPC_A24ND	VMEbus A24 non-supervisory data address modifier.
EPC_A24SD	VMEbus A24 supervisory data address modifier.
EPC_A24NP	VMEbus A24 non-supervisory program address modifier.
EPC_A24SP	VMEbus A24 supervisory program address modifier.
EPC_A32ND	VMEbus A32 non-supervisory data address modifier.
EPC_A32SD	VMEbus A32 supervisory data address modifier.
EPC_A32NP	VMEbus A32 non-supervisory program address modifier.
EPC_A32SP	VMEbus A32 supervisory program address modifier.

The following constants define valid values for the *Byte_Ordering* parameter:

Constant	Description
EPC_IBO	Intel (80X86) byte ordering.
EPC_MBO	Motorola (68000) byte ordering.

EPC and VXLink hardware provide one or more 64-Kbyte dynamically configured bus windows. The table below enumerates the bus memory mapping attributes supported by the dynamically configured bus window hardware:

Address_Mod	*Byte_Ordering*	*Base_Address* Range
EPC_A16N **EPC_A16S**	**EPC_MBO** **EPC_IBO**	0x000000000
EPC_A24ND **EPC_A24NP** **EPC_A24SD** **EPC_A24SP**	**EPC_MBO** **EPC_IBO**	0x000000000, 0x00010000, ..., 0x00FF0000
EPC_A32ND **EPC_A32NP** **EPC_A32SD** **EPC_A32SP**	**EPC_MBO** **EPC_IBO**	0x000000000, 0x00010000, ..., 0xFFFF0000

The function rounds the specified *Base_Address* down to the nearest bus window size boundary (64 Kbytes) and sets the size of the mapping to the size of the bus window (64 Kbytes). Mapping an address range larger than the bus window size requires multiple mappings. Also, mapping an address range that spans a bus window size boundary requires multiple mappings.

EPC hardware also provides a number of statically configured bus memory windows. Use **EpcMapBusMemory** to map bus memory using statically configured bus memory window hardware.

Return Value The function returns a Bus Management return value:

EPC_INV_ADDMOD	The specified *Address_Mod* parameter is invalid.
EPC_INV_BORDER	The specified *Byte_Ordering* parameter is invalid.
EPC_INV_PTR	The specified *Mapped_Ptr_Ptr* parameter is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_OS_ERROR	An operating system error occurred.
EPC_OUT_OF_RSRCS	The underlying operating system currently contains insufficient resources to create the specified mapping.
EPC_SUCCESS	The function completed successfully.

See Also **EpcCopyData, EpcGetMappingAttributes, EpcMapBusMemory, EpcOpenSession, EpcPopData, EpcPushData, EpcUnmapBusMemory.**

Example See **EpcCopyData.**

EpcMapEpcTriggers

2

Description Maps one interface trigger line to another.

C Synopsis

```
#include "busmgr.h"

short
EpcMapEpcTriggers( unsigned long  Session_ID,
                   unsigned long  In_Trigger_Mask,
                   unsigned long  Out_Trigger_Mask);
```

Session_ID	*Session_ID* specifies a session.
In_Trigger_Mask	*In_Trigger_Mask* specifies an input interface trigger line.
Out_Trigger_Mask	*Out_Trigger_Mask* specifies output interface trigger lines.

Visual Basic Synopsis

```
Declare Function
EpcMapEpcTriggers% Lib "bmvxiw16.dll"
            (ByVal Session_ID&,
            ByVal In_Trigger_Mask&,
            ByVal Out_Trigger_Mask&)
```

Remarks **EpcMapEpcTriggers** maps the interface trigger lines specified by *Out_Trigger_Mask* as outputs of the interface trigger line specified by *In_Trigger_Mask*.

The parameters *In_Trigger_Mask* is a constant specifying an input interface trigger line. The parameter *Out_Trigger_Mask* is an OR'd mask of constants specifying output interface trigger lines.

The table below enumerates valid trigger mapping combinations for an EPC-7 interface:

In_Trigger_Mask	*Out_Trigger_Mask*	Description
EPC_EXT_TRIG0	**EPC_TTL_TRIG0** ... **EPC_TTL_TRIG7**	Maps external trigger 0 as input to a single TTL trigger line.
EPC_TTL_TRIG0 ... **EPC_TTL_TRIG7**	**EPC_EXT_TRIG0**	Maps a single TTL trigger line as input to external trigger 0.

The table below enumerates valid trigger mapping combinations for a VXLink interface:

In_Trigger_Mask	*Out_Trigger_Mask*	Description
EPC_EXT_TRIG0	0x00000000	Unmaps external trigger 0.
EPC_EXT_TRIG0	**EPC_TTL_TRIG0** ... **EPC_TTL_TRIG7**	Maps external trigger 0 as input to a single TTL trigger line.
0x00000000	**EPC_EXT_TRIG1**	Unmaps external trigger 1.
EPC_TTL_TRIG0 ... **EPC_TTL_TRIG7**	**EPC_EXT_TRIG1**	Maps a single TTL trigger line as input to external trigger 0.

When an external trigger line is mapped as input to one or more interface trigger lines, asserting the external trigger line asserts all of the mapped interface trigger lines. Likewise, deasserting the external trigger line deasserts all of the mapped interface trigger lines.

When one or more interface trigger lines are mapped as input to an external trigger line, asserting one of the interface trigger lines asserts the mapped external trigger line. Likewise, deasserting one of the interface trigger lines deasserts the mapped external trigger line.

An EPC-7 interface provides a single bi-directional external trigger. The external trigger is always mapped; it cannot be unmapped. Specifying a mapping for external trigger 0 overrides the previous mapping. By default, TTL trigger 1 is mapped as an output to external trigger 0.

A VXLink interface provides two unidirectional external triggers. External trigger 0 is an input-only trigger and external trigger 1 is an output-only trigger. The external triggers can be independently mapped or unmapped. By default, both external triggers are unmapped.

Return Value The function returns a EPConnect return value:

EPC_INV_MASK	Either *In_Trigger_Mask* or *Out_Trigger_Mask* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetEpcTriggerMapping, EpcOpenSession.**

EpcMapSharedMemory

Description Creates a shared memory mapping.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcMapSharedMemory
        (unsigned long          Session_ID,
        unsigned long  FAR *    Base_Address_Ptr,
        unsigned long  FAR *    Size_Ptr,
        void HUGE *  FAR *      Mapped_Ptr_Ptr);
```

Session_ID	*Session_ID* specifies a bus session.
Base_Address_Ptr	*Base_Address_Ptr* points to a location where the base address attribute of the shared memory mapping will be placed.
Size_Ptr	*Size_Ptr* points to a location where the size attribute of the shared memory mapping, in bytes, will be placed.
Mapped_Ptr_Ptr	*Mapped_Ptr_Ptr* points to a location where a pointer to the base of the desired memory will be placed.

Visual Basic Synopsis

```
Declare Function
EpcMapSharedMemory% Lib "bmvxiw16.dll"
                        (ByVal Session_ID&,
                        Base_Address_Ptr&,
                        Size_Ptr&,
                        Mapped_Ptr_Ptr As Any)
```

Remarks **EpcMapSharedMemory** creates a shared memory mapping and places the base address attribute of the memory mapping, the size attribute of the memory mapping, and a pointer to the base of the memory in the locations pointed to by *Base_Address_Ptr*, *Size_Ptr*, and *Mapped_Ptr_Ptr*, respectively.

The values in the locations pointed to by *Base_Address_Ptr* and *Size_Ptr* define a range of addresses α, where:

**Base_Address_Ptr* <= α <= **Base_Address_Ptr* + **Size_Ptr* - 1;

The value in the location pointed to by *Base_Address_Ptr* specifies a physical local address. To determine the corresponding physical VMEbus address, the value should be added to the base address of the slave memory. Use **EpcGetSlaveMapping** to determine the base address of the slave memory.

A shared memory area is a global resource. **EpcMapSharedMemory** and **EpcUnmapSharedMemory** map and unmap the entire shared memory area. Once a session maps the shared memory area, it cannot be mapped again until the original session unmaps it.

An interface must contain dual-ported slave memory to support a shared memory area. Only the EPC-7 supports shared memory area functionality.

Return Value

The function returns a Bus Management return value:

EPC_INV_PTR	One or more of the *Base_Address_Ptr*, *Size_Ptr*, and *Mapped_Ptr_Ptr* parameters is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another bus session.
EPC_OS_ERROR	An operating system error occurred.
EPC_OUT_OF_RSRCS	The underlying operating system currently contains insufficient resources to create the specified mapping.
EPC_SUCCESS	The function completed successfully.

See Also EpcCopyData, EpcGetMappingAttributes, EpcGetSlaveMapping, EpcOpenSession, EpcUnmapSharedMemory.

Example See EpcCopyData.

EpcOpenSession

Description Creates a session.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcOpenSession( unsigned long FAR *Session_ID_Ptr);
```

Session_ID_Ptr — *Session_ID_Ptr* points to a location where a handle to the session will be placed.

Visual Basic Synopsis

```
Declare Function
EpcOpenSession% Lib "bmvxiw16.dll" (Session_ID_Ptr&)
```

Remarks **EpcOpenSession** creates a session and places a handle to the session in the location pointed to by *Session_ID_Ptr*.

By default, a newly created session does not lock shared interface hardware and has no enabled events, installed event handlers, or memory mappings.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The specified *Session_ID_Ptr* parameter is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_OUT_OF_RSRCS	The underlying operating system currently contains insufficient resources to open a session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcCloseSession, EpcLockSession, EpcMapBusMemory, EpcMapSharedMemory, EpcSetEventEnableMask, EpcSetEventHandler.**

Example See **EpcCloseSession.**

EpcPopData

Pops a block of data from a single memory location to consecutive memory locations.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcPopData( unsigned long *          Session_ID,
            void HUGE *              Source_ptr,
            void HUGE *              Dest_ptr,
            unsigned long            Size,
            unsigned short           Data_Width
            unsigned long  FAR *     Actual_Size_Ptr);
```

Session_ID — *Session_ID* specifies a bus location.

Source_Ptr — *Source_Ptr* specifies the address of a FIFO queue from which data will be popped.

Dest_Ptr — *Dest_Ptr* specifies the address of a data buffer into which data will be popped.

Size — *Size* specifies the number of data bytes to pop.

Data_Width — *Data_Width* specifies the number of data bits to pop per bus access.

Actual_Size_Ptr — *Actual_Size_Ptr* specifies a location where the actual number of bytes popped will be placed.

Remarks

EpcPopData efficiently pops blocks of data from a single memory location to consecutive memory locations using the attributes of pointes *Source_Ptr* and *Dest_Ptr*. The intended use of the function is popping large blocks of data from a FIFO queue.

The *Size* parameter shouold always express the number of bytes to be popped, regardless of the specified *Data_Width* parameter. Passing a zero *Size* parameter results in no data being popped.

Remarks

The following constants define valid values for the *Data_Width* parameter:

Constant	Description
EPC_8_BIT	8-bit data width
EPC_8_BIT_ODD	8-bit data width, odd bytes only
EPC_16_BIT	16-bit data width
EPC_32_BIT	32-bit data width
EPC_FASTCOPY	To increase pop performance, don't check for intermediate bus errors. This constant can not be used alone; it must be OR'd with one of the preceding constants.

The function returns the actual number of bytes popped in the location pointed to by *Actual_Size_Ptr*.

The function operates ocrrectly using both unmapped pointers and memory mapped pointers for Source_Ptr and Dest_Ptr. Local-to-local, local-to-VME, VME-to-local, and VME-to-VME pops all execute properly.

For a pop to complete, any *Source_Ptr* or *Dest_Ptr* that corresponds to a VMEbus addresses must be aligned on an address boundary equivalent to the specified *Data_Width*. Otherwise, the function returns an **EPC_INV_ALIGN** error. For example, if both *Source_Ptr* and *Dest_Ptr* correspond to VMEbus memory and *Data_Width* is **EPC_16_BIT**, then both *Source_Ptr* and *Dest_Ptr* must correspond to VMEbus addresses aligned on a 16-bit boundary for the pop to complete successfully.

For a 16-bit or 32-bit pop to complete under DOS or Windows, no individual data element may span a segment boundary. Otherwise, the function returns an **EPC_INV_ALIGN** error. For example, if *Data_Width* is **EPC_16_BIT** and *Size* is greater than 64 Kbytes, both *Source_Ptr* and *Dest_Ptr* must be aligned on a 16-bit boundary for the pop operation to complete successfully.

Return Value The function returns an EPConnect return value:

EPC_BERR	A bus error occurred during the pop.
EPC_INV_ALIGN	*Size* is not a multiple of *Data_Width*, *Source_Ptr* is mapped to a VMEbus address and is not aligned on a *Data_Width* boundary, *Dest_Ptr* is mapped to a VMEbus address and is not aligned on a *Data_Width* boundary, or a 16-bit or 32-bit data element spans a segment boundary or.
EPC_INV_PTR	One or more of *Source_Ptr*, *Dest_Ptr*, or *Actual_Size_Ptr* is invalid.
EPC_INV_RANGE	The address range defined by *Source_Ptr* and *Data_Width* and/or the address range defined by *Dest_Ptr* and *Size* contains bus addresses that are not currently mapped.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_INV_WIDTH	The *Data_Width* parameter is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcCopyData, EpcGetMappingAttributes, EpcMapBusMemory, EpcMapSharedMemory, EpcOpenSession, EpcPushData, EpcUnmapBusMemory, EpcUnmapSharedMemory.**

Example See **EpcCopyData.**

EpcPulseEpcLines

Description Pulses EPC control lines.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcPulseEpcLines(unsigned long *Session_ID*, unsigned long *Line_Mask*);

Session_ID	*Session_ID* specifies a session.
Line_Mask	*Line_Mask* specifies a mask of EPC control lines.

Visual Basic Synopsis

Declare Function
EpcPulseEpcLines% Lib "bmvxiw16.dll" (ByVal *Session_ID*&, ByVal *Line_Mask*&)

Remarks **EpcPulseEpcLines** pulses (asserts and deasserts as an atomic operation) the EPC control lines specified by *Line_Mask*.

Line_Mask is an OR'd mask of the following constants, where a set bit indicates that the function should pulse the corresponding interface control line:

Constant	Description
EPC_SYSFAIL	SYSFAIL.
EPC_SYSRESET	SYSRESET.

The function directly affects the interface control line state. Interface control line state reflects the state of bits in the interface's control line drive registers. Actual bus control line state is an OR'd combination of the states all devices on the bus. If the interface asserts a control line, the actual bus control line transitions from deasserted to asserted only if all other devices on the bus have previously deasserted the line. Likewise, if the interface deasserts a control line, the actual bus control line transitions from asserted to deasserted only if all devices on the bus have previously deasserted the line.

When pulsing the SYSRESET interface control line, the function leaves the line asserted for at least 200 milliseconds (in accordance with bus specifications). Whether pulsing the SYSRESET actual bus control line resets an EPC-7 or EPC-8 depends on the value of the interface's **EPC_SYSRESET_IN** miscellaneous attribute bit (see **EpcSetMiscAttributes**). (EPC-7 and EPC-8 only)

Whether pulsing the SYSFAIL interface control line pulses the SYSFAIL actual bus control line depends on the value of the interface's **EPC_SYSFAIL_OUT** miscellaneous attribute bit (see **EpcSetMiscAttributes**).

To pulse SYSFAIL on an EPC-7, EPC-8, or VXLink interface, the function deasserts then asserts the interface's **EPC_PASS** miscellaneous attribute bit (see **EpcSetMiscAttributes**). After pulsing SYSFAIL, the interface's **EPC_PASS** miscellaneous attribute remains asserted.

Return Value

The function returns a Bus Management return value:

EPC_INV_MASK	The parameter *Line_Mask* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also EpcGetBusLines, EpcGetEpcLines, EpcOpenSession, EpcSetEpcLines, EpcSetMiscAttributes.

Example See **EpcAssertInterrupt**.

EpcPulseEpcTriggers

Description Pulses interface trigger lines.

C Synopsis

```
#include "busmgr.h"

short
EpcPulseEpcTriggers(unsigned long Session_ID, unsigned long
Trigger_Mask);
```

Session_ID	*Session_ID* specifies a session.
Trigger_Mask	*Trigger_Mask* specifies a mask of interface trigger lines.

Visual Basic Synopsis

```
Declare Function
EpcPulseEpcTriggers% Lib "bmvxiw16.dll"
(ByVal Session_ID&, ByVal Trigger_Mask&)
```

Remarks **EpcPulseEpcTriggers** pulses (asserts and deasserts as an atomic operation) the interface trigger lines specified by *Trigger_Mask*.

Trigger_Mask is an OR'd mask of the following constants, where a set bit indicates that the function should pulse the corresponding interface trigger line:

Constant	Description
EPC_ECL_TRIG0	ECL trigger 0 (EPC-7 only).
EPC_ECL_TRIG1	ECL trigger 1 (EPC-7 only).
EPC_TTL_TRIG0	TTL trigger 0 (EPC-7 and VXLink only).
...	
EPC_TTL_TRIG7	TTL trigger 7 (EPC-7 and VXLink only).

The function directly affects the interface trigger line state. Interface trigger line state reflects the state of bits in the interface's trigger line drive registers. Actual bus trigger line state is an OR'd combination of the states all devices on the bus. If the interface asserts a trigger line, the actual bus trigger line transitions from deasserted to asserted only if all other devices on the bus have previously deasserted the line. Likewise, if the interface deasserts a trigger line, the actual bus trigger line transitions from asserted to deasserted only if all devices on the bus have previously deasserted the line.

Return Value

The function returns a EPConnect return value:

EPC_INV_MASK	The parameter *Trigger_Mask* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also

EpcGetBusTriggers, **EpcGetEpcTriggers**, **EpcOpenSession**, **EpcSetEpcTriggers**.

EpcPushData

2

Pushes a block of data from consecutive memory locations to a single memory location.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcPushData( unsigned long          Session_ID,
             void HUGE *            Source_Ptr,
             void HUGE *            Dest_Ptr,
             unsigned long          Size,
             unsigned short         Data_Width,
             unsigned long FAR *    Actual_Size_Ptr);
```

Session_ID	*Session_ID* specifies a bus session.
Source_Ptr	*Source_Ptr* specifies the address of a data buffer from which data will be pushed.
Dest_Ptr	*Dest_Ptr* specifies the address of a FIFO queue to which data will be pushed.
Size	*Size* specifies the number of data bytes to push.
Data_Width	*Data_Width* specifies the number of data bits to push per bus access.
Actual_Size_Ptr	*Actual_Size_Ptr* specifies a location where the actual number of bytes pushed will be placed.

Remarks

EpcPushData efficiently pushes blocks of data from consecutive memory locations to a single memory location using the attributes of pointers *Source_Ptr* and *Dest_Ptr*. The intended use of the function is pushing large blocks of data to a FIFO queue.

The *Size* parameter should always express the number of bytes to be pushed, regardless of the specified *Data_Width* parameter. Passing a zero *Size* parameter results in no data being pushed.

The following constants define valid values for the *Data_Width* parameter:

Constant	Description
EPC_8_BIT	8-bit data width
EPC_8_BIT_ODD	8-bit data width, odd bytes only
EPC_16_BIT	16-bit data width
EPC_32_BIT	32-bit data width
EPC_FASTCOPY	To increase push performance, don't check for intermediate bus errors. This constant cannot be used alone; it must be OR'd with one of the preceding constants.

The function returns the actual number of bytes pushed in the location pointed to by *Actual_Size_Ptr*.

The function operates correctly using both unmapped pointers and memory mapped pointers for *Source_Ptr* and *Dest_Ptr*. local-to-local, local-to-VME, VME-to-local, and VME-to-VME pushes all execute properly.

For a push to complete, the specified *Size* must be aligned on a boundary equivalent to the specified *Data_Width*. Otherwise, the function returns an **EPC_INV_ALIGN** error. For example, if *Data_Width* is **EPC_16_BIT**, then *Size* must be a multiple of two for the push to complete successfully. If *Data_Width* is **EPC_32_BIT**, then *Size* must be a multiple of four for the push to complete successfully.

For a push to complete, any *Source_Ptr* or *Dest_Ptr* that corresponds to a VMEbus addresses must be aligned on an address boundary equivalent to the specified *Data_Width*. Otherwise, the function returns an **EPC_INV_ALIGN** error. For example, if both *Source_Ptr* and *Dest_Ptr* correspond to VMEbus memory and *Data_Width* is **EPC_16_BIT**, then both *Source_Ptr* and *Dest_Ptr* must correspond to VMEbus addresses aligned on a 16-bit boundary for the push to complete successfully.

For a 16-bit or 32-bit push to complete under DOS or Windows, no individual data element may span a segment boundary. Otherwise, the function returns an **EPC_INV_ALIGN** error. For example, if *Data_Width* is **EPC_16_BIT** and *Size* is greater than 64 Kbytes, both *Source_Ptr* and *Dest_Ptr* must be aligned on a 16-bit boundary for the push operation to complete successfully.

Return Value

The function returns a EPConnect return value:

EPC_BERR	A bus error occurred during the push.
EPC_INV_ALIGN	*Size* is not a multiple of *Data_Width*, *Source_Ptr* is mapped to a VMEbus address and is not aligned on a *Data_Width* boundary, *Dest_Ptr* is mapped to a VMEbus address and is not aligned on a *Data_Width* boundary, or a 16-bit or 32-bit data element spans a segment boundary.
EPC_INV_PTR	One or more of *Source_Ptr*, *Dest_Ptr*, or *Actual_Size_Ptr* is invalid.
EPC_INV_RANGE	The address range defined by *Source_Ptr* and *Size* and/or the address range defined by *Dest_Ptr* and *Data_Width* contains bus addresses that are not currently mapped.
EPC_INV_SESSION	The specified *Session_ID* is invalid.

EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_INV_WIDTH	The *Data_Width* parameter is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcCopyData, EpcGetMappingAttributes, EpcMapBusMemory, EpcMapSharedMemory, EpcOpenSession, EpcPopData, EpcUnmapBusMemory, EpcUnmapSharedMemory.**

Example See **EpcCopyData.**

EpcSetBusAttributes

Description Defines the interface's bus management attributes.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSetBusAttributes( unsigned long  Session_ID,
                     unsigned short Bus_Enable,
                     unsigned short Bus_Arb_Mode,
                     unsigned short Bus_Arb_Priority,
                     unsigned short Bus_Release);
```

Session_ID	*Session_ID* specifies a session.
Bus_Enable	*Bus_Enable* specifies the interface's bus enable attribute.
Bus_Arb_Mode	*Bus_Arb_Mode* specifies the interface's bus arbitration mode.
Bus_Arb_Priority	*Bus_Arb_Priority* specifies the interface's bus arbitration priority.
Bus_Release	*Bus_Release* specifies the interface's bus release mode.

Visual Basic Synopsis

```
Declare Function
EpcSetBusAttributes% Lib "bmvxiw16.dll"
                     (ByVal Session_ID&,
                     ByVal Bus_Enable%,
                     ByVal Bus_Arb_Mode%,
                     ByVal Bus_Arb_Priority%,
                     ByVal Bus_Release%)
```

Remarks

EpcSetBusAttributes defines the interface's bus management attributes.

Bus_Enable specifies the interface's bus enable attribute. The interface's bus enable attribute determines whether accesses made by the interface reach the bus. Valid *Bus_Enable* values are:

Bus_Enable	Description
EPC_DISABLE_BUS	Disable bus accesses for the interface (Supported on EPC-7 and EPC-8 only).
EPC_ENABLE_BUS	Enable bus accesses for the interface.

Bus_Arb_Mode specifies the interface's bus arbitration mode. The interface's bus arbitration mode defines how the interface arbitrates bus collisions. The interface's bus arbitration mode only affects bus accesses if the interface has been designated the VMEbus slot-1 controller or VXIbus slot-0 controller. Valid *Bus_Arb_Mode* values are:

Bus_Arb_Mode	Description
EPC_PRIORITY	Priority bus arbitration.
EPC_ROUND_ROBIN	Round-robin bus arbitration.

Bus_Arb_Priority specifies the interface's bus arbitration priority. The interface's bus arbitration priority defines the priority level at which the interface arbitrates for the bus. Possible values placed at *Bus_Arb_Priority* are:

Bus_Arb_Priority	Description
EPC_PRIORITY0	Bus arbitration priority 0.
EPC_PRIORITY1	Bus arbitration priority 1.
EPC_PRIORITY2	Bus arbitration priority 2.
EPC_PRIORITY3	Bus arbitration priority 3.

Bus_Release specifies the interface's bus release mode. The interface's bus release mode determines when the interface requests and/or releases the bus. Valid *Bus_Release* values are:

Bus_Release	Description
EPC_ROR	"Release On Request" bus release mode.
EPC_RONR	"Request On No Request" bus release mode.

Return Value The function returns a Bus Management return value:

EPC_INV_ARB_MODE	The parameter *Bus_Arb_Mode* is invalid.
EPC_INV_ARB_PRIO	The parameter *Bus_Arb_Priority* is invalid.
EPC_INV_ENABLE	The parameter *Bus_Enable* is invalid.
EPC_INV_RELEASE	The parameter *Bus_Release* is invalid.
EPC_INV_SESSION	The parameter *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_INV_TIMEOUT	An invalid timeout was encountered.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetBusAttributes**, **EpcOpenSession**.

Example See **EpcGetBusAttributes**.

EpcSetEpcLines

Description Defines the interface control line state.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcSetEpcLines(unsigned long *Session_ID*, unsigned long *Line_Mask*);

Session_ID	*Session_ID* specifies a session.
Line_Mask	*Line_Mask* specifies an interface control line state.

Visual Basic Synopsis

Declare Function
EpcSetEpcLines% Lib "bmvxiw16.dll" (ByVal *Session_ID&*, ByVal *Line_Mask&*)

Remarks **EpcSetEpcLines** defines the interface control line state as specified by *Line_Mask*.

Line_Mask is either zero or an OR'd bit mask of the following constants. A set bit indicates that the function should assert the corresponding interface control line. A clear bit indicates that the function should deassert the corresponding interface control line:

Constant	Description
EPC_SYSFAIL	SYSFAIL.
EPC_SYSRESET	SYSRESET.

The function directly affects the interface control line state. Interface control line state reflects the state of bits in the interface's control line drive registers. Actual bus control line state is an OR'd combination of the states all devices on the bus. If the interface asserts a control line, the actual bus control line transitions from deasserted to asserted only if all other devices on the bus have previously deasserted the line. Likewise, if the interface deasserts a control line, the actual bus control line transitions from asserted to deasserted only if all devices on the bus have previously deasserted the line.

Whether asserting the SYSRESET actual bus control line resets the interface on an EPC-7 or EPC-8 depends on the value of the interface's **EPC_SYSRESET_IN** miscellaneous attribute bit (see **EpcSetMiscAttributes**).

Whether asserting or deasserting the SYSFAIL interface control line asserts or deasserts the SYSFAIL actual bus control line depends on the value of the interface's **EPC_SYSFAIL_OUT** miscellaneous attribute bit (see **EpcSetMiscAttributes**).

To assert or deassert SYSFAIL on an EPC-7, EPC-8, or VXLink interface, the function deasserts or asserts the interface's **EPC_PASS** miscellaneous attribute bit (see **EpcSetMiscAttributes**). After asserting SYSFAIL, the interface's **EPC_PASS** miscellaneous attribute remains deasserted. Likewise, after deasserting SYSFAIL, the interface's **EPC_PASS** miscellaneous attribute remains asserted.

Return Value

The function returns a Bus Management return value:

EPC_INV_MASK	The parameter *Line_Mask* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also EpcGetBusLines, EpcGetEpcLines, EpcOpenSession, EpcPulseEpcLines, EpcSetMiscAttributes.

Example See EpcAssertInterrupt.

EpcSetEpcMODID

Description Defines interface MODID line state.

C Synopsis

```
#include "busmgr.h"

short
EpcSetEpcMODID(unsigned long Session_ID, unsigned long
MODID_Mask);
```

Session_ID	*Session_ID* specifies a session.
MODID_Mask	*MODID_Mask* specifies an interface MODID line state.

Visual Basic Synopsis

```
Declare Function
EpcSetEpcMODID% Lib "bmvxiw16.dll"
(ByVal Session_ID&, ByVal MODID_Mask&)
```

Remarks **EpcSetEpcMODID** defines the interface MODID line state as specified by *MODID_Mask*.

MODID_Mask is either zero or an OR'd bit mask of the following constants. A set bit indicates that the function should assert the corresponding interface MODID line. A clear bit indicates that the function should deassert the corresponding interface MODID line:

Constant	Description
EPC_MODID0	MODID line 0 (EPC-7 and VXLink only).
...	
EPC_MODID12	MODID line 12 (EPC-7 and VXLink only).

Only the VXIbus slot-0 controller device can assert or deassert the actual bus MODID lines. When an interface is the VXIbus slot-0 controller, defining the interface MODID line state also defines the actual bus MODID line state. When an interface is not the VXIbus slot-0 controller, defining the interface MODID line state has no effect on the actual bus MODID lines.

Return Value The function returns an EPConnect return value:

EPC_INV_MASK	The parameter *MODID_Mask* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetBusMODID, EpcOpenSession.**

EpcSetEpcTriggers

Description Defines the interface trigger line state.

C Synopsis

```
#include "busmgr.h"

short
EpcSetEpcTriggers(unsigned long Session_ID, unsigned long
Trigger_Mask);
```

Session_ID	*Session_ID* specifies a session.
Trigger_Mask	*Trigger_Mask* specifies an interface bus control line state.

Visual Basic Synopsis

```
Declare Function

EpcSetEpcTriggers% Lib "bmvxiw16.dll" (ByVal Session_ID&,
ByVal Trigger_Mask&)
```

Remarks **EpcSetEpcTriggers** defines the interface trigger line state as specified by *Trigger_Mask*.

Trigger_Mask is either zero or an OR'd bit mask of the following constants. A set bit indicates that the function should assert the corresponding interface trigger line.

A clear bit indicates that the function should deassert the corresponding interface trigger line:

Constant	Description
EPC_ECL_TRIG0	ECL trigger 0 (EPC-7 only).
EPC_ECL_TRIG1	ECL trigger 1 (EPC-7 only).
EPC_TTL_TRIG0	TTL trigger 0 (EPC-7 and VXLink only).
...	
EPC_TTL_TRIG7	TTL trigger 7 (EPC-7 and VXLink only).

The function directly affects the interface trigger line state. interface trigger line state reflects the state of bits in the interface's trigger line drive registers. Actual bus trigger line state is an OR'd combination of the states all devices on the bus. If the interface asserts a trigger line, the actual bus trigger line transitions from deasserted to asserted only if all other devices on the bus have previously deasserted the line. Likewise, if the interface deasserts a trigger line, the actual bus control line transitions from asserted to deasserted only if all devices on the bus have previously deasserted the line.

Return Value

The function returns a EPConnect return value:

EPC_INV_MASK	The parameter *Trigger_Mask* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also

EpcGetBusTriggers, **EpcGetEpcTriggers**, **EpcOpenSession**, **EpcPulseEpcTriggers**.

EpcSetEventEnableMask

Description Defines a session's enabled event mask attribute.

C Synopsis

#include "busmgr.h"

short
EpcSetEventEnableMask(unsigned long *Session_ID*, unsigned long *Event_Mask*);

Session_ID	*Session_ID* specifies a session.
Event_Mask	*Event_Mask* specifies a mask of enabled events.

Visual Basic Synopsis

Declare Function
EpcSetEventEnableMask% Lib "bmvxiw16.dll" (ByVal *Session_ID&*, ByVal *Event_Mask&*)

Remarks **EpcSetEventEnableMask** sets the specified session's enabled event mask attribute to *Event_Mask*.

The *Event_Mask* parameter is a bit mask where each bit corresponds to an event. The *Event_Mask* parameter should be either zero or an OR'd combination of the following constants:

Event	Description
EPC_MSG_INT	Message interrupt (EPC-7 and EPC-8 only)
EPC_VME1_INT	VMEbus interrupt 1
...	
EPC_VME7_INT	VMEbus interrupt 7
EPC_SIGNAL_INT	VXIbus signal FIFO interrupt
EPC_TTL_TRIG0_INT	VXIbus TTL Trigger 0 interrupt (EPC-7 only)
...	
EPC_TTL_TRIG7_INT	VXIbus TTL Trigger 7 interrupt (EPC-7 only)
EPC_SYSRESET_ERR	VMEbus SYSRESET error
EPC_ACFAIL_ERR	VMEbus power failure error
EPC_BERR_ERR	VMEbus access error
EPC_SYSFAIL_ERR	VMEbus SYSFAIL error
EPC_WATCHDOG_ERR	Watchdog timer expiration error (EPC-7 and EPC-8 only)
EPC_EXT_TRIG0_INT	External trigger 0 interrupt (VXLink only)
EPC_EXT_TRIG1_INT	External trigger 1 interrupt (VXLink only)

Return Value The function returns a Bus Management return value:

EPC_INV_MASK	*Event_Mask* contains enabled events that are not valid for this interface.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetEventEnableMask, EpcOpenSession.**

Example See **EpcGetEventEnableMask.**

EpcSetEventHandler

Description Defines an entry in a session's event handler array.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSetEventHandler( unsigned long  Session_ID,
                    unsigned long  Event_Mask,
                    void (FAR *    Event_Handler)
                          (unsigned long,
                           unsigned long,
                           unsigned long),
                    void FAR *     Stack_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Event_Mask	*Event_Mask* specifies an event.
Event_Handler	*Event_Handler* specifies an event handler.
Stack_Ptr	*Stack_Ptr* specifies an event handler stack pointer.

Visual Basic Synopsis

```
Declare Function
EpcSetEventHandler% Lib "bmvxiw16.dll"
                    (ByVal Session_ID&,
                    ByVal Event_Mask&,
                    Event_Handler As Any,
                    Stack_Ptr As Any);
```

Remarks **EpcSetEventHandler** sets the specified session's specified event handler array entry to *Event_Handler* and the event handler's stack pointer to *Stack_Ptr*.

The *Event_Mask* parameter is a bit mask where each bit corresponds to an event. The *Event_Mask* parameter should be one of the following constants:

Event	Description
EPC_MSG_INT	Message interrupt (EPC-7 and EPC-8 only)
EPC_VME1_INT	VMEbus interrupt 1
...	
EPC_VME7_INT	VMEbus interrupt 7
EPC_SIGNAL_INT	VXIbus signal FIFO interrupt
EPC_TTL_TRIG0_INT	VXIbus TTL Trigger 0 interrupt (EPC-7 only)
...	
EPC_TTL_TRIG7_INT	VXIbus TTL Trigger 7 interrupt (EPC-7 only)
EPC_SYSRESET_ERR	VMEbus SYSRESET error
EPC_ACFAIL_ERR	VMEbus power failure error
EPC_BERR_ERR	VMEbus access error
EPC_SYSFAIL_ERR	VMEbus SYSFAIL error
EPC_EXT_TRIG0_INT	External trigger 0 interrupt (VXLink only)
EPC_EXT_TRIG1_INT	External trigger 1 interrupt (VXLink only)
EPC_WATCHDOG_ERR	Watchdog timer expiration error (EPC-7 and EPC-8 only)

The *Event_Handler* parameter is a pointer to an event handler function with the following call semantics:

```
void FAR
EventHandlerFunction(unsigned long Session_ID,
            unsigned long Handler_Mask,
            unsigned long Handler_Data);
```

The event handler function should return to the caller using a normal RET instruction. It should not attempt to return using an IRET instruction.

The value passed in the event handler function's *Session_ID* parameter specifies the session that received the event. The value passed in the event handler function's *Handler_Mask* parameter specifies the event that caused execution of the event handler function.

Whether or not the event handler function receives a meaningful *Handler_Data* parameter depends on the value of *Handler_Mask*:

Handler_Mask	*Handler_Data*
EPC_MSG_INT	0
EPC_VME1_INT ... **EPC_VME7_INT**	VMEbus interrupt status/id (zero-extended to 32 bits)
EPC_SIGNAL_INT	VXIbus signal data (zero extended to 32 bits)
EPC_TTL_TRIG0_INT ... **EPC_TTL_TRIG7_INT**	0
EPC_ACFAIL_ERR	0
EPC_BERR_ERR	0
EPC_SYSFAIL_ERR	0
EPC_WATCHDOG_ERR	0
EPC_SYSRESET_ERR	0
EPC_EXT_TRIG0_INT	0
EPC_EXT_TRIG1_INT	0

The *Stack_Ptr* parameter is a pointer to the bottom of a block of memory reserved for use as a stack.

Defining a NULL event handler and/or event handler stack pointer effectively removes any previously assigned event handler and event handler stack pointer.

Defining an event handler and an event handler stack pointer does not enable or disable reception of the corresponding event. A separate call to **EpcSetEventEnableMask** is required.

Bus Management for Windows calls an event handler exactly once for each occurrence of its corresponding event and disables virtual processor interrupts before an event handler is called. The table below describes the algorithm used by EPConnect/VXI in processing each event type:

Event	Algorithm
EPC_MSG_INT	For each session with the event enabled and a handler installed, the IRQ handler disables the event and calls the installed event handler. To receive additional message interrupt events, a session must re-enable the event. To avoid redundant message interrupt events, a session should only re-enable the event after receiving a word serial command (using **EpcSrvReceiveWSCommand**) or sending a word serial command response (using **EpcSrvSendWSResponse**).
EPC_VME1_INT ... **EPC_VME7_INT**	The IRQ handler acknowledges the VMEbus interrupt and gets the status/id data. For each session with the event enabled and a handler installed, the IRQ handler calls the installed event handler. Additional events occur whenever additional VMEbus interrupts are asserted on the bus.

EPC_SIGNAL_INT	The IRQ handler gets the signal data from the signal FIFO. For each session with the event enabled and a handler installed, the IRQ handler calls the installed event handler Additional events occur whenever a device writes to the interface's signal FIFO.
EPC_TTL_TRIG0_INT ... **EPC_TTL_TRIG7_INT**	For each session with the event enabled and a handler installed, the IRQ handler disables the event and calls the installed event handler. To receive additional TTL trigger interrupt events for a specific TTL trigger, a session must re-enable the event. To ensure that a previous TTL trigger assertion does not cause redundant events, a session should wait for the deassertion of the corresponding TTL trigger latch bit (using **EpcGetMiscAttributes**) before re-enabling a TTL trigger interrupt event.

EPC_ACFAIL_ERR	For each session with the event enabled and a handler installed, the IRQ handler disables the event and calls the installed event handler. To receive additional ACFAIL error events, a session must re-enable the event. To ensure that the previous ACFAIL assertion does not cause redundant events, a session should wait for the deassertion of the ACFAIL bus control line (using **EpcGetBusLines**) before re-enabling the ACFAIL error event.
EPC_BERR_ERR	The IRQ handler clears the BERR condition. For each session with the event enabled and a handler installed, the IRQ handler calls the installed event handler. Additional BERR error events occur whenever the interface makes a bus access that terminates with a BERR condition.

EPC_SYSFAIL_ERR	For each session with the event enabled and a handler installed, the IRQ handler disables the event and calls the installed event handler. To receive additional SYSFAIL error events, a session must re-enable the event. To ensure that the previous SYSFAIL assertion does not cause a redundant event, a session should wait for the deassertion of the SYSFAIL bus control line (using **EpcGetBusLines**) before re-enabling the SYSFAIL error event.
EPC_WATCHDOG_ERR	For each session with the event enabled and a handler installed, the IRQ handler disables the event and calls the installed event handler. To receive additional watchdog timer error events, a session must re-enable the watchdog timer error event. To ensure that the previous watchdog timer expiration does not cause redundant events, a session should reset the watchdog timer (using **EpcWatchdogTimer**) before re-enabling the watchdog timer error event.

EPC_SYSRESET_ERR	The IRQ handler re-initializes the hardware interface. For each session with the event enabled and a handler installed, the IRQ handler disables the event and calls the installed event handler. To receive additional SYSRESET error events, a session must re-enable the event. To ensure that the previous SYSRESET assertion does not cause redundant events, a session should wait for the deassertion of the SYSRESET bus control line (using **EpcGetBusLines**) before re-enabling the SYSRESET error event.
EPC_EXT_TRIG0_INT EPC_EXT_TRIG1_INT	For each session with the event enabled and a handler installed, the IRQ handler disables the event and calls the installed event handler. Additional events occur whenever additional external trigger events are detected.

Return Value The function returns a Bus Management return value:

EPC_INV_MASK	*Event_Mask* contains more than one event or contains an event that is not valid for this EPC.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetBusLines, EpcGetEventHandler, EpcGetMiscAttributes, EpcOpenSession, EpcSetEventEnableMask, EpcSrvReceiveWSCommand, EpcSrvSendWSResponse, EpcWatchdogTimer.**

Example See **EpcGetEventEnableMask.**

EpcSetLockingTimeout

Description Defines a session's locking timeout.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSetLockingTimeout(unsigned long Session_ID, unsigned long
Timeout);
```

Session_ID	*Session_ID* specifies a session.
Timeout	*Timeout* specifies a locking timeout.

Visual Basic Synopsis

```
Declare Function
EpcSetLockingTimeout% Lib "bmvxiw16.dll" (ByVal
Session_ID&, ByVal Timeout&)
```

Remarks **EpcSetLockingTimeout** defines the specified session's locking timeout.

Timeout specifies the session's locking timeout, in milliseconds.

By default, a session has a locking timeout of zero milliseconds. When the session encounters a locking conflict, an **EPC_LOCKED** error is returned immediately.

Return Value The function returns a Bus Management return value:

EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetLockingTimeout**, **EpcLockSession**, **EpcOpenSession**.

Example See **EpcGetLockingTimeout**.

EpcSetMiscAttributes

Description Defines the interface's miscellaneous configuration attributes.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcSetMiscAttributes(unsigned long *Session_ID*, unsigned long *Misc_Mask*);

Session_ID	*Session_ID* specifies a session.
Misc_Mask	*Misc_Mask* specifies miscellaneous interface configuration attributes.

Visual Basic Synopsis

Declare Function
EpcSetMiscAttributes% Lib "bmvxiw16.dll" (ByVal *Session_ID&*, ByVal *Misc_Mask&*)

Remarks **EpcSetMiscAttributes** defines miscellaneous interface configuration attributes.

Misc_Mask is either zero or an OR'd bit mask of the following constants, where a set bit indicates that the function should assert the corresponding miscellaneous interface attribute bit. A clear bit indicates that the function should deassert the corresponding miscellaneous interface attribute bit:

Constant	Description
EPC_DIR	Word serial byte transfer protocol DIR bit. Asserting the bit indicates that the interface is ready to receive data from its commander device. Supported on EPC-7 and EPC-8 only.

EPC_DOR	Word serial byte transfer protocol DOR bit. Asserting the bit indicates that the interface is ready to send data to its commander device. Supported on EPC-7 and EPC-8 only.
EPC_ERR	Word serial protocol ERR* bit. Asserting the bit indicates to the commander device that the interface has detected a word serial protocol error. Supported on EPC-7 and EPC-8 only.
EPC_LOCK	Message-based device Locked* bit. Asserting the bit indicates that the commander device has locked access to the interface from other local sources. Supported on EPC-7 and EPC-8 only.
EPC_MULTIPLE_LOCK	Word serial protocol extension multiple commander lock bit. When asserted, the first commander to read the asserted bit from the interface's Response register can safely send a word serial command. Supported on EPC-7 and EPC-8 only.
EPC_PASS	Device initialization PASSED bit. Asserting the bit indicates that the interface has passed self-test.
EPC_READY	Device initialization READY bit. Asserting the bit indicates that the interface is ready to begin normal operation.
EPC_RESET	Interface reset bit. Asserting the bit places the interface in VXI "soft reset" state.

EPC_RRDY	Word serial protocol Read Ready bit. Asserting the bit indicates to a commander device that the interface has a word serial response in its message register. Supported on EPC-7 and EPC-8 only.
EPC_RSRC_MGR	Interface resource manager execution bit. Asserting the bit indicates that resource manager execution is complete.
EPC_STICKY_BERR	"Sticky" bus error bit. When asserted, the bit indicates that a bus error has occurred since the bit was last deasserted. This bit cannot be asserted directly by software; it can only be deasserted.
EPC_SYSFAIL_OUT	SYSFAIL output enable bit. When asserted, the interface can assert SYSFAIL. When deasserted, the interface cannot assert SYSFAIL.
EPC_SYSRESET_IN	SYSRESET input enable bit. When asserted, asserting SYSRESET resets the interface. When deasserted, asserting SYSRESET does not reset the interface.
EPC_WRDY	Word serial protocol Write Ready bit. Asserting the bit indicates to a commander device that the interface is ready to receive a word serial command. Supported on EPC-7 and EPC-8 only.

Deasserting **EPC_PASS** while asserting **EPC_SYSFAIL_OUT** causes the interface to assert SYSFAIL on the bus.

Return Value The function returns a Bus Management return value:

EPC_INV_MASK	The parameter *Misc_Mask* is invalid.
EPC_INV_SESSION	The parameter *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_INV_TIMEOUT	An invalid timeout was encountered.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetMiscAttributes**, **EpcOpenSession**.

Example See **EpcGetBusAttributes**.

EpcSetSessionData

Description Defines a session's application-specified data.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSetSessionData( unsigned long Session_ID,
                   unsigned long Session_Data);
```

Session_ID	*Session_ID* specifies an open session.
Session_Data	*Session_Data* specifies the session's application-specified data.

Visual Basic Synopsis

```
Declare Function
EpcSetSessionData% Lib "bmvxiw16.dll" (ByVal Session_ID&,
ByVal Session_Data&)
```

Remarks **EpcSetSessionData** defines the specified session's application-specified data.

The application-specified data is a 4-byte quantity.

Typically, an application uses **EpcSetSessionData** to store a pointer to one of its data structures. Later, the application uses **EpcGetSessionData** to quickly retrieve the pointer during performance-critical operations (like event handling).

Return Value The function returns a Bus Management return value:

EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcGetSessionData, EpcOpenSession.**

Example See **EpcCloseSession.**

EpcSetSlaveMapping

Description Defines the interface's slave memory mapping.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSetSlaveMapping( unsigned long  Session_ID,
                    unsigned short Address_Space,
                    unsigned long  Base_Address);
```

Session_ID	*Session_ID* specifies a session.
Address_Space	*Address_Space* specifies a slave memory address space.
Base_Address	*Base_Address* specifies a slave memory base address.

Visual Basic Synopsis

```
Declare Function
EpcSetSlaveMapping% Lib "bmvxiw16.dll" (ByVal Session_ID&,
                    ByVal Address_Space%,
                    ByVal Base_Address&)
```

Remarks **EpcSetSlaveMapping** defines the mapping of the interface's slave memory to the bus.

Address_Space specifies whether the interface's slave memory appears on the bus, and if so, in which address space. *Base_Address* specifies the base address of the interface's slave memory in the given *Address_Space*.

Valid combinations of *Address_Space* and *Base_Address* are dependent on the interface type:

Interface Type	*Address_Space*	*Base_Address*
EPC-7	**EPC_DISABLED**	N/A
	EPC_A24	0x00000000, 0x00400000, ..., 0x00C00000
	EPC_A32	0x00000000, 0x01000000, ..., 0xFF000000
EPC-8	**EPC_DISABLED**	N/A
VXLink	**EPC_DISABLED**	N/A

A24 base addresses are aligned on a 4 Mbyte boundary, and only the first 4 Mbytes of the interface's slave memory is mapped to the bus. A32 base addresses are aligned on a 16 Mbyte boundary, and only the first 16 Mbytes of the interface's slave memory is mapped to the bus.

Return Value

The function returns a Bus Management return value:

EPC_INV_BASE	The parameter *Base_Address* is invalid.
EPC_INV_SESSION	The parameter *Session_ID* is invalid.
EPC_INV_SPACE	The parameter *Address_Space* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_INV_TIMEOUT	An invalid timeout was encountered.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also

EpcGetSlaveMapping, EpcOpenSession.

Example

See **EpcGetBusAttributes.**

EpcSetULA

Description Defines the interface's unique logical address.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSetULA(unsigned long Session_ID, unsigned short ULA);
```

Session_ID	*Session_ID* specifies a session.
ULA	*ULA* specifies the interface's unique logical address.

Visual Basic Synopsis

```
Declare Function
EpcSetULA% Lib "bmvxiw16.dll" (ByVal Session_ID&, ByVal
ULA%)
```

Remarks **EpcSetULA** defines the interface's unique logical address.

Valid unique logical address values are 0x00 through 0xFF.

Return Value The function returns a Bus Management return value:

EPC_INV_SESSION	The parameter *Session_ID* is invalid.
EPC_INV_TIMEOUT	An invalid timeout was encountered.
EPC_INV_ULA	The parameter *ULA* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also EpcGetULA, EpcOpenSession.

Example See EpcGetBusAttributes.

EpcSrvEnableWSCommand

Description Enables word serial command reception.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSrvEnableWSCommand
        (unsigned long Session_ID,
        unsigned short Enable_Next_Command);
```

Session_ID	*Session_ID* specifies a session.
Enable_Next_Command	*Enable_Next_Command* specifies the type of word serial command reception to enable.

Visual Basic Synopsis

```
Declare Function
EpcSrvEnableWSCommand% Lib "bmvxiw16.dll" (ByVal
Session_ID&, ByVal Enable_Next_Command%)
```

Remarks **EpcSrvEnableWSCommand** configures the interface hardware to receive a word serial command.

The following constants specify valid values for the *Enable_Next_Command* parameter:

Constant	Description
EPC_DISABLE_ALL	Disable word serial command reception.
EPC_ENABLE_WRDY	Enable word serial command reception by asserting WRDY and deasserting both DIR and DOR.
EPC_ENABLE_DIR	Enable word serial command reception and data input by asserting both WRDY and DIR and deasserting DOR (EPC-7 and EPC-8 only).
EPC_ENABLE_DOR	Enable word serial command reception and data output by asserting both WRDY and DOR and deasserting DIR (EPC-7 and EPC-8 only).
EPC_ENABLE_ALL	Enable word serial command reception, data input, and data output by asserting WRDY, DIR, and DOR (EPC-7 and EPC-8 only).

Disabling word serial command reception when it is already enabled and without receiving a word serial command can result in a word serial protocol violation by allowing the commander device to write an unexpected word serial command.

On an EPC-7, enabling word serial command reception when an outgoing word serial command response remains unread in the interface's message registers can result in a word serial protocol violation by allowing the commander device to write over the word serial command response.

Enabling word serial command reception when it is already enabled can result in a word serial protocol violation. In particular, enabling word serial command reception with DIR deasserted when word serial command reception is already enabled with DIR asserted can generate a DIR violation. Likewise, enabling word serial command reception with DOR deasserted when word serial command reception is already enabled with DOR asserted can generate a DOR violation.

EPConnect does not support enabling word serial command reception on a VXLink interface. Attempting to use the function on a VXLink interface results in an **EPC_INV_HW** error.

Return Value

The function returns a Bus Management return value:

EPC_INV_ENABLE	The parameter *Enable_Next_Command* is invalid.
EPC_INV_HW	The interface does not support enabling word serial command reception.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also

EpcOpenSession, **EpcSrvReceiveWSCommand**.

EpcSrvReceiveWSCommand

Description Receives a word serial command from a commander device.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSrvReceiveWSCommand
        (unsigned long          Session_ID,
        void FAR *              Command_Ptr,
        unsigned short FAR *    Command_Size_Ptr,
        unsigned short          Enable_Next_Command,
        unsigned long           Timeout);
```

Session_ID	*Session_ID* specifies a session.
Command_Ptr	*Command_Ptr* specifies a location where the word serial command will be placed.
Command_Size_Ptr	*Command_Size_Ptr* specifies a location where the size of the word serial command will be placed.
Enable_Next_Command	*Enable_Next_Command* specifies whether to enable the interface hardware to receive the another word serial command.
Timeout	*Timeout* specifies the number of milliseconds to wait for a word serial command.

Visual Basic Synopsis

```
Declare Function
EpcSrvReceiveWSCommand% Lib "bmvxiw16.dll"
                        (ByVal Session_ID&,
                        Command_Ptr As Any,
                        Command_Size_Ptr%,
                        ByVal Enable_Next_Command%,
                        ByVal Timeout&)
```

Remarks

EpcSrvReceiveWSCommand receives a word serial command and places the command and its size in the locations pointed to by *Command_Ptr* and *Command_Size_Ptr*, respectively. The function then configures the interface hardware for future word serial command reception.

Command_Size_Ptr points to a location where the function places the size of the received word serial command:

Command_Size_Ptr	**Description**
EPC_16_BIT	Received a 16-bit word serial command.
EPC_32_BIT	Received a 32-bit long word serial command (EPC-7 only).

The following constants specify valid values for the *Enable_Next_Command* parameter:

Constant	Description
EPC_DISABLE_ALL	Disable word serial command reception.
EPC_ENABLE_WRDY	Enable word serial command reception by asserting WRDY and deasserting both DIR and DOR.
EPC_ENABLE_DIR	Enable word serial command reception and data input by asserting both WRDY and DIR and deasserting DOR.
EPC_ENABLE_DOR	Enable word serial command reception and data output by asserting both WRDY and DOR and deasserting DIR .
EPC_ENABLE_ALL	Enable word serial command reception, data input, and data output by asserting WRDY, DIR, and DOR.

Word serial command reception must be enabled before attempting to receive a word serial command. Otherwise, **EpcSrvReceiveWSCommand** returns invalid word serial command data. Use **EpcSrvEnableWSCommand** to enable initial word serial command reception.

Occasionally, it's useful to receive a word serial command without destroying the contents of the interface's message registers. To receive a word serial command without destroying the contents of the interface's message registers, use **EpcSrvReceiveWSCommand** with an *Enable_Next_Command* parameter value of **EPC_DISABLE_ALL**. This allows a subsequent call to **EpcSrvReceiveWSCommand** to receive the same word serial command. Note that either enabling word serial command reception or (on an EPC-7) sending a word serial command response overwrites the contents of the interface's message registers, destroying any data preserved there.

On an EPC-7 or EPC-8, the function returns **EPC_DIR_ERR** when a Byte Available or Trigger word serial command is received and the interface is not enabled for data input (e.g., interface's DIR bit is clear). Likewise, on an EPC-7 or EPC-8, the function returns **EPC_DOR_ERR** when a Byte Request word serial command is received and the interface is not enabled for data output (e.g., the interface's DOR bit is clear). Use **EpcSrvSendWSProtocolError** to send protocol errors to the commander device.

EPConnect does not support receiving a word serial command on a VXLink interface. Attempting to use the function on a VXLink interface results in an **EPC_INV_HW** error.

Return Value

The function returns a Bus Management return value:

EPC_DIR_ERR	A word serial command protocol DIR violation error occurred.
EPC_DOR_ERR	A word serial command protocol DOR violation error occurred.
EPC_INV_ENABLE	The parameter *Enable_Next_Command* is invalid.
EPC_INV_HW	The interface does not support enabling word serial command reception.
EPC_INV_PTR	One or more of parameters *Command_Ptr* and *Command_Size_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_RECV_TIMEOUT	A timeout occurred waiting for a word serial command.
EPC_SUCCESS	The function completed successfully.

See Also EpcOpenSession, EpcSrvEnableWSCommand, EpcSrvSendWSResponse.

EpcSrvSendProtocolEvent

Description Sends a protocol event to the commander device.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSrvSendProtocolEvent(  unsigned long   Session_ID,
                          unsigned short  ULA,
                          unsigned long   Method_Mask,
                          unsigned short  Protocol_Event);
```

Session_ID	*Session_ID* specifies a session.
ULA	*ULA* specifies the unique logical address of the commander device.
Method_Mask	*Method_Mask* specifies the method to use for sending the protocol event.
Protocol_Event	*Protocol_Event* specifies a protocol event.

ULA is unused when *Method_Mask* specifies a VMEbus interrupt.

Visual Basic Synopsis

```
EpcSrvSendProtocolEvent% Lib "bmvxiw16.dll"
                         (ByVal Session_ID&,
                         ByVal ULA%,
                         ByVal Method_Mask&,
                         ByVal Protocol_Event%)
```

Remarks **EpcSrvSendProtocolEvent** sends the VMEbus protocol event *Protocol_Event* to the commander device at unique logical address *ULA*.

Method_Mask specifies the method to use for sending the protocol event. Valid values are:

Method_Mask	Description
EPC_VME1_INT	VMEbus interrupt 1 (EPC-7 only).
...	
EPC_VME7_INT	VMEbus interrupt 7 (EPC-7 only).
EPC_SIGNAL_REG	Write to the commander device's signal register.

The function always overwrites the lower eight bits of the specified *Protocol_Event* parameter with the unique logical address of the interface.

On an EPC-7, using a VMEbus interrupt to send a protocol event requires the use of the EPC-7's message high register. Receiving 32-bit long word serial commands also requires the use of the EPC-7's message high register. Therefore, using a VMEbus interrupt to send a protocol event while simultaneously receiving 32-bit long word serial commands can have unpredictable results. Note, however, that no conflict occurs when using a VMEbus interrupt to send a protocol event while simultaneously receiving 16-bit word serial commands.

Return Value The function returns a Bus Management return value:

EPC_BERR	A bus error occurred writing the protocol event into the commander device's signal register.
EPC_INV_ASSERT	The interface is already asserting a VMEbus interrupt.
EPC_INV_EVENT	The parameter *Protocol_Event* is invalid.
EPC_INV_METHOD	The parameter *Method_Mask* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_INV_ULA	The parameter *ULA* is invalid.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_SUCCESS	The function completed successfully.

See Also EpcOpenSession.

EpcSrvSendWSProtocolError

Description Sends a word serial protocol error to the commander device.

C Synopsis

```
#include "busmgr.h"

short
EpcSrvSendWSProtocolError
                    (unsigned long Session_ID,
                    unsigned short Protocol_Error,
                    unsigned short Enable_Next_Command,
                    unsigned long  Timeout);
```

Session_ID	*Session_ID* specifies a session.
Protocol_Error	*Protocol_Error* specifies a *Read Protocol Error* word serial command response.
Enable_Next_Command	*Enable_Next_Command* specifies whether to enable the interface hardware to receive the another word serial command.
Timeout	*Timeout* specifies the number of milliseconds to wait for a word serial command. *Timeout* also specifies the number of milliseconds to wait for a commander to read a word serial command response.

Visual Basic Synopsis

```
Declare Function
EpcSrvSendWSProtocolError% Lib "bmvxiw16.dll"
                    (ByVal Session_ID&,
                    ByVal Protocol_Error%,
                    ByVal Enable_Next_Command%,
                    ByVal Timeout&)
```

Remarks

EpcSrvSendWSProtocolError notifies a commander device that a word serial protocol error has occurred by asserting the interface's response register Write Ready and Err* bits. The function then either:

- receives a READ PROTOCOL ERROR word serial command, deasserts the interface's response register Err* bit, sends the specified *Protocol_Error* response, and optionally enables reception of the next word serial command, or
- receives an ABORT NORMAL OPERATION word serial command, deasserts the interface's response register Err* bit, and returns **EPC_RECV_ANO**.
- receives a CLEAR word serial command, deasserts the interface's response register Err* bit, and returns **EPC_RECV_CLEAR**.
- receives an END NORMAL OPERATION word serial command, deasserts the interface's response register Err* bit, and returns **EPC_RECV_ENO.**

All word serial commands received while the interface is waiting for either a READ PROTOCOL ERROR word serial command, an ABORT NORMAL OPERATION, a CLEAR, or an END NORMAL OPERATION are discarded.

The following constants specify valid values for the *Enable_Next_Command* parameter:

Constant	Description
EPC_DISABLE_ALL	Disable word serial command reception.
EPC_ENABLE_WRDY	Enable word serial command reception by asserting WRDY and deasserting both DIR and DOR.
EPC_ENABLE_DIR	Enable word serial command reception and data input by asserting both WRDY and DIR and deasserting DOR.
EPC_ENABLE_DOR	Enable word serial command reception and data output by asserting both WRDY and DOR and deasserting DIR.
EPC_ENABLE_ALL	Enable word serial command reception, data input, and data output by asserting WRDY, DIR, and DOR.

On an EPC-7, any outgoing word serial command response must be read from the interface's message registers by the interface's commander device before attempting to notify a commander device that a word serial protocol error has occurred. Otherwise, additional word serial protocol violations can occur. Successful completion of **EpcSrvSendWSResponse** indicates that an outgoing word serial command response has been read from the interface's message registers.

EPConnect does not support sending a word serial protocol error on VXLink interfaces. Attempting to use the function on an unsupported interface results in an **EPC_INV_HW** error.

Return Value

The function returns a EPConnect return value:

EPC_INV_ENABLE	The parameter *Enable_Next_Command* is invalid.
EPC_INV_ERROR	The parameter *Protocol_Error* is invalid.

EPC_INV_HW	The interface hardware does not support sending a word serial protocol error.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The Bus Manager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_RECV_ANO	The interface received an ABORT NORMAL OPERATION command.
EPC_RECV_CLEAR	The interface received a CLEAR word serial command.
EPC_RECV_ENO	The interface received an END NORMAL OPERATION word serial command.
EPC_RECV_TIMEOUT	A timeout occurred receiving a word serial command.
EPC_SEND_TIMEOUT	A timeout occurred sending a word serial command response.
EPC_SUCCESS	The function completed successfully.

See Also **EpcOpenSession, EpcSrvSendWSResponse.**

EpcSrvSendWSResponse

Description Sends a word serial command response to the commander device.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSrvSendWSResponse
        (unsigned long   Session_ID,
        void  FAR *      Response_Ptr,
        unsigned short   Response_Size,
        unsigned short   Enable_Next_Command,
        unsigned long    Timeout);
```

Session_ID	*Session_ID* specifies a session.
Response_Ptr	*Response_Ptr* specifies the location of a word serial command response.
Response_Size	*Response_Size* specifies the size of the word serial command response.
Enable_Next_Command	*Enable_Next_Command* specifies whether to enable the interface hardware to receive the another word serial command.
Timeout	*Timeout* specifies the number of milliseconds to wait for a commander to read the word serial command response.

Visual Basic Synopsis

```
Declare Function
EpcSrvSendWSResponse% Lib "bmvxiw16.dll"
                (ByVal Session_ID&,
                Response_Ptr As Any,
                ByVal Response_Size%,
                ByVal Enable_Next_Command%,
                ByVal Timeout&)
```

Remarks

EpcSrvSendWSResponse optionally sends the word serial command response at the location pointer to by *Response_Ptr* to a commander device. The function then configures the interface hardware for future word serial command reception.

Response_Size specifies the size of the word serial command response:

Response_Size	Description
EPC_16_BIT	Send a 16-bit word serial command response.
EPC_32_BIT	Send a 32-bit long word serial command response. (EPC-7 only)

The following constants specify valid values for the *Enable_Next_Command* parameter:

Constant	Description
EPC_DISABLE_ALL	Disable word serial command reception.
EPC_ENABLE_WRDY	Enable word serial command reception by asserting WRDY and deasserting both DIR and DOR.
EPC_ENABLE_DIR	Enable word serial command reception and data input by asserting both WRDY and DIR and deasserting DOR.
EPC_ENABLE_DOR	Enable word serial command reception and data output by asserting both WRDY and DOR and deasserting DIR.
EPC_ENABLE_ALL	Enable word serial command reception, data input, and data output by asserting WRDY, DIR, and DOR.

On an EPC-7, sending a word serial command response while word serial command reception is enabled can result in a word serial protocol violation by allowing the commander device to write over the word serial command response.

Occasionally, it is useful to ensure that a word serial response has been read without destroying the contents of the interface's message registers. To ensure that a word serial response has been read without destroying the contents of the interface's message registers, use **EpcSrvSendWSResponse** with a *Response_Ptr* parameter value of null and an *Enable_Next_Command* parameter value of **EPC_DISABLE_ALL**. Such a call tests that a word serial command response has been read from the interface's message registers without destroying the contents of the registers. Note that either enabling word serial command reception or sending a word serial command response overwrites the contents of the interface's message registers, destroying any data preserved there.

The function returns **EPC_MULTIPLE_ERR** if the *Response_Ptr* parameter is not null and previously sent response data remains unread in the interface's message registers.

The function returns **EPC_SEND_TIMEOUT** if a commander device does not read the word serial command response within the specified timeout time. If this error occurs, the word serial command response remains in the interface message register.

EPConnect does not support sending a word serial command response on a VXLink interface. Attempting to use the function on a VXLink interface results in an **EPC_INV_HW** error.

Return Value The function returns a Bus Management return value:

EPC_INV_ENABLE	The parameter *Enable_Next_Command* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_HW	The interface does not support sending a word serial command response.
EPC_INV_SIZE	The parameter *Response_Size* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another session.
EPC_MULTIPLE_ERR	A word serial protocol multiple queries error occurred.
EPC_SEND_TIMEOUT	A timeout occurred waiting for a commander to read the word serial command response.
EPC_SUCCESS	The function completed successfully.

See Also **EpcOpenSession, EpcSrvEnableWSCommand.**

EpcSwap16

Description Byte-swaps a 16-bit value.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSwap16(unsigned short FAR *Value_Ptr);
```

Value_Ptr	*Value_Ptr* specifies a location containing a 16-bit value to byte-swap.

Visual Basic Synopsis

```
Declare Function
EpcSwap16% Lib "bmvxiw16.dll" (Value_Ptr%)
```

Remarks **EpcSwap16** byte-swaps the 16-bit value in the location pointed to by *Value_Ptr*.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Value_Ptr* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcSwapBuffer, EpcSwap32, EpcSwap48, EpcSwap64, EpcSwap80.**

Example See **EpcSwapBuffer.**

EpcSwap32

2

Description Byte-swaps a 32-bit value.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSwap32(unsigned long FAR*Value_Ptr);
```

Value_Ptr	*Value_Ptr* specifies a location containing a 32-bit value to byte-swap.

Visual Basic Synopsis

```
Declare Function
EpcSwap32% Lib "bmvxiw16.dll" (Value_Ptr&)
```

Remarks **EpcSwap32** byte-swaps the 32-bit value in the location pointed to by *Value_Ptr*.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Value_Ptr* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcSwapBuffer**, **EpcSwap16**, **EpcSwap48**, **EpcSwap64**, **EpcSwap80**.

Example See **EpcSwapBuffer**.

EpcSwap48

Description Byte-swaps a 48-bit value.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcSwap48(void **FAR***Value_Ptr*);

Value_Ptr	*Value_Ptr* specifies a location containing a 48-bit value to byte-swap.

Visual Basic Synopsis

Declare Function
EpcSwap48% Lib "bmvxiw16.dll" (*Value_Ptr* As Any)

Remarks **EpcSwap48** byte-swaps the 48-bit value in the location pointed to by *Value_Ptr*.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Value_Ptr* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcSwapBuffer, EpcSwap16, EpcSwap32, EpcSwap64, EpcSwap80.**

Example See **EpcSwapBuffer.**

EpcSwap64

2

Description Byte-swaps a 64-bit value.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSwap64(void FAR*Value_Ptr);
```

Value_Ptr — *Value_Ptr* specifies a location containing a 64-bit value to byte-swap.

Visual Basic Synopsis

```
Declare Function
EpcSwap64% Lib "bmvxiw16.dll" (Value_Ptr As Any)
```

Remarks **EpcSwap64** byte-swaps the 64-bit value in the location pointed to by *Value_Ptr*.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Value_Ptr* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcSwapBuffer**, **EpcSwap16**, **EpcSwap32**, **EpcSwap48**, **EpcSwap80**.

Example See **EpcSwapBuffer**.

EpcSwap80

Description Byte-swaps an 80-bit value.

C Synopsis

```
#include "busmgr.h"

short
EpcSwap80(void FAR*Value_Ptr);
```

Value_Ptr — *Value_Ptr* specifies a location containing a 80-bit value to byte-swap.

Visual Basic Synopsis

```
Declare Function
EpcSwap80% Lib "bmvxiw16.dll" (Value_Ptr As Any)
```

Remarks **EpcSwap80** byte-swaps the 80-bit value in the location pointed to by *Value_Ptr*.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Value_Ptr* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also **EpcSwapBuffer, EpcSwap16, EpcSwap32, EpcSwap48, EpcSwap64.**

Example See **EpcSwapBuffer.**

EpcSwapBuffer

Description Byte-swaps a buffer of data.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcSwapBuffer(void FAR    * Buffer_Ptr,
              unsigned long  Buffer_Size,
              unsigned short Width);
```

Buffer_Ptr	*Buffer_Ptr* specifies the location of a buffer of data elements to byte-swap.
Buffer_Size	*Buffer_Size* specifies the size of the specified data buffer, in bytes.
Width	*Width* specifies the width of the individual data elements in the specified data buffer.

Visual Basic Synopsis

```
Declare Function
EpcSwapBuffer% Lib "bmvxiw16.dll" (Buffer_Ptr As Any, ByVal
Buffer_Size&, ByVal Width%)
```

Remarks **EpcSwapBuffer** byte-swaps the array of data elements at the location pointed to by *Buffer_Ptr*.

Width specifies the width of the individual data elements in the specified data buffer, in bytes. Valid values are:

Width	Description
EPC_8_BIT	The specified buffer contains *Buffer_Size* 8-bit data elements.
EPC_16_BIT	The specified buffer contains *Buffer_Size*/2 16-bit data elements.
EPC_32_BIT	The specified buffer contains *Buffer_Size*/4 32-bit data elements.
EPC_48_BIT	The specified buffer contains *Buffer_Size*/6 48-bit data elements.
EPC_64_BIT	The specified buffer contains *Buffer_Size*/8 64-bit data elements.
EPC_80_BIT	The specified buffer contains *Buffer_Size*/10 80-bit data elements.

The function assumes that all of the data elements in the specified buffer are the same size.

The function does not limit *Buffer_Size* to less than 64 Kbytes, nor does it make any attempt to detect the end of the buffer segment. The function wraps around to the beginning of the buffer segment if *Buffer_Size* is too large.

Return Value

The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Buffer_Ptr* is invalid.
EPC_INV_SIZE	The parameter *Buffer_Size* is not a multiple of *Width*.
EPC_INV_WIDTH	The parameter *Width* is invalid.
EPC_SUCCESS	The function completed successfully.

See Also

EpcSwap16, EpcSwap32, EpcSwap48, EpcSwap64, EpcSwap80.

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * byteord.c -- Bus Management Library byte order functions sample code.
 */

#include "busmgr.h"

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
ByteOrdSample(void);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * GLOBAL DATA...
 */

unsigned char Value16[] = { 0x00, 0x11 };
unsigned char Value32[] = { 0x00, 0x11, 0x22, 0x33 };
unsigned char Value48[] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55 };
unsigned char Value64[] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77 };
unsigned char Value80[] = { 0x00, 0x11, 0x22, 0x33, 0x44,
                            0x55, 0x66, 0x77, 0x88, 0x99 };

/*
 * CODE...
 */

short FAR
ByteOrdSample(void)
{
    /*
     * Byte-swap a 16-bit value, then byte-swap it back to its original order.
     *
     * NOTES:
     *      1.  For the sake of example, the code uses both EpcSwap16() and
     *          EpcSwapBuffer() to swap the data.
     */

    EpcSwap16((unsigned short FAR *) Value16);
    EpcSwapBuffer((void FAR *) Value16, sizeof(Value16), EPC_16_BIT);

    /*
     * Byte-swap a 32-bit value, then byte-swap it back to its original order.
     *
     * NOTES:
     *      1.  For the sake of example, the code uses both EpcSwap32() and
     *          EpcSwapBuffer() to swap the data.
     */

    EpcSwap32((unsigned long FAR *) Value32);
    EpcSwapBuffer((void FAR *) Value32, sizeof(Value32), EPC_32_BIT);

    /*
     * Byte-swap a 48-bit value, then byte-swap it back to its original order.
     *
```

```
     * NOTES:
     *      1.  For the sake of example, the code uses both EpcSwap48() and
     *          EpcSwapBuffer() to swap the data.
     */

    EpcSwap48((void FAR *) Value48);
    EpcSwapBuffer((void FAR *) Value48, sizeof(Value48), EPC_48_BIT);

    /*
     * Byte-swap a 64-bit value, then byte-swap it back to its original order.
     *
     * NOTES:
     *      1.  For the sake of example, the code uses both EpcSwap64() and
     *          EpcSwapBuffer() to swap the data.
     */

    EpcSwap64((void FAR *) Value64);
    EpcSwapBuffer((void FAR *) Value64, sizeof(Value64), EPC_64_BIT);

    /*
     * Byte-swap a 80-bit value, then byte-swap it back to its original order.
     *
     * NOTES:
     *      1.  For the sake of example, the code uses both EpcSwap80() and
     *          EpcSwapBuffer() to swap the data.
     */

    EpcSwap80((void FAR *) Value80);
    EpcSwapBuffer((void FAR *) Value80, sizeof(Value80), EPC_80_BIT);
    WinPrintf("SUCCESS:  ByteOrdSample() complete.\n");
    return (EPC_SUCCESS);
}
```

EpcUnlockSession

Description Unlocks shared interface hardware for a session.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcUnlockSession(unsigned long Session_ID);
```

Session_ID	*Session_ID* specifies a session.

Visual Basic Synopsis

```
Declare Function
EpcUnlockSession% Lib "bmvxiw16.dll" (ByVal Session_ID&)
```

Remarks **EpcUnlockSession** unlocks shared interface hardware for the specified *Session_ID*.

Return Value The function returns a Bus Management return value:

EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_NOT_LOCKED	Shared interface hardware is not locked by the specified session.
EPC_SUCCESS	The function completed successfully.

See Also **EpcLockSession**, **EpcOpenSession**.

Example See **EpcGetLockingTimeout**.

EpcUnmapBusMemory

Description Destroys a bus memory mapping.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcUnmapBusMemory( unsigned long          Session_ID,
                   volatile void HUGE *Mapped_Ptr);
```

Session_ID	*Session_ID* specifies a bus session.
Mapped_Ptr	*Mapped_Ptr* specifies a pointer to mapped bus memory.

Visual Basic Synopsis

```
Declare Function
EpcUnmapBusMemory% Lib "bmvxiw16.dll" (ByVal
Session_ID&, ByVal Mapped_Ptr As Any)
```

Remarks **EpcUnmapBusMemory** destroys a bus memory mapping.

Return Value The function returns a Bus Management return value:

EPC_INV_MAP	The specified *Mapped_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_USAGE	The mapping specified by *Mapped_Ptr* is in use by another thread.
EPC_OS_ERROR	An operating system error occurred.
EPC_SUCCESS	The function completed successfully.

See Also **EpcMapBusMemory**, **EpcOpenSession**.

Example See **EpcCopyData**.

EpcUnmapSharedMemory

Description Destroys a shared memory mapping.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcUnmapSharedMemory( unsigned long          Session_ID,
                      volatile void HUGE *Mapped_Ptr);
```

Session_ID	*Session_ID* specifies a bus session.
Mapped_Ptr	*Mapped_Ptr* specifies a pointer to mapped shared memory.

Visual Basic Synopsis

```
Declare Function
EpcUnmapSharedMemory% Lib "bmvxiw16.dll" (ByVal
Session_ID&, ByVal Mapped_Ptr As Any)
```

Remarks **EpcUnmapSharedMemory** destroys a shared memory mapping.

Return Value The function returns a Bus Management return value:

EPC_INV_MAP	The specified *Mapped_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_INV_USAGE	The mapping specified by *Mapped_Ptr* is in use by another thread.
EPC_OS_ERROR	An operating system error occurred.
EPC_SUCCESS	The function completed successfully.

See Also **EpcMapSharedMemory**, **EpcOpenSession**.

Example See **EpcCopyData**.

EpcValidateBusMapping

Validates a bus memory mapping that uses dynamically configured bus window hardware.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcValidateBusMapping( unsigned long          Session_ID,
                       volatile void HUGE *Mapped_Ptr);
```

Session_ID	*Session_ID* specifies a bus session.
Mapped_Ptr	*Mapped_Ptr* specifies a pointer to mapped bus memory.

Remarks

EpcValidateBusMapping configures an interface's dynamically configured bus window hardware with the attributes of the specified bus memory mapping.

The function supports direct bus access for bus memory mappings created using **EpcMapBusMemoryExt**. The function is a no-op for bus memory mappings created using **EpcMapBusMemory**.

Direct bus access using dynamically configured bus window hardware requires using **EpcValidateBusMapping** before a group direct bus accesses to ensure that the bus window references the desired bus memory.

On an operating system that supports multiple processes or threads (any non-DOS operating system), direct bus access using dynamically configured bus windows also requires a mechanism for protecting the bus window hardware from reconfiguration during the bus accesses. In a non-preemptive environment (like Windows), simply insuring that all direct bus accesses complete before giving up the processor is sufficient.

In an environment with a preemptive scheduling algorithm (like LynxOS or OS/2), direct bus access using dynamically configured bus window hardware also requires using **EpcLockSession** and **EpcValidateBusMapping** before a group of direct bus accesses and **EpcUnlockSession** after the direct bus access is complete. Using **EpcLockSession** and **EpcUnlockSession** insures that another thread does not reconfigure the dynamically configured bus window hardware during the access.

Return Value

The function returns a EPConnect return value:

EPC_INV_MAP	The specified *Mapped_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present.
EPC_LOCKED	Shared interface hardware is locked by another bus session.
EPC_OS_ERROR	An operating system error occurred.
EPC_SUCCESS	The function completed successfully.

See Also

EpcMapBusMemory, EpcMapBusMemoryExt, EpcLockSession, EpcOpenSession, EpcUnlockSession.

EpcVerifyEnvironment

Description Verifies and queries the EPConnect environment.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcVerifyEnvironment(struct **EpcEnvironment FAR** **Environment_Ptr*);

Environment_Ptr — *Environment_Ptr* specifies a location where data describing the EPConnect environment will be placed.

Visual Basic Synopsis

Declare Function
EpcVerifyEnvironment% Lib "bmvxiw16.dll" (*Environment_Ptr* As Any)

Remarks **EpcVerifyEnvironment** verifies the EPConnect environment and places data describing the environment in the structure pointed to by *Environment_Ptr*.

The returned EPConnect environment structure contains a complete description of the underlying hardware and software. The structure also contains a complete description of the Bus Management Library features supported by the underlying software and hardware:

```
struct EpcEnvironment
{
   /* Hardware, firmware, and software revision attributes.          */

   unsigned char HWRevision;          /* Hardware revision number:    */
                                      /*     EPC_7                    */
                                      /*     EPC_8                    */
                                      /*     VXLink_ISA               */
   unsigned char BIOSMajorRevision;   /* BIOS major revision number.  */
   unsigned char BIOSMinorRevision;   /* BIOS minor revision number.  */
   unsigned char SWMajorRevision;     /* Software major revision number. */
   unsigned char SWMinorRevision;     /* Software minor revision number. */
```

```
                                /* Memory mapping attributes.                  */

unsigned char IsHWByteSwap;     /* Is the interface capable of hardware byte */
                                /* swapping?                                   */
                                /*     TRUE                                    */
                                /*     FALSE                                   */
unsigned short   AddressMod;    /* Valid address modifiers (OR'd               */
                                /* combination of):                            */
                                /*     EPC_A16S                                */
                                /*     EPC_A24SD                               */
                                /*     EPC_A32SD                               */
unsigned short   ByteOrder;     /* Valid byte ordering values (OR'd            */
                                /* combination of):                            */
                                /*     EPC_IBO                                 */
                                /*     EPC_MBO                                 */
unsigned short   DataWidth;     /* Valid data widths (OR'd combination of):    */
                                /*     EPC_8_BIT                               */
                                /*     EPC_8_BIT_ODD                           */
                                /*     EPC_16_BIT                              */
                                /*     EPC_32_BIT                              */
                                /*     EPC_FASTCOPY                            */
unsigned short MinA16Address;   /* Minimum accessible A16 address.             */
unsigned short MaxA16Address;   /* Maximum accessible A16 address.             */
unsigned long MinA24Address;    /* Minimum accessible A24 address.             */
unsigned long MaxA24Address;    /* Maximum accessible A24 address.             */
unsigned long MinA32Address;    /* Minimum accessible A32 address.             */
unsigned long MaxA32Address;    /* Maximum accessible A32 address.             */

/* Event attributes. */

unsigned long EventMask;        /* Valid events (OR'd combination of):         */
                                /*     EPC_MSG_INT             */
                                /*     EPC_VME1_INT            */
                                /*     ...                     */
                                /*     EPC_VME7_INT            */
                                /*     EPC_SIGNAL_INT          */
                                /*     EPC_TTL_TRIG0_INT       */
                                /*     ...                     */
                                /*     EPC_TTL_TRIG7_INT       */
                                /*     EPC_ACFAIL_ERR          */
                                /*     EPC_BERR_ERR            */
                                /*     EPC_SYSFAIL_ERR         */
                                /*     EPC_WATCHDOG_ERR        */
                                /*     EPC_SYSRESET_ERR        */

/* EPC bus configuration attributes. */

unsigned short   BusEnable;         /* Valid bus enable attributes (OR'd       */
                                    /* combination of):                        */
                                    /*     EPC_DISABLE_BUS                     */
                                    /*     EPC_ENABLE_BUS                      */
unsigned short   BusArbMode;        /* Valid bus arbitration mode attributes   */
                                    /* (OR'd combination of):                  */
                                    /*     EPC_PRIORITY                        */
                                    /*     EPC_ROUND_ROBIN                     */
unsigned short   BusArbPriority;    /* Valid bus arb priority attributes       */
                          /* (OR'd combination of):                            */
                          /*     EPC_PRIORITY0                                 */
                          /*     EPC_PRIORITY1                                 */
                          /*     EPC_PRIORITY2                                 */
                          /*     EPC_PRIORITY3                                 */
unsigned short   BusRelease;/* Valid bus release attributes (OR'd             */
                          /* combination of):                                  */
```

```
                        /*     EPC_ROR                                  */
                        /*     EPC_RONR                                 */

/* Miscellaneous configuration attributes. */

unsigned char IsBERRAssertion; /* Is  software capable of asserting the    */
                        /* sticky BERR bit?                 */
                        /*     TRUE                         */
                        /*     FALSE                        */
unsigned long GetMiscMask;  /* Miscellaneous attributes that can be     */
                        /* queried (OR'd combination of):           */
                        /*     EPC_DIR                      */
                        /*     EPC_DOR                      */
                        /*     EPC_ERR                      */
                        /*     EPC_LOCK                     */
                        /*     EPC_PASS                     */
                        /*     EPC_PIPELINE_BUSY            */
                        /*     EPC_READY                    */
                        /*     EPC_RRDY                     */
                        /*     EPC_RSRC_MGR                 */
                        /*     EPC_STICKY_BERR              */
                        /*     EPC_SYSFAIL_OUT              */
                        /*     EPC_SYSRESET_IN              */
                        /*     EPC_TTL_LATCH0               */
                        /*     ...                          */
                        /*     EPC_TTL_LATCH7               */
                        /*     EPC_WATCHDOG                 */
                        /*     EPC_WRDY                     */
unsigned long SetMiscMask;  /* Miscellaneous attributes that can be     */
                        /* defined (OR'd combination of):           */
                        /*     EPC_DIR                      */
                        /*     EPC_DOR                      */
                        /*     EPC_ERR                      */
                        /*     EPC_LOCK                     */
                        /*     EPC_PASS                     */
                        /*     EPC_READY                    */
                        /*     EPC_RESET                    */
                        /*     EPC_RRDY                     */
                        /*     EPC_RSRC_MGR                 */
                        /*     EPC_STICKY_BERR              */
                        /*     EPC_SYSFAIL_OUT              */
                        /*     EPC_SYSRESET_IN              */
                        /*     EPC_WRDY                     */


/* Slave memory configuration attributes. */

unsigned char IsSelfAccess; /* Is  the I/F capable of slave memory   */
                        /* self-accesses (via the VMEbus)?          */
                        /*     TRUE                             */
                        /*     FALSE                            */
unsigned short SlaveSpace;  /* Valid slave memory address spaces    */
                        /* (OR'd combination of):               */
                        /*     EPC_DISABLED                     */
                        /*     EPC_A24                          */
                        /*     EPC_A32                          */
unsigned short SlaveWidth;  /* Valid slave memory access widths (OR'd*/
                        /* combination of):                     */
                        /*     EPC_8_BIT                        */
                        /*     EPC_16_BIT                       */
                        /*     EPC_32_BIT                       */
unsigned long MinA24Slave;  /* Minimum A24 slave memory base        */
                        /* address.                             */
```

```
unsigned long MaxA24Slave;          /* Maximum A24 slave memory base   */
                                    /* address.                        */
unsigned long A24SlaveBaseInc;      /* A24 slave memory base increment*/
unsigned long MinA32Slave;          /* Minimum A32 slave memory base   */
                                    /* address.                        */
unsigned long MaxA32Slave;          /* Maximum A32 slave memory base   */
                                    /* address.                        */
unsigned long A32SlaveBaseInc;      /* A32 slave memory base increment*/


/* unique logical address configuration attributes.                    */
unsigned char MinULA;               /* Minimum valid ULA.              */
unsigned char MaxULA;               /* Maximum valid ULA.              */

/* Bus line attributes. */

unsigned long GetBusLineMask;   /* Bus control lines that can be queried */
                                /* (OR'd combination of):                */
                                /*     EPC_ACFAIL                        */
                                /*     EPC_SYSFAIL                       */
                                /*     EPC_SYSRESET                      */
unsigned long GetBusMODIDMask;  /* Bus MODID lines that can be queried   */
                                /* (OR'd combination of):                */
                                /*     EPC_SLOT_MODID                    */
                                /*     EPC_MODID0                        */
                                /*     ...                               */
                                /*     EPC_MODID12                       */
unsigned long GetBusTriggerMask;   /* Bus trigger lines that can be queried */
                                /* (OR'd combination of):                */
                                /*     EPC_ECL_TRIG0                     */
                                /*     EPC_ECL_TRIG1                     */
                                /*     EPC_TTL_TRIG0                     */
                                /*     ...                               */
                                /*     EPC_TTL_TRIG7                     */

/* EPC line attributes. */

unsigned long GetEpcLineMask;   /* I/f control lines that can be queried */
                                /* (OR'd combination of):                */
                                /*     EPC_SYSFAIL                       */
                                /*     EPC_SYSRESET                      */
unsigned long SetEpcLineMask;   /* I/f control lines that can be defined */
                                /* (OR'd combination of):                */
                                /*     EPC_SYSFAIL                       */
                                /*     EPC_SYSRESET                      */
unsigned long SetEpcMODIDMask;  /* I/f MODID lines that can be defined   */
                                /* (OR'd combination of):                */
                                /*     EPC_MODID0                        */
                                /*     ...                               */
                                /*     EPC_MODID12                       */
unsigned long GetEpcTriggerMask;   /* I/f trigger lines that can be queried */
                                /* (OR'd combination of):                */
                                /*     EPC_ECL_TRIG0                     */
                                /*     EPC_ECL_TRIG1                     */
                                /*     EPC_TTL_TRIG0                     */
                                /*     ...                               */
                                /*     EPC_TTL_TRIG7                     */
unsigned long SetEpcTriggerMask;   /* I/f trigger lines that can be queried */
                                /* (OR'd combination of):                */
                                /*     EPC_ECL_TRIG0                     */
                                /*     EPC_ECL_TRIG1                     */
                                /*     EPC_TTL_TRIG0                     */
                                /*     ...                               */
                                /*     EPC_TTL_TRIG7                     */
unsigned long InTriggerMask;    /* I/f trigger lines used as an          */
```

```
                                /* input trigger in a trigger mapping    */
                                /* operation (OR'd combination of):      */
                                /*     EPC_TTL_TRIG0                     */
                                /*     ...                               */
                                /*     EPC_TTL_TRIG7                     */
unsigned long OutTriggerMask[EPC_TRIGGER_CNT];
                                /* I/f trigger lines that can be used as */
                                /* output triggers in a trigger mapping  */
                                /* operation (one array element for each */
                              /* potential input trigger;  each entry is an   */
                              /* OR'd combination of):                        */
                              /*     EPC_TTL_TRIG0                            */
                              /*     ...                                      */
                              /*     EPC_TTL_TRIG7                            */

/* Watchdog timer attributes. */

unsigned short   WatchdogCfg;  /* Valid watchdog timer configuration       */
                              /* constants (OR'd combination of):         */
                              /*     EPC_WDT_RESET                        */
                              /*     EPC_WDT_FAST_ERROR                   */
                              /*     EPC_WDT_FAST_RESET                   */
                              /*     EPC_WDT_SLOW_ERROR                   */
                              /*     EPC_WDT_SLOW_RESET                   */

/* Servant attributes. */

unsigned char IsProtocolError;     /* Is the EPC capable of signaling a    */
                              /* protocol error to its commander?         */
                              /*     TRUE                                 */
                              /*     FALSE                                */
unsigned short   WSSize;      /* Valid word serial command/response       */
                              /* sizes (OR'd combination of):             */
                              /*     EPC_16_BIT                           */
                              /*     EPC_32_BIT                           */
                              /*     EPC_48_BIT                           */
unsigned short   EnableNextCommand;   /* Valid word serial command enable */
                              /* constants (OR'd combination of):         */
                              /*     EPC_DISABLE_ALL                      */
                              /*     EPC_ENABLE_WRDY                      */
                              /*     EPC_ENABLE_DIR                       */
                              /*     EPC_ENABLE_DOR                       */
                              /*     EPC_ENABLE_ALL                       */
unsigned long MethodMask;     /* Valid methods for sending protocol       */
                              /* events (OR'd combination of):            */
                              /*     EPC_VME1_INT                         */
                              /*     ...                                  */
                              /*     EPC_VME7_INT                         */
                              /*     EPC_SIGNAL_REG                       */
unsigned short   IRQ;         /* PC-AT IRQ used.                          */
unsigned short   IOBase;      /* I/O base address.                        */
unsigned long WindowBase;     /* Bus window base address.                 */
unsigned long WindowSize;     /* Bus window size, in bytes.               */
```

```
    /* Dynamic memory mapping attributes. */

    unsigned char IsHWByteSwapExt; /* Is  the interface  capable of hardware    */
                                   /* byte swapping?                            */
                                   /*     TRUE                                  */
                                   /*     FALSE                                 */
    unsigned short    AddressModExt;       /* Valid address modifiers (OR'd
    */
                                   /* combination of):                          */
                                   /*     EPC_A16N                              */
                                   /*     EPC_A16S                              */
                                   /*     EPC_A24ND                             */
                                   /*     EPC_A24NP                             */
                                   /*     EPC_A24SD                             */
                                   /*     EPC_A24SP                             */
                                   /*     EPC_A32ND                             */
                                   /*     EPC_A32NP                             */
                                   /*     EPC_A32SD                             */
                                   /*     EPC_A32SP                             */
    unsigned short    ByteOrderExt,     /* Valid byte ordering values (OR'd      */
                                   /* combination of):                          */
                                   /*     EPC_IBO                               */
                                   /*     EPC_MBO                               */
    unsigned short    DataWidthExt;     /* Valid data widths (OR'd combination of):
    */
                                   /*     EPC_8_BIT                             */
                                   /*     EPC_8_BIT_ODD                         */
                                   /*     EPC_16_BIT                            */
                                   /*     EPC_32_BIT                            */
                                   /*     EPC_FASTCOPY                          */
    unsigned short    MinA16AddressExt;/* Minimum accessible A16 address. */
    unsigned short    MaxA16AddressExt;/* Maximum accessible A16 address. */
    unsigned long MinA24AddressExt;/* Minimum accessible A24 address.    */
    unsigned long MaxA24AddressExt;/* Maximum accessible A24 address.    */
    unsigned long MinA32AddressExt;/* Minimum accessible A32 address.    */
    unsigned long MaxA32AddressExt;/* Maximum accessible A32 address.    */

    unsigned long Reserved[6];         /* Reserved area (for future expansion). */
};
```

Return Value The function returns a Bus Management return value:

EPC_INV_HW	EPConnect does not support this revision of interface hardware.
EPC_INV_PTR	The parameter *Environment_Ptr* is invalid.
EPC_INV_SW	The BusManager device driver is not present or there is a revision mismatch between the Bus Management Library and the BusManager VxD.
EPC_SUCCESS	The function completed successfully.

Example See **EpcGetErrorString**.

EpcWaitForEvent

2

Description Wait for an event to occur.

C Synopsis

```
#include "busmgr.h"

short FAR PASCAL
EpcWaitForEvent(  unsigned long          Session_ID,
                  unsigned long          Timeout,
                  unsigned long          Wait_Mask,
                  unsigned long FAR *    Event_Mask_Ptr,
                  unsigned long FAR *    Event_Data_Ptr);
```

Session_ID	*Session_ID* specifies a session.
Timeout	*Timeout* specifies the number of milliseconds to wait for an enabled event to occur.
Wait_Mask	*Wait_Mask* specifies the events to await.
Event_Mask_Ptr	*Event_Mask_Ptr* specifies a location where an event mask that specifies the occurring event will be placed.
Event_Data_Ptr	*Event_Data_Ptr* specifies a location where event data from the occurring event will be placed.

Visual Basic Synopsis

```
Declare Function
EpcWaitForEvent% Lib "bmvxiw16.dll"
                 (ByVal Session_ID&,
                 ByVal Timeout&,
                 ByVal Wait_Mask&,
                 Event_Mask_Ptr&,
                 Event_Data_Ptr&)
```

Remarks

EpcWaitForEvent waits at least *Timeout* milliseconds for one of the events specified by *Wait_Mask* to occur, then places an event mask identifying the event and the event's data in the locations pointed to by *Event_Mask_Ptr* and *Event_Data_Ptr*, respectively.

The *Wait_Mask* parameter is a bit mask where each bit corresponds to an event. The *Wait_Mask* parameter should be an OR'd combination of the following constants:

Event	Description
EPC_MSG_INT	Message (EPC-7 and EPC-8 only)
EPC_VME1_INT	VMEbus interrupt 1
...	
EPC_VME7_INT	VMEbus interrupt 7
EPC_SIGNAL_INT	VXIbus signal FIFO interrupt
EPC_TTL_TRIG0_INT	VXIbus TTL Trigger 0 interrupt (EPC-7 only)
...	
EPC_TTL_TRIG7_INT	VXIbus TTL Trigger 7 interrupt (EPC-7 only)
EPC_SYSRESET_ERR	VMEbus SYSRESET error
EPC_ACFAIL_ERR	VMEbus power failure error
EPC_BERR_ERR	VMEbus access error
EPC_SYSFAIL_ERR	VMEbus SYSFAIL error
EPC_WATCHDOG_ERR	Watchdog timer expiration error (EPC-7 and EPC-8 only)
EPC_EXT_TRIG0_INT	External trigger 0 interrupt (VXLink only)
EPC_EXT_TRIG1_INT	External trigger 1 interrupt (VXLink only)

The value that **EpcWaitForEvent** places in the location pointed to by *Event_Mask_Ptr* can be one the following constants:

**Event_Mask_Ptr*	Description
EPC_MSG_INT	Message interrupt (EPC-7 and EPC-8 only)
EPC_VME1_INT	VMEbus interrupt 1
...	
EPC_VME7_INT	VMEbus interrupt 7
EPC_SIGNAL_INT	VXIbus signal FIFO interrupt
EPC_TTL_TRIG0_INT	VXIbus TTL Trigger 0 interrupt (EPC-7 only)
...	
EPC_TTL_TRIG7_INT	VXIbus TTL Trigger 7 interrupt (EPC-7 only)
EPC_SYSRESET_ERR	VMEbus SYSRESET error
EPC_ACFAIL_ERR	VMEbus power failure error
EPC_BERR_ERR	VMEbus access error
EPC_SYSFAIL_ERR	VMEbus SYSFAIL error
EPC_WATCHDOG_ERR	Watchdog timer expiration error (EPC-7 and EPC-8 only)
EPC_EXT_TRIG0_INT	External trigger 0 interrupt (VXLink only)
EPC_EXT_TRIG1_INT	External trigger 1 interrupt (VXLink only)

Whether the value that **EpcWaitForEvent** places in the location pointed to by *Event_Data_Ptr* is meaningful depends on the event:

**Event_Mask_Ptr*	**Event_Data_Ptr*
EPC_MSG_INT	0
EPC_VME1_INT	VMEbus interrupt
...	status/id
EPC_VME7_INT	(zero-extended to 32 bits)
EPC_SIGNAL_INT	VXIbus signal data (zero-extended to 32 bits)
EPC_TTL_TRIG0_INT	0
...	0
EPC_TTL_TRIG7_INT	0
EPC_SYSRESET_ERR	0
EPC_ACFAIL_ERR	0
EPC_BERR_ERR	0
EPC_SYSFAIL_ERR	0
EPC_WATCHDOG_ERR	0
EPC_EXT_TRIG0_INT	0
EPC_EXT_TRIG1_INT	0

When the specified timeout expires before an event occurs, the locations pointed to by *Event_Mask_Ptr* and *Event_Data_Ptr* contain undefined values.

If a session has an event handler and event handler stack pointer defined for an enabled event, an occurrence of the event satisfies the wait condition and invokes the event handler. The order of execution of the two threads is undefined.

Waiting for an event does not enable reception of the corresponding event. A separate, preceding call to **EpcSetEventEnableMask** is required.

Return Value The function returns a Bus Management return value:

EPC_INV_MASK	*Wait_Mask* contains an event that is not valid for this interface.
EPC_INV_PTR	One or more of the parameters *Event_Mask_Ptr* and *Event_Data_Ptr* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_SUCCESS	The function completed successfully.
EPC_TIMEOUT	The specified timeout period expired before an enabled event occurred.

See Also **EpcOpenSession, EpcSetEventEnableMask.**

Example See **EpcGetEventEnableMask.**

EpcWatchdogTimer

Description Modifies the interface's watchdog timer configuration.

C Synopsis

#include "busmgr.h"

short FAR PASCAL
EpcWatchdogTimer(unsigned long *Session_ID*, unsigned short *WatchdogCfg*);

Session_ID	*Session_ID* specifies a session.
WatchdogCfg	*WatchdogCfg* specifies a watchdog timer configuration.

Visual Basic Synopsis

Declare Function
EpcWatchdogTimer% Lib "bmvxiw16.dll" (ByVal *Session_ID*&, ByVal *Watchdog_Cfg*%)

Remarks **EpcWatchdogTimer** modifies the configuration of the interface's watchdog timer.

WatchdogCfg specifies the configuration of the interface's watchdog timer.

Valid values are:

Constant	Description
EPC_WDT_RESET	Reset the watchdog timer without modifying either the watchdog timer period or the operation that occurs upon watchdog timer expiration expiration (EPC-7 and EPC-8 only).
EPC_WDT_FAST_ERROR	Reset the watchdog timer, use the short watchdog timer period, and generate a watchdog timer error event upon watchdog timer expiration expiration (EPC-7 and EPC-8 only).
EPC_WDT_FAST_RESET	Reset the watchdog timer, use the short watchdog timer period, and reset the EPC upon watchdog timer expiration expiration (EPC-7 and EPC-8 only).
EPC_WDT_SLOW_ERROR	Reset the watchdog timer, use the long watchdog timer period, and generate a watchdog timer error event upon watchdog timer expiration expiration (EPC-7 and EPC-8 only).
EPC_WDT_SLOW_RESET	Reset the watchdog timer, use the long watchdog timer period, and reset the EPC upon watchdog timer expiration expiration (EPC-7 and EPC-8 only).

The actual length of an interface's watchdog timer period varies depending on the type of the interface:

Interface Type	Short Watchdog Timer Period	Long Watchdog Timer Period
EPC-7	210 milliseconds	6.7 seconds
EPC-8	128 milliseconds	8.2 seconds

If an interface's watchdog timer is not reset within the current watchdog timer period, a watchdog timer error event occurs. An application can enable the event (using **EpcSetEventEnableMask**) and install an event handler (using **EpcSetEventHandler**) to gracefully handle the error.

The interface's watchdog timer is typically reset in sections of code that execute frequently and/or execute at regular time intervals. Note that other bus operations may reset the watchdog timer as a side-effect of their execution.

By default, an interface's watchdog timer is configured to use the long watchdog timer period and generate a watchdog error event upon expiration.

EPC-5 and VXLink hardware do not include watchdog timer support. Attempting to use the function on an EPC-5 or VXLink interface results in an **EPC_INV_CFG** error.

Return Value

The function returns a Bus Management return value:

EPC_INV_CFG	The parameter *WatchdogCfg* is invalid.
EPC_INV_SESSION	The specified *Session_ID* is invalid.
EPC_INV_SW	The BusManager device driver is not present or there is a revision mismatch between the Bus Management Library and the BusManager VxD.
EPC_SUCCESS	The function completed successfully.

See Also

EpcOpenSession, EpcSetEventEnableMask, EpcSetEventHandler.

Example

```
/*
 * Copyright 1994 by RadiSys Corporation.  All rights reserved.
 */

/*
 * watchdog.c -- Bus Management Library EPC watchdog timer functions sample
 *               code.
 */

#include "busmgr.h"

/*
 * FUNCTION PROTOTYPES...
 */

short FAR
WatchdogSample(void);

int FAR
WinPrintf(char FAR *Format_Ptr, ...);

/*
 * CODE...
 */

short FAR
WatchdogSample(void)
{
    char                  err_string[ERROR_STRING_SZ];
    short                 err_num;
    unsigned long         event_data;
    unsigned long         event_mask;
    unsigned long         session_id;
    struct EpcEnvironment environment;

    /*
     * Verify the EPConnect environment.
     */

    if ((err_num = EpcVerifyEnvironment(&environment)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcVerifyEnvironment() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * Open a session.
     */

    if ((err_num = EpcOpenSession(&session_id)) != EPC_SUCCESS)
    {
        EpcGetErrorString(err_num, err_string);
        WinPrintf("FAILURE:  EpcOpenSession() error == %s (%d).\n",
                  err_string,
                  err_num);
        return (err_num);
    }

    /*
     * If watchdog timer functionality is supported on this EPC, configure the
```

```
     * EPC to generate a watchdog timer error event using a short timeout, then
     * wait for up to one second (1000 ms) for the event to occur.
     */

    if ((environment.WatchdogCfg & EPC_WDT_FAST_ERROR) != 0 &&
        (environment.WatchdogCfg & EPC_WDT_SLOW_ERROR) != 0  )
    {
        EpcWatchdogTimer(session_id, EPC_WDT_FAST_ERROR);
        EpcWaitForEvent(session_id,
                        1000,
                        EPC_WATCHDOG_ERR,
                        &event_mask,
                        &event_data);
        if ((event_mask & EPC_WATCHDOG_ERR) == 0)
        {
            WinPrintf("Watchdog timer event DID NOT occurred.\n");
        }
        else
        {
            WinPrintf("Watchdog timer event occurred.\n");
        }
        EpcWatchdogTimer(session_id, EPC_WDT_SLOW_ERROR);
    }

    /*
     * Close the session and return.
     */

    EpcCloseSession(session_id);
    WinPrintf("SUCCESS:  WatchdogSample() complete.\n");
    return (EPC_SUCCESS);
}
```

3. On-Line Resource Manager

3

EPConnect provides an On-Line Resource Manager (OLRM) interface for querying configuration and state information about an instrument system and devices within that instrument system. Configuration information is typically static (i.e., established by a Start-Up Resource Manager (SURM) at system initialization time, and not changed thereafter). Device state information is typically dynamic (i.e., reflects the run-time state changes of a device as it is used by an executing application).

The following OLRM functions are available:

Function	Description
OlrmGetArbAttr	Queries an arbitrary string attribute.
OlrmGetBoolAttr	Queries a boolean attribute.
OlrmGetNmByGPA	Queries the device name corresponding to a GPIB address.
OlrmGetNmByNA	Queries the device name corresponding to a network address.
OlrmGetNmByULA	Queries the device name corresponding to a VXIbus unique logical address.
OlrmGetNumAttr	Queries a numeric attribute.
OlrmGetStrAttr	Queries a string attribute.

To use OLRM to query a specific device's attributes, an application must first know the name of the device. An application can use **OlrmGetNmByGPA**, **OlrmGetNmByNA**, or **OlrmGetNmByULA** to query a device's name from its address. Once the application knows the device's name, it can use **OlrmGetArbAttr**, **OlrmGetBoolAttr**, **OlrmGetNumAttr**, and **OlrmGetStrAttr** to query individual device attributes.

3

3.1 Functions By Name

This section contains an alphabetical listing of the OLRM library functions. Each listing describes the function, gives its invocation sequence and arguments, discusses its operation, and lists its returned values. Where usage of the function may not be clear, an example with comments is given.

OlrmGetArbAttr

Description Queries an arbitrary string attribute.

C Synopsis

```
#include "busmgr.h"
#include "olrm.h"

short
OlrmGetArbAttr(charFAR * Name_Ptr,
               charFAR *  Arb_Attribute_Ptr,
               charFAR *  Arb_Result_Ptr);
```

Name_Ptr	*Name_Ptr* specifies a device name.
Arb_Attribute_Ptr	*Arb_Attribute_Ptr* specifies an arbitrary string attribute.
Arb_Result_Ptr	*Arb_Result_Ptr* specifies the location of a buffer where the specified string attribute will be placed.

Visual Basic Synopsis

```
Declare Function
OlrmGetNmByGPA% Lib "olrmw16.lib"
                              (ByVal Primary%,
                              ByVal Secondary%
                              ByVal Name_Ptr$)
```

Remarks **OlrmGetArbAttr** queries an arbitrary string attribute of the specified device and places the result in the buffer pointed to by *Arb_Result_Ptr*. The function allows an application to obtain attribute information that is not accessible via the standard set of integer search keys, particularly attribute information about non-GPIB/non-VME devices.

Name_Ptr is a null-terminated ASCII string specifying a device name.

Arb_Attribute_Ptr is a null-terminated ASCII string specifying the string attribute to query.

Arb_Result_Ptr specifies the location a buffer where the function places the result of the arbitrary string attribute query. The buffer must be at least **ATTRIBUTE_SZ** bytes long.

The result of an arbitrary string attribute query is always a null-terminated ASCII character string. Depending on the specified string attribute, the result string may represent a decimal number, a hexadecimal number, a binary number, a bit mask, or a string of characters. It is the responsibility of the application to interpret the result string.

Return Value The function returns a Bus Management return value:

EPC_INV_ATTR	The parameter *Arb_Attribute_Ptr* is invalid.
EPC_INV_NAME	A device with the specified name does not exist.
EPC_INV_PTR	One or more of the parameters *Name_Ptr* and *Arb_Result_Ptr* is invalid.
EPC_OS_ERROR	The resource manager database file could not be read.
EPC_SUCCESS	The function completed successfully.

See Also **OlrmGetBoolAttr**, **OlrmGetNumAttr**, **OlrmGetStrAttr**.

OlrmGetBoolAttr

Description Queries a boolean attribute.

C Synopsis

```
#include "busmgr.h"
#include "olrm.h"

short
OlrmGetBoolAttr(char FAR * Name_Ptr,
                short Bool_Attribute,
                unsigned short FAR * Bool_Result_Ptr);
```

Name_Ptr	*Name_Ptr* specifies a device name.
Bool_Attribute	*Bool_Attribute* specifies a boolean attribute.
Bool_Result_Ptr	*Bool_Result_Ptr* specifies a location where the specified boolean attribute will be placed.

Visual Basic Synopsis

```
Declare Function
OlrmGetBoolAttr% Lib "olrmw16.dll"
                 (ByVal Name_Ptr$$,
                 (ByVal Bool_Attribute%
                 Bool_Result_Ptr%)
```

Remarks **OlrmGetBoolAttr** queries a boolean attribute of the specified device and places the result in the location pointed to by *Bool_Result_Ptr*.

Name_Ptr is a null-terminated ASCII string specifying a device name.

Bool_Attribute specifies the boolean attribute to query. Valid values for VXIbus devices are:

Bool_Attribute	**Bool_Result_Ptr*
VXI_ISMEMACT	Memory active (accessible).
VXI_ISMODID	MODID line asserted.
VXI_ISREADY	Ready for normal operation.
VXI_ISPASSED	Passed self test.
VXI_MEM_ISNONPRIV	Non-privileged access capability (memory devices only).
VXI_MEM_ISBLKTR	Block transfer capability (memory devices only).
VXI_MEM_ISNONVOL	Non-volatile RAM memory (memory devices only).
VXI_MEM_ISELPROG	EPROM memory (memory devices only).
VXI_MEM_ISD32TR	D32 transfer capability (memory devices only).
VXI_MPR_ISCMDR	Commander capability (message-based devices only).
VXI_MPR_ISSIG	Signal register present (message-based devices only).
VXI_MPR_ISMSTR	VXIbus master capability (message-based devices only).
VXI_MPR_ISINTR	Interrupter capability (message-based devices only).
VXI_MPR_ISFHS	FHS capability (message-based devices only).
VXI_MPR_ISSHMEM	Shared memory protocol capability (message-based devices only).
VXI_MRP_ISRG	Response generation capability (message-based devices only).

VXI_MRP_ISEG	Event generation capability (message-based devices only).
VXI_MRP_ISPI	Programmable interrupter capability (message-based devices only).
VXI_MRP_ISPH	Programmable handler capability (message-based devices only).
VXI_MRP_ISTRG	Supports word serial *Trigger* command (message-based devices only).
VXI_MRP_ISI4	Supports IEEE 488.2 instrument protocol (message-based devices only).
VXI_MRP_ISINST	Supports VXIbus instrument protocol (message-based devices only).
VXI_MRP_ISELW	Extended long word serial capability (message-based devices only).
VXI_MRP_ISLW	Long word serial capability (message-based devices only).
VXI_MRR_ISDOR	DOR asserted (message-based devices only).
VXI_MRR_ISDIR	DIR asserted (message-based devices only).
VXI_MRR_ISERR	Word serial protocol error detected (message-based devices only).
VXI_MRR_ISRRDY	Read Ready asserted (message-based devices only).
VXI_MRR_ISWRDY	Write Ready asserted (message-based devices only).
VXI_MRR_ISFHSACT	FHS protocol active (message-based devices only).

VXI_MRR_ISLOCKED	Local lockout active (message-based devices only).

No valid *Bool_Attribute* values are defined for GPIB network devices.

If the requested boolean attribute is false, the function places a zero at the location pointed to by *Bool_Result_Ptr*. Otherwise, the function places a non-zero value at the location.

Return Value The function returns a Bus Management return value:

EPC_INV_ATTR	The parameter *Bool_Attribute* is invalid.
EPC_INV_NAME	A device with the specified name does not exist.
EPC_INV_PTR	One or more of the parameters *Name_Ptr* and *Bool_Result_Ptr* is invalid.
EPC_NO_DATA	The resource management database does not contain the requested attribute.
EPC_SUCCESS	The function completed successfully.

See Also **OlrmGetArbAttr**, **OlrmGetNumAttr**, **OlrmGetStrAttr**.

OlrmGetNmByGPA

Description Queries the device name corresponding to a GPIB address.

C Synopsis

```
#include "busmgr.h"
#include "olrm.h"

short
OlrmGetNmByGPA(short Primary, short Secondary, char FAR
*Name_Ptr);
```

Primary	*Primary* specifies a GPIB primary address.
Secondary	*Secondary* specifies a GPIB secondary address.
Name_Ptr	*Name_Ptr* specifies the location of a buffer where the name of the device corresponding to the specified GPIB address will be placed.

Visual Basic Synopsis

```
Declare Function
OlrmGetNmByGPA% Lib "olrmw16.dll"
                    (ByVal Primary%,
                    ByVal Secondary%,
                    ByVal Name_Ptr$)
```

Remarks **OlrmGetNmByGPA** queries the name of the device corresponding to the specified GPIB address and places the name in the buffer pointed to by *Name_Ptr*.

Primary specifies a GPIB primary address. Valid values are 0 through 30, inclusive.

Secondary specifies a GPIB secondary address. Valid values are -1 and 0 through 30, inclusive. If *Secondary* is -1, the function searches for a GPIB device with the specified primary address and no secondary address. Otherwise, the function searches for a GPIB device with the specified primary address and the specified secondary address.

Name_Ptr specifies the location of a buffer where a device's name will be placed. The buffer must be at least **DEVNAME_SZ** byte long.

Return Value The function returns a Bus Management return value:

EPC_INV_GPA	One or more of the parameters *Primary* and *Secondary* is invalid.
EPC_INV_PTR	The parameter *Name_Ptr* is invalid.
EPC_NO_DEVICE	A device corresponding to the specified GPIB address does not exist.
EPC_SUCCESS	The function completed successfully.

See Also **OlrmGetNmByNA, OlrmGetNmByULA.**

OlrmGetNmByNA

Description Queries the device name corresponding to a network address.

C Synopsis

```
#include "busmgr.h"
#include "olrm.h"

short
OlrmGetNmByNA(char FAR *Net_Address_Ptr, char FAR
*Name_Ptr);
```

Net_Address_Ptr	*Net_Address_Ptr* specifies a network address string.
Name_Ptr	*Name_Ptr* specifies the location of a buffer where the name of the device corresponding to the specified network address will be placed.

Visual Basic Synopsis

```
Declare Function
OlrmGetNmByNA% Lib "olrmw16.dll"
                    (ByVal Net_Address_Ptr$,
                    ByVal Name_Ptr$)
```

Remarks **OlrmGetNmByNA** queries the name of the device corresponding to the specified network address and places the name in the buffer pointed to by *Name_Ptr*.

Net_Address_Ptr specifies the location of a null-terminated ASCII string representing a network address.

Name_Ptr specifies the location of a buffer where a device's name will be placed. The buffer must be at least **DEVNAME_SZ** byte long.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	One or more of the parameters *Net_Address_Ptr* and *Name_Ptr* is invalid.
EPC_NO_DEVICE	A device corresponding to the specified network address does not exist.
EPC_SUCCESS	The function completed successfully.

See Also **OlrmGetNmByGPA, OlrmGetNmByULA.**

OlrmGetNmByULA

Description Queries the device name corresponding to a VXIbus unique logical address.

C Synopsis

```
#include "busmgr.h"
#include "olrm.h"

short
OlrmGetNmByULA(unsigned short ULA, char FAR *Name_Ptr);
```

ULA	*ULA* specifies a VXIbus unique logical address.
Name_Ptr	*Name_Ptr* specifies the location of a buffer where the name of the device corresponding to the specified VXIbus unique logical address will be placed.

Visual Basic Synopsis

```
Declare Function
OlrmGetNmByULA% Lib "olrmw16.dll"
                    (ByVal ULA%,
                    ByVal Name_Ptr$)
```

Remarks **OlrmGetNmByULA** queries the name of the device corresponding to the specified VXIbus unique logical address and places the name in the buffer pointed to by *Name_Ptr*.

ULA specifies a VXIbus unique logical address. Valid values are 0 through 255, inclusive.

Name_Ptr specifies the location of a buffer where a device's name will be placed. The buffer must be at least **DEVNAME_SZ** byte long.

Return Value The function returns a Bus Management return value:

EPC_INV_PTR	The parameter *Name_Ptr* is invalid.
EPC_NO_DEVICE	A device corresponding to the specified VMEbus unique logical address does not exist.
EPC_SUCCESS	The function completed successfully.

See Also **OlrmGetNmByGPA, OlrmGetNmByNA.**

OlrmGetNumAttr

Description Queries a numeric attribute.

C Synopsis

```
#include "busmgr.h"
#include "olrm.h"

short
OlrmGetNumAttr(char FAR * Name_Ptr,
               short Num_Attribute,
               unsigned long FAR * Num_Result_Ptr);
```

Name_Ptr	*Name_Ptr* specifies a device name.
Num_Attribute	*Num_Attribute* specifies a numeric attribute.
Num_Result_Ptr	*Num_Result_Ptr* specifies a location where the specified numeric attribute will be placed.

Visual Basic Synopsis

```
Declare Function
OlrmGetNumAttr% Lib "olrmw16.dll"
                (ByVal Name_Ptr$$,
                ByVal Num_Attribute%,
                ByVal Num_Result_Ptr&)
```

Remarks **OlrmGetNumAttr** queries a numeric attribute of the specified device and places the result in the location pointed to by *Num_Result_Ptr*.

Name_Ptr is a null-terminated ASCII string specifying a device name.

Num_Attribute specifies the numeric attribute to query. Valid values for VXIbus devices are:

Num_Attribute	**Num_Result_Ptr*
VXI_ULA	Unique logical address.
VXI_IDREG	ID register.
VXI_DTREG	Device type register.
VXI_STREG	Status register.
VXI_OFFREG	Offset register.
VXI_MNFID	Manufacturer ID.
VXI_MODCOD	Model code.
VXI_DEVCLASS	Device class. 0=memory, 1=extended, 2=message-based, 3=register-based.
VXI_ADRSP	Address space. 0=A16/A24, 1=A16/A32, 3=A16 only.
VXI_A16BASE	A16 memory base.
VXI_A24BASE	A24 memory base (A16/A24 devices only).
VXI_A24SIZE	A24 memory size, in bytes (A16/A24 devices only).
VXI_A32BASE	A32 memory base (A16/A32 devices only).
VXI_A32SIZE	A32 memory size, in bytes (A16/A32 devices only).
VXI_STATDD	Status register device dependent bits. Defined and reserved bits are masked to zero.
VXI_BUSNUM	Bus mainframe number.
VXI_SLOTNUM	Slot number.
VXI_CMDRLA	Unique logical address of commander device.

VXI_S0LA	Unique logical address of slot-0 device.
VXI_SVARSZ	Servant area size.
VXI_HDLRMAP1 ... VXI_HDLRMAP7	Interrupt line assigned to handlers 1 through 7. A zero result indicates the handler is not assigned for the device.
VXI_INTRMAP1 ... VXI_INTRMAP7	Interrupt line assigned to interrupters 1 through 7. A zero result indicates the interrupter is not assigned for the device.
VXI_MEM_ATTREG	Attribute register (memory devices only).
VXI_MEM_TYPE	Memory type (memory devices only). 1 = ROM, 2 = other, 3 = RAM.
VXI_MEM_SPEED	Minimum memory access time, in nanoseconds (memory devices only).
VXI_MEM_DD	Attribute register device dependent bits (memory devices only). Defined and reserved bits are masked to zero.
VXI_SBC_REG	Subclass register (extended devices only)
VXI_SBC_RES	Reserved subclass ID (extended devices only).
VXI_SBC_MNFID	Subclass manufacturer ID (extended devices only).
VXI_SBC_MFSBC	Manufacturer subclass (extended devices only).
VXI_MSG_PTOREG	Protocol register (message-based devices only).
VXI_MSG_RSPREG	Response register (message-based devices only).
VXI_MSG_DHIREG	Data high register (message-based devices only). Warning: querying this register may modify device state.

VXI_MSG_DLOREG	Data low register (message-based devices only). Warning: querying this register may modify device state.
VXI_MSG_PRDD	Protocol register device dependent bits (message-based devices only). Defined and reserved bits are masked to zero.
VXI_MSG_RDPR	Word serial *Read Protocol* command response (message-based devices only).
VXI_MSG_RDPRDD	Word serial *Read Protocol* command response device dependent bits (message-based devices only) Defined and reserved bits are masked to zero.
VXI_MSG_RRDD	Response register device dependent bits (message-based devices only) Defined and reserved bits are masked to zero.

No valid *Num_Attribute* values are defined for GPIB network devices.

Return Value The function returns a Bus Management return value:

EPC_INV_ATTR	The parameter *Num_Attribute* is invalid.
EPC_INV_NAME	A device with the specified name does not exist.
EPC_INV_PTR	One or more of the parameters *Name_Ptr* and *Num_Result_Ptr* is invalid.
EPC_NO_DATA	The resource management database does not contain the requested attribute.
EPC_SUCCESS	The function completed successfully.

See Also **OlrmGetArbAttr, OlrmGetBoolAttr, OlrmGetStrAttr.**

OlrmGetStrAttr

Description Queries a string attribute.

C Synopsis

```
#include "busmgr.h"
#include "olrm.h"

short
OlrmGetStrAttr(char FAR * Name_Ptr,
               short Str_Attribute,
               char FAR * Str_Result_Ptr);
```

Name_Ptr	*Name_Ptr* specifies a device name.
Str_Attribute	*Str_Attribute* specifies a string attribute.
Str_Result_Ptr	*Str_Result_Ptr* specifies the location of a buffer where the specified string attribute will be placed.

Visual Basic Synopsis

```
Declare Function
OlrmGetStrAttr% Lib "olrmw16.dll"
                (ByVal Name_Ptr$,
                ByVal Str_Attribute%,
                ByVal Str_Result_Ptr$)
```

Remarks **OlrmGetStrAttr** queries a string attribute of the specified device and places the result in the buffer pointed to by *Str_Result_Ptr*.

Name_Ptr is a null-terminated ASCII string specifying a device name.

Str_Attribute specifies the string attribute to query. Valid values for VXIbus devices are:

Str_Attribute	**Str_Result_Ptr*
VXI_CMDRNM	Name of commander device.
VXI_MFNM	Manufacturer name.
VXI_MODNM	Model name.
VXI_S0NM	Name of slot-0 device.

Valid *Str_Attribute* values for network devices are:

Str_Attribute	**Str_Result_Ptr*
NET_ADDRESS	Network address.

No valid *Str_Attribute* values are defined for GPIB devices.

Str_Result_Ptr specifies the location a buffer where the function places the result of the string attribute query. The buffer must be at least **ATTRIBUTE_SZ** bytes long.

Return Value The function returns a Bus Management return value:

EPC_INV_ATTR	The parameter *Str_Attribute* is invalid.
EPC_INV_NAME	A device with the specified name does not exist.
EPC_INV_PTR	One or more of the parameters *Name_Ptr* and *Str_Result_Ptr* is invalid.
EPC_NO_DATA	The resource management database does not contain the requested attribute.
EPC_SUCCESS	The function completed successfully.

See Also **OlrmGetArbAttr, OlrmGetBoolAttr, OlrmGetNumAttr.**

NOTES

4. Advanced Topics

This chapter discusses topics of interest to advanced application programmers. Topics include:

- Byte Ordering and Data Representation
- Handler Operations
- Event Handler Execution Under Windows
- Event Handler Implementation
- TTL Trigger Interrupt Handling on an EPC-7
- Using the backward-compatibility library

4.1 Byte Ordering and Data Representation

Byte ordering adds complexity to the VXIbus interface. Many VXIbus devices use the data formats of Motorola microprocessors. Others, including RadiSys EPC controllers, use the data format of Intel microprocessors. Although the Motorola and Intel microprocessors use the same data types, the hardware representations of these data types differ.

Figure 4-1 shows how the same sequence of bytes in memory is interpreted by Intel and Motorola microprocessors. Memory value 11 is at the lowest address and memory value AA is at the highest address. The data widths shown correspond to the data operand sizes found on both microprocessors.

Memory Value	Intel Order	Data Width	Motorola Order
11	11	8 bits	11
22	2211	16 bits	1122
33			
44	44332211	32 bits	11223344
55			
66	665544332211	48 bits	112233445566
77			
88	8877665544332211	64 bits	1122334455667788
99			
AA	AA99887766554433221l	80 bits	1122334455667788AA

Figure 4-1. Byte Order Example.

4.1.1 Byte Swapping Functions

The **EpcSwap*** functions convert 16-bit, 32-bit, 48-bit, 64-bit and 80-bit data between Intel and Motorola byte orders (8-bit data does not require conversion).

4.1.2 Correcting Data Structure Byte Ordering

Even if byte-swapping occurs during a transfer, byte ordering problems occur when data is copied between Motorola and Intel memory using a different data width than the width of the operand itself. This situation occurs when a data structure containing mixed-type fields is copied in a single operation.

The following code fragment illustrates how to use the **EpcSwap*** functions to correct the byte order in the local copy of the data structure:

```
struct DataStructure
{
        char        Field8;
        char        Field16[2];
        char        Field32[4];
        char        Field48[6];
        char        Field64[8];
        char        Field80[10];
} data;

/* Copy the data structure to EPC memory from the VXIbus. */

(void) EpcCopyData(Session_ID,
                   (void HUGE *) Mapped_Ptr,
                   (void HUGE *) &data,
                   sizeof(struct DataStructure),
                   EPC_8_BIT,
                   &Actual_Size);

/* Byte-swap the individual structure fields (data.Field8 */
/* is an 8-bit field, so it is already correct).          */

(void) EpcSwap16((unsigned short FAR  *) data.Field16);
(void) EpcSwap32((unsigned long FAR *) data.Field32);
(void) EpcSwap48((void FAR *) data.Field48);
(void) EpcSwap64((void FAR *) data.Field64);
(void) EpcSwap80((void FAR *) data.Field80);
```

In the above example, the data structure was copied from VXIbus memory one byte at a time. To copy data from EPC memory to Motorola-ordered VXIbus memory, byte-swap the fields of the structure in local memory (using the above byte swapping functions) and copy the data using the **EpcCopyData** function.

It is sometimes more efficient to copy blocks of data using data transfer width greater than the expected data width. If you use a greater data transfer width to copy data structures containing mixed-type fields to/from Motorola-order memory, do not use the EPC's hardware byte-swapping feature. Swap the data structure fields individually.

4.2 Event Handler Execution

These conditions must be true before an application's event handlers can execute:

- The application must use **EpcSetEventHandler** to install an error handler.
- The application must call **EpcSetEventEnableMask** to enable event reception.
- An event must occur.

The Bus Management API discards all events that occur before the application installs an event handler.

When an application installs an event handler and enables event reception, the event handler processes events as soon as they are received. The installed event handler executes as part of an interrupt thread, with virtual processor interrupts disabled, and using the installed event handler stack.

4.3 Event Handler Operations Under Windows

Event handlers can execute as part of an interrupt thread under Windows. This feature implies that an event handler can only call fully reentrant Bus Management, "C" library, Windows, DOS, and BIOS support functions.

Bus Management Library and ORLM library functions are fully reentrant and may be called from an event handler or any application code that executes as part of an interrupt thread. Note, however, that when called reentrantly, EPConnect Bus Management Library functions cannot yield the processor to other tasks while in a timing loop.

The following "C" library functions are reentrant under Microsoft "C" Version 6.0, and may be called from an event handler or any application code that executes as part of an interrupt thread (it is likely that this list is different for other releases of the Microsoft "C" compiler and for compilers from other vendors):

abs	**memccpy**	**strct**	**strnset**
atoi	**memchr**	**strchr**	**strrchr**
atol	**memcmp**	**strcmp**	**strrev**
bsearch	**memcpy**	**strcmpi**	**strset**
chdir	**memicmp**	**strcpy**	**strstr**
getpid	**memmove**	**stricmp**	**strupr**
halloc	**memset**	**strlen**	**swab**
hfree	**mkdir**	**strlwr**	**tolower**
itoa	**movedata**	**strncat**	**toupper**
labs	**putch**	**strncmp**	
lfind	**rmdir**	**strncpy**	
lsearch	**segread**	**strnicmp**	

No Windows functions are fully reentrant. As such, none of the Windows functions should be called from an event handler.

Not all DOS and BIOS functions are fully reentrant. However, mechanisms exist (the "InDos" and "CriticalError" flags) for avoiding DOS reentrancy by delaying background processing until DOS is not in use.

4.4 Event Handler Implementation

An event handler is called as part of an interrupt thread with its own stack. Care must be taken during implementation to avoid several pitfalls.

Since an event handler function is called as part of an interrupt thread, the event handler function must reload the data segment register (DS) with its data segment upon entry. Any of the following three methods will correctly load the data segment register for an event handler function:

1. Explicitly declare the event handler function to be an exported function in the EXPORTS section of the application's module definition (.DEF) file.

2. Explicitly declare the event handler function to be an exported function using the "C" language "_export" function declaration.

3. Explicitly declare that the event handler function reloads the data segment register upon entry using the "C" language "_loadds" function declaration.

Since an event handler function is called using the installed event handler stack, an event handler function written in "C" must be compiled with the assumption that the data segment is not equivalent to the stack segment (DS != SS). Otherwise, a catastrophic failure can occur when the event handler function is called. For Microsoft compilers, use the "/Alfw" memory model parameter. For Borland compilers, use the "-ml" memory model parameter.

Since an event handler function is called using the installed event handler stack, an event handler function written in "C" must be compiled with automatic stack checking disabled. Otherwise, a catastrophic failure will occur when the event handler is called. For Microsoft compilers, use the "/Gs" parameter. For Borland compilers, avoid using the "-N" parameter.

Since an event handler function is generally performance-critical, its code and data segments should be carefully defined for maximum performance. For maximum performance, define the event handler function's code and data segments to be PRELOAD, FIXED, and NONDISCARDABLE in the application's module definition (.DEF) file.

4.5 TTL Trigger Interrupts on an EPC-7

Receiving and processing TL trigger interrupts on an EPC-7 requires software intervention. EPC-7 hardware generates a TTL trigger interrupt when all of the following conditions are true:

- A bit in the TTL trigger interrupt enable register is set. The Bus Management Library function **EpcSetEventEnableMask** sets and/or clears the register's bits.
- The corresponding bit in the TTL trigger latch register is clear.
- The corresponding TTL trigger line is asserted for at least 30 nanoseconds.

The main complication in this scenario is that a bit in the TTL trigger latch register cannot be cleared until the corresponding TTL trigger line is deasserted. In order to clear a bit in the register, the register must be read while the corresponding TTL trigger line is deasserted. TTL trigger line assertion is not necessarily under EPC control.

The operation of the EPC-7 TTL trigger latch register has three potential side effects for software:

- If a TTL trigger interrupt remains enabled after receiving the initial interrupt and clearing the TTL trigger latch register, the CPU can be monopolized by redundant TTL trigger interrupts.
- If a TTL trigger latch register bit is not cleared before enabling the corresponding TTL trigger interrupt, it is possible to receive an interrupt for a TTL trigger that was asserted, latched, and deasserted long before the TTL trigger interrupt was enabled.
- If a TTL trigger latch register bit is not cleared after receiving the corresponding TTL trigger interrupt, the EPC will not latch subsequent TTL trigger line assertions and, therefore, will miss subsequent TTL trigger interrupts.

To avoid the first side effect, the Bus Management for Windows implementation globally disables a TTL trigger interrupt upon reception. In addition, the Bus Manager Library implementation provides sufficient functionality to avoid the other two side effects.

To avoid the side effect of receiving extraneous TTL trigger interrupts, execute **EpcGetMiscAttributes** before calling **EpcGetEventEnableMask** and **EpcSetEventEnableMask** to enable TTL trigger interrupts for a session.

For example:


```
void FAR PASCAL
EnableTTLTriggerInterrupts(unsigned long Session_ID, unsigned
long Event_Mask)
{
       unsigned long mask1;
       unsigned long mask2;

        /*
         * Wait for corresponding TTL trigger latch register
         * bits to clear, then enable the TTL trigger
         * interrupts.
         */

       mask1 = Event_Mask << 4;
       for (;;)
       {
              EpcGetMiscAttributes(Session_ID, &mask2);
              if ((mask1 & mask2) == 0)
              {
                     break;
              }
       }
       EpcGetEventEnableMask(Session_ID, &mask1);
       EpcSetEventEnableMask(Session_ID, mask1 | Event_Mask);
}
```

To avoid the side effect of missing multiple TTL trigger interrupts from the same TTL trigger, re-enable the interrupt immediately after receiving a TTL trigger interrupt, preferably as part of the event handler function itself. For example:

```
       void FAR
       TTLTriggerInterruptHandler(  unsigned long Session_ID,
                                    unsigned long Event_Mask,
                                    unsigned long Event_Data)
       {
            /*
             *Re-enable the TTL trigger interrupt.
```

```
        */
          EnableTTLTriggerInterrupts(Session_ID,Event_Mask;
        /*
         * Execute other event handler tasks...
         */
}
```

4.6 Backward-Compatibility Library

The Backward Compatibility Library (**EPCDICW.DLL**) and its corresponding import library (**EPCDICW.LIB**) provide a level of compatibility between the Windows and DOS programming interfaces. Most of the functions available in the DOS Bus Management Library are available in the Windows Backward-Compatibility Library with identical calling conventions. However, the functionality the two libraries provide is not strictly identical. Differences between the Windows Backward-Compatibility Library and the DOS Bus Management Library include the following:

- No Message Delivery System (MDS) functionality is available in the Windows Backward-Compatibility Library. If using MDS support under Windows, the application should be ported to SICL. If MDS support under Windows is a **requirement**, users should not upgrade beyond EPConnect/VXI for DOS version 3.11.

- The Windows Backward Compatibility Library supports **A16S**, **A24SD**, **A24S**, **A32SD**, and **A32S** VMEbus address modifiers only. Attempting to use other VMEbus address modifiers (**A16N**, **A16U**, **A24ND**, **A24NP**, **A24N**, **A24U**, **A24SP**, **A32ND**, **A32NP**, **A32N**, **A32U** and **A32SP**) under Windows results in an **ERR_FAIL** error and/or a null mapped pointer.

- The Windows Backward Compatibility Library can access the first gigabyte of A32 space only (addresses 0x00000000 through 0x3FFFFFFF). Attempting to map higher A32 space addresses under Windows results in an **ERR_FAIL** error and/or a null mapped pointer.

- The Windows Backward Compatibility Library automatically disables persistent interrupt and error events when they are received. Automatic disabling of persistent events prevents the generation of multiple,

redundant events. Persistent events include **EPC_MSG_INTR**, **EPC_TTL_TRIG*_INTR**, **EPC_SYSFAIL_ERR**, **EPC_ACFAIL_ERR**, and **EPC_WATCHDOG_ERR**. Under Windows, when one of these persistent events occurs, it must be re-enabled by the application before it will be received again.

- The Windows Backward Compatibility Library supports standard servant word serial communications only. The RadiSys-specific protocol for multiple-commander word serial communications is not supported. Under Windows, the multiple-commander arming codes (**EPC_WSRCV_DISARM**, **EPC_WSRCV_ARM**, and **EPC_WSRCV_ARMandENABLE**) and the single-commander arming codes (**EPC_WSRCV_FDISARM**, **EPC_WSRCV_FARM**, and **EPC_WSRCV_FARMandENABLE**) are equivalent.

5. Support and Service

5.1 In North America

5.1.1 Technical Support

RadiSys maintains a technical support phone line at (503) 646-1800 that is staffed weekdays (except holidays) between 8 AM and 5 PM Pacific time. If you have a problem outside these hours, you can leave a message on voice-mail using the same phone number. You can also request help via electronic mail or by FAX addressed to RadiSys Technical Support. The RadiSys FAX number is (503) 646-1850. The RadiSys E-mail address on the Internet is *support@radisys.com*. If you are sending E-mail or a FAX, please include information on both the hardware and software being used and a detailed description of the problem, specifically how the problem can be reproduced. We will respond by E-mail, phone or FAX by the next business day.

Technical Support Services are designed for customers who have purchased their products from RadiSys or a sales representative. If your RadiSys product is part of a piece of OEM equipment, or was integrated by someone else as part of a system, support will be better provided by the OEM or system vendor that did the integration and understands the final product and environment.

6.1.2 Bulletin Board

RadiSys operates an electronic bulletin board (BBS) 24 hours per day to provide access to the latest drivers, software updates and other information. The bulletin board is not monitored regularly, so if you need a fast response please use the telephone or FAX numbers listed above.

The BBS operates at up to 14400 baud. Connect using standard settings of eight data bits, no parity, and one stop bit (8, N, 1). The telephone number is (503) 646-8290.

5.2 Other Countries

Contact the sales organization from which you purchased your RadiSys product for service and support.

Index

I

NOTES

SICL for Windows

Programmer's

Reference Guide

RadiSys® Corporation

15025 S.W. Koll Parkway

Beaverton, OR 97006

Phone: (503) 646-1800

FAX: (503) 646-1850

07-0223-01

December 1994

May 1994

Software License and Warranty

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THE DISKETTE OR DISK UNIT PACKAGE. BY OPENING THE PACKAGE, YOU INDICATE THAT YOU ACCEPT THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, YOU SHOULD PROMPTLY RETURN THE UNOPENED PACKAGE, AND YOU WILL BE REFUNDED.

LICENSE

You may:

1. Use the product on a single computer;
2. Copy the product into any machine-readable or printed form for backup or modification purposes in support of your use of the product on a single computer;
3. Modify the product or merge it into another program for your use on the single computer—any portion of this product merged into another program will continue to be subject to the terms and conditions of this agreement;
4. Transfer the product and license to another party if the other party agrees to accept the terms and conditions of this agreement—if you transfer the product, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copy not transferred, including all modified versions and portions of the product contained in or merged into other programs.

You must reproduce and include the copyright notice on any copy, modification, or portion merged into another program.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PRODUCT OR ANY COPY, MODIFICATION, OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION, OR MERGED PORTION OF THE PRODUCT TO ANOTHER PARTY, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

TERM

The license is effective until terminated. You may terminate it at any time by destroying the product and all copies, modifications, and merged portions in any form. The license will also terminate upon conditions set forth elsewhere in this agreement or if you fail to comply with any of the terms or conditions of this agreement. You agree upon such termination to destroy the product and all copies, modifications, and merged portions in any form.

LIMITED WARRANTY

RadiSys Corporation ("RadiSys") warrants that the product will perform in substantial compliance with the documentation provided. However, RadiSys does not warrant that the functions contained in the product will meet your requirements or that the operation of the product will be uninterrupted or error-free.

RadiSys warrants the diskette(s) on which the product is furnished to be free of defects in materials and workmanship under normal use for a period of ninety (90) days from the date of shipment to you.

LIMITATIONS OF REMEDIES

RadiSys' entire liability shall be the replacement of any diskette that does not meet RadiSys' limited warranty (above) and that is returned to RadiSys.

IN NO EVENT WILL RADISYS BE LIABLE FOR ANY DAMAGES, INCLUDING LOST PROFITS OR SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THE PRODUCT EVEN IF RADISYS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

GENERAL

You may not sublicense the product or assign or transfer the license, except as expressly provided for in this agreement. Any attempt to otherwise sublicense, assign, or transfer any of the rights, duties, or obligations hereunder is void.

This agreement will be governed by the laws of the state of Oregon.

If you have any questions regarding this agreement, please contact RadiSys by writing to RadiSys Corporation, 15025 SW Koll Parkway, Beaverton, Oregon 97006.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATION BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

NOTES

Table of Contents

NOTES

1. Introducing SICL for Windows

This manual is intended for programmers using the SICL for Windows programming interface to develop enhanced mode Windows applications that control I/O modules via the VXI or GPIB interfaces on an EPC or on a PC with a VXLink card. You are expected to have read the *EPConnect/VXI for DOS & Windows User's Guide* for an understanding of what is in EPConnect/VXI, how to configure it with Windows, and how to use the Start-Up Resource Manager (SURM). You are not expected to have in-depth knowledge of Windows.

SICL for Windows is designed to execute under enhanced mode Windows only. It will not execute properly under Windows standard mode. It is also designed to execute on EPC-7, EPC-8 or VXLink hardware. It will not execute properly on an EPC-2.

This chapter introduces you to the RadiSys® Standard Instrument Control Library (SICL) for Windows. In it you will find the following:

- What is in this manual and how to use it
- What is SICL for Windows?
- Programming, Compiling and Linking
- What to do next

1

1.1 How This Manual is Organized

This manual has five chapters:

Chapter 1, *Introduction*, introduces SICL for Windows and this manual.

Chapter 2, *Function Descriptions*, describes the major categories of SICL functions and gives complete descriptions of each SICL function. The function descriptions also contain supporting examples or references to an example that demonstrates use of the function. Function descriptions are alphabetic by function name.

Chapter 3, *Advanced Topics*, provides information for developing advanced applications.

Chapter 4, *I/O Formatting*, describes input and output formats for formatted I/O functions.

Chapter 5, *SICL Errors*, lists and describes the error codes returned by SICL functions.

Chapter 6, *Support and Service*, describes how to contact RadiSys Technical Support for support of SICL for Windows.

1.2 What is SICL For Windows?

SICL for Windows is an implementation of the SICL standard as defined by Hewlett Packard. It is a runtime library for use by C/C++ programmers that are developing instrument control applications that run on a RadiSys VXIbus Embedded Personal Computer (EPC®) or a Hewlett-Packard VXLink card. SICL for Windows (referred to as SICL in this manual) is written for use with ANSI standard C/C++ compilers (for example, Microsoft C/C++ and Borland C/C++).

The library contains functions for Windows-based applications running on a VXIbus embedded controller to control VXIbus instruments or General Purpose Interface Bus (GPIB) instruments. An instrument control connection is called a session. Sessions can be to a single instrument (device) or to a particular bus (interface), VXIbus or GPIB.

SICL functions allow C/C++ programmers to take full advantage of the connected instrument capabilities, including:

- Sending and receiving messages.
- Requesting a status byte from a device.
- Receiving asynchronous service requests (SRQ) from devices.
- Clearing a device or interface.
- Locking and unlocking devices and interfaces.
- Controlling time-outs.
- Controlling interrupt, service request (SRQ), and error handling.
- Using symbolic names for devices and interfaces.
- Formatted and unformatted I/O.
- Bus mapping and copy functions
- Register based command messages

1.2.1 Conformance to the SICL Standard

SICL for Windows conforms to revision 3.8 of the Hewlett-Packard SICL standard.

For VXI, this implementation supports level 2F: device and interface sessions for both non-formatted and formatted I/O. This implementation of SICL does not support communications with commanders.

For GPIB, this implementation supports level 3F: device and interface sessions for both non-formatted and formatted I/O. The GPIB implementation includes commander support.

1.2.2 Portability

Applications written using SICL easily port to other environments with little or no change, as long as the new environment supports an equivalent level of the SICL standard.

1.2.3 Transparency

SICL defines one consistent interface for communicating with both VXIbus and GPIB devices. In addition, SICL supports symbolic naming of devices and interfaces. These features allow applications that communicate with one instrument on one interface (VXI or GPIB) to communicate with an equivalent instrument on the other interface without program modification or recompilation.

1.2.4 SICL for Windows Architecture

Figure 1-1 is a diagram of the SICL for Windows software architecture that shows how the architecture relates to the VXIbus and GPIB hardware and where SICL resides in the architecture. User-written Windows applications can access the VXI hardware using SICL or the Bus Management library.

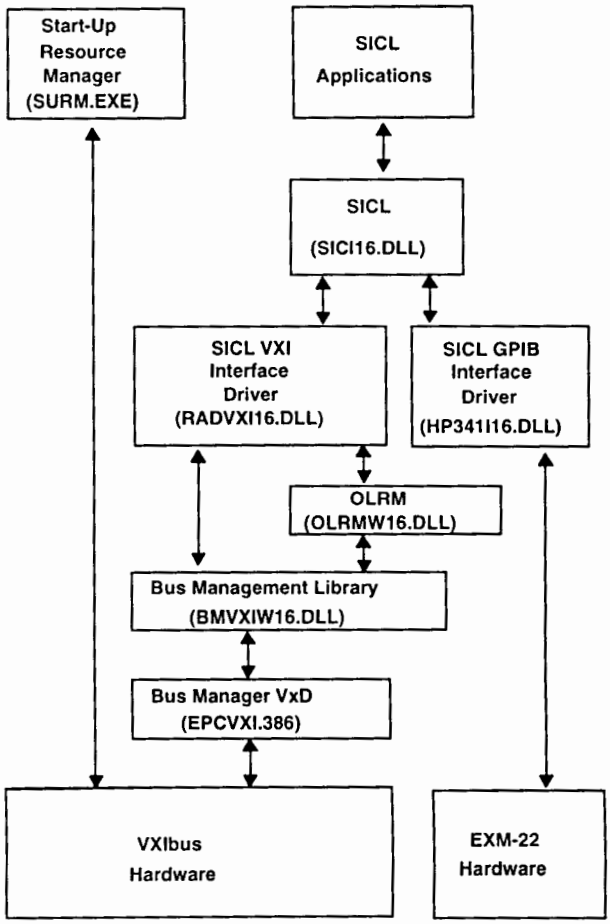

Figure 1-1. SICL for Windows Software Architecture.

1.2.5 SURM

The Start-Up Resource Manager (SURM) determines the physical content of the system and configures the devices. It is typically the first program to run after DOS boots. The SURM is the EPConnect implementation of the resource manager defined in the VXIbus specification. However, SURM extends the specification definition to include non-VXIbus devices, such as GPIB instruments. The SURM uses the **DEVICES** file to obtain device information not directly available from the devices. SURM accesses VXIbus devices in the system directly.

1.2.6 SICL

The SICL interface is independent of the operating system, the hardware platform, and the communication interface. Programs that use SICL port easily to another controller platform as long as the new platform also uses a compatible SICL library. Portability is both at the source code level and at the interface level. Programs written to communicate with an instrument on a given interface can be used to communicate with an equivalent instrument on another interface without modification.

1.2.7 SICL VXI and GPIB Interface Drivers

The SICL VXI and GPIB interface drivers provide interface and hardware specific support to SICL. They implement the portions of SICL functionality that are either interface or hardware specific.

1.2.8 OLRM

The On-Line Resource Management library (**OLRMW16.DLL**) provides user applications with access to results of the resource management process, as well as retrieving status information from devices over the VXIbus. A C/C++ language interface is provided to access OLRM data. OLRM accesses the VXIbus through the Bus Management library and the BusManager VxD.

1.2.9 Bus Management Library and BusManager VxD

The Bus Management library and BusManager VxD are at the foundation of EPConnect. They provide the lowest level interface to the VXIbus hardware through their function libraries. These functions allow you to:

- Control VXIbus word serial registers.
- Send word serial commands of all sizes.
- Transfer blocks of data to and from VXIbus devices, with BERR detection.
- Control EPC Slave memory
- Query EPC driver, firmware, and hardware version or type.

The Bus Management library is used primarily for functionality that isn't provided by SICL. For portability and ease-of-use, user programs should use SICL whenever possible.

1.3 Programming, Compiling and Linking

This section contains information about programming with SICL for Windows. Included is a list of the header files provided, the programming interfaces, and compiling and linking hints.

1.3.1 SICL.H Header File

For C and C++ programs, you must include the **SICL.H** header file at the beginning of every file that contains SICL function calls. This header file contains the SICL function prototypes and the definitions for all SICL constants and error codes.

Figure 1-2 shows the structure of **SICL.H**. It contains two sections: one defining standard constants, structures, and functions and another defining non-standard constants, structures, and functions.

```
#ifndef SICL_H
#define SICL_H
...body of the standard header file...

#ifndef STD_SICL

...body of non-standard header file...
#endif  /* STD_SICL */

#endif  /* SICL_H */
```

Figure 1-2. Default SICL.H File

An **#if/#endif** pair surrounds the contents of the **SICL.H** header file so that you can include the file multiple times without causing compiler errors.

The include file also contains **extern "C"{}** bracketing for the C++ compiler. Because **extern "C"** is strictly a C++ keyword, it is also bracketed and only visible when compiling under C++ and not standard C. If your compiler does not define the **_cplusplus** manifest constant, you are required to bracket the **SICL.H** file with **extern "C"** when compiling C++ SICL programs.

1.3.2 SRQ Interrupt and Error Handler Declarations

Custom error, SRQ, and interrupt handler (callback) functions installed using SICL's **ionerror**, **ionsrq**, and **ionintr** functions should be declared using the SICL modifier **SICLCALLBACK**, which is defined as "**_export _far _pascal**" in **SICL.H**. Failure to do this usually causes a general protection fault (GPF) error at the time the handler is called.

Additionally, if you are developing an application using the QuickWin feature provided with Microsoft compilers and are installing a custom handler, you must also use the **_loadds** modifier with your handler declaration.

1.3.3 Compiling and Linking SICL for Windows Applications

The following is a summary of important compiler-specific considerations for several of the popular Windows 3.1 compiler products.

Ensure that **SICL.H** is in the compiler search path by doing one of the following:

1. Specify the entire file pathname when including the header file in the source file.

2. Specify **C:\SICL\C** as part of the header file search path parameter at compiler invocation time.

3. Specify **C:\SICL\C** as part of the header file search path environment variable.

When linking a SICL for Windows application or DLLs, the link must include the appropriate SICL library files. The SICL library is **SICL16.LIB**. In addition, applications must specify either **MSAPP16.LIB** for Microsoft C/C++ compilers and **BCAPP16.LIB** for Borland C/C++ compilers. A DLL must specify either **MSDLL16.LIB** for Microsoft C/C++ compilers or **BCDLL16.LIB** for Borland C/C++ compilers.

Ensure that the SICL libraries are in the linker search path by doing one of the following:

1. Specify the entire library pathname when linking object files.

2. Specify the **C:\SICL\C** directory as part of the linker library search path.

NOTE: For specific compiler and/or linker options, refer to your compiler's documentation.

1.3.4 Considerations when using Visual Basic

SICL has a special function, **siclcleanup**, to ensure that Windows performs the necessary cleanup required when a SICL program completes execution. Each SICL application should call **siclcleanup** before exiting. The best place to call **siclcleanup** is in the **Form_Unload** routine of the Start Up form in a Visual Basic program. This is where **siclcleanup** is called in all of the Visual Basic example programs.

Any string that is used as a buffer for an **iread** or **iwrite** call must be preceded with the **ByVal** Visual Basic reserved word.

To load and run an existing Visual Basic application, first run Visual Basic. Then open the project file for the program you want to run by selecting **File | Open Project** from the Visual Basic menu. Visual Basic project files have a .MAK file extension. Once you have opened the application's proejct file, you can run the application by pressing either F5 or the Run button on the Visual Basic Toolbar.

Note that you can create a standalone executable (.EXE) version of this program by selecting **File | Make EXE File** from the Visual Basic menu. Once this is done, your application can be run stand-alone just like any other .EXE file without having to run Visual Basic.

Error Handling. When a SICL call results in an error, the error is communicated to Visual Basic by setting Visual Basic's **Err** variable to the SICL error code, and **Error$** is set to a human-readable string that corresponds to **Err**. This allows SICL to be integrated in with Visual Basic's built-in error handling capabilities. SICL programs written in Visual Basic can set up error handlers with the Visual Basic **On Error** statement.

The SICL **inoerror** function for C programs is not used with Visual Basic. Similarly, the **I_ERROR_EXIT** and **I_ERROR_NO_EXIT** default handlers used in C programs are not defined for Visual Basic.

When an error occurs within a Visual Basic program, the default behavior is to display a dialog box indicating the error and then halt the program. If you want your program to intercept errors and keep executing, you will need to install an error handler with the **On Error** statement. For example:

```
On Error GoTo MyErrorHandler
```

will cause your program to jump to code at the label **MyErrorHandler** when an error occurs. Note that the error handling code must exist within the subroutine or function where the error handler was declared.

If you don't want to call an error handler or have your application terminate when an error occurs, you can use the **OnError** statement to tell Visual Baic to ignore errors. For example:

```
On Error Resume Next
```

tells Visual Basic to proceed to the statement following the statement in which an error occurs. In this case, you could call the Visual Basic **Err** function in subsequent lines to find out which error occurred.

Visual Basic error handlers are only active within the scope of the subroutine or funciton in which they are declared. Each Visual Basic subroutine or function that wants an error handler must declare its own error handler. Note that this is different than the way SICL error handlers installed with **inoerror** work in C programs. An error handler installed with **inoerror** remains active within the scope of the whole C program.

1.4 What to do Next

Follow these instructions to begin compiling and linking SICL for Windows applications:

1. If SICL for Windows is not pre-installed on your system, install and configure EPConnect using the procedures in Chapter 2 of the *EPConnect/VXI for DOS & Windows User's Guide.*

2. If necessary, refer to the error messages in Chapter 4 of this manual for corrective action information about device driver installation errors.

3. Use the function descriptions in Chapter 2 of this manual for details about a function and/or its parameters to develop applications. Most functions have accompanying examples that demonstrate the function's use.

4. Refer to the sample programs.

2. Function Descriptions

2

This chapter lists the SICL functions by category and by name. It is for the programmer who needs a particular fact, such as what function performs a specific task or what a function's arguments are.

The first section lists the functions categorically by the task each performs. It also gives you a brief description of what each function does. The second section lists the functions alphabetically and describes each function in detail.

2.1 Functions by Category

The categorical listing provides an overview of the operations performed by the SICL functions. Included with each category is a description of the operations performed, a listing of the functions in the category, and a brief description of each function.

The categories of the SICL library functions include:

- Session Handling
- Unformatted I/O
- Formatted I/O
- Asynchronous Event Control
- Memory Mapping
- Memory Mapped I/O
- Error Handling
- Locking
- Timeouts
- Device and Interface Control
- VXI Interface Control
- GPIB Interface Control
- Version Control
- Microsoft Windows control

2.1.1 Session Handling

Session handling functions open sessions, get/set information about sessions, and close sessions. Interaction with devices and interfaces is session-based. Opening a session returns a session handle which is used in subsequent calls to the device or interface.

Session handling functions include the following:

iclose	Closes a session.
igetaddr	Gets a pointer to the session's address string.
igetdata	Gets a pointer to a session's application data structure.
igetdevaddr	Gets a device address.
igetintfsess	Opens an interface session for the interface corresponding to a specific device.
igetintftype	Gets a session's interface type.
igetlu	Gets a session's logical unit.
igetluinfo	Gets information describing a particular logical unit.
igetlulist	Gets a list of valid logical unit numbers.
igetsesstype	Gets a session's type.
iopen	Opens a session.
isetdata	Stores a pointer to the session data structure.

2.1.2 Unformatted I/O

Unformatted I/O provides a method to send and receive arbitrary blocks of data to and from a device or interface. No buffering, formatting or conversion is performed. Using unformatted I/O provides the greatest control when accessing a device or interface.

Do not mix the unformatted I/O function calls with formatted I/O calls within a session.

Unformatted I/O functions include the following:

igettermchr	Gets a session's current termination character.
iread	Reads data from a device or interface.
itermchr	Specifies a session's termination character.
iwrite	Writes data to a device or interface.

2.1.3 Formatted I/O

Formatted I/O eliminates the need to convert internal C data types to data types understood by a particular device or interface. Format strings direct formatting and conversion. These format strings are similar to format strings found in standard C **printf** and **scanf** functions. All formatting and conversion operations are compatible with IEEE 488.2 character and number formats. Formatted I/O operations use buffers to queue data into large blocks to improve performance.

The formatted I/O functions are buffered. There are two non-buffered and non-formatted I/O functions called **iread** and **iwrite**. These are raw I/O functions and do not intermix with the formatted I/O functions.

If raw I/O must be mixed, use the **ifread/ifwrite** functions. They have the same parameters as **iread** and **iwrite**, but write raw output to the formatted I/O buffers.

The formatted I/O functions convert data under the control of the format string. the format string specifies how the argument is converted before it is input or output. The %F format string is not supported.

Do not mix the formatted I/O function calls with unformatted I/O function calls within a session.

The **iprintf, ivprintf,** and **ifwrite** functions and the write portion of the **ipromptf** function use the write buffer. When the write buffer is full or when it receives an EOI it is flushed (its contents are sent to the device or interface). It also flushes immediately after the write portion of an **ipromptf** call.

The **iscanf, ivscanf,** and **ifread** functions and the read portion of the **ipromptf** function use the read buffer. The read buffer flushes (discards its contents) automatically before the write portion of an **ipromptf** call and after an **iflush** call.

The functions **iflush, isetbuf,** and **isetubuf** control read/write buffer operations. The functions **ifread** and **ifwrite** don't do formatting, but are listed here because they use the read/write buffers.

Formatted I/O functions include the following:

iflush	Flushes formatted I/O read and/or write buffers.
ifread	Reads data from a device or interface.
ifwrite	Writes data to a device or interface.
iprintf	Formats and writes data to a device or interface.
ipromptf	Writes formatted data to and reads formatted response from a device or interface.
iscanf	Reads and formats data from a device or interface.
isetbuf	Sets the size of formatted I/O read and write buffers.
isetubuf	Sets the formatted I/O read or write buffer to a user-supplied buffer.
isprintf	Formats and writes data to a buffer.
isscanf	Reads and formats data from a buffer.
isvprintf	Formats and writes data to a buffer using a standard **va_list** parameter.
isvscanf	Reads and formats data from a buffer using a standard **va_list** parameter.
ivprintf	Formats and writes data to a device or interface using a standard **va_list** parameter.
ivpromptf	Writes formatted data to and reads formatted response from a device or interface using a standard **va_list** parameter.
ivscanf	Reads and formats data from a device or interface using a standard **va_list** parameter.

2.1.4 Asynchronous Event Control

An asynchronous event is an event that can occur anytime during the execution of a program. In SICL, an asynchronous event occurs when a service request (SRQ) occurs or an enabled interrupt occurs.

Asynchronous event control functions include the following:

igetonintr	Queries a session's current interrupt handler.
igetonsrq	Queries a session's current SRQ handler.
iintroff	Disables SRQ and interrupt event processing.
iintron	Enables SRQ and interrupt event processing.
ionintr	Installs a session's interrupt handler.
ionsrq	Installs a session's SRQ handler.
isetintr	Enables and disables interrupt reception.
iwaithdlr	Waits for an SRQ or interrupt handler function to execute.

2.1.5 Memory Mapping

The memory mapping functions map and unmap portions of VXIbus memory space into a SICL application's address space, and get memory space mapping information.

Memory mapping functions include the following:

imap	Maps a portion of a VXIbus memory space into an application's address space.
imapinfo	Queries address space mapping capabilities for the specified interface.
iunmap	Deletes an address space mapping.

2.1.6 Memory Mapped I/O

The memory mapped I/O functions copy bytes, words, and longwords from one location to another. The locations can be either a sequence of memory locations or a FIFO register. The locations can reside in any VXI address space or in local PC space. Note that SICL memory mapped I/O will not operate properly using a VXLink card. Functions compatible with SICL memory mapped I/O are marked with an asterisk.

Memory mapped I/O functions include the following:

ibblockcopy	Copies bytes from one set of sequential memory locations to another.
ibeswap*	Byte-swaps a buffer of data from Motorola (big-endian) byte order to the native byte order of the EPC.
ibpeek	Reads a byte from a mapped address.
ibpoke	Writes a byte to a mapped address.
ibpopfifo	Copies bytes from a single memory location (FIFO register) to sequential memory locations.
ibpushfifo	Copies bytes from sequential memory locations to a single memory location (FIFO register).
ilblockcopy	Copies a block of 32-bit words from one set of sequential memory locations to another.
ileswap*	Byte-swaps a buffer of data from Intel (little-endian) byte order to the native byte order of the EPC.
ilpeek	Reads a 32-bit word stored at a mapped address.
ilpoke	Writes a 32-bit word to a mapped address.
ilpopfifo	Copies 32-bit words from a single memory location (FIFO register) to sequential memory locations.
ilpushfifo	Copies 32-bits words from sequential memory locations to a single memory location (FIFO register).

iswap*	Byte-swaps a buffer of data.
iwblockcopy	Copies blocks of 16-bit words from one set of sequential memory locations to another.
iwpeek	Reads a 16-bit word from a mapped address.
iwpoke	Writes a 16-bit word to a mapped address.
iwpopfifo	Copies 16-bit words from a single memory location (FIFO register) to sequential memory locations.
iwpushfifo	Copies 16-bits words from sequential memory locations to a single memory location (FIFO register).

2.1.7 Error Handling

Most SICL functions can generate errors. Functions usually return a special value (a null pointer or a non-zero return value) to indicate an error. In addition, the application program can designate an error handler function to execute when an error occurs.

Error handling functions include the following:

icauseerr	Set a process' most recent error number.
igeterrno	Gets a process' most recent error number.
igeterrstr	Gets an error string.
igetonerror	Queries the current error handler.
ionerror	Installs an error handler.

2.1.8 Locking

A device or interface can be locked by a session to prevent access by another session. Locking is useful when multiple threads attempt simultaneous device or interface access. A locked device or interface can cause the accessing thread to suspend or generate an error.

Locking functions include the following:

igetlockwait	Gets a session's current lock-wait flag.
ilock	Locks a device or interface.
isetlockwait	Determines whether accessing a locked device or interface suspends the calling thread or generates an error.
iunlock	Unlocks a device or interface.

Locking affects these SICL functions:

iclear	**igpibsendcmd**	**isetstb**
iflush	**igpibsett1delay**	**isetubuf**
ifread	**ilocal**	**itrigger**
ifwrite	**ilock**	**ivprintf**
igpibatnctl	**imap**	**ivpromptf**
igpibgett1delay	**iprintf**	**ivscanf**
igpibllo	**ipromptf**	**ivxitrigoff**
igpibpassctl	**iread**	**ivxitrigon**
igpibppoll	**ireadstb**	**ivxitrigroute**
igpibppollconfig	**iremote**	**ivxiws**
igpibppollresp	**iscanf**	**iwrite**
igpibrenctl	**isetbuf**	**ixtrig**

Because SICL allows multiple sessions on the same device or interface, opening a session does not give you exclusive access to the device or interface. In some cases this is not an issue, but should be a consideration if you are concerned with program portability.

The SICL **ilock** function is used to lock an interface or device. The SICL **iunlock** function is used to unlock an interface or device.

Locks are performed on a per-session (device, interface or commander) basis. If a session within a given process locks a device or interface, then that device or interface can only be accessed from that session.

Locks can be nested. The device or interface only becomes unlocked when the same number of unlocks are done as the number of locks. Doing an unlock without a lock returns the error **I_ERR_NOLOCK**.

Locking an interface (from an interface session) restricts other device and interface sessions from accessing this interface. Locking a device restricts other device sessions from accessing this device; however, other interface sessions may continue to access the interface for this device. Locking a commander (from a commander session) restricts other commander sessions from accessing this commander.

NOTE: It is possible for an interface session to access a device locked from a device session.

Not all SICL routines are affected by locks. Some routines that simply set or return session parameters never touch the interface hardware and therefore work without locks.

Actions of Locking

If a session tries to perform an SICL function that obeys locks on an interface or device that is currently locked by another session, the default action is to suspend the call until the lock is released or, if a timeout is set, until it times out.

This action can be changed with the **isetlockwait** function. If the **isetlockwait** function is called with the flag parameter set to 0 (zero), the default action is changed. Rather than causing SICL functions to suspend, an error will be returned.

To return to the default action, to suspend and wait for an unlock, call the **isetlockwait** function with the flag set to any non-zero value.

Locking in a multi-user environment

In a multi-user/multi-process environment where devices are being shared, it is a good idea to use locking to ensure exclusive use of a particular device or set of devices. In general, it is not friendly behavior to lock a device at the beginning of an application and unlock it at the end. This can result in deadlock or long waits by others who want to use the resource. The recommended way to use locking is per transaction. Per transaction means that you lock before you setup the device, then unlock after all the desired data has been acquired. When sharing a device, you cannot assume the state of the device, so the beginning of each transaction should have any setup need to configure the device or devices to be used.

2.1.9 Timeouts

A timeout value is the time interval to wait for an operation to complete before aborting. When an operation aborts because of a timeout, the aborted function returns an error indicating that the call timed out.

Timeout functions include the following:

igettimeout	Gets a session's current timeout value.
itimeout	Sets a session's timeout value.

Timeouts affect these SICL functions:

iclear	**igpibsendcmd**	**isetubuf**
iflush	**igpibsett1delay**	**itrigger**
ifread	**ilocal**	**ivprintf**
ifwrite	**ilock**	**ivpromptf**
igpibatnctl	**imap**	**ivscanf**
igpibgett1delay	**iprintf**	**ivxitrigoff**
igpibllo	**ipromptf**	**ivxitrigon**
igpibpassctl	**iread**	**ivxitrigroute**
igpibppoll	**ireadstb**	**ivxiwaitnormop**
igpibppollconfig	**iremote**	**ivxiws**
igpibppollresp	**iscanf**	**iwaithdlr**
igpibrenctl	**isetbuf**	**iwrite**
	isetstb	**ixtrig**

2.1.10 Device and Interface Control

The device and interface control category contains functions that perform control operations common to different interface types. It also contains functions that set local and remote access to the devices.

Device and interface control functions include the following:

iabort	Aborts an I/O operation in progress on another thread.
iclear	Clears a device or an interface.
icmd	Send a command to a SICL interface driver.
ihint	Defines the type of communication a device driver should use.
ilocal	Puts a device in local mode.
ireadstb	Reads the status byte from a device.
iremote	Puts a device in remote mode.
isetstb	Sets this controller's status byte.
itrigger	Sends a trigger to a device or interface.
ixtrig	Asserts and deasserts one or more triggers on an interface.

2.1.11 VXI Interface

The VXI interface function category contains control functions specific to the VXIbus only.

VXI interface functions include the following:

ivxibusstatus	Gets VXIbus status.
ivxigettrigroute	Gets a current trigger routing.
ivxirminfo	Gets VXI device information.
ivxiservants	Gets a list of VXI servants.
ivxitrigoff	Deasserts VXIbus trigger lines.
ivxitrigon	Asserts VXIbus trigger lines.
ivxitrigroute	Routes VXIbus trigger lines.
ivxiwaitnormop	Waits for normal operation of a VXI interface.
ivxiws	Sends a word serial command to a VXI device.

2.1.12 GPIB Interface

The GPIB interface function category contains control functions specific to GPIB only.

GPIB interface functions include the following:

igpibatnctl	Controls the state of the ATN line during GPIB writes.
igpibbusaddr	Changes the bus address of the GPIB interface card.
igpibbusstatus	Gets GPIB status.
igpibgett1delay	Retrieves the t1 delay on the GPIB interface.
igpibllo	Puts all GPIB devices into local-lockout mode.
igpibpassctl	Passes active controller status to another GPIB interface.
igpibppoll	Executes a parallel poll.
igpibppollconfig	Configures a GPIB device's response to a parallel poll.
igpibpollresp	Sets the state of the PPOLL bit when polled by the commander.
igpibrenctl	Controls the state of the GPIB REN line.
igpibsendcmd	*Writes command bytes to a GPIB interface.*
igpibsett1delay	Sets the t1 delay on the GPIB interface.

2.1.13 Version Control

The version control category contains a function to check the version of the SICL library.

iversion	Returns the SICL version of the library that the application was linked to.

2.1.14 Microsoft Windows Control

The Microsoft Windows control category contains a function to make sure all SICL I/O resources are released before a Windows 3.1 SICL application terminates.

_siclcleanup	Releases Windows 3.1 I/O resources before terminating.

2.2 Functions by Name

This section contains an alphabetical listing of the SICL library functions. Each listing describes the function, gives its invocation sequence and arguments, discusses its operation, and lists its returned values. Where usage of the function may not be clear, an example with comments is given.

iabort

Description Aborts an I/O operation in progress on another thread.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iabort(INST id);
```

id Session handle.

Visual Basic Synopsis

```
Declare Sub iabort Lib "sicl16.dll" (ByVal id As Integer)
```

Remarks This function aborts an I/O operation in progress on another thread specified by *id*.

The function is valid only for device sessions.

Windows supports a single thread per task. Therefore, on Windows, this function has no effect.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iread, iwrite**

ibblockcopy

Description Copies bytes from one set of sequential memory locations to another.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ibblockcopy(INST id, unsigned char _far *src, unsigned char
    _far *dest, unsigned long count);
```

id	Session handle.
src	Source address.
dest	Destination address.
count	Number of bytes to copy.

Visual Basic Synopsis

Declare Sub **ibblockcopy** Lib "sicl16.dll" (ByVal *id* As Integer, *src* As Any, *dest* As Any, ByVal *cnt* As Long)

Remarks This function copies bytes from successive memory locations beginning at *src* into successive memory locations beginning at *dest*. *Count* specifies the number of data bytes to transfer. *Id* identifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibpeek**, **ibpoke**, **ibpopfifo**, **ibpushfifo**, **ilblockcopy**, **imap**, **iwblockcopy**

Example

```
/*
 * ibblock.c:  this example uses ibblockcopy() to read a VXI register of the
 *             device configured as ULA 0.  The bit encoding of this register
 *         is defined by the VXI specification.  For this particular
 *         example, the program is using the Device Class bits.
 */

#include <windows.h>
#include "sicl.h"

#define    VXI_REG_OFFSET  0xC000

char _far *Strings[] =
{
     "Memory",
     "Extended",
     "Message Based",
     "Register Based"
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_ibblockcopy(void)
{
     volatile char _far * mapped_ptr;
     unsigned char        id_reg_high;
     int             error_number;
     INST            id;

     /* Open a VXI interface session. */

     id = iopen("vxi");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Map in A16 space */

     mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
     if (mapped_ptr == NULL)
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Copy the ID register of the device at ULA 0 and determine  */
     /* the device's class.                                        */

     error_number = ibblockcopy(    id,
                         (unsigned char *)
                         (mapped_ptr + VXI_REG_OFFSET),
                         &id_reg_high,
```

```
                        1);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  ibblockcopy().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         iclose(id);
         return (error_number);
    }
    WinPrintf("Class of device at ULA 0 is %s.\n",
              Strings[id_reg_high >> 6]);
    iclose(id);
    return (error_number);
}
```

ibeswap

Description Byte-swaps a buffer of data from Motorola (big-endian) byte order to the native byte order of the EPC.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ibeswap (char _far *buf, unsigned long length, int datasize);
```

buf Address of data buffer.

length Length of the buffer, in bytes.

datasize Size of data elements in the buffer, in bytes.

Visual Basic Synopsis

```
Declare Sub ibeswap Lib "sicl16.dll" (addr As Any, ByVal length
As Long, ByVal datasize As Integer)
```

Remarks This function byte-swaps a buffer of equal-sized data elements. *Length* specifies the overall size of the buffer and *datasize* specifies the size of the individual data elements in the buffer.

Length must be a multiple of *datasize*.

Datasize may be 1, 2, 4 or 8 bytes.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ileswap, iswap**

Example See **iswap**.

ibpeek

Description Reads a byte from a mapped address.

C Synopsis

#include "sicl.h"

unsigned char SICLAPI
ibpeek(volatile unsigned char _far **addr*);

addr Address of byte.

Visual Basic Synopsis

Declare Function **ibpeek** Lib "sicl16.dll" Alias "vbibpeek" (ByVal *addr* As Long) As Integer

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call.

This function does not detect bus errors caused by its use.

Return Value The function returns the 8-bit value stored at *addr*.

See Also **ibpoke, ilpeek, imap, iwpeek**

Example

```
/*
 * ibpeek.c:    this example uses ibpeek() to read a VXI register of the device
 *         configured as ULA 0.  The bit encoding of this register is
 *         defined by the VXI specification.  In this particular example,
 *         the program is using the Address Space bits.
 */

#include <windows.h>
#include "sicl.h"

#define    VXI_REG_OFFSET  0xC000

char _far *Strings[] =
{
      "A16/A24",
      "A16/A32",
      "RESERVED",
      "A16 Only"
};
```

```
void
WinPrintf(char _far *Format_String, ...);

int
sample_ibpeek(void)
{
     volatile char _far * mapped_ptr;
     unsigned char        id_reg;
     int             error_number;
     INST            id;

     /* Open a VXI interface session. */

     id = iopen("vxi");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Map in A16 space */

     mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
     if (mapped_ptr == NULL)
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Read the ID register of the device at ULA 0 and determine  */
     /* the device's address space.                                */

     id_reg   =   ibpeek((volatile   unsigned   char   _far   *)   (mapped_ptr   +
VXI_REG_OFFSET));
     WinPrintf("Address space of device at ULA 0 is %s.\n",
               Strings[(id_reg & 0x30) >> 4]);
     iclose(id);
     return (I_ERR_NOERROR);
}
```

ibpoke

Description Writes a byte to a mapped address.

C Synopsis

#include "sicl.h"

void SICLAPI
ibpoke(volatile unsigned char _far ***dest*****, unsigned char** ***value*****);**

dest	Destination address.
value	Byte to write.

Visual Basic Synopsis

Declare Sub **ibpoke** Lib "sicl16.dll" Alias "vbibpoke" (ByVal *addr* As Long, ByVal *value* As Integer)

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call.

Return Value The function returns no value.

See Also **ibpeek, ilpoke, imap, iwpoke**

Example

```
/*
 * ibpoke.c:    this example uses ibpoke() to write to a VXI register of the
 *         device configured as ULA 0.  This example assumes the device
 *         at ULA 0 is an EPC-7.
 */

#include <windows.h>
#include "sicl.h"

#define    VXI_REG_OFFSET  0xC000

void
WinPrintf(char _far *Format_String, ...);

int
sample_ibpoke(void)
{
     volatile char _far * mapped_ptr;
     int             error_number;
     INST            id;
```

```
    /* Open a VXI interface session. */

    id = iopen("vxi");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        return (error_number);
    }

    /* Map in A16 space */

    mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
    if (mapped_ptr == NULL)
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        iclose(id);
        return (error_number);
    }

    /* Clear the high bit of the EPC-7's Status/Control Register, */
    /* causing the EPC-7 to ignore A32 accesses.                  */

    mapped_ptr += VXI_REG_OFFSET + 5;
    ibpoke(   (volatile unsigned char _far *) mapped_ptr,
        (unsigned char) (ibpeek((volatile unsigned char _far *) mapped_ptr) &
~0x80));
    iclose(id);
    return (I_ERR_NOERROR);
}
```

ibpopfifo

Description Copies bytes from a single byte-wide memory location (FIFO register) to sequential memory locations.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ibpopfifo(INST id, unsigned char _far *fifo, unsigned char _far
    *dest, unsigned long count);
```

id	Session handle.
fifo	FIFO pointer.
dest	Destination address.
count	Number of bytes to copy.

Visual Basic Synopsis

Declare Sub **ibpopfifo** Lib "sicl16.dll" (ByVal *id* As Integer, *fifo* As Any, *dest* As Any, ByVal *cnt* As Long)

Remarks This function copies *count* bytes from *fifo* into successive memory locations beginning at *dest*. *Count* specifies the number of data bytes to transfer. *Id* identifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibpushfifo, ilpopfifo, imap, iwpopfifo**

Example

```
/*
 * ibpop.c:     this example uses ibpopfifo() to read from a hypothetical
 *          FIFO at address 0 in A16 space.
 */

#include <windows.h>
#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ibpopfifo(void)
{
     volatile char _far * mapped_ptr;
     unsigned char        fifo_data[5];
     int                  error_number;
     INST                 id;

     /* Open a VXI interface session. */

     id = iopen("vxi");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Map in A16 space */

     mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
     if (mapped_ptr == NULL)
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Read the FIFO 5 times, storing the values into fifo_data[]. */

     error_number = ibpopfifo( id,
                          (unsigned char *) mapped_ptr,
                          fifo_data,
                          sizeof(fifo_data));
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  ibpopfifo().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     iclose(id);
     return (error_number);
}
```

ibpushfifo

Description Copies bytes from sequential memory locations to a single memory location (FIFO register).

C Synopsis

```
#include "sicl.h"

int SICLAPI
ibpushfifo(INST id, unsigned char _far *src, unsigned char _far
    *fifo, unsigned long count);
```

id	Session handle.
src	Source address.
fifo	FIFO pointer.
count	Number of bytes to copy.

Visual Basic Synopsis

Declare Sub **ibpushfifo** Lib "sicl16.dll" (ByVal *id* As Integer, *src* As Any, *fifo* As Any, ByVal *cnt* As Long)

Remarks This function copies *count* bytes from the sequential memory locations beginning at *src* into the FIFO at *fifo*. *Count* specifies the number of data bytes to transfer. *Id* specifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibpopfifo, ilpushfifo, imap, iwpushfifo**

Example

```
/*
 * ibpush.c:    this example uses ibpushfifo() to write to a hypothetical
 *         FIFO at address 0 in A16 space.
 */

#include <windows.h>
#include "sicl.h"

unsigned char fifo_data[] =
{
      0x53, 0x61, 0x6D, 0x70, 0x6C, 0x65, 0x44, 0x61, 0x74, 0x61
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_ibpushfifo(void)
{
      volatile char _far * mapped_ptr;
      int             error_number;
      INST            id;

      /* Open a VXI interface session. */

      id = iopen("vxi");
      if (id == ((INST) 0))
      {
            error_number = igeterrno();
            WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
            return (error_number);
      }

      /* Map in A16 space */

      mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
      if (mapped_ptr == NULL)
      {
            error_number = igeterrno();
            WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
            iclose(id);
            return (error_number);
      }

      /* Write the FIFO 10 times, storing the values from fifo_data[]. */

      error_number = ibpushfifo(     id,
                        fifo_data,
                        (unsigned char *) mapped_ptr,
                        sizeof(fifo_data));
      if (error_number != I_ERR_NOERROR)
      {
            WinPrintf("FAILURE:  ibpushfifo().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
      }
      iclose(id);
      return (error_number);
}
```

icauseerr

Description Sets a process' most recent error number.

C Synopsis

#include "sicl.h"

void SICLAPI
icauseerr(INST *id*, **int** *error*, **int** *callhandler*);

id	Session handle.
error	Error number.
callhandler	A flag indicating whether or not to call the process' currently installed error handler.

Visual Basic Synopsis

Declare Sub **icauseerr** Lib "sicl16.dll" Alias "vbcauseerr" (ByVal *id* As Integer, ByVal *errcode* As Integer, ByVal *flag* As Integer)

Remarks The function sets the calling process' most recent error number to *error* for creating user-defined errors. If *error* is not **I_ERR_NOERROR** and *callhandler* is non-zero and the process has an error handler installed, the function also calls the installed error handler. A process' most recent error number can be queried using **igeterrno**. A process' error handler can be set using **ionerror** and queried using **igetonerror**.

Return Value The function does not return a value.

See Also **igeterrno, igeterrstr, igetonerror, ionerror**

Example See **ionerror.**

iclear

Description Clears a device or an interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iclear(INST id);
```

id Session handle.

Visual Basic Synopsis

```
Declare Sub iclear Lib "sicl16.dll" (ByVal id As Integer)
```

Remarks The function flushes the session's formatted I/O read and write buffers and performs a "clear" operation.

For VXI device sessions, the function issues a DEVICE CLEAR word serial command to the device. The function only supports message-based VXI devices. Other VXI devices cause an error.

For VXI interface sessions, the function issues a SYSRESET signal (SYSRESET is pulsed).

For GPIB device sessions, the function issues a selected device clear command to the device.

For GPIB interface sessions, the function issues an interface clear signal (IFC is pulsed).

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iclose, iopen, itimeout**

Example

```
/*
 * iclear.c:  call iclear() to assert IFC (GPIB).
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_iclear(void)
{
     int  error_number;
     INST id;

     /* Open a GPIB interface session. */

     id = iopen("gpib");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Pulse IFC. */

     error_number = iclear(id);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  iclear().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     iclose(id);
     return (error_number);
}
```

iclose

Description Closes a session.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iclose(INST id);
```

id Session handle.

Visual Basic Synopsis

```
Declare Sub iclose Lib "sicl16.dll" (ByVal id As Integer)
```

Remarks This function invalidates the session specified by *id.*

Closing a session releases all resources associated with the session, including locks, mapped VXIbus memory, and enabled interrupts.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * iclose.c:    this example uses explicit calls to iclose() to release a
 *         session's resources.
 */

#include <windows.h>
#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_iclose(void)
{
     volatile char _far * mapped_ptr;
     int             error_number;
     INST            id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Map in vxisink's A16 registers. */

     mapped_ptr = imap(id, I_MAP_VXIDEV, 0, 0, NULL);
     if (mapped_ptr == NULL)
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     else
     {
          error_number = I_ERR_NOERROR;
     }

     /* Close the session.  Once closed, both the mapped pointer   */
     /* and the session id are no longer valid.                    */

     iclose(id);
     return (error_number);
}
```

icmd

Description Sends a command to a SICL interface driver.

C Synopsis

```
#include "sicl.h"

int SICLAPI
icmd(INST id, long command, int allsize, int onesize, void _far
    *data);
```

id	Interface session handle.
command	SICL TULIP driver command.
allsize	Combined size of all command data elements, in bytes.
onesize	Size of each command data element, in bytes.
data	Location of command data.

Visual Basic Synopsis

Declare Sub **icmd** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *cmd* As Long, ByVal *datalen* As Integer, ByVal *datawidth* As Integer, *pdata* As Any)

Remarks This function sends a command directly to the SICL interface driver corresponding to the specified session's interface.

The function provides access to functionality that, while required for correct operation, is not part of the SICL standard.

The SICL for Windows implementation provides non-standard SICL VXI interface driver commands to allow correct processing of VXIbus TTL trigger interrupts. For further information, refer to Chapter 3, *Advanced Topics*.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ionintr, iopen, isetintr**

iflush

Description Flushes formatted I/O read and/or write buffers.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iflush(INST id, int buffermask);
```

id	Session handle.
buffermask	Selects the buffer(s) to clear.

Visual Basic Synopsis

Declare Sub **iflush** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *mask* As Integer)

Remarks This function flushes the session's read buffer and/or write buffer. *Buffermask* must be an OR'd combination of the following constants:

Constant	Description
I_BUF_READ	Discard the contents of the session's read buffer. If data is discarded and the last byte does not contain an END indicator, read from the device or interface until an **END** indicator is read. Discarding the read buffer ensures that the next buffered input function reads data directly from the device rather than reading data that was previously buffered.
I_BUF_WRITE	Write the contents of the write buffer to the device or interface.

I_BUF_DISCARD_READ

Discard the contents of the session's read buffer without performing any I/O. Cannot be used in conjunction with **I_BUF_READ**.

I_BUF_DISCARD_WRITE

Discard the contents of the session's write buffer without performing any I/O. Cannot be used in conjunction with **I_BUF_WRITE**.

If a specified buffer is empty or has already been flushed, this call has no effect.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **isetbuf, isetubuf, itimeout**

Example

```
/*
 * iflush.c:  use iflush() to explicitly flush a session's write buffer.
 */

#include "sicl.h"

#define BUFFER_SIZE  64

void
WinPrintf(char _far *Format_String, ...);

int
sample_iflush(void)
{
     int  error_number;
     INST id;

#if !defined(I_SICL_FMTIO)
     WinPrintf("Formatted I/O is not supported.\n");
     return (I_ERR_NOERROR);
#endif

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Create a write buffer for the session. */

     error_number = isetbuf(id, I_BUF_WRITE, BUFFER_SIZE);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  isetbuf().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Write data to the write buffer.  Use "-t" to prevent an    */
     /* implicit buffer flush.                          */

     (void) iprintf(id,"Test Data%-t\n");

     /* Explicitly flush the write buffer. */

     error_number = iflush(id, I_BUF_WRITE);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  iflush().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     iclose(id);
     return (error_number);
}
```

ifread

Description Reads data from a device or interface via the formatted I/O buffer.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ifread(INST id, char _far*buf, unsigned long bufsize, int _far
    *reason, unsigned long _far *actualcnt);
```

id	Session handle.
buf	Pointer to the data buffer.
bufsize	Number of data bytes to read.
reason	Pointer to the location where the function stores the cause of read termination.
actualcnt	Pointer to a location where the function stores the actual number of bytes read from the device or interface.

Visual Basic Synopsis

Declare Sub **ifread** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *buf* As Any, ByVal *bufsize* As Long, *reason* As Any, *actual* As Long)

Remarks This function reads *bufsize* bytes from the formatted I/O read buffer of the session specified by *id* and stores them into the buffer beginning at *buf*. It performs no formatting or data conversion.

Data is read from the read buffer until empty, then data is read from the device. If the buffer is empty, **ifread** reads data from the device until a termination condition is met.

Reading ends when *bufsize* bytes are read, an END indicator is received, a termination character is received, or a timeout occurs. **ifread** blocks until one of these four conditions is met.

If *reason* is not null, the function stores a bit mask describing why the read terminated in the referenced memory location. The following constants define valid bits in the mask pointed to by *reason*:

Constant	Description
I_TERM_CHR	Termination character received (see **itermchr**)
I_TERM_END	END indicator received
I_TERM_MAXCNT	*Bufsize* bytes read

If *actualcnt* is not null, the function stores the number of bytes read in the referenced memory location.

For VXI device sessions, the function generates BYTE REQUEST word serial commands. The function only supports message-based VXI devices; other VXI devices cause an error.

For VXI interface sessions, the function generates an **I_ERROR_NOTSUPP** error.

For GPIB device sessions, the function first causes all devices to unlisten. Then, it issues the interface's listen address, followed by the device's talk address. Finally, the function reads the data bytes.

For GPIB interface sessions, the function reads data from a GPIB interface without performing any addressing.

To avoid unpredictable results, do not mix buffered input function calls (**ifread, ipromptf, iscanf, ivpromptf, ivscanf**) and unbuffered input function calls (**iread**) within the same session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ifwrite, igettermchr, ivpromptf, iread, iscanf, itermchr, itimeout, ivpromptf, ivscanf**

Example

```
/*
 * ifread.c:    this example calls ifread() to read an instrument's response
 *         without waiting.
 */

#include "sicl.h"

#define BUFFER_SIZE  64

void
WinPrintf(char _far *Format_String, ...);

int
sample_ifread(void)
{
     char       buffer[BUFFER_SIZE] = { 0 };
     int        error_number;
     int        reason;
     unsigned long   read_count;
     INST       id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Write a command to the device. */

     (void) iprintf(id, "IDN?");

     /* Read and print the device's response. */

     error_number = ifread(    id,
                    buffer,
                    BUFFER_SIZE,
                    &reason,
                    &read_count);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  ifread().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }
     buffer[read_count] = '\0';
     WinPrintf("Response data read from \"vxisink\" = %s.\n", buffer);
     WinPrintf("Read termination reason(s):\n");
     if ((reason & I_TERM_CHR) != 0)
     {
          WinPrintf("\tI_TERM_CHR.\n");
     }
     if ((reason & I_TERM_END) != 0)
     {
          WinPrintf("\tI_TERM_END.\n");
     }
```

```
    if ((reason & I_TERM_MAXCNT) != 0)
    {
        WinPrintf("\tI_TERM_MAXCNT.\n");
    }
    iclose(id);
    return (error_number);
}
```

ifwrite

Description Writes data to a device or interface via the formatted I/O buffer.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ifwrite(INST id, char _far *buf, unsigned long bufsize, int end,
    unsigned long _far *actualcnt);
```

id	Session handle.
buf	Pointer to the data buffer.
bufsize	Length, in bytes, of data buffer.
end	END indicator flag.
actualcnt	Pointer to a location where the function stores the actual number of bytes written.

Visual Basic Synopsis

Declare Sub **ifwrite** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *buf* As Any, ByVal *datalen* As Long, ByVal *endi* As Integer, *actual* As Long)

Remarks This function writes the *bufsize* bytes at *buf* to the formatted I/O write buffer of the session specified by *id*. It performs no formatting or data conversion. If *end* is zero, the data is not written to the device until the write buffer is full.

Writing ends when *bufsize* bytes are written or a timeout occurs. This function blocks until one of these two conditions is met.

If *end* is non-zero, the function writes an END indicator with the last data byte. If *end* is zero, the function does not write an END indicator with the last data byte.

If *actualcnt* is not null, the function stores the number of data bytes written in the referenced memory location.

For VXI device sessions, the function generates BYTE AVAILABLE word serial commands. The function supports only message-based VXI devices; other VXI devices generate an error.

For VXI interface sessions, the function generates an **I_ERR_NOTSUPP** error.

For GPIB device sessions, the function first causes all devices to unlisten. Then, it issues the interface's talk address, followed by the device's listen address. Finally, the function writes the data.

For GPIB interface sessions, the function writes bytes directly to the interface without performing any addressing. The ATN line state determines whether the bytes are interpreted as data or command bytes.

To avoid unpredictable results, do not mix buffered output function calls (**ifwrite, iprintf, ipromptf, ivprintf, ivpromptf**) and unbuffered output function calls (**iwrite**) within the same session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ifread, iprintf, ipromptf, itimeout, ivprintf, ivpromptf, iwrite**

Example

```
/*
 * ifwrite.c:  This example calls ifwrite() to write to an instrument.
 */

#include "sicl.h"

#define    BUFFER_SIZE     3
#define    EOI         1

char DataBuffer[] = "RST";

void
WinPrintf(char _far *Format_String, ...);

int
sample_ifwrite(void)
{
      int         error_number;
      unsigned long   actual_count;
      INST        id;

      /* Open a VXI device session. */
```

```
    id = iopen("vxisink");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        return (error_number);
    }

    /* Write a buffer of data to the device. */

    error_number = ifwrite(id, DataBuffer, BUFFER_SIZE, EOI, &actual_count);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  ifwrite().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
    }
    else
    {
        WinPrintf("%d bytes written to \"vxisink\".\n",
                BUFFER_SIZE);
    }
    iclose(id);
    return (error_number);
}
```

igetaddr

Description Gets a pointer to the session's address string.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetaddr(INST id, char _far * _far *address);
```

id	Session handle.
address	Pointer to the address of a location where the function stores the session's address string.

Visual Basic Synopsis

Declare Sub **igetaddr** Lib "sicl16.dll" Alias "vbgetaddr" (ByVal *id* As Integer, ByVal *addr* As String)

Remarks This function returns a pointer to the address string of the session specified by *id*. The returned address is the address that was passed to **iopen** when SICL opened the session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * igetaddr.c:  use igetaddr() to query a session's name.
*/

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_igetaddr(void)
{
      char _far *     address_ptr;
      int       error_number;
      INST      id;
```

```
    /* Open a VXI device session. */

    id = iopen("vxisink");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        return (error_number);
    }

    /* Query and print the session's address string. */

    error_number = igetaddr(id, &address_ptr);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  igetaddr().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
    }
    else
    {
        WinPrintf("Session address string = \"%s\".\n",
                  address_ptr);
    }
    iclose(id);
    return (error_number);
}
```

igetdata

Description Gets a pointer to a session's application data structure.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetdata(INST id, void _far* _far *data);
```

id	Session handle.
data	Pointer to a location where the function stores the application-specific data structure.

Visual Basic Synopsis

None

Remarks This function queries an application-specific data structure from the session specified by *id* and places it at the location specified by *data*. The **isetdata** function establishes the application-specific data structure.

The application-specific data structure is a 4-byte memory block. Its contents are application-specific. Typically, it contains a pointer to an application data structure.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **isetdata**

Example

```
/*
 * igetdata.c:  use isetdata()/igetdata() to cache application pointers.
 */

#include "sicl.h"

#define DEV_CNT      10
#define DEV_TYPE_CNT 2
```

```
char *Strings[] =
{
     "vdevx",
     "gdevx"
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_igetdata(void)
{
     int  dev_type;
     int  dev_number;
     int  error_number;
     int  lu;
     int  primary;
     int  secondary;
     int  session    = 0;
     INST id    = (INST) 0;
     INST prev_id    = (INST) 0;
     INST next_id    = (INST) 0;

     /* Open device sessions with names gdev[0-9] and vdev[0-9].   */
     /* Using the cached data field, make a linked list of sessions.    */

     for (dev_type = 0; dev_type < DEV_TYPE_CNT; dev_type += 1)
     {
          for (dev_number = 0; dev_number < DEV_CNT; dev_number += 1)
          {
               *(Strings[dev_type] + 4) = (char) (dev_number + '0');
               id = iopen(Strings[dev_type]);
               if (id == ((INST) 0))
               {
                    error_number = igeterrno();
                    WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                              igeterrstr(error_number),
                              error_number);
                    break;
               }

               /* Add the session to the list. */

               if (next_id == ((INST) 0))
               {
                    next_id = id;
               }
               if (prev_id != ((INST) 0))
               {
                    error_number  =  isetdata(prev_id,  (void  _far  *)  ((unsigned
long) id));
                    if (error_number != I_ERR_NOERROR)
                    {
                         WinPrintf("FAILURE:  isetdata().  Error = %s (%d).\n",
                                   igeterrstr(error_number),
                                   error_number);
                         iclose(id);
                         break;
                    }
               }
               prev_id = id;
          }
     }
```

```
    /* Traverse the session chain, printing primary address and   */
    /* logical unit data and closing the sessions.                */

    id = next_id;
    while (id != 0)
    {
        igetdata(id, (void _far * _far *) &next_id);
        igetlu(id, &lu);
        igetdevaddr(id, &primary, &secondary);
        iclose(id);
        id = next_id;
        WinPrintf("Session %d:  logical unit = %d, primary address = %d.\n",
                  session++,
                  lu,
                  primary);
    }
    return (I_ERR_NOERROR);
}
```

igetdevaddr

Description Gets a device address.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetdevaddr(INST id, int _far *primary, int _far *secondary);
```

id	Device session handle.
primary	Pointer to a location where the function stores the session's primary address.
secondary	Pointer to a location where the function stores the session's secondary address.

Visual Basic Synopsis

Declare Sub **igetdevaddr** Lib "sicl16.dll" (ByVal *id* As Integer, *prim* As Integer, *sec* As Integer)

Remarks The function returns the primary and secondary addresses of the session specified by *id* in the locations specified by *primary* and *secondary*, respectively.

The function is valid only for device sessions.

For VXI devices, *primary* is the device's ULA and "secondary" is -1.

If a GPIB device session's secondary address does not exist, *secondary* is set to –1.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * igetdev.c:  call igetdevaddr() to obtain a device session's primary and
 *             secondary addresses.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_igetdevaddr(void)
{
     int  error_number;
     int  primary;
     int  secondary;
     INST id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query and print the session's primary address. */

     error_number = igetdevaddr(id, &primary, &secondary);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetdevaddr().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     else
     {
          WinPrintf("Session \"vxisink\" primary address   = %d",
                    primary);
          WinPrintf("                   secondary address = %d",
                    secondary);
     }
     iclose(id);
     return (error_number);
}
```

igeterrno

Description Gets an error number.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igeterrno(void);
```

Visual Basic Synopsis

Declare Sub **igetdevaddr** Lib "sicl16.dll" (ByVal *id* As Integer, *prim* As Integer, *sec* As Integer)

Return Value The function returns the return value of the process' most recent SICL event.

If a SICL function fails, the value returned by **igeterrno** affects the failure. If a subsequent SICL function succeeds, **igeterrno** still reflects the failure that occurred in the initial function.

If no error occurred in the preceding function, **igeterrno** returns **I_ERR_NOERROR**.

See Also **igeterrstr**

Example See **ionerror**.

igeterrstr

Description Gets an error string.

C Synopsis

```
#include "sicl.h"

char _far * SICLAPI
igeterrstr(int error);
```

error Error number.

Visual Basic Synopsis

Declare Function **igeterrstr** Lib "sicl16.dll" Alias "vbgeterrstr" (ByVal *errcode* As Integer) As String

Remarks This function returns a pointer to an ASCII string corresponding to the error number specified by *error*.

If passed an invalid error code, the function returns a null pointer.

See Also **igeterrno**

Example See **ionerror**.

igetintfsess

Description Opens an interface session for the interface corresponding to a specific device.

C Synopsis

```
#include "sicl.h"

INST SICLAPI
igetintfsess(INST id);
```

id Device session handle.

Visual Basic Synopsis

Declare Function **igetintfsess** Lib "sicl16.dll" (ByVal *id* As Integer) As Integer

Remarks The function opens a session for communicating with the interface corresponding to the device session *id*.

The interface session handle returned by this function should not be used in a call to **iclose**. The interface session will be closed automatically when the device session specified by *id* is closed.

Multiple calls to this function using the same device session *id* parameter will return the same interface session handle.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * intfsess.c:  use igetintfsess() to open an interface session.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);
```

```
int
sample_igetintfsess(void)
{
     int  error_number;
     INST dev_id;
     INST itf_id;

     /* Open a VXI device session. */

     dev_id = iopen("vxisink");
     if (dev_id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Get a corresponding interface session. */

     itf_id = igetintfsess(dev_id);
     if (itf_id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  igetintfsess().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(dev_id);
          return (error_number);
     }

     /* Open a GPIB device session. */

     dev_id = iopen("gpibsink");
     if (dev_id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Get a corresponding interface session. */

     itf_id = igetintfsess(dev_id);
     if (itf_id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  igetintfsess().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     else
     {
          error_number = I_ERR_NOERROR;
     }
     iclose(dev_id);
     return (error_number);
}
```

igetintftype

Description Gets a session's interface type.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetintftype(INST id, int _far *intftype);
```

id	Session handle.
intftype	Pointer to a location where the function stores the interface type.

Visual Basic Synopsis

```
Declare Sub igetintftype Lib "sicl16.dll" (ByVal id As Integer,
pdata As Integer)
```

Remarks This function places the interface type of the session specified by *id* in the location specified by *intftype*. The following are valid interface type constants:

Constant	Description
I_INTF_GPIB	GPIB interface
I_INTF_VXI	VXI interface

The function is valid only for interface sessions.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * igetintf.c:  call igetintftype() to obtain the device session's interface
 *          type.
 */

#include "sicl.h"

#define DIM(x)       (sizeof(x)/sizeof(char *))

char *Name_Strings[] = { "7", "16", "gpibsink", "vxisink" };
char *Type_Strings[] = { "I_INTF_GPIB", "I_INTF_VXI" };

void
WinPrintf(char _far *Format_String, ...);

int
sample_igetintftype(void)
{
     int  error_number;
     int  index;
     int  type;
     INST id;

     error_number = I_ERR_NOERROR;
     for (index = 0; index < DIM(Name_Strings); index += 1)
     {
          id = iopen(Name_Strings[index]);
          if (id == ((INST) 0))
          {
               continue;
          }
          error_number = igetintftype(id, &type);
          if (error_number != I_ERR_NOERROR)
          {
               WinPrintf("FAILURE:  igetintftype().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
          }
          else
          {
               WinPrintf("Session \"%s\" interface type = %s.\n",
                         Name_Strings[index],
                         Type_Strings[type]);
          }
          iclose(id);
     }
     return (error_number);
}
```

igetlockwait

Description Gets a session's current lock-wait flag.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetlockwait(INST id, int _far *waitflag);
```

id	Session handle.
waitflag	Pointer to the location where the function stores the lock-wait flag.

Visual Basic Synopsis

Declare Sub **igetlockwait** Lib "sicl16.dll" (ByVal *id* As Integer, *flag* As Integer)

Remarks This function places the current state of the lock-wait flag of the session specified by *id* in the location specified by *waitflag*. The **isetlockwait** function sets the session's lock-wait flag state.

When a session's lock-wait flag is non-zero and a locking conflict occurs, the session waits for its previously specified timeout period for the lock to be released. If the lock-wait flag is zero and a locking conflict occurs, **I_ERR_LOCKED** is returned.

By default, a session waits for a conflicting lock to be released (its lock-wait flag is non-zero).

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ilock, isetlockwait, iunlock**

Example

```
/*
 * igetlock.c:  call igetlockwait() to obtain the session's lock wait flag.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_igetlockwait(void)
{
     int  error_number;
     int  wait_flag;
     INST id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query and print the session's lock wait flag. */

     error_number = igetlockwait(id, &wait_flag);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetlockwait().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     else
     {
          WinPrintf("Lock wait flag = %d.\n",
                    wait_flag);
     }
     iclose(id);
     return (error_number);
}
```

igetlu

Description Gets a session's logical unit.

C Synopsis

#include "sicl.h"

int SICLAPI
igetlu(INST *id*, **int _far** **lu*);

id	Session handle.
lu	Pointer to the location where the function stores the logical unit.

Visual Basic Synopsis

Declare Sub **igetlu** Lib "sicl16.dll" (ByVal *id* As Integer, *lu* As Integer)

Remarks This function places the logical unit of the session specified by *id* in the location specified by *lu*.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igetluinfo, igetlulist, iopen**

Example See **igetdata**.

igetluinfo

Description Gets information describing a particular logical unit.

C Synopsis

int SICLAPI
igetluinfo(int *lu*, **struct luinfo _far** **luinfo*);

lu	Logical unit number.
luinfo	Pointer to the location where the function stores the logical unit information.

Visual Basic Synopsis

Declare Sub **igetluinfo** Lib "sicl16.dll" Alias "vbgetluinfo" (ByVal *lu* As Integer, result As *lu_info*)

Remarks This function places information specific to logical unit *lu* at the location specified by *luinfo*.

Logical unit information is returned in the format of the **luinfo** structure. The **luinfo** structure is defined in **sicl.h.** There are four fields which must be present; other fields are optional. The required fields are:

```
struct luinfo
{
   ...
   long logical_unit;
   char symname[32];
   char cardname[32];
   long intftype;
   ...
};
```

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, igetlu, igetlulist**

Example See **igetlulist**.

igetlulist

Description Gets a list of valid logical unit numbers.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetlulist(int _far *_far *lu);
```

lu Pointer to the address of a location where the function stores the address of a list of valid logical unit numbers.

Visual Basic Synopsis

Declare Sub **igetlulist** Lib "sicl16.dll" Alias "vbgetlulist" (*list*() As Integer)

Remarks This function places the address of a list of logical unit numbers in the location specified by *lu*.

The valid logical unit list is terminated by an entry containing -1.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, igetlu, igetluinfo**

Example

```
/*
 * igetluli.c: this example uses igetlulist() and igetluinfo() to query the
 *         list of valid logical units.
 */

#include "sicl.h"

#define LIST_SIZE    256

char *Strings[] =
{
      "I_INTF_GPIB",
      "I_INTF_VXI",
      "I_INTF_RS232",
      "I_INTF_GPIO"
};
```

```
void
WinPrintf(char _far *Format_String, ...);

int
sample_igetlulist(void)
{
     int       error_number;
     int _far *list_ptr;
     struct lu_info  info;

     /* Query and print a list of logical units. */

     error_number = igetlulist(&list_ptr);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetlulist().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }
     WinPrintf("Logical unit list:\n");
     while (*list_ptr != -1)
     {
          error_number = igetluinfo(*list_ptr++, &info);
          if (error_number != I_ERR_NOERROR)
          {
               WinPrintf("FAILURE:  igetluinfo().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
               return (error_number);
          }
          WinPrintf("\tLogical unit %d:\n",
                    info.logical_unit);
          WinPrintf("\t\tInterface type = %s\n",
                    Strings[info.intftype]);
          WinPrintf("\t\tSymbolic name  = \"%s\"\n",
                    info.symname);
          WinPrintf("\t\tCard name      = \"%s\"\n",
                    info.cardname);
     }
     return (I_ERR_NOERROR);
}
```

igetonerror

Description Queries a session's current error handler.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetonerror(errorproc_t _far *errorhandler);
```

errorhandler Pointer to a location where the function stores the current error handler.

Visual Basic Synopsis

None

Remarks This function queries the process' current error handler. The **ionerror** function defines the error handler.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ionerror**

Example See **ionerror**

igetonintr

Description Queries a session's current interrupt handler.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetonintr(INST id, intrhandler_t _far*intrhandler);
```

id	Session handle.
intrhandler	Pointer to a location where the function stores the current interrupt handler.

Visual Basic Synopsis

None

Remarks This function queries the current interrupt handler in use by the device or interface session specified by *id*. The **ionintr** function defines a device's interrupt handler.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ionintr**

Example See **ionintr**

igetonsrq

Description Queries a session's current service request (SRQ) handler.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetonsrq(INST id, srqhandler_t _far *srqhandler);
```

id	Session handle.
srqhandler	Pointer to a location where the function stores the current SRQ handler.

Visual Basic Synopsis

None

Remarks This function queries the current SRQ handler of the session specified by *id*. The function **ionsrq** defines the session's SRQ handler.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ionsrq**

Example See **ionsrq**

igetsesstype

Description Gets a session's type.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igetsesstype(INST id, int _far *sessiontype);
```

id	Session handle.
sessiontype	Pointer to the location where the functions stores the session's type.

Visual Basic Synopsis

```
Declare Sub igetsesstype Lib "sicl16.dll" (ByVal id As Integer,
pdata As Integer)
```

Remarks This function places the session type of the session specified by *id* in the location specified by *sessiontype*. The following are valid *sessiontype* constants:

Constant	Description
I_SESS_DEV	Device session
I_SESS_INTF	Interface session
I_SESS_CMDR	Commander session

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * igetsess.c:  call igetsesstype() to query a session's type.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);
```

```
int
sample_igetsesstype(void)
{
     int  error_number;
     int  type;
     INST id;

     /* Open a GPIB device session. */

     id = iopen("gpibsink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query and print the session's type. */

     error_number = igetsesstype(id, &type);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetsesstype().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }
     WinPrintf("Session \"gpibsink\" is");
     if (type == I_SESS_DEV)
     {
          WinPrintf(" a device session.\n");
     }
     else
     {
          WinPrintf(" an interface session.\n");
     }
     iclose(id);

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query and print the session's type. */

     error_number = igetsesstype(id, &type);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetsesstype().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }
```

```
    WinPrintf("Session \"vxisink\" is");
    if (type == I_SESS_DEV)
    {
        WinPrintf(" a device session.\n");
    }
    else
    {
        WinPrintf(" an interface session.\n");
    }
    iclose(id);
    return (error_number);
}
```

igettermchr

Description Gets a session's current termination character.

C Synopsis

#include "sicl.h"

int SICLAPI
igettermchr(INST *id*, **int _far** **termchr*);*

id	Session handle.
termchr	Pointer to a location where the functions stores the current termination character.

Visual Basic Synopsis

Declare Sub **igettermchr** Lib "sicl16.dll" (ByVal *id* As Integer, *tchr* As Integer)

Remarks This function places the current termination character of the session specified by *id* in the location specified by *termchr*.

The default termination character for a session is −1 (no termination character set). Use **itermchr** to set a termination character.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ifread, iread, itermchr**

Example

```
/*
 * igetterm.c:  call igettermchr()/itermchr() to query/define a session's
 *         termination character.
 */

#include "sicl.h"

#define SESSION_TERM_CHAR '\n'

void
WinPrintf(char _far *Format_String, ...);

int
sample_igettermchr(void)
{
```

```
    int  error_number;
    int  term_char;
    INST id;

    /* Open a VXI device session. */

    id = iopen("vxisink");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        return (error_number);
    }

    /* Query the session's termination character. */

    error_number = igettermchr(id, &term_char);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  igettermchr().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        iclose(id);
        return (error_number);
    }

    /* Define the session's termination character. */

    error_number = itermchr(id, SESSION_TERM_CHAR);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  itermchr().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
    }
    iclose(id);
    return (error_number);
}
```

igettimeout

Description Gets a session's current timeout value.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igettimeout(INST id, long _far *timeout);
```

id	Session handle.
timeout	Pointer to a location where the function stores the timeout value.

Visual Basic Synopsis

Declare Sub **igettimeout** Lib "sicl16.dll" (ByVal *id* As Integer, *tval* As Long)

Remarks This function places the current timeout value of the session specified by *id* in the location specified by *timeout*. Timeout values are specified in milliseconds.

The default timeout value for a session is 0 (no timeout set). A *timeout* value less than zero also indicates that no timeout is set. Use **itimeout** to set a session timeout value.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **itimeout**

Example

```
/*
 * igettime.c: call igettimeout()/itimeout() to query/define a session's
 *         timeout value.
 */

#include "sicl.h"

#define SESSION_TIMEOUT   500

void
WinPrintf(char _far *Format_String, ...);
```

```
int
sample_igettimeout(void)
{
     int  error_number;
     long timeout;
     INST id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query the session's timeout value. */

     error_number = igettimeout(id, &timeout);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igettimeout().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Define the session's timeout value. */

     error_number = itimeout(id, SESSION_TIMEOUT);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  itimeout().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     iclose(id);
     return (error_number);
}
```

igpibatnctl

Description Controls the state of the ATN line.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibatnctl(INST id, int atnstate);
```

id	GPIB interface session handle.
atnstate	ATN line state.

Visual Basic Synopsis

Declare Sub **igpibatnctl** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *atnval* As Integer)

Remarks This function sets the state of the ATN line. Note that the state of the ATN line is modified by future reads and writes.

This function is valid only for GPIB interface sessions.

Setting *atnstate* equal to zero deasserts the ATN line. Setting *atnstate* to a non-zero value asserts the ATN line.

Use **iwrite** and **igpibsendcmd** to actually send bytes while controlling the state of ATN.

The state of the ATN line is undefined following all other SICL calls.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iclear, iflush, ifwrite, iprintf, ipromptf, isetbuf, isetubuf, ivprintf, ivpromptf, iwrite**

Example

```
/*
 * gpibatn.c:  this example uses igpibatnctl() to configure the ATN line for
 *         commands or data.
 */

#include "sicl.h"

#define ATN_DATA     0
#define ATN_COMMAND  1

void
WinPrintf(char _far *Format_String, ...);

int
sample_igpibatnctl(void)
{
     int  error_number;
     INST id;

     /* Open a GPIB interface session. */

     id = iopen("gpib");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Deassert the ATN line. */

     error_number = igpibatnctl(id, ATN_DATA);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igpibatnctl().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Send data bytes. */

     iprintf(id, "Test Data\n");
     iclose(id);
     return (I_ERR_NOERROR);
}
```

igpibbusaddr

Description Changes the bus address of the GPIB interface card.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibbusaddr(INST id, int address);
```

id	GPIB interface session handle.
address	GPIB address.

Visual Basic Synopsis

Declare Sub **igpibbusaddr** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *busaddr* As Integer)

Remarks This function changes the GPIB interface card's address.

Address must contain a valid GPIB address.

This function only works on GPIB interface sessions.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

igpibbusstatus

Description Gets GPIB status.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibbusstatus(INST id, int request, int _far *result);
```

id	GPIB interface session handle.
request	Status request.
result	Pointer to the location where the function stores the GPIB interface status.

Visual Basic Synopsis

```
Declare Sub igpibbusstatus Lib "sicl16.dll" (ByVal id As Integer,
ByVal request As Integer, result As Integer)
```

Remarks This function places the GPIB interface status requested by *request* in the location specified by *result*. The following are valid constants for *request*:

Constant	Description
I_GPIB_BUS_REM	Get the interface remote state (1 = remote, 0 = not remote).
I_GPIB_BUS_SRQ	Get the SRQ state (1 = SRQ asserted, 0 = SRQ not asserted).
I_GPIB_BUS_NDAC	Get the NDAC state (1 = NDAC asserted; 0 = NDAC not asserted).
I_GPIB_BUS_SYSCTLR	Get the interface system controller state (1 = system controller, 0 = not system controller).

I_GPIB_BUS_ACTCTLR	Get the interface active controller state (1 = active controller, 0 = not active controller).
I_GPIB_BUS_TALKER	Get interface addressed-to-talk state (1 =addressed-to-talk, 0 = not addressed-to-talk).
I_GPIB_BUS_LISTENER	Get interface addressed-to-listen state (1 = addressed-to-listen, 0 = not addressed-to-listen).
I_GPIB_BUS_ADDR	Get the interface primary bus address.
I_GPIB_BUS_LINES	Get current GPIB control line state: bit 0 set if SRQ asserted bit 1 set if NDAC asserted bit 2 set if ATN asserted bit 3 set if DAV asserted bit 4 set if NRFD asserted bit 5 set if EOI asserted bit 6 set if IFC asserted bit 7 set if REN asserted bit 8 set if in remote state bit 9 set if in local lockout (LLO) mode bit 10 set if active controller bit 11 set if addressed to talk bit 12 set if addressed to listen

The function queries the state of the GPIB interface as sensed by the interface hardware at a specific point in time. An application should not use **igpibbusstatus** as a general purpose bus analyzer, for two reasons. First, not all interface hardware can accurately sense the state of all GPIB interface lines at all times. Second, the state of the GPIB interface may change between the time the state is queried and the time an application receives the results of a query.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * gpibstat.c: this example calls igpibbusstatus() to display GPIB bus status
 *         information.
 */

#include "sicl.h"

#define DIM(x)  (sizeof(x)/sizeof(int))

int Requests[] =
{
      I_GPIB_BUS_REM,
      I_GPIB_BUS_SRQ,
      I_GPIB_BUS_NDAC,
      I_GPIB_BUS_SYSCTLR,
      I_GPIB_BUS_ACTCTLR,
      I_GPIB_BUS_TALKER,
      I_GPIB_BUS_LISTENER,
      I_GPIB_BUS_ADDR,
      I_GPIB_BUS_LINES
};

char _far *Strings[] =
{
        "I_GPIB_BUS_REM      ",
        "I_GPIB_BUS_SRQ      ",
        "I_GPIB_BUS_NDAC     ",
        "I_GPIB_BUS_SYSCTLR  ",
        "I_GPIB_BUS_ACTCTLR  ",
        "I_GPIB_BUS_TALKER   ",
        "I_GPIB_BUS_LISTENER",
        "I_GPIB_BUS_ADDR     ",
        "I_GPIB_BUS_LINES    "
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_igpibbusstatus(void)
{
      int   error_number;
```

```
    int  result;
    int  index;
    INST id;

    /* Open a GPIB interface session. */

    id = iopen("gpib");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf( "FAILURE:  iopen().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        return (error_number);
    }

    /* Request and print GPIB status. */

    for (index = 0; index < DIM(Requests); index++)
    {
        error_number = igpibbusstatus( id,
                            Requests[index],
                            &result);
        if (error_number != I_ERR_NOERROR)
        {
            WinPrintf( "FAILURE:  igpibbusstatus().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
            break;
        }
        WinPrintf("%s = 0x%08X.\n", Strings[index], result);
    }
    iclose(id);
    return (error_number);
}
```

igpibgett1delay

Description Retrieves the delay on the GPIB interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibgett1delay(INST id, int _far *delay);
```

id	GPIB interface session handle.
delay	Time, in nanoseconds.

Visual Basic Synopsis

```
Declare Sub igpibgett1delay Lib "sicl16.dll" (ByVal id As Integer,
delay As Integer)
```

Remarks This function retrieves the current setting of t1 *delay* on the GPIB interface specified by *id*. The value returned is the time of t1 *delay* in nanoseconds.

Return Value This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also **igpibsett1delay**

igpibllo

Description Puts all GPIB devices into local-lockout mode.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibllo(INST id);
```

id GPIB interface session handle.

Visual Basic Synopsis

Declare Sub **igpibllo** Lib "sicl16.dll" (ByVal *id* As Integer)

Remarks This function sends the GPIB LLO (local lockout) command to all devices on the GPIB interface of the session specified by *id*.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, itimeout**

Example

```
/*
 * igpibllo.c: this example uses igpibllo() to put all GPIB devices into
 *         local-lockout mode.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_igpibllo(void)
{
     int  error_number;
     INST id;

     /* Open a GPIB interface session. */

     id = iopen("gpib");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
```

```
                igeterrstr(error_number),
                error_number);
        return (error_number);
    }

    /* Send the LLO command. */

    error_number = igpibllo(id);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  igpibllo().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
    }
    iclose(id);
    return (error_number);
}
```

igpibpassctl

Description Passes active controller status to another GPIB interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibpassctl(INST id, int busaddress);
```

id	GPIB interface session handle.
busaddress	GPIB address of new active controller interface.

Visual Basic Synopsis

```
Declare Sub igpibpassctl Lib "sicl16.dll" (ByVal id As Integer,
ByVal busaddr As Integer)
```

Remarks This function passes active controller state from the GPIB interface of the session specified by *id* to the GPIB interface whose address is *busaddress*.

Busaddress must be between zero and 30, inclusive.

Although the interface can pass active controller status, the interface always assumes it is the system controller and can regain active controller status by asserting REN and performing an IFC.

Note that passing control fundamentally alters the behavior of the SICL driver on this interface. Having passed control, no other process will be able to execute most SICL calls on this interface until active control is regained. Closing the SICL session that passed control has no effect on this global state.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, itimeout**

Example

```
/*
 * passctl.c:  this example uses igpibpassctl() to pass active control to
 *         another GPIB interface.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_igpibpassctl(void)
{
     int  error_number;
     int  primary;
     int  secondary;
     INST id;

     /* Open a GPIB device session by name. */

     id = iopen("gpibsink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Get the device's primary address. */

     error_number = igetdevaddr(id, &primary, &secondary);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetdevaddr().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }
     iclose(id);

     /* Open a GPIB interface session by name. */

     id = iopen("gpib");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Pass active controller status to the device. */

     error_number = igpibpassctl(id, primary);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igpibpassctl().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
```

```
    }
    iclose(id);
    return (error_number);
}
```

igpibppoll

Description Executes a parallel poll.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibppoll(INST id, unsigned int _far *polldata);
```

id	GPIB interface session handle.
polldata	Pointer to the location where the function stores the parallel poll result.

Visual Basic Synopsis

Declare Sub **igpibppoll** Lib "sicl16.dll" (ByVal *id* As Integer, *result* As Integer)

Remarks This function executes a parallel poll of the GPIB interface of the session referenced by *id*. The parallel poll results are placed in the lower eight bits of the location specified by *polldata*.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, igpibppollconfig, igpibppollresp, itimeout**

Example

```
/*
 * gpibpoll.c:  this example calls igpibpollconfig() to configure a device's
 *         response to a parallel poll.  Additionally, it calls
 *         igpibppoll() to verify correct execution of the poll
 *         configuration call.
 */

#include "sicl.h"

#define POLL_CONFIG  0x47 /* GPIB response line 7, no service req. */

void
WinPrintf(char _far *Format_String, ...);

int
sample_igpibppoll(void)
{
     int       error_number;
     unsigned int    poll_data;
     INST      id;

     /* Open a GPIB device session. */

     id = iopen("gpibsink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Configure the device's parallel poll response and close the     */
     /* session.                                                        */

     error_number = igpibppollconfig(id, POLL_CONFIG);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igpibppollconfig().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }
     iclose(id);

     /* Open a GPIB interface session. */

     id = iopen("gpib");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Execute a parallel poll. */

     error_number = igpibppoll(id, &poll_data);
     if (error_number != I_ERR_NOERROR)
```

```
    {
        WinPrintf("FAILURE:  igpibppoll().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        iclose(id);
        return (error_number);
    }
    if (poll_data != 0x80)
    {
        WinPrintf("FAILURE:      parallel   poll   received   0x%08X,   expected
0x%08X.\n",
                poll_data,
                1 << (POLL_CONFIG & 0x0f));
    }
    else
    {
        WinPrintf("Poll data = 0x%08X",
                poll_data);
    }
    iclose(id);
    return (error_number);
}
```

igpibppollconfig

Description Configures a GPIB device's response to a parallel poll.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibppollconfig(INST id, int configparam);
```

id	GPIB device or commander session handle.
configparam	Device configuration.

Visual Basic Synopsis

Declare Sub **igpibppollconfig** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *cval* As Integer)

Remarks This function configures the parallel poll response of the GPIB device session specified by *id*. *Configparam* specifies the GPIB device's response to future parallel polls.

Specifying *configparam* equal to -1 disables the device from responding to parallel polling. Specifying *configparam* greater than or equal to zero enables the device's response to a parallel poll. The lower four bits of *confgiparam* configure the parallel poll response. Bits 0, 1, and 2 specify the GPIB response lines. Bit 3 specifies the meaning of a parallel poll response (1=service request, 0=no service request.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, itimeout**

Example See **igpibppoll**

igpibppollresp

Description Sets the state of a device's PPOLL response bit.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibppollresp(INST id, int val);
```

id	GPIB device or commander session handle.
value	State of the PPOLL bit.

Visual Basic Synopsis

Declare Sub **igpibppollresp** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *sval* As Integer)

Remarks This function checks for errors and returns.

This function sets the state of the parallel poll in the specified device's parallel poll response bit for subsequent parallel polls by its commander.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, igpibppoll, igpibppollconfig**

igpibrenctl

Description Controls the state of the GPIB REN line.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibrenctl(INST id, int renstate);
```

id	GPIB interface session handle.
renstate	REN line state.

Visual Basic Synopsis

Declare Sub **igpibrenctl** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *ren* As Integer)

Remarks This function defines the REN line state of the GPIB interface of the session specified by *id*.

Specifying a *renstate* equal to zero deasserts the REN line. Specifying *renstate* as non-zero asserts the REN line.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, itimeout**

Example

```
/*
 * gpibren.c:  this example uses igpibrenctl() to configure the GPIB REN line.
 */

#include "sicl.h"

#define REN_DEASSERT 0
#define REN_ASSERT   1

void
WinPrintf(char _far *Format_String, ...);

int
sample_igpibrenctl(void)
{
      int  error_number;
      INST id;

      /* Open a GPIB interface session. */

      id = iopen("gpib");
      if (id == ((INST) 0))
      {
            error_number = igeterrno();
            WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                      igeterrstr(error_number),
                      error_number);
            return (error_number);
      }

      /* Assert the REN line. */

      error_number = igpibrenctl(id, REN_ASSERT);
      if (error_number != I_ERR_NOERROR)
      {
            WinPrintf("FAILURE:  igpibrenctl().  Error = %s (%d).\n",
                      igeterrstr(error_number),
                      error_number);
      }
      iclose(id);
      return (error_number);
}
```

igpibsendcmd

Description Writes command bytes to a GPIB interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibsendcmd(INST id, char _far *buffer, int buffersize);
```

id	GPIB interface session handle.
buffer	Pointer to a data source buffer.
buffersize	Data buffer size, in bytes.

Visual Basic Synopsis

Declare Sub **igpibsendcmd** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *buf* As String, ByVal *length* As Integer)

Remarks This function writes data from the buffer pointed to by *buffer* to the GPIB interface of the session specified by *id* with the ATN line asserted. *Buffersize* specifies the number of data bytes in the buffer.

The function does not parse the command data in the specified buffer. Sending command data that changes the state of the GPIB interface may not be correctly reflected in the EPC's GPIB hardware state. Therefore, do not use the function to change the state of the GPIB interface. For example, to pass active controller status, use **igpibpassctl** rather than simply sending command data via **igpibsendcmd**.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**, **itimeout**

Example

```
/*
 * gpibcmd.c:  this example uses igpibsendcmd() to send commands to the GPIB
 *         interface.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_igpibsendcmd(void)
{
     char command_buf[5] = { 0 };
     int  buf_length;
     int  dev_primary;
     int  dev_secondary;
     int  error_number;
     int  itf_primary;
     INST dev_id;
     INST itf_id;

     /* Open a GPIB interface session. */

     itf_id = iopen("gpib");
     if (itf_id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query the GPIB interface primary address. */

     error_number = igpibbusstatus( itf_id,
                         I_GPIB_BUS_ADDR,
                         &itf_primary);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igpibbusstatus().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(itf_id);
          return (error_number);
     }

     /* Open a GPIB device session. */

     dev_id = iopen("gpibsink");
     if (dev_id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(itf_id);
          return (error_number);
     }

     /* Query the GPIB device's primary and secondary addresses. */
```

```
    error_number = igetdevaddr(dev_id, &dev_primary, &dev_secondary);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  igetdevaddr().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         iclose(dev_id);
         iclose(itf_id);
         return (error_number);
    }

    /* Send GPIB commands preparing the device to listen. */

    command_buf[0] = 0x3F;                              /* UNL */
    command_buf[1] = (char) (itf_primary + 0x40);       /* MTA */
    command_buf[2] = (char) (dev_primary + 0x20);       /* LAG */
    if (dev_secondary == -1)
    {
         buf_length = 3;
    }
    else
    {
         command_buf[3] = (char) (dev_secondary + 0x60);     /* SCG */
         buf_length = 4;
    }
    error_number = igpibsendcmd(itf_id, command_buf, buf_length);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  igpibsendcmd().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
    }
    iclose(dev_id);
    iclose(itf_id);
    return (error_number);
}
```

igpibsett1delay

Description Sets the t1 delay on the GPIB interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
igpibsett1delay(INST id, int delay);
```

id	GPIB interface session handle.
delay	Time, in nanoseconds.

Visual Basic Synopsis

Declare Sub **igpibsett1delay** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *delay* As Integer)

Remarks This function sets the t1 delay on the GPIB interface specified by *id*. The value is the time of t1 delay in nanoseconds, and should be no less than **I_GPIB_T1DELAY_MIN** or no greater than **I_GPIB_T1DELAY_MAX**.

Note that most GPIB interfaces only support a small number of t1 delays, so the actual value used by the interface could be different than that specified in the **igpibsett1delay** function. You can query the actual value used by calling the **igpibgett1delay** function.

Return Value The function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also **igpibgett1delay**

ihint

Description Defines the type of communication a device driver should use.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ihint(INST id, int hint);
```

id	Session handle.
hint	Communications type.

Visual Basic Synopsis

```
Declare Sub ihint Lib "sicl16.dll" (ByVal id As Integer, ByVal hint
As Integer)
```

Remarks This function defines the methodology to use in communicating with an interface.

Valid *hint* constants are:

Constant	Description
I_HINT_DONTCARE	No communications preference.
I_HINT_IO	Optimize I/O performance, possibly at the expense of system performance.
I_HINT_SYSTEM	Optimize system performance, possibly at the expense of I/O performance.
I_HINT_USEDMA	Use DMA, if possible.
I_HINT_USEINTR	Use interrupts, if possible.
I_HINT_USEPOLL	Use polling, if possible.

The hint parameter is only a suggestion to the driver software, and may be ignored.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

iintroff

Description Disables SRQ and interrupt event processing.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iintroff(void);
```

Visual Basic Synopsis

None

Remarks This function disables processing of SRQ and interrupt events for the calling process.

When event processing is disabled, SRQ and interrupt events are queued.

By default, SRQ and interrupt event processing is enabled.

Use **iintron** to re-enable SRQ and interrupt event processing.

SRQ and interrupt event disabling can be nested. Each call to **iintroff** should be paired with one, and only one, call to **iintron**.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iintron**

Example See **ionintr**.

iintron

Description Enables SRQ and interrupt event processing.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iintron(void);
```

Visual Basic Synopsis

None

Remarks This function enables processing of SRQ and interrupt events for the calling process.

By default, SRQ and interrupt event processing is enabled.

Use **iintroff** to disable SRQ and interrupt event processing.

Attempting to enable SRQ and interrupt event processing when it is already enabled results in an **I_ERR_OS** error.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iintroff, ionintr, ionsrq, isetintr**

Example See **ionintr**.

ilblockcopy

Description Copies a block of 32-bit words from one set of sequential memory locations to another.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ilblockcopy(INST id, unsigned long _far *src, unsigned long
    _far *dest, unsigned long count, int swap);
```

id	Session handle.
src	Source pointer.
dest	Destination pointer.
count	Number of 32-bit words to copy.
swap	Byte swap flag.

Visual Basic Synopsis

Declare Sub **ilblockcopy** Lib "sicl16.dll" (ByVal *id* As Integer, *src* As Any, *dest* As Any, ByVal *cnt* As Long, ByVal *swap* As Integer)

Remarks Copies 32-bit words from successive memory locations beginning at *src* into successive memory locations beginning at *dest*. *Count* specifies the number of 32-bit words to transfer. *Id* specifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Whether or not byte-swapping occurs depends upon the source and destination of the copy operation. The swap flag is ignored.

The following scenarios are possible when accessing EPC and VXIbus memory:

src	*dest*	*Result*
EPC	EPC	No byte-swapping
EPC	VXI	One byte-swap
VXI	EPC	One byte-swap
VXI	VXI	Two byte-swaps (equals no byte-swapping)

For byte-swapping to work properly, all 32-bit VXIbus accesses must be aligned on a 32-bit boundary.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibblockcopy, ilpeek, ilpoke, ilpopfifo, ilpushfifo, imap, iwblockcopy**

Example

```
/*
 * ilblock.c:  this example uses ilblockcopy() to read/write this EPC's
 *         slave memory via the VXIbus.
 */

#include <windows.h>
#include "sicl.h"

#define NO_BYTE_SWAP 0
#define BYTE_SWAP    1

void
WinPrintf(char _far *Format_String, ...);

int
sample_ilblockcopy(void)
{
     volatile char _far * mapped_ptr;
     int             error_number;
     unsigned long        address_space;
     unsigned long        base_address;
     unsigned long        memory_data;
     INST            id;

     /* Open a VXI interface session. */

     id = iopen("vxi");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
```

```
                error_number);
        return (error_number);
}

/* Query the location of our slave memory. */

error_number = ivxibusstatus(  id,
                        I_VXI_BUS_SHM_ADDR_SPACE,
                        &address_space);
if (error_number != I_ERR_NOERROR)
{
        WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        iclose(id);
        return (error_number);
}
if (address_space == 0)
{
        WinPrintf("FAILURE:  the EPC's slave memory is not enabled.\n");
        iclose(id);
        return (error_number);
}
error_number = ivxibusstatus(  id,
                        I_VXI_BUS_SHM_PAGE,
                        &base_address);
if (error_number != I_ERR_NOERROR)
{
        WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        iclose(id);
        return (error_number);
}
iclose(id);

/* Open a VXI device session. */

id = iopen("vxisink");
if (id == ((INST) 0))
{
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        return (error_number);
}

/* Map in the first 64K of the EPC's slave memory. */

if (address_space == 24)
{
        mapped_ptr = imap(   id,
                        I_MAP_A24,
                        (unsigned int) (base_address >> 8),
                        1,
                        NULL);
}
else
{
        mapped_ptr = imap(   id,
                        I_MAP_A32,
                        (unsigned int) base_address,
                        1,
                        NULL);
```

```
    }
    if (mapped_ptr == NULL)
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        iclose(id);
        return (error_number);
    }

    /* Read a 32-bit value from physical address 0 of EPC memory  */
    /* via the VXIbus, then write the value back.                  */

    error_number = ilblockcopy(    id,
                        (unsigned long *) mapped_ptr,
                        &memory_data,
                        1,
                        BYTE_SWAP);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  ilblockcopy().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        iclose(id);
        return (error_number);
    }
    error_number = ilblockcopy(    id,
                        &memory_data,
                        (unsigned long *) mapped_ptr,
                        1,
                        BYTE_SWAP);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  ilblockcopy().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
    }
    iclose(id);
    return (error_number);
}
```

ileswap

Description Byte-swaps a buffer of data from Intel (little-endian) byte order to the native byte order of the EPC.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ileswap(char _far * buf, unsigned long length, int datasize);
```

buf	Pointer to a data buffer.
length	Length of the data buffer, in bytes.
datasize	Size of data elements in the data buffer, in bytes.

Visual Basic Synopsis

Declare Sub **ileswap** Lib "sicl16.dll" (*addr* As Any, ByVal *length* As Long, ByVal *datasize* As Integer)

Remarks Since the native byte order of an EPC is Intel (little-endian) byte order, this function simply checks the parameters for errors and returns.

Length must be a multiple of datasize.

Datasize may be 1, 2, 4, or 8 bytes.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibeswap, iswap**

Example See **iswap**

ilocal

Description Puts a device in local mode.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ilocal(INST id);
```

id Session handle.

Visual Basic Synopsis

```
Declare Sub ilocal Lib "sicl16.dll" (ByVal id As Integer)
```

Remarks For VXI device sessions, the function issues a CLEAR LOCK word serial command to the device. The function only supports message-based VXI devices; other VXI devices cause an error.

For GPIB device sessions, the function addresses the device to listen, then sends the GTL (go to local) command.

This function supports only device sessions. Specifying an interface session is an error.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iremote, itimeout**

Example

```
/*
 * ilocal.c:  this example uses ilocal() to put a GPIB device into local mode.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ilocal(void)
{
     int  error_number;
     INST id;

     /* Open a GPIB device session. */

     id = iopen("gpibsink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Send a GTL (Go To Local) command to the session's device */

     error_number = ilocal(id);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  ilocal().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     iclose(id);
     return (error_number);
}
```

ilock

Description Locks a device or interface session.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ilock(INST id);
```

id Session handle.

Visual Basic Synopsis

```
Declare Sub ilock Lib "sicl16.dll" (ByVal id As Integer)
```

Remarks This function locks the session specified by *id* to prevent device or interface access by other sessions.

Locking an interface session prevents all other device and interface sessions from accessing an interface. Only the locking session can access the interface.

Locking a device session prevents other device sessions from accessing a device. Only the locking session can access the device. Locking a device session does not prevent other device sessions from accessing other devices, nor does it prevent interface sessions from accessing the interface (or any device on the interface).

Locks can be nested. Each **ilock** call must be paired with a corresponding **iunlock** call.

Locking conflict resolution for a session is determined using **isetlockwait**.

Locking affects these SICL functions:

iclear	**igpibsendcmd**	**isetstb**
iflush	**igpibsett1delay**	**isetubuf**
ifread	**ilocal**	**itrigger**
ifwrite	**ilock**	**ivprintf**
igpibatnctl	**imap**	***ivpromptf***
igpibgett1delay	**iprintf**	**ivscanf**
igpibllo	**ipromptf**	**ivxitrigoff**
igpibpassctl	**iread**	**ivxitrigon**
igpibppoll	**ireadstb**	**ivxitrigroute**
igpibppollconfig	**iremote**	**ivxiws**
igpibppollresp	**iscanf**	**iwrite**
igpibrenctl	**isetbuf**	**ixtrig**

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **itimeout**, **iunlock**

Example

```
/*
 * ilock.c:  this example uses ilock()/iunlock() to lock access to a device.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ilock(void)
{
     int  error_number;
     INST id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Lock the session */

     error_number = ilock(id);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  ilock().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Critical section code goes here... */

     /* Explicitly unlock the session. */

     error_number = iunlock(id);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  iunlock().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     iclose(id);
     return (error_number);
}
```

ilpeek

Description Reads a 32-bit word from a mapped address.

C Synopsis

#include "sicl.h"

unsigned long SICLAPI
ilpeek(volatile unsigned long _far ***addr*);**

addr Address of a 32-bit word.

Visual Basic Synopsis

Declare Function **ilpeek** Lib "sicl16.dll" Alias "ilpeek" (ByVal *addr* As Long) As Long

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call. Byte swapping is always performed.

For byte-swapping to work properly, all 32-bit VXIbus accesses must be aligned on a 32-bit boundary.

Return Value The function returns the 32-bit word stored at *addr*.

See Also **ibpeek, ilpoke, imap, iwpeek**

Example

```
/*
 * ilpeek.c:    this example uses ilpeek()/ilpoke() to read/write this EPC's
 *         slave memory via the VXIbus.
 */

#include <windows.h>
#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ipeek(void)
{
     volatile char _far * mapped_ptr;
     int             error_number;
     unsigned long        address_space;
     unsigned long        base_address;
     unsigned long        memory_data;
```

```
INST            id;

/* Open a VXI interface session. */

id = iopen("vxi");
if (id == ((INST) 0))
{
     error_number = igeterrno();
     WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     return (error_number);
}

/* Query the location of our slave memory. */

error_number = ivxibusstatus(  id,
                  I_VXI_BUS_SHM_ADDR_SPACE,
                  &address_space);
if (error_number != I_ERR_NOERROR)
{
     WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     iclose(id);
     return (error_number);
}
if (address_space == 0)
{
     WinPrintf("FAILURE:  the EPC's slave memory is not enabled.\n");
     iclose(id);
     return (error_number);
}
error_number = ivxibusstatus(  id,
                  I_VXI_BUS_SHM_PAGE,
                  &base_address);
if (error_number != I_ERR_NOERROR)
{
     WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     iclose(id);
     return (error_number);
}
iclose(id);

/* Open a VXI device session. */

id = iopen("vxisink");
if (id == ((INST) 0))
{
     error_number = igeterrno();
     WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     iclose(id);
     return (error_number);
}

/* Map in the first 64K of the EPC's slave memory. */

if (address_space == 24)
{
     mapped_ptr = imap(   id,
                  I_MAP_A24,
```

```
                        (unsigned int) (base_address >> 8),
                        1,
                        NULL);
    }
    else
    {
        mapped_ptr = imap(   id,
                        I_MAP_A32,
                        (unsigned int) base_address,
                        1,
                        NULL);
    }
    if (mapped_ptr == NULL)
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        iclose(id);
        return (error_number);
    }

    /* Read a 32-bit value from physical address 0 of EPC memory  */
    /* via the VXIbus, then write the value back.                  */

    memory_data = ilpeek((volatile unsigned long _far *) mapped_ptr);
    ilpoke((volatile unsigned long _far *) mapped_ptr, memory_data);
    iclose(id);
    return (I_ERR_NOERROR);
}
```

ilpoke

Description Writes a 32-bit word to a mapped address.

C Synopsis

```
#include "sicl.h"

void SICLAPI
ilpoke(volatile unsigned long _far *dest, unsigned long value);
```

dest	Destination address.
value	32-bit word to write.

Visual Basic Synopsis

Declare Sub **ilpoke** Lib "sicl16.dll" Alias "ilpoke" (ByVal *addr* As Long, ByVal *value* As Long)

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call. Byte swapping is always performed.

For byte-swapping to work properly, all 32-bit VXIbus accesses must be aligned on a 32-bit boundary.

Return Value The function returns no value.

See Also **ibpoke**, **ilpeek**, **imap**, **iwpoke**

Example See **ilpeek**

ilpopfifo

Description Copies 32-bit words from a single memory location (FIFO register) to sequential memory locations.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ilpopfifo(INST id, unsigned long _far *fifo, unsigned long _far
    *dest, unsigned long count, int swap);
```

id	Session handle.
fifo	FIFO pointer.
dest	Destination address.
count	Number of 32-bit words to copy.
swap	Byte swap flag.

Visual Basic Synopsis

Declare Sub **ilpopfifo** Lib "sicl16.dll" (ByVal *id* As Integer, *fifo* As Any, *dest* As Any, ByVal *cnt* As Long, ByVal *swap* As Integer)

Remarks This function copies *count* 32-bit words from *fifo* into sequential memory locations beginning at *dest*. *Count* specifies the number of 32-bit words to transfer. *Id* specifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Whether or not byte-swapping occurs depends upon the source and destination of the copy operation. The swap flag is ignored. The following scenarios are possible when accessing EPC and VXIbus memory:

src	*dest*	Result
EPC	EPC	No byte-swapping
EPC	VXI	One byte-swap
VXI	EPC	One byte-swap
VXI	VXI	Two byte-swaps (equals no byte-swapping)

For byte-swapping to work properly, all 32-bit VXIbus accesses must be aligned on a 32-bit boundary.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibpopfifo, ilpushfifo, imap, iwpopfifo**

Example

```
/*
 * ilpop.c:     this example uses ilpopfifo() to read from a hypothetical
 *         FIFO at address 0 in A16 space.
 */

#include <windows.h>
#include "sicl.h"

#define NO_BYTE_SWAP 0
#define BYTE_SWAP    1

void
WinPrintf(char _far *Format_String, ...);

int
sample_ipopfifo(void)
{
     volatile char _far * mapped_ptr;
     unsigned long        fifo_data[5];
     int             error_number;
     INST            id;

     /* Open a VXI interface session. */

     id = iopen("vxi");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }
```

```
    /* Map in A16 space */

    mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
    if (mapped_ptr == NULL)
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        iclose(id);
        return (error_number);
    }

    /* Read the FIFO 5 times, storing the values into fifo_data[]. */

    error_number = ilpopfifo( id,
                        (unsigned long *) mapped_ptr,
                        fifo_data,
                        sizeof(fifo_data) / sizeof(unsigned long),
                        BYTE_SWAP);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  ilpopfifo().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
    }
    iclose(id);
    return (error_number);
}
```

ilpushfifo

Description Copies 32-bit words from sequential memory locations to a single 32-bit wide memory location (FIFO register).

C Synopsis

```
#include "sicl.h"

int SICLAPI
ilpushfifo(INST id, unsigned long _far *src, unsigned long _far
    *fifo, unsigned long count, int swap);
```

id	Session handle.
src	Source address.
fifo	FIFO pointer.
count	Number of 32-bit words to copy.
swap	Byte swap flag.

Visual Basic Synopsis

Declare Sub **ilpushfifo** Lib "sicl16.dll" (ByVal *id* As Integer, *src* As Any, *fifo* As Any, ByVal *cnt* As Long, ByVal *swap* As Integer)

Remarks Copies *count* 32-bit words from the sequential memory locations beginning at *src* into the FIFO at *fifo*. *Count* specifies the number of 32-bit words to transfer. *Id* specifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Whether or not byte-swapping occurs depends upon the source and destination of the copy operation. The swap flag is ignored.

The following scenarios are possible when accessing EPC and VXIbus memory:

src	*dest*	Result
EPC	EPC	No byte-swapping
EPC	VXI	One byte-swap
VXI	EPC	One byte-swap
VXI	VXI	Two byte-swaps (equals no byte-swapping)

For byte-swapping to work properly, all 32-bit VXIbus accesses must be aligned on a 32-bit boundary.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibpushfifo, ilpopfifo, imap, iwpushfifo**

Example

```
/*
 * ilpush.c:    this example uses ilpushfifo() to write to a hypothetical
 *         FIFO at address 0 in A16 space.
 */

#include <windows.h>
#include "sicl.h"

#define NO_BYTE_SWAP 0
#define BYTE_SWAP    1

unsigned long fifo_data[] =
{
     0x53616D70, 0x6C654461, 0x74615361, 0x6D706C65, 0x44617461
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_ipushfifo(void)
{
     volatile char _far * mapped_ptr;
     int             error_number;
     INST            id;

     /* Open a VXI interface session. */

     id = iopen("vxi");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
```

```
    }

    /* Map in A16 space */

    mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
    if (mapped_ptr == NULL)
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        iclose(id);
        return (error_number);
    }

    /* Write the FIFO 5 times, storing the values from fifo_data[]. */

    error_number = ilpushfifo(     id,
                        fifo_data,
                        (unsigned long *) mapped_ptr,
                        sizeof(fifo_data) / sizeof(unsigned long),
                        BYTE_SWAP);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  ilpushfifo().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
    }
    iclose(id);
    return (error_number);
}
```

imap

Description Maps a portion of a VXIbus memory space into an application's address space.

C Synopsis

```
#include "sicl.h"

char _far * SICLAPI
imap(INST id, int mapspace, unsigned int pagestart, unsigned int
    pagecnt, char _far *suggestedaddress);
```

id	Session handle.
mapspace	Address space to map.
pagestart	Starting page number.
pagecnt	Number of pages to map.
suggestedaddress	User suggested pointer to the mapped memory location.

Visual Basic Synopsis

Declare Function **imap** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *mapspace* As Integer, ByVal *pagestart* As Integer, ByVal *pagecnt* As Integer, ByVal *suggested* As Long) As Long

Remarks The address space to be mapped depends on *id* and *mapspace*. The following are valid constants for *mapspace*:

Constant	Description
I_MAP_A16	Map the A16 address space. Valid for VXI device and interface sessions.
I_MAP_A24	Map the A24 address space (page size 64K bytes). Valid for VXI device and interface sessions.
I_MAP_A32	Map the A32 address space (page size 64K bytes). Valid for VXI device and interface sessions.

I_MAP_VXIDEV	Map a VXI device's configuration registers. Valid only for VXI device sessions.
I_MAP_EXTEND	Map a VXI device's A24/A32 memory (page size 64 Kbytes). Valid only for VXI device sessions.
I_MAP_SHARED	Map the EPC's shared memory (page size 64 Kbytes). Valid for all VXI sessions.

Pagestart is the offset, in 64K pages, into the specified address space. *Pagecnt* is the amount of memory, in 64K pages, to map.

The *suggestedaddress* parameter is not used.

When *mapspace* is either **I_MAP_A16** or **I_MAP_VXIDEV**, the *pagestart* and *pagecnt* variables are not used.

EPC hardware limits **I_MAP_A32** mapping to the lower 1 Gigabyte of A32 space (pages 0 through 0x3FFF, inclusive).

When *mapspace* is **I_MAP_EXTEND**, the device's A16 registers determine the location of the address space.

Use **imapinfo** to calculate a valid *pagestart* and *pagecnt* for a given address space.

Although **imap** returns a pointer to the designated portion of VXIbus, the pointer cannot be used directly because the byte order is not defined. Byte order is defined when the returned pointer is used in a memory-mapped I/O function.

Unmap an address space when it is no longer needed to free operating system and/or hardware resources.

The action taken by **imap** when insufficient resources are available to complete a mapping depends on the session's lock wait flag (as set using **isetlockwait**). If the session's lock wait flag is zero, then **imap** returns **I_ERR_LOCKED.** If the session's lock wait flag is non-zero, **imap** suspends execution of the calling thread until sufficient resources become available or the session's timeout expires. If the session's timeout expires before sufficient resources become available, the function returns **I_ERR_TIMEOUT.**

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **imapinfo, iopen, isetlockwait, iunmap**

Example

```
/*
 * imap.c: this example uses imap()/iunmap() to map a VXI device's A16
 *         registers into the application's memory space.
 */

#include <windows.h>
#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_imap(void)
{
     volatile char _far *mapped_ptr;
     int             error_number;
     INST            id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Map in the device's A16 registers and print the device's   */
     /* manufacturer id.                                          */

     mapped_ptr = imap(id, I_MAP_VXIDEV, 0, 0, NULL);
     if (mapped_ptr == NULL)
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
```

```
        return (error_number);
    }
    WinPrintf("Device \"vxisink\" manufacturer ID = 0x%04X",
            iwpeek((volatile unsigned short _far *) mapped_ptr) & 0x0FFF);

    /* Explicitly unmap the device's A16 registers. */

    error_number = iunmap(id, (char _far *) mapped_ptr, 0, 0, 0);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  iunmap().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
    }
    iclose(id);
    return (error_number);
}
```

imapinfo

Description Queries address space mapping capabilities for the specified interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
imapinfo(INST id, int mapspace, int _far *numwindows, int _far
    *windowsize);
```

id	Session handle.
mapspace	Address space.
numwindows	Pointer to a location where the function stores the total number of mapping windows.
windowsize	Pointer to a location where the function stores the mapping window size, in pages.

Visual Basic Synopsis

Declare Sub **imapinfo** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *mapspace* As Integer, *numwindows* As Integer, *winsize* As Integer)

Remarks This function queries the number of mapping windows available and the size of each window for the specified *mapspace*. It does not identify which windows are in use by another process.

Use **imap** to access bus memory through the mapping windows.

The following constants define valid values for *mapspace*:

Constant	Description
I_MAP_A16	The A16 address space
I_MAP_A24	The A24 address space (page size 64K bytes)
I_MAP_A32	The A32 address space (page size 64K bytes)

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **imap, iopen**

Example

```
/*
 * imapinfo.c: this example calls imapinfo() to determine the EPC's mapping
 *         window(s).
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_imapinfo(void)
{
     INST id;
     int  error_number;
     int  window_count;
     int  window_size;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query and print the EPC's A16, A24, and A32 window attributes. */

     error_number = imapinfo(id, I_MAP_A16, &window_count, &window_size);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  imapinfo().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
```

```
        return (error_number);
    }
    WinPrintf("The VXI interface supports %d %d-page A16 windows.\n",
              window_count,
              window_size);
    error_number = imapinfo(id, I_MAP_A24, &window_count, &window_size);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  imapinfo().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        iclose(id);
        return (error_number);
    }
    WinPrintf("The VXI interface supports %d %d-page A24 windows.\n",
              window_count,
              window_size);
    error_number = imapinfo(id, I_MAP_A32, &window_count, &window_size);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  imapinfo().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
    }
    else
    {
        WinPrintf("The VXI interface supports %d %d-page A32 windows.\n",
                  window_count,
                  window_size);
    }
    iclose(id);
    return (error_number);
}
```

ionerror

Description Installs an error handler.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ionerror(errorproc_t errorhandler);
```

errorhandler Pointer to an error handler function.

Visual Basic Synopsis

None

Remarks This function installs the function specified by *errorhandler* as the function to call when an error occurs.

The SICL library assumes error handler functions have the following calling semantics:

```
void SICLCALLBACK
errorhandler(INST id, int error);
```

where *id* identifies the device or interface session generating the error and *error* is an error constant defining the error. SICL defines two default error handlers:

Error Handler	Description
I_ERROR_EXIT	Writes an error message to the SICL message log and terminates the process.
I_ERROR_NO_EXIT	Writes an error message to the SICL message log and allows process to continue.

The SICL message log can be viewed using the application **C:\SICL\BIN\ILOG.EXE.**

Installing a null error handler removes the current error handler.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igetonerror**

Example

```
/*
 * igetone.c:  this example uses the error handler functions to manipulate
 *         the error handler.
 */

#include <windows.h>
#include "sicl.h"

volatile int HandlerExecuted;

void
WinPrintf(char _far *Format_String, ...);

void SICLCALLBACK
ErrorHandler(INST Id, int Error_Number)
{
     char _far *Address_Ptr;

     HandlerExecuted = 1;
     igetaddr(Id, &Address_Ptr);
     WinPrintf("Error %s detected for session \"%s\".\n",
               igeterrstr(Error_Number),
               Address_Ptr);
}

int
sample_ionerror(void)
{
     int       error_number;
     errorproc_t    old_handler;
     INST      id;

     /* Open a VXI interface session. */

     id = iopen("vxi");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query and save the previously installed error handler. */

     error_number = igetonerror(&old_handler);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetonerror().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
```

```
        iclose(id);
        return (error_number);
    }

    /* Install the error handler ErrorHandler(). */

    error_number = ionerror((errorproc_t) ErrorHandler);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  ionerror().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        iclose(id);
        return (error_number);
    }

    /* Intentionally generate a SICL error and verify handler execution. */

    HandlerExecuted = 0;
    imap(id, I_MAP_VXIDEV, 0, 0, NULL);
    if (HandlerExecuted == 0)
    {
        WinPrintf("FAILURE:  error handler did not execute!\n");
    }
    else
    {
        WinPrintf("Error handler successfully executed.\n");
    }

    /* Force a user-defined error and verify handler execution. */

    HandlerExecuted = 0;
    icauseerr(id, I_ERR_SYNTAX, 1);
    if (HandlerExecuted == 0)
    {
        WinPrintf("FAILURE:  error handler did not execute!\n");
    }
    else
    {
        WinPrintf("Error handler successfully executed.\n");
    }

    /* Restore the original the error handler. */

    error_number = ionerror(old_handler);
    if (error_number != I_ERR_NOERROR)
    {
        WinPrintf("FAILURE:  ionerror().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
    }
    iclose(id);
    return (error_number);
}
```

ionintr

Description Installs a session's interrupt handler.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ionintr(INST id, intrhandler_t intrhandler);
```

id	Session handle.
intrhandler	Pointer to an interrupt handler function.

Visual Basic Synopsis

None

Remarks This function installs the function specified by *intrhandler* as the function to call when the device or interface session specified by *id* processes an interrupt event.

The SICL library assumes that interrupt handler functions have the following calling semantics:

```
void SICLCALLBACK
intrhandler(INST id, long data1, long data2);
```

where *id* identifies the device or interface session receiving the interrupt, *data1* identifies the interrupt (**I_INTR_TRIG, I_INTR_VXI_SIGNAL**, etc.). *Data2* has meaning only for **I_INTR_GPIB_TLAC** interrupts to GPIB commander sessions, and **I_INTR_TRIG** interrupts to VXI interface sessions.

For **I_INTR_GPIB_TLAC** interrupts to GPIB commander sessions, *Data2* is a bit mask where bit 0 indicates whether the device is addressed to listen and bit 1 indicates whether the device is addressed to talk.

For **I_INTR_TRIG** interrupts to VXI interface sessions, *Data2* identifies the trigger causing the interrupt.

On an EPC-7, *Data2* may be one of the following:

Constant	Description
I_TRIG_TTL0	TTL trigger 0.
I_TRIG_TTL1	TTL trigger 1.
I_TRIG_TTL2	TTL trigger 2.
I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.
I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

On a VXLink interface, *Data2* may be one of the following:

Constant	Description
I_TRIG_TTL0	External input trigger.
I_TRIG_TTL1	External output trigger.
I_TRIG_TTL2	TTL trigger 2.
I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.
I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

Proper VXI TTL trigger interrupt operation on an EPC-7 requires software intervention. Refer to Chapter 3, *Advanced Topics*, for additional information.

This function does not enable interrupt reception or processing. See **isetintr** to disable/enable interrupt reception and **iintroff** and **iintron** to disable and enable interrupt processing, respectively. By default, interrupt processing is enabled.

Note the difference between interrupt reception and interrupt processing. Refer to Chapter 3, *Advanced Topics*, for more information.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igetonintr, iintroff, iintron, isetintr, ivxitrigroute**

Example

```
/*
 * ionintr.c:  this example sets an interrupt handler, then generates and
 *         processes an interrupt (this program assumes it's executing on
 *         an EPC-7).
 */

#include "olrm.h"
#include "sicl.h"

#define ENABLE_INTERRUPT  1
#define NO_CAUSE_GIVEN          0xFF00
#define SIG_REG_OFFSET          0x0008

volatile int HandlerExecuted;

void
WinPrintf(char _far *Format_String, ...);

int
GenerateInterrupt(INST Id)
{
     volatile char _far * mapped_ptr;
     int             error_number;
     unsigned long        olrm_data;

     /* Generate an I_INTR_VXI_SIGNAL interrupt by writing a  */
     /* NO CAUSE GIVEN event from the servant device to the   */
     /* EPC's signal register.                                */

     (void) OlrmGetNumAttr("vxisink", VXI_ULA, &olrm_data);
     mapped_ptr = imap(Id, I_MAP_VXIDEV, 0, 0, NULL);
     if (mapped_ptr == NULL)
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }
     iwpoke(    (volatile unsigned short _far *) (mapped_ptr + SIG_REG_OFFSET),
          (unsigned short) (olrm_data | NO_CAUSE_GIVEN));
     iunmap(Id, (char _far *) mapped_ptr, 0, 0, 0);
}

void SICLCALLBACK
InterruptHandler(INST Id, long Data1, long Data2)
{
     char _far *Address_Ptr;

     HandlerExecuted = 1;
     igetaddr(Id, &Address_Ptr);
     WinPrintf("Session \"%s\" processing interrupt.\n", Address_Ptr);
}

int
```

```
sample_ionintr(void)
{
     int       error_number;
     intrhandler_t  old_handler;
     INST      id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query the session's interrupt handler. */

     error_number = igetonintr(id, &old_handler);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetonintr().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Define the session's interrupt handler. */

     error_number = ionintr(id, (intrhandler_t) InterruptHandler);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  ionintr().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Enable the I_INTR_VXI_SIGNAL interrupt for the session. */

     error_number = isetintr(id, I_INTR_VXI_SIGNAL, ENABLE_INTERRUPT);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  isetintr().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Disable handler execution. */

     iintroff();

     /* Generate an I_INTR_VXI_SIGNAL interrupt. */

     HandlerExecuted = 0;
     GenerateInterrupt(id);

     /* Verify no interrupt handler execution. */
```

```
    if (HandlerExecuted != 0)
    {
         WinPrintf("FAILURE:  interrupt handler executed!\n");
         iclose(id);
         return (error_number);
    }

    /* Wait for and verify interrupt handler execution. */

    error_number = iwaithdlr(1000);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  iwaithdlr().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         iclose(id);
         return (error_number);
    }
    if (HandlerExecuted == 0)
    {
         WinPrintf("FAILURE:  interrupt handler did not execute!\n");
         iclose(id);
         return (error_number);
    }

    /* Generate an I_INTR_VXI_SIGNAL interrupt. */

    HandlerExecuted = 0;
    GenerateInterrupt(id);

    /* Verify no interrupt handler execution. */

    if (HandlerExecuted != 0)
    {
         WinPrintf("FAILURE:  interrupt handler executed!\n");
         iclose(id);
         return (error_number);
    }

    /* Enable handler execution and verify that the interrupt     */
    /* handler executed.                                          */

    iintron();
    if (HandlerExecuted == 0)
    {
         WinPrintf("FAILURE:  interrupt handler did not execute!\n");
    }
    iclose(id);
    return (error_number);
}
```

ionsrq

Description Installs a session's service request (SRQ) handler.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ionsrq(INST id, srqhandler_t srqhandler);
```

id	Session handle.
srqhandler	Pointer to an SRQ handler function.

Visual Basic Synopsis

None

Remarks If *id* specifies a device session, this function installs the function specified by *srqhandler* as the function to call when the corresponding device generates a service request. If *id* specifies an interface session, the function installs the function specified by *srqhandler* as the function to call when any device on the corresponding interface generates a service request.

The SICL library assumes that SRQ handler functions have the following calling semantics:

```
void SICLCALLBACK
srqhandler(INST id);
```

where *id* identifies the device requesting service.

SRQ reception is always enabled. This function does not enable or disable SRQ processing. Use **iintroff** to disable SRQ processing and **iintron** to enable SRQ processing. By default, SRQ processing is enabled.

Note the difference between SRQ reception and SRQ processing. Refer to Chapter 3, *Advanced Topics*, for more information.

If a process has two or more sessions that refer to the same device and a SRQ request occurs, the SRQ handlers for each of the different sessions are called.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igetonsrq, iintroff, iintron, ireadstb**

Example

```
/*
 * ionsrq.c:    this example sets an SRQ handler, then generates and processes
 *         an SRQ (this example assumes it's executing on an EPC-7).
 */

#include "olrm.h"
#include "sicl.h"

#define REQUEST_TRUE 0xFD00
#define SIG_REG_OFFSET    0x0008

volatile int HandlerExecuted;

void
WinPrintf(char _far *Format_String, ...);

int
GenerateSRQ(INST Id)
{
     volatile char _far * mapped_ptr;
     int              error_number;
     unsigned long        olrm_data;

     /* Generate a SRQ by writing a REQUEST TRUE event from the    */
     /* servant device to the EPC's signal register.               */

     (void) OlrmGetNumAttr("vxisink", VXI_ULA, &olrm_data);
     mapped_ptr = imap(Id, I_MAP_VXIDEV, 0, 0, NULL);
     if (mapped_ptr == NULL)
     {
          error_number = igeterrno();
          WinPrintf( "FAILURE:  imap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }
     iwpoke(    (volatile unsigned short _far *) (mapped_ptr + SIG_REG_OFFSET),
          (unsigned short) (olrm_data | REQUEST_TRUE));
     iunmap(Id, (char _far *) mapped_ptr, 0, 0, 0);
}

void SICLCALLBACK
SRQHandler(INST Id)
{
     char *Address_Ptr;

     HandlerExecuted = 1;
     igetaddr(Id, &Address_Ptr);
     WinPrintf("Session \"%s\" processing SRQ.\n", Address_Ptr);
```

```
}

int
sample_ionsrq(void)
{
     int       error_number;
     srqhandler_t    old_handler;
     INST      id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Query the session's SRQ handler. */

     error_number = igetonsrq(id, &old_handler);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  igetonsrq().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Define the session's SRQ handler. */

     error_number = ionsrq(id, (srqhandler_t) SRQHandler);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  ionsrq().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }

     /* Disable handler execution. */

     iintroff();

     /* Generate an SRQ. */

     HandlerExecuted = 0;
     GenerateSRQ(id);

     /* Verify no SRQ handler execution. */

     if (HandlerExecuted != 0)
     {
          WinPrintf("FAILURE:  SRQ handler executed!\n");
          iclose(id);
          return (error_number);
     }

     /* Wait for and verify SRQ handler execution. */
```

```
    error_number = iwaithdlr(1000);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  iwaithdlr().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         iclose(id);
         return (error_number);
    }
    if (HandlerExecuted == 0)
    {
         WinPrintf("FAILURE:  SRQ handler did not execute!\n");
         iclose(id);
         return (error_number);
    }

    /* Generate an SRQ. */

    HandlerExecuted = 0;
    GenerateSRQ(id);

    /* Verify no SRQ handler execution. */

    if (HandlerExecuted != 0)
    {
         WinPrintf("FAILURE:  SRQ handler executed!\n");
         iclose(id);
         return (error_number);
    }

    /* Enable handler execution and verify that the SRQ */
    /* handler executed.                                */

    iintron();
    if (HandlerExecuted == 0)
    {
         WinPrintf("FAILURE:  SRQ handler did not execute!\n");
    }
    iclose(id);
    return (error_number);
}
```

iopen

Description Opens a session.

C Synopsis

```
#include "sicl.h"

INST SICLAPI
iopen(char _far *addr);
```

addr Address string

Visual Basic Synopsis

Declare Function **iopen** Lib "sicl16.dll" Alias "vbopen" (ByVal *addr* As String) As Integer

Remarks This function opens a session for communicating with the device or interface specified by the address string *addr*. *Addr* cannot be null.

An address string for interfaces has this form:

logical-unit | *symbolic-name*

where *logical-unit* is an integer greater than zero and less than 32767 and *symbolic-name* is any sequence of letters, digits, underscores, periods, and dashes that begins with a letter. The following are valid interface addresses:

7 An interface at *logical-unit* 7

vxi A *symbolic-name* for the VXIbus interface

An address string for devices has this form:

(i/f-address"," primary-address ["," *secondary-address*])| *symbolic-name*

where *i/f-address* is *logical-unit* | *symbolic-name* (the same as the address string for interfaces), *primary-address* is interface specific (normally a positive integer, but can be a string or sequence of bytes), *secondary-address* is also interface specific, and *symbolic-name* is any sequence of letters, digits, underscores, periods, and dashes that begins with a letter.

The following are valid device addresses:

7,23	*I/f-address* is *logical-unit* 7 and *primary-address* of the device is 23.
vxi,128	*I/f-address* is *symbolic-name* "vxi" and *primary-address* is *ula* 128.
meter	The device has *symbolic-name* "meter."

An address string for commanders has the following form:

i/f-address ",cmdr"

where *I/f-address* is *logical-unit* | *symbolic-name* (the same as the address string for interfaces).

The following are valid commander addresses:

7,cmdr	*I/f-address* is logical-unit 7.
vxi,cmdr	*I/f-address* is symbolic-name "vxi"

Logical units, symbolic interface names, and the corresponding device driver names are defined in the **SICL.INI** file. By default, the **SICL.INI** file defines the following interfaces:

```
[Aliases]
GPIB=hp341i
gpib=hp341i
HPIB=hp341i
hpib=hp341i
VXI=radvxi
vxi=radvxi
MXI=radvxi
mxi=radvxi
[PARAMS]
hp341i=LU,Name,Interface,Slot,BusAddr,Switches,SysCtl,IRQ
radvxi=LU,Name,Interface
[INTF0]
LU=7
name=gpib
Interface=gpib
Slot=0
BusAddr=0
Switches=0b1100
SysCtl=1
IRQ=5

[INTFL]
Lu=16
Name=vxi
Interface=vxi
```

Symbolic device names are defined in the **DEVICES** file. If no configured name matches the device, a device is assigned a symbolic name by the SURM. The SURM-assigned names may change if the system configuration is changed.

If an interface and a device have the same name, the session opens as an interface session because interface names are searched first.

Address strings that begin with ASCII digits "0" through "9" are considered logical units.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iclose**

Example

```
/*
 * iopen.c:  use iopen() to open some sessions.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_iopen(void)
{
     int  error_number;
     INST id;

     /* Open a VXI interface session by name. */

     id = iopen("vxi");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }
     iclose(id);

     /* Open a VXI device session by address. */

     id = iopen("vxi,1");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }
     iclose(id);

     /* Open a VXI device session by name. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }
     iclose(id);

     /* Open a GPIB interface session by name. */

     id = iopen("gpib");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
```

```
        return (error_number);
    }
    iclose(id);

    /* Open a GPIB device session by address. */

    id = iopen("gpib,1");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        return (error_number);
    }
    iclose(id);

    /* Open a GPIB device session by name. */

    id = iopen("gpibsink");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
        return (error_number);
    }
    iclose(id);
    return (I_ERR_NOERROR);
}
```

iprintf

Description Formats and writes data to a device or interface.

C Synopsis

```
# include "sicl.h"

int SICLAPIV
iprintf(INST id, const char _far *format [, argument...]);
```

id	Session handle.
format	Pointer to a format control string.
argument	Optional arguments to format string.

Visual Basic Synopsis

None

Remarks This function writes characters and values to the device or interface of the session specified by *id*. *Format* is a string of ordinary characters, escape character sequences, and format specifications that control how to format and convert each *argument*. Refer to Chapter 4, *I/O Formatting*, for additional information.

Format specifications always begin with the percent sign (%) and are processed left to right. The first format specification causes the first *argument* value to be converted and written. The second format specification causes conversion and writing of the second *argument*, and so forth. To avoid unpredictable results, there must be an *argument* for each format specification. If there are more *arguments* than format specifications, the excess *arguments* are ignored.

Formatted data may be written to a formatted I/O write buffer, or directly to a device. Refer to **isetbuf** and **isetubuf** for additional information.

To avoid unpredictable results, do not mix buffered output function calls (**ifwrite, iprintf, ipromptf, ivprintf, ivpromptf**) and unbuffered output function calls (**iwrite**) within the same session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iflush, ifwrite, ipromptf, iscanf, isetbuf, isetubuf, isprintf, isvprintf, ivprintf, iwrite**

Example

```
/*
 * iprintf.c:  this program uses iprintf() to send data to a device.
 */

#include "sicl.h"

char _far *    BeginString     = "BEGIN";
char       EndCharacter    = ';';
int        BlockData[4]    = { 1, 2, 3, 4 };
int        Integer         = 1;
double          DoublePrecision = 3825.1e+15;

void
WinPrintf(char _far *Format_String, ...);

int
CheckIPrintfError(int Conversion_Count)
{
     int error_number;

     if (Conversion_Count == 1)
     {
          return (I_ERR_NOERROR);
     }
     error_number = igeterrno();
     WinPrintf("FAILURE:  iprintf().  Unexpected number of conversions.\n");
     WinPrintf("          Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     return (error_number);
}

int
sample_iprintf(void)
{
     int  error_number;
     INST id;

#if !defined(I_SICL_FMTIO)
     WinPrintf("Formatted I/O is not supported.\n");
     return (I_ERR_NOERROR);
#endif

     /* Open a device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
```

```
    }

    /* Send data to the device. */

    error_number = CheckIPrintfError(iprintf(id, "%s\n", BeginString));
    if (error_number != I_ERR_NOERROR)
    {
         iclose(id);
         return (error_number);
    }
    error_number = CheckIPrintfError(iprintf(id, "%@Hd\n", Integer));
    if (error_number != I_ERR_NOERROR)
    {
         iclose(id);
         return (error_number);
    }
    error_number = CheckIPrintfError(iprintf(id, "%e\n", DoublePrecision));
    if (error_number != I_ERR_NOERROR)
    {
         iclose(id);
         return (error_number);
    }
    error_number = CheckIPrintfError(iprintf(id, "%@Bg\n", DoublePrecision));
    if (error_number != I_ERR_NOERROR)
    {
         iclose(id);
         return (error_number);
    }
    error_number = CheckIPrintfError(iprintf(id, "%4B\n", BlockData));
    if (error_number != I_ERR_NOERROR)
    {
         iclose(id);
         return (error_number);
    }
    error_number = CheckIPrintfError(iprintf(id, "%C", EndCharacter));
    iclose(id);
    return (error_number);
}
```

ipromptf

Description Writes formatted data to and reads formatted data from a device or interface.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
ipromptf(INST id, const char _far *writeformat, const char _far
    *readformat [,argument]...);
```

id	Session handle.
writeformat	Pointer to write format.
readformat	Pointer to read format.
argument	Optional arguments and/or pointer(s) to location(s) where the function stores the formatted data.

Visual Basic Synopsis

None

Remarks This function performs both an **iprintf** function and an **iscanf** function in a single call. First data is formatted and written to the device, then it is read.

W*riteformat* points to a format specification string that writes data to the device or interface of the session specified by *id*. It uses the number of *argument*s necessary to satisfy the format specification. The write format specification is identical to the **iprintf** format specification. Refer to Chapter 4, *I/O Formatting,* for additional information.

Readformat points to a read data format specification string that reads data from the device or interface of the session specified by *id*. *Readformat* uses the remaining arguments to satisfy the read format specification. The read format specification is identical to the **iscanf** format specification. Refer to Chapter 4, *I/O Formatting*, for additional information.

When **ipromptf** is executed, the read buffer is discarded, **iprintf** is executed, the write buffer is sent to the device, and finally **iscanf** is executed.

Interrupts that occur while a read is being executed are not processed until the read completes.

To avoid unpredictable results, do not mix buffered I/O function calls (**ifread, ifwrite, iprintf, ipromptf, iscanf, ivprintf, ivpromptf ivscanf**) and unbuffered I/O function calls (**iread, iwrite**) within the same session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iprintf, iscanf, ivpromptf**

Example

```
/*
 * iprompt.c:  this example calls ipromptf() to program and read an instrument.
 */

#include "sicl.h"

#define BUFFER_SIZE  64

void
WinPrintf(char _far *Format_String, ...);

int
sample_ipromptf(void)
{
     char data_buffer[BUFFER_SIZE];
     int  conversion_count;
     int  error_number;
     INST id;

#if !defined(I_SICL_FMTIO)
     WinPrintf("Formatted I/O is not supported.\n");
     return (I_ERR_NOERROR);
#endif

     /* Open a device session. */
```

If *reason* is not null, the function stores a bit mask describing why the read terminated in the referenced memory location. The following constants define valid bits in the mask specified by *reason*:

Constant	Description
I_TERM_CHR	Termination character received (see **itermchr**)
I_TERM_END	END indicator received
I_TERM_MAXCNT	*Bufsize* bytes read

If *actualcnt* is not null, the function stores the number of bytes read in the referenced memory location.

For VXI device sessions, the function generates BYTE REQUEST word serial commands to read data. The function only supports message-based VXI devices; other VXI devices cause an error.

For VXI interface sessions, the function generates an **I_ERROR_NOTSUPP** error.

For GPIB device sessions, the function first causes all devices to unlisten. Then, it issues the interface's listen address, followed by the device's talk address. Finally, the function reads the data bytes.

For GPIB interface sessions, the function reads data from a GPIB interface without performing any addressing.

To avoid unpredictable results, do not mix buffered input function calls (**ifread, ipromptf, iscanf, ivpromptf, ivscanf**) and unbuffered input function calls (**iread**) within the same session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ifread, igettermchr, ipromptf, iscanf, itermchr, itimeout, ivpromptf, ivscanf, iwrite**

iread

Description Reads data from a device or interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iread(INST id, char _far *buf, unsigned long bufsize, int _far
    *reason, unsigned long _far *actualcnt);
```

id	Session handle.
buf	Pointer to the data buffer.
bufsize	Number of data bytes to read.
reason	Pointer to the location where the functions stores the cause of read termination bit mask.
actualcnt	Pointer to a location where the function stores the actual number of bytes read from the device or interface.

Visual Basic Synopsis

```
Declare Sub iread Lib "sicl16.dll" (ByVal id As Integer, buf As
Any, ByVal bufsize As Long, reason As Any, actual As Long)
```

Remarks This function reads *bufsize* bytes from the device or interface of the session specified by *id* and stores them into the buffer beginning at *buf*. It performs no buffering, formatting or data conversion.

Reading ends when *bufsize* bytes are read, an END indicator is received, a termination character is received, or a timeout occurs. This function blocks until one of these four conditions is met.

```
    id = iopen("vxisink");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        return (error_number);
    }

    /* Write a command and read the reply. */

    conversion_count = ipromptf(id, "IDN?", "%s", data_buffer);
    if (conversion_count == 1)
    {
        error_number = I_ERR_NOERROR;
        WinPrintf("Data read from \"vxisink\" = \"%s\"\n",
                data_buffer);
    }
    else
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  ipromptf().  Unexpected  number  of
conversions.\n");
        WinPrintf("          Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
    }
    iclose(id);
    return (error_number);
}
```

Example

```
/*
 * iread.c:     this example calls iread() to read an instrument's response
 *         without waiting.
 */

#include "sicl.h"

#define BUFFER_SIZE  64

void
WinPrintf(char _far *Format_String, ...);

int
sample_iread(void)
{
     char       buffer[BUFFER_SIZE] = { 0 };
     int        error_number;
     int        reason;
     unsigned long   read_count;
     INST       id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Write a command to the device. */

     iprintf(id, "IDN?");

     /* Read and print the device's response. */

     error_number = iread(     id,
                    buffer,
                    BUFFER_SIZE,
                    &reason,
                    &read_count);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  iread().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          iclose(id);
          return (error_number);
     }
     buffer[read_count] = '\0';
     WinPrintf("Response data read from \"vxisink\" = %s.\n", buffer);
     WinPrintf("Read termination reason(s):\n");
     if ((reason & I_TERM_CHR) != 0)
     {
          WinPrintf("\tI_TERM_CHR.\n");
     }
     if ((reason & I_TERM_END) != 0)
     {
          WinPrintf("\tI_TERM_END.\n");
     }
```

```
    if ((reason & I_TERM_MAXCNT) != 0)
    {
        WinPrintf("\tI_TERM_MAXCNT.\n");
    }
    iclose(id);
    return (error_number);
}
```

ireadstb

Description Reads the status byte from a device.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ireadstb(INST id, unsigned char _far *statusbyte);
```

id	Device session handle.
statusbyte	Pointer to a location where the function stores the device's status byte.

Visual Basic Synopsis

Declare Sub **ireadstb** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *stb* As String)

Remarks This function reads the device status byte of the session specified by *id* and is valid only for device sessions.

For VXI device sessions, the function issues a READ STB word serial command. The function only supports message-based VXI devices; other VXI devices cause an error.

For GPIB device sessions, the function issues a GPIB serial poll (SPOLL) command.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **isetstb**, **itimeout**

Example

```
/*
 * ireadstb.c:  this example uses ireadstb() to read a device's status byte.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ireadstb(void)
{
     int        error_number;
     unsigned char   status_byte;
     INST       id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Read the device's status byte. */

     error_number = ireadstb(id, &status_byte);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  ireadstb().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     else
     {
          WinPrintf("Status byte = 0x%02X", status_byte);
     }
     iclose(id);
     return (error_number);
}
```

iremote

Description Puts a device in remote mode.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iremote(INST id);
```

id Session handle.

Visual Basic Synopsis

```
Declare Sub iremote Lib "sicl16.dll" (ByVal id As Integer)
```

Remarks This function places the device of the session specified by *id* into remote mode and is valid only for device sessions.

For VXI device sessions, the function issues a SET LOCK word serial command. The function only supports message-based VXI devices; other VXI devices cause an error.

For GPIB device sessions, the function addresses the device to listen.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ilocal**

Example

```
/*
 * iremote.c:  this example uses iremote() to issue a Set Lock word serial
 *         command.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_iremote(void)
{
      int  error_number;
      INST id;

      /* Open a VXI device session. */

      id = iopen("vxisink");
      if (id == ((INST) 0))
      {
            error_number = igeterrno();
            WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
            return (error_number);
      }

      /* Generate a Set Lock word serial command to the device. */

      error_number = iremote(id);
      if (error_number != I_ERR_NOERROR)
      {
            WinPrintf("FAILURE:  iremote().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
      }
      iclose(id);
      return (error_number);
}
```

iscanf

Description Reads and formats data from a device or interface.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
iscanf(INST id, const char _far *format [, void _far
    *argument...]);
```

id	Session handle.
format	Pointer to a format control string.
argument	Pointer(s) to location(s) where the function stores the formatted data.

Visual Basic Synopsis

None

Remarks This function reads a series of characters and values from the device or interface session specified by *id*. The characters and values are read into the locations specified by *argument*. *Format* is a string of ordinary characters and format specifications that control how to format and convert characters from the specified device or interface. Refer to Chapter 4, *I/O Formatting*, for additional information.

Format specifications always begin with the percent sign (%) and are read left to right. Characters outside the format specification are expected to match the sequence of characters from the device or interface. The matching characters from the device or interface are scanned but not stored. If a scanned character does not match the format specification, **iscanf** terminates.

The first format specification causes the first input field from the device or interface to be converted and written to the location specified by the first *argument*. The second format specification causes conversion of the second input field from the device or interface to be converted and written to the location specified by the second *argument*, and so forth. There must be enough format specifications and arguments for the input field being read for the results to be predictable. Excess format specifications and arguments are ignored.

Formatted data may be read from both the formatted I/O read buffer and directly from a device. Refer to **isetbuf** and **isetubuf** for additional information.

To avoid unpredictable results, do not mix buffered input function calls (**ifread, ipromptf, iscanf, ivpromptf, ivscanf**) and unbuffered input function calls (**iread**) within the same session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iflush, ifread, ipromptf, iread, isscanf, isvscanf, isetbuf, isetubuf, ivscanf**

Example

```
/*
 * iscanf.c:    this program illustrates input formatting with iscanf().  The
 *         program prints to a device that simply echoes all input. The
 *         printed value should be identical to the scanned value.
 */

#include <string.h>
#include "sicl.h"

char _far *     PrintString     = "Test String";
char       ScanString[16];
double          PrintDouble     = 3825.1e+7;
double          ScanDouble;

void
WinPrintf(char _far *Format_String, ...);

int
CheckIPrintfError(int Conversion_Count)
{
     int error_number;

     if (Conversion_Count == 1)
```

```
    {
        return (I_ERR_NOERROR);
    }
    error_number = igeterrno();
    WinPrintf("FAILURE:  iprintf().  Unexpected number of conversions.\n");
    WinPrintf("          Error = %s (%d).\n",
            igeterrstr(error_number),
            error_number);
    return (error_number);
}

int
CheckIScanfError(int Conversion_Count)
{
    int error_number;

    if (Conversion_Count == 1)
    {
        return (I_ERR_NOERROR);
    }
    error_number = igeterrno();
    WinPrintf("FAILURE:  iscanf().  Unexpected number of conversions.\n");
    WinPrintf("          Error = %s (%d).\n",
            igeterrstr(error_number),
            error_number);
    return (error_number);
}

int
sample_iscanf(void)
{
    int  error_number;
    INST id;

#if !defined(I_SICL_FMTIO)
    WinPrintf("Formatted I/O is not supported.\n");
    return (I_ERR_NOERROR);
#endif

    /* Open a device session. */

    id = iopen("vxisink");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        return (error_number);
    }
    itimeout(id, 500);

    /* Test string formatting. */

    error_number = CheckIPrintfError(iprintf(id, "%s\n", PrintString));
    if (error_number != I_ERR_NOERROR)
    {
        iclose(id);
        return (error_number);
    }
    error_number = CheckIScanfError(iscanf(id, "%s\n", &ScanString));
    if (error_number != I_ERR_NOERROR)
    {
        iclose(id);
        return (error_number);
```

```
    }
    WinPrintf("Printed string = \"%s\", Scanned string = \"%s\".\n",
              PrintString,
              ScanString);
    if (strcmp(PrintString, ScanString) != 0)
    {
         WinPrintf("FAILURE:  string data mismatch.\n");
         iclose(id);
         return (error_number);
    }
    iflush(id,I_BUF_READ);

    /* Test floating point formatting. */

    error_number = CheckIPrintfError(iprintf(id, "%e\n", PrintDouble));
    if (error_number != I_ERR_NOERROR)
    {
         iclose(id);
         return (error_number);
    }
    error_number = CheckIScanfError(iscanf(id,"%e", &ScanDouble));
    if (error_number != I_ERR_NOERROR)
    {
         iclose(id);
         return (error_number);
    }
    WinPrintf("Printed value = %e, Scanned value = %e.\n",
              PrintDouble,
              ScanDouble);
    if (PrintDouble != ScanDouble)
    {
         WinPrintf("FAILURE:  floating point data mismatch.\n");
    }
    iclose(id);
    return (error_number);
}
```

isetbuf

Description Sets the size of formatted I/O read and/or write buffers.

C Synopsis

```
#include "sicl.h"

int SICLAPI
isetbuf(INST id, int buffermask, int buffersize);
```

id	Session handle.
buffermask	Buffer selection mask.
buffersize	Buffer size, in bytes.

Visual Basic Synopsis

```
Declare Sub isetbuf Lib "sicl16.dll" (ByVal id As Integer, ByVal
mask As Integer, ByVal size As Integer)
```

Remarks This function flushes the current read buffer and/or write buffer and sets the read buffer and/or write buffer size for the device or interface session specified by *id*.

Buffermask is an OR'd combination of the following buffer selection constants:

Constant	Description
I_BUF_READ	Discard the contents of the session's current read buffer. If data is discarded and the last byte does not contain an END indicator, read from the device or interface until and END indicator is read and set a new read buffer size.

Discarding the contents of the current read buffer ensures that the next call to buffered input functions reads data directly from the device rather than reading data that was previously buffered.

I_BUF_WRITE Write the contents of the session's current write buffer to the device or interface and set a new write buffer size.

Specifying a *buffersize* equal to zero disables buffering and all reads and writes take place directly to the device or interface.

Specifying a *buffersize* greater than zero creates a new buffer of the specified size. The write buffer is written to the device or interface anytime a buffer fills or when the END indicator is placed in the buffer. The read buffer retains data until explicitly flushed using **iflush**.

Specifying a *buffersize* less than zero creates a buffer of the absolute value of the specified size. The write buffer is written to the device or interface anytime the buffer fills, when the END indicator is placed in the buffer, or at the end of each formatted output function (**ifwrite, iprintf, ipromptf, ivprintf, ivpromptf**). The read buffer flushes data at the end of every formatted input function (**ifread, ipromptf, iscanf, ivpromptf,** and **ivscanf**).

Default read and write buffers sizes are **I_READ BUF_SZ** and **I_WRITE_BUF_SZ**, respectively. Closing and reopening a session flushes the buffers and resets their length to the defaults.

If the function fails and the returned value is **I_ERR_NORSRC**, the buffer size for the buffers specified by *buffermask* are set to zero.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iflush, iprintf, ipromptf, iscanf, isetubuf, ivprintf, ivpromptf, ivscanf**

Example

```
/*
 * isetbuf.c:  this program used isetbuf() to illustrate the effect of the
 *         write buffer size on iprintf().
 */

#include "sicl.h"

#define SIZE_ARRAY_LGTH   3

char _far *     BeginString     = "BEGIN";
char        EndCharacter    = ';';
int         BlockData[4]    = { 1, 2, 3, 4 };
int         BufferSize[]    = { -100, 0, 100 };
int         Integer         = 1;
double          DoublePrecision = 3825.1e+15;

void
WinPrintf(char _far *Format_String, ...);

int
CheckIPrintfError(int Conversion_Count)
{
     int error_number;

     if (Conversion_Count == 1)
     {
          return (I_ERR_NOERROR);
     }
     error_number = igeterrno();
     WinPrintf("FAILURE:  iprintf().  Unexpected number of conversions.\n");
     WinPrintf("          Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     return (error_number);
}

int
sample_isetbuf(void)
{
     int  error_number;
     int  index;
     INST id;

#if !defined(I_SICL_FMTIO)
     WinPrintf("Formatted I/O is not supported.\n");
     return (I_ERR_NOERROR);
#endif

     /* Open a device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Send data to the device in using various write buffer sizes. */

     for (index = 0; index < SIZE_ARRAY_LGTH; index += 1)
```

```
    {
        error_number = isetbuf(   id,
                        I_BUF_WRITE,
                        BufferSize[index]);
        if (error_number != I_ERR_NOERROR)
        {
            WinPrintf("FAILURE:  isetbuf().  Error = %s (%d).\n",
                        igeterrstr(error_number),
                        error_number);
            break;
        }
        error_number = CheckIPrintfError(iprintf(id, "%s\n", BeginString));
        if (error_number != I_ERR_NOERROR)
        {
            break;
        }
        error_number = CheckIPrintfError(iprintf(id, "%@Hd\n", Integer));
        if (error_number != I_ERR_NOERROR)
        {
            break;
        }
        error_number = CheckIPrintfError(iprintf(id, "%e\n", DoublePrecision));
        if (error_number != I_ERR_NOERROR)
        {
            break;
        }
        error_number       =       CheckIPrintfError(iprintf(id,       "%@Bg\n",
DoublePrecision));
        if (error_number != I_ERR_NOERROR)
        {
            break;
        }
        error_number = CheckIPrintfError(iprintf(id, "%4B\n", BlockData));
        if (error_number != I_ERR_NOERROR)
        {
            break;
        }
        error_number = CheckIPrintfError(iprintf(id,"%C", EndCharacter));
        if (error_number != I_ERR_NOERROR)
        {
            break;
        }

        /* For write buffer sizes > 0, the buffer is only   */
        /* flushed when the buffer is full or the END indicator  */
        /* is placed into the buffer.  The buffer is being  */
        /* implicitly flushed by placing "\n" into the buffer.   */

        if (BufferSize[index] > 0)
        {
            iprintf(id, "\n");
        }
    }
    iclose(id);
    return (error_number);
}
```

isetdata

Description Stores a pointer to the session data structure.

C Synopsis

```
#include "sicl.h"

int SICLAPI
isetdata(INST id, void _far *data);
```

id	Session handle.
data	Application-specific data.

Visual Basic Synopsis

None

Remarks This function defines an application-specific data structure associated with the session specified by *id*. The data structure can be queried with the **igetdata** function.

The session data structure is a 4-byte memory block. Its contents are application-specific. Typically, it contains a pointer to an application data structure.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igetdata**

Example See **igetdata**.

isetintr

Description Enables and disables interrupt reception.

C Synopsis

```
#include "sicl.h"

int SICLAPI
isetintr(INST id, int intrtype, long intrenable);
```

id	Session handle.
intrtype	Interrupt type.
intrenable	Interrupt enable flag.

Visual Basic Synopsis

None

Remarks This function enables or disables interrupt reception for the interrupt type specified by *intrtype* for the session specified by *id*.

The following are valid constants for *intrtype*:

Constant	Description
I_INTR_DEVCLR	Interrupt when a commander sends a device clear to this device (GPIB commander session only).
I_INTR_GPIB_GET	Interrupt when a commander sends a GET command to the device (GPIB commander session only).
I_INTR_GPIB_IFC	Interrupt on GPIB interface clear (GPIB interface sessions only).
I_INTR_GPIB_PPOLLCONFIG	Interrupt when a commander changes the device's PPOLL configuration (GPIB commander session only).

I_INTR_GPIB_REMLOC

Interrupt when a commander places the device in remote or local mode (GPIB commander session only).

I_INTR_GPIB_TLAC Interrupt when a commander addresses the device to talk, untalk, listen, or unlisten (GPIB commander session only).

I_INTR_INTFACT Interrupt when an interface becomes active (GPIB interface sessions only).

I_INTR_INTFDEACT Interrupt when an interface deactivates (GPIB interface sessions only).

I_INTR_OFF Disable all interrupts.

I_INTR_STB Interrupt when a commander reads this device's status byte (GPIB commander session only).

I_INTR_TRIG Interrupt on a trigger (GPIB interface sessions; also, VXI interface sessions on an EPC-7).

I_INTR_VXI_SIGNAL Interrupt on a VXI signal or a VME interrupt from a servant VXI device (VXI device sessions only).

I_INTR_VXI_VME Interrupt on a VME interrupt from a non-servant device (VXI interface sessions only).

I_INTR_VXI_UNKSIG Interrupt on a VXI signal from a non-servant device (VXI interface sessions only).

When *intrenable* is zero, the function disables the interrupts specified by *intrtype*; a value other than zero enables the selected interrupt. When *intrtype* is **I_INTR_OFF**, *intrenable* is ignored.

When *intrtype* is **I_INTR_TRIG** and *id* specifies a VXI interface session, *intrenable* becomes a bit mask that specifies one or more trigger interrupts. Setting *intrenable* to zero disables the trigger interrupt.

On an EPC-7, the following are valid constants for *intrenable* when *intrtype* is **I_INTR_TRIG:**

Constant	Description
I_TRIG_STD	Standard trigger.
I_TRIG_TTL0	TTL trigger 0.
I_TRIG_TTL1	TTL trigger 1.
I_TRIG_TTL2	TTL trigger 2.
I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.
I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

On a VXLink interface, the following are valid constants for *intrenable* when *intrtype* is **I_INTR_TRIG:**

Constant	Description
I_TRIG_STD	Standard trigger.
I_TRIG_TTL0	External Input Trigger.
I_TRIG_TTL1	External Output Trigger.
I_TRIG_TTL2	TTL trigger 2.
I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.
I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

The VXI trigger(s) corresponding to the **I_TRIG_STD** constant can be modified using **ivxitrigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Proper VXI trigger interrupt operation on an EPC-7 requires software intervention. Refer to Chapter 3, *Advanced Topics*, for additional information.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igetonintr, iintron, iintroff, ionintr, ivxitrigroute**

Example See **igetonintr.**

isetlockwait

Description Determines whether accessing a locked device or interface suspends the calling thread or generates an error.

C Synopsis

```
#include "sicl.h"

int SICLAPI
isetlockwait(INST id, int waitflag);
```

id	Session handle.
waitflag	Lock wait flag.

Visual Basic Synopsis

Declare Sub **isetlockwait** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *flag* As Integer)

Remarks The function sets the state of the lock-wait flag of the session specified by *id* to the value specified by *waitflag*.

When a session's lock-wait flag is non-zero and a locking conflict occurs, the session waits for its previously specified timeout period for the lock to be released. If the lock-wait flag is zero and a locking conflict occurs, **I_ERR_LOCKED** is returned.

By default, a session waits for conflicting locks to be released (its lock-wait flag is non-zero).

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igetlockwait**, **ilock**, **iunlock**

isetstb

Description Sets this controller's status byte.

C Synopsis

#include "sicl.h"

int SICLAPI
isetstb(INST *id*, **unsigned char** *statusbyte);*

id	GPIB commander session handle.
statusbyte	Status byte.

Visual Basic Synopsis

Declare Sub **isetstb** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *stb* As Integer)

Remarks The function sets the status byte of the device specified by *id* to *statusbyte*.

The VXIbus interface driver supports SICL standard level 2F (support for device and interface sessions only). Therefore, this function always returns an error for a VXI session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ireadstb**

isetubuf

Description Sets the formatted I/O read or write buffers to a user-specified buffer.

C Synopsis

```
#include "sicl.h"

int SICLAPI
isetubuf(INST id, int buffermask, int buffersize, char _far *buf);
```

id	Session handle.
buffermask	Buffer selection mask.
buffersize	Buffer size, in bytes.
buf	Pointer to a data buffer.

Visual Basic Synopsis

None

Remarks This function flushes either the read buffer or the write buffer of the device or interface session specified by *id*, then sets the buffer to *buf*.

Buffermask may be either of following buffer selection constants:

Constant	Description
I_BUF_READ	Discard the contents of the session's current read buffer. If data is discarded and the last byte does not contain an END indicator, read from the device or interface until and END indicator is read. Set the new read buffer. Cannot be used in conjunction with **I_BUF_WRITE**.

Discarding the contents of the current read buffer ensures that the next buffered input function call reads data directly from the device rather than reading data that was previously buffered.

I_BUF_WRITE Write the contents of the session's current write buffer to the device and set the new write buffer. Cannot be used in conjunction with **I_BUF_READ**.

Specifying a *buffersize* equal to zero disables buffering and all reads and writes take place directly to the device or interface.

Specifying a *buffersize* greater than zero installs the new buffer with the specified size. The write buffer is written to the device or interface anytime a buffer fills or when the END indicator is placed in the buffer. The read buffer retains data until explicitly flushed using **iflush**.

Specifying a *buffersize* less than zero installs the new buffer with the absolute value of the specified size. The write buffer is written to the device or interface anytime the buffer fills, when an END indicator is placed in the buffer, or at the end of each formatted output function (**iwrite, iprintf, ipromptf, ivprintf, ivpromptf**). The read buffer flushes data at the end of every formatted input function (**ifread, ipromptf, iscanf, ivpromptf, ivscanf**).

Default read and write buffers sizes are **I_READ BUF_SZ** and **I_WRITE_BUF_SZ**, respectively. Closing and reopening a session flushes the buffers and resets their length to the defaults.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iflush, ipromptf, iprintf, iscanf**

Example

```
/*
 * isetubuf.c:  this program used isetubuf() to illustrate the effect of the
 *         write buffer size on iprintf().
 */
```

2

```
#include <malloc.h>
#include <windows.h>
#include "sicl.h"

#define BUFFER_SIZE  64
#define SIZE_ARRAY_LGTH   3

char _far *     BeginString          = "BEGIN";
char       EndCharacter         = ';';
int        BlockData[4]         = { 1, 2, 3, 4 };
int        BufferSize[]         = { -BUFFER_SIZE, 0, BUFFER_SIZE };
int        Integer              = 1;
double          DoublePrecision      = 3825.1e+15;

void
WinPrintf(char _far *Format_String, ...);

int
CheckIPrintfError(int Conversion_Count)
{
     int error_number;

     if (Conversion_Count == 1)
     {
          return (I_ERR_NOERROR);
     }
     error_number = igeterrno();
     WinPrintf("FAILURE:  iprintf().  Unexpected number of conversions.\n");
     WinPrintf("          Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     return (error_number);
}

int
sample_isetbuf(void)
{
     char _far *     buffer;
     int        error_number;
     int        index;
     INST       id;

#if !defined(I_SICL_FMTIO)
     WinPrintf("Formatted I/O is not supported.\n");
     return (I_ERR_NOERROR);
#endif

     /* Open a device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Send data to the device in using various write buffer sizes. */

     error_number = I_ERR_NOERROR;
     for (index = 0; index < SIZE_ARRAY_LGTH; index += 1)
     {
          /* Allocate a buffer */
```

```
          if ((buffer = (char _far *) malloc(BUFFER_SIZE)) == NULL)
          {
               WinPrintf("FAILURE:  buffer allocation.\n");
               break;
          }
          error_number = isetubuf(  id,
                              I_BUF_WRITE,
                              BufferSize[index],
                              buffer);
          if (error_number != I_ERR_NOERROR)
          {
               WinPrintf("FAILURE:  isetubuf().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
               free(buffer);
               break;
          }
          error_number = CheckIPrintfError(iprintf(id, "%s\n", BeginString));
          if (error_number != I_ERR_NOERROR)
          {
               free(buffer);
               break;
          }
          error_number = CheckIPrintfError(iprintf(id, "%@Hd\n", Integer));
          if (error_number != I_ERR_NOERROR)
          {
               free(buffer);
               break;
          }
          error_number = CheckIPrintfError(iprintf(id, "%e\n", DoublePrecision));
          if (error_number != I_ERR_NOERROR)
          {
               free(buffer);
               break;
          }
          error_number       =       CheckIPrintfError(iprintf(id,       "%@Bg\n",
DoublePrecision));
          if (error_number != I_ERR_NOERROR)
          {
               free(buffer);
               break;
          }
          error_number = CheckIPrintfError(iprintf(id, "%4B\n", BlockData));
          if (error_number != I_ERR_NOERROR)
          {
               free(buffer);
               break;
          }
          error_number = CheckIPrintfError(iprintf(id, "%C", EndCharacter));
          if (error_number != I_ERR_NOERROR)
          {
               free(buffer);
               break;
          }

          /* For write buffer sizes > 0, the buffer is only   */
          /* flushed when the buffer is full or the END indicator  */
          /* is placed into the buffer.  The buffer is being  */
          /* implicitly flushed by placing "\n" into the buffer.   */

          if (BufferSize[index] > 0)
          {
              iprintf(id, "\n");
          }
```

```
        free(buffer);
    }
    iclose(id);
    return (error_number);
}
```

isprintf

Description Formats and writes data to a buffer.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
isprintf(char _far * buf, const char _far *format [, argument...]);
```

buf	Pointer to a data buffer.
format	Pointer to a format control string.
argument	Optional arguments to format string.

Visual Basic Synopsis

None

Remarks This function writes characters and values to the buffer specified by *buf*. *Format* is a string of ordinary characters, escape character sequences, and format specifications that control how to format and convert each *argument*. Refer to Chapter 4, *I/O Formatting*, for additional information.

Format specifications always begin with the percent sign (%) and are processed left to right. The first format specification causes the first *argument* value to be converted and written. The second format specification causes conversion and writing of the second *argument*, and so forth. To avoid unpredictable results, there must be an *argument* for each format specification. If there are more *arguments* than format specifications, the excess *arguments* are ignored.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iprintf, isvprintf, isscanf**

Example See **iprintf**

isscanf

Description Reads and formats data from a buffer.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
isscanf(char _far *buf, const char _far *format [, _far
    *argument...]);
```

id	Session handle.
format	Pointer to a data buffer.
argument	Pointer(s) to location(s) where the function stores the formatted data.

Visual Basic Synopsis

None

Remarks This function reads a series of characters and values from the buffer specified by *buf*. The characters and values are read into the locations specified by *argument*. *Format* is a string of ordinary characters and format specifications that control how to format and convert characters from the specified device or interface. Refer to Chapter 4, *I/O Formatting*, for more information.

Format specifications always begin with the percent sign (%) and are read left to right. Characters outside the format specification are expected to match the sequence of characters from *buf*. The matching characters from *buf* are scanned but not stored. If a scanned character does not match the format specification, **isscanf** terminates.

The first format specification causes the first input field from *buf* to be converted and written to the location specified by the first *argument*. The second format specification causes conversion of the second input field from *buf* to be converted and written to the location specified by the second *argument*, and so forth. There must be enough format specifications and arguments for the input field being read for the results to be predictable. Excess format specifications and arguments are ignored.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iscanf, isvscanf, isprintf**

Example See **iscanf**

isvprintf

Description Formats and writes data to a buffer using a standard **va_list** parameter.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
isvprintf(char _far * buf, const char _far *format, va_list
    argument);
```

buf	Pointer to a data buffer.
format	Pointer to a format control string.
argument	Arguments to format string.

Visual Basic Synopsis

Declare Function **isvprintf** Lib "sicl16.dll" Alias "vbsvprintf" (ByVal *user_buf* As String, ByVal *fmt* As String, *ap* As Any) As Integer

Remarks This function writes characters and values to the buffer specified by *buf*. *Format* is a string of ordinary characters, escape character sequences, and format specifications that control how to format and convert each *argument*.

The **va_list** type is an ANSI standard mechanism for passing a variable number of arguments. It allows the prediction of the number of function parameters.

Format specifications always begin with the percent sign (%) and are processed left to right. The first format specification causes the first *argument* value to be converted and written. The second format specification causes conversion and writing of the second *argument*, and so forth. To avoid unpredictable results, there must be an *argument* for each format specification. If there are more *arguments* than format specifications, the excess *arguments* are ignored.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iprintf, isprintf, isvscanf**

Example See **ivprintf**

isvscanf

Description Reads and formats data from a buffer using a standard **va_list** parameter.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
isvscanf(char _far * buf, const char _far *format, va_list
    argument);
```

buf	Pointer to a data buffer.
format	Pointer to a format control string.
argument	Location(s) where the function stores the formatted data.

Visual Basic Synopsis

Declare Function **isvscanf** Lib "sicl16.dll" Alias "vbsvscanf" (ByVal *user_buf* As String, ByVal *fmt* As String, *ap* As Any) As Integer

Remarks This function reads a series of characters and values from the buffer specified by *buf*. The characters and values are read into the locations specified by *argument*. *Format* is a string of ordinary characters and format specifications that control how to format and convert characters from the specified device or interface. Refer to Chapter 4, *I/O Formatting*, for additional information.

The **va_list** type is an ANSI standard mechanism for passing a variable number of arguments. It allows the prediction of the number of function parameters.

Format specifications always begin with the percent sign (%) and are read left to right. Characters outside the format specification are expected to match the sequence of characters from *buf*. The matching characters from *buf* are scanned but not stored. If a scanned character does not match the format specification, **isvscanf** terminates.

The first format specification causes the first input field from *buf* to be converted and written to the location specified by the first *argument*. The second format specification causes conversion of the second input field from *buf* to be converted and written to the location specified by the second *argument*, and so forth. There must be enough format specifications and arguments for the input field being read for the results to be predictable. Excess format specifications and arguments are ignored.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iscanf, isscanf**

Example See **ivscanf**

iswap

Description Byte-swaps a buffer of data.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iswap(char _far *buf, unsigned long length, int datasize);
```

buf Pointer to a data buffer.

length Length of the buffer, in bytes.

datasize Size of data elements in the buffer, in bytes.

Visual Basic Synopsis

Declare Sub **iswap** Lib "sicl16.dll" (*addr* As Any, ByVal *length* As Long, ByVal *datasize* As Integer)

Remarks This function byte-swaps a buffer of equal-sized data elements. *Length* specifies the overall size of the buffer and *datasize* specifies the size of individual data elements in the buffer.

Length must be a multiple of *datasize*.

Datasize may be 1, 2, 4, or 8 bytes.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibeswap, ileswap**

Example

```
/*
 * iswap.c:     use ibeswap()/ileswap()/iswap() to swap data.  Note that
 *         ileswap() is a NOP on an EPC.
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sicl.h>

unsigned long DataBuffer[] = { 0x00112233, 0xCCDDEEFF };

void
main(void)
{
     int error_number;

     /* Print original data. */

     fprintf(  stdout,
               "Original 32-bit data        = 0x%08X, 0x%08X\n",
               DataBuffer[0],
               DataBuffer[1]);

     /* Execute ileswap() and print data. */

     error_number = ileswap(   (char *) DataBuffer,
                    sizeof(DataBuffer),
                    sizeof(unsigned long));
     if (error_number != I_ERR_NOERROR)
     {
          fprintf(  stderr,
                    "FAILURE:  ileswap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          exit(-1);
     }
     fprintf(  stdout,
               "32-bit data after ileswap() = 0x%08X, 0x%08X\n",
               DataBuffer[0],
               DataBuffer[1]);

     /* Execute ibeswap() and print data. */

     error_number = ibeswap(   (char *) DataBuffer,
                    sizeof(DataBuffer),
                    sizeof(unsigned long));
     if (error_number != I_ERR_NOERROR)
     {
          fprintf(  stderr,
                    "FAILURE:  ibeswap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          exit(-2);
     }
     fprintf(  stdout,
               "32-bit data after ibeswap() = 0x%08X, 0x%08X\n",
               DataBuffer[0],
               DataBuffer[1]);

     /* Execute iswap() and print data. */

     error_number = iswap(     (char *) DataBuffer,
                    sizeof(DataBuffer),
                    sizeof(unsigned long));
     if (error_number != I_ERR_NOERROR)
     {
          fprintf(  stderr,
                    "FAILURE:  iswap().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
```

```
        exit(-3);
    }
    fprintf(   stdout,
               "32-bit data after iswap()    = 0x%08X, 0x%08X\n",
               DataBuffer[0],
               DataBuffer[1]);
    exit(0);
}
```

itermchr

Description Specifies a session's termination character.

C Synopsis

```
#include "sicl.h"

int SICLAPI
itermchr(INST id, int termchar);
```

id	Session handle.
termchar	Termination character.

Visual Basic Synopsis

Declare Sub **itermchr** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *tchr* As Integer)

Remarks This function specifies the termination character for the session specified by *id*. The functions **ifread, ipromptf**, **iread, iscanf, isscanf, isvscanf**, **ivpromptf,** and **ivscanf** use the termination character to signal the end of a read operation.

Use the **igettermchr** function to get the current termination character.

Valid *termchr* values are −1 and 0 through 255, inclusive. The value −1 (default) indicates that no termination character is set. A value of 0 through 255 is a termination character.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igettermchr, ifread, ipromptf, iread, iscanf, isscanf, isvscanf, ivpromptf, ivscanf**

Example See **igettermchr**.

itimeout

Description Set a session's timeout value.

C Synopsis

```
#include "sicl.h"

int SICLAPI
itimeout(INST id, long timeout);
```

id	Session handle.
timeout	Timeout interval, in milliseconds.

Visual Basic Synopsis

Declare Sub **itimeout** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *tval* As Long)

Remarks This function specifies the timeout value for the session specified by *id*. A timeout value is the time interval to wait for an operation to complete before aborting. When an operation aborts because of a timeout, the aborted function returns an error indicating that the call timed out. Time-outs affect these SICL functions:

iclear	**igpibsendcmd**	**isetubuf**
iflush	**igpibsett1delay**	**itrigger**
ifread	**ilocal**	**ivprintf**
ifwrite	**ilock**	**ivpromptf**
igpibatnctl	**imap**	**ivscanf**
igpibgett1delay	**iprintf**	**ivxitrigoff**
igpibllo	**ipromptf**	**ivxitrigon**
igpibpassctl	**iread**	**ivxitrigroute**
igpibppoll	**ireadstb**	**ivxiwaitnormop**
igpibppollconfig	**iremote**	**ivxiws**
igpibppollresp	**iscanf**	**iwaithdlr**
igpibrenctl	**isetbuf**	**iwrite**
	isetstb	**ixtrig**

The *timeout* value is in milliseconds. A *timeout* value of less than or equal to zero indicates an infinite timeout. The default *timeout* value is 0.

Use **igettimeout** to get a session's current *timeout* value.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **igettimeout**

Example See **igettimeout**.

itrigger

Description Sends a trigger command to a device or interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
itrigger(INST id);
```

id Session handle.

Visual Basic Synopsis

```
Declare Sub itrigger Lib "sicl16.dll" (ByVal id As Integer)
```

Remarks This function sends a trigger command to the device or interface of the session specified by *id*. When *id* specifies a device session, the trigger is sent to the device of the session and is dependent on the interface (VXI or GPIB), but the trigger is an addressed trigger. When *id* specifies an interface session, the trigger is interface specific.

For VXI device sessions, the function issues a TRIGGER word serial command. The function only supports message-based VXI devices; other VXI devices cause an error.

For VXI interface sessions, the function asserts and deasserts the trigger defined by **I_TRIG_STD**.

For GPIB device sessions, the function issues an addressed Group Execute Trigger (GET) command.

For GPIB interface sessions, the function issues a Group Execute Trigger (GET) command without performing any addressing. The user should use **igpibsendcmd** to set up those listeners to receive the trigger.

The VXIbus triggers corresponding to the **I_TRIG_STD** constant can be modified using **ivxitrigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **itimeout, ixtrig**

Example

```
/*
 * itrigger.c: this example uses itrigger() to send a trigger to a device.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_itrigger(void)
{
     int  error_number;
     INST id;

     /* Open a VXI device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Send a trigger to the device. */

     error_number = itrigger(id);
     if (error_number != I_ERR_NOERROR)
     {
          WinPrintf("FAILURE:  itrigger().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     iclose(id);
     return (error_number);
}
```

iunlock

Description Unlocks a device or interface session.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iunlock(INST id);
```

id Session handle.

Visual Basic Synopsis

```
Declare Sub iunlock Lib "sicl16.dll" (ByVal id As Integer)
```

Remarks This function unlocks the session specified by *id*.

Closing a session implicitly unlocks the session.

Attempting to unlock a device or interface session that is not locked generates an error.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ilock**

Example See **ilock**.

iunmap

Description Deletes an address space mapping.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iunmap(INST id, char _far *mapaddress, int mapspace, unsigned
    int pagestart, unsigned int pagecnt);
```

id	Session handle.
mapaddress	Mapped address pointer.
mapspace	Mapping address space.
pagestart	Starting page number.
pagecnt	Number of mapped pages.

Visual Basic Synopsis

Declare Sub **iunmap** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *addr* As Long, ByVal *mapspace* As Integer, ByVal *pagestart* As Integer, ByVal *pagecnt* As Integer)

Remarks *Mapaddress* is a pointer returned by a previous **imap** call.

Mapaddress completely describes the mapping to SICL. The *mapspace*, *pagestart*, and *pagecnt* parameters are ignored.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **imap**, **imapinfo**, **iopen**

Example See **imap**

iversion

Description Returns the SICL library version data.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iversion(int _far * SICLversion, int _far * implversion);
```

SICLversion	Pointer to a location where the function stores the supported SICL specification version number.
implversion	Pointer to a location where the function stores the SICL DLL implementation version number.

Visual Basic Synopsis

Declare Sub **iversion** Lib "sicl16.dll" (*specversion* As Integer, *implversion* As Integer)

Remarks This function returns both the version of the SICL specification supported by the SICL DLL and the version of the SICL DLL implementation.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

ivprintf

Description Formats and writes data to a device or interface using a standard **va_list** parameter.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
ivprintf(INST id, const char _far *format,va_list argument);
```

id	Session handle.
format	Pointer to a format control string.
argument	Optional arguments.

Visual Basic Synopsis

Declare Function **ivprintf** Lib "sicl16.dll" Alias "vbvprintf" (ByVal *id* As Integer, ByVal *fmt* As String, *ap* As Any) As Integer

Remarks This function writes characters and values to the device or interface of the session specified by *id*. *Format* is a string of ordinary characters, escape character sequences, and format specifications that control how to format and convert each *argument*. Refer to Chapter 4, *I/O Formatting*, for additional information.

The **va_list** type is an ANSI standard mechanism for passing a variable number of arguments. It allows the prediction of the number of function parameters.

Format specifications always begin with the percent sign (%) and are processed left to right. The first format specification causes the first *argument* value to be converted and written. The second format specification causes conversion and writing of the second *argument*, and so forth. To avoid unpredictable results, use an *argument* for each format specification. If there are more *arguments* than format specifications, excess *arguments* are ignored.

To avoid unpredictable results, do not mix buffered output function calls (**ifwrite, iprintf, ipromptf, ivprintf, ivpromptf**) and unbuffered output function calls (**iwrite**) within the same session.

Formatted data may be written to a formatted I/O write buffer, or directly to a device. Refer to **isetbuf** and/or **isetubuf** for additional information.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iflush, ifwrite, iprintf, ipromptf, isetbuf, isetubuf, isprintf, isvprintf, ivpromptf, ivscanf, iwrite**

Example

```
/*
 * ivprintf.c: this program uses ivprintf() to send data to a device.
 */

#include "sicl.h"

char _far *     BeginString     = "BEGIN";
char       EndCharacter    = ';';
int        BlockData[4]    = { 1, 2, 3, 4 };
int        Integer         = 1;
double          DoublePrecision = 3825.1e+15;

void
WinPrintf(char _far *Format_String, ...);

int
CheckIVPrintfError(int Conversion_Count)
{
     int error_number;

     if (Conversion_Count == 1)
     {
          return (I_ERR_NOERROR);
     }
     error_number = igeterrno();
     WinPrintf("FAILURE:  ivprintf().  Unexpected number of conversions.\n");
     WinPrintf("          Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     return (error_number);
}

int
IVPrintfWrapper(INST Id, char *Format_Ptr, ... )
{
     int  conversion_count;
     va_list    arguments;

     va_start(arguments, Format_Ptr);
     conversion_count = ivprintf(Id, Format_Ptr, arguments);
     va_end(arguments);
```

```
     return (conversion_count);
}

int
sample_ivprintf(void)
{
     int  error_number;
     INST id;

#if !defined(I_SICL_FMTIO)
     WinPrintf("Formatted I/O is not supported.\n");
     return (I_ERR_NOERROR);
#endif

     /* Open a device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Send data to the device. */

     error_number = CheckIVPrintfError(IVPrintfWrapper(id, "%s\n", BeginString));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     error_number = CheckIVPrintfError(IVPrintfWrapper(id, "%@Hd\n", Integer));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     error_number      =      CheckIVPrintfError(IVPrintfWrapper(id,      "%e\n",
DoublePrecision));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     error_number      =      CheckIVPrintfError(IVPrintfWrapper(id,      "%@Bg\n",
DoublePrecision));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     error_number = CheckIVPrintfError(IVPrintfWrapper(id, "%4B\n", BlockData));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     error_number = CheckIVPrintfError(IVPrintfWrapper(id, "%C", EndCharacter));
     iclose(id);
     return (error_number);
}
```

ivpromptf

Description Writes formatted data to and reads formatted data from a device or interface using a standard **va_list** parameter.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
ivpromptf(INST id, const char _far *writeformat, const char
    _far *readformat,va_list argument);
```

id	Session handle.
writeformat	Pointer to write format.
readformat	Pointer to read format.
argument	Optional arguments.

Visual Basic Synopsis

None

Remarks This function performs both an **ivprintf** function and an **ivscanf** function in a single call. First data is written , then it is read.

W*riteformat* points to a format specification string that writes data to the device or interface of the session specified by *id*. It uses the number of *argument*s necessary to satisfy the format specification. The write format specification is identical to the **ivprintf** format specification. Refer to Chapter 4, *I/O Formatting*, for additional information.

Readformat points to a format specification string that reads data from the device or interface of the session specified by *id*. *Readformat* uses the remaining arguments to satisfy the read format specification. The read format specification is identical to the **ivscanf** format specification. Refer to Chapter 4, *I/O Formatting*, for additional information.

The **va_list** is an ANSI standard mechanism for passing a variable number of arguments. It allows the prediction of the number of function parameters.

When **ivpromptf** is executed, the read buffer is discarded, **ivprintf** is executed, the write buffer is sent to the device, and **ivscanf** is executed.

Interrupts that occur while a read is being executed are not processed until the read completes.

To avoid unpredictable results, do not mix buffered I/O function calls (**ifread, ifwrite, iprintf, ipromptf, iscanf, ivprintf, ivpromptf, ivscanf**) and unbuffered I/O function calls (**iread, iwrite**) within the same session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ipromptf, ivprintf, ivscanf**

Example

```
/*
 * ivprompt.c: this example calls ivpromptf() to program and read an
 *         instrument.
 */

#include "sicl.h"

#define BUFFER_SIZE  64

void
WinPrintf(char _far *Format_String, ...);

int
IVPromptfWrapper(INST Id, char *Write_Format_Ptr, char *Read_Format_Ptr, ...)
{
      int   conversion_count;
      va_list    arguments;

      va_start(arguments, Read_Format_Ptr);
      conversion_count = ivpromptf(  Id,
                           Write_Format_Ptr,
                           Read_Format_Ptr,
                           arguments);
      va_end(arguments);
      return (conversion_count);
}

int
sample_ivpromptf(void)
```

```
{
     char data_buffer[BUFFER_SIZE];
     int  conversion_count;
     int  error_number;
     INST id;

#if !defined(I_SICL_FMTIO)
     WinPrintf("Formatted I/O is not supported.\n");
     return (I_ERR_NOERROR);
#endif

     /* Open a device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }

     /* Write a command and read the reply. */

     conversion_count = IVPromptfWrapper(id, "IDN?", "%s", data_buffer);
     if (conversion_count == 1)
     {
          error_number = I_ERR_NOERROR;
          WinPrintf("Data read from \"vxisink\" = \"%s\"\n",
                    data_buffer);
     }
     else
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:       ivpromptf().       Unexpected    number    of
conversions.\n");
          WinPrintf("          Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
     }
     iclose(id);
     return (error_number);
}
```

ivscanf

Description Reads and formats data from a device or interface using a standard **va_list** parameter.

C Synopsis

```
#include "sicl.h"

int SICLAPIV
ivscanf(INST id, const char _far *format, va_list argument);
```

id	Session handle.
format	Pointer to a format control string.
argument	Optional arguments.

Visual Basic Synopsis

Declare Function **ivscanf** Lib "sicl16.dll" Alias "vbvscanf" (ByVal *id* As Integer, ByVal *fmt* As String, *ap* As Any) As Integer

Remarks This function reads a series of characters and values from the device or interface session specified by *id*. The characters and values are read into the locations specified by *argument*. *Format* is a string of ordinary characters and format specifications that control how to format and convert characters from the specified device or interface. Refer to Chapter 4, *I/O Formatting*, for additional information.

The **va_list** is an ANSI standard mechanism for passing a variable number of arguments. It allows the prediction of the number of function parameters.

Format specifications always begin with the percent sign (%) and are read left to right. Characters outside the format specification are expected to match the sequence of characters from the device or interface. The matching characters from the device or interface are scanned but not stored. If a scanned character does not match the format specification, **ivscanf** terminates.

The first format specification causes the first input field from the device or interface to be converted and written to the location specified by the first *argument*. The second format specification causes conversion of the second input field from the device or interface to be converted and written to the location specified by the second *argument*, and so forth. There must be enough format specifications and arguments for the input field being read for the results to be predictable. Excess format specifications and arguments are ignored.

Formatted data may be read from a formatted I/O read buffer rather than directly from a device. Refer to **isetbuf** and **isetubuf** for additional information.

To avoid unpredictable results, do not mix buffered input function calls (**ifread, ipromptf, iscanf, ivpromptf, ivscanf**) and unbuffered input function calls (**iread**) within the same session.

Return Value

The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also

iflush, iread, iscanf, isetbuf, isetubuf, ivprintf, ivpromptf

Example

```
/*
 * ivscanf.c:  this program illustrates input formatting with ivscanf().  The
 *         program prints to a device that simply echoes all input. The
 *         printed value should be identical to the scanned value.
 */

#include "sicl.h"

char _far *     PrintString     = "Test String";
char        ScanString[16];
double          PrintDouble     = 3825.1e+7;
double          ScanDouble;

void
WinPrintf(char _far *Format_String, ...);
```

```
int
CheckIVPrintfError(int Conversion_Count)
{
     int error_number;

     if (Conversion_Count == 1)
     {
         return (I_ERR_NOERROR);
     }
     error_number = igeterrno();
     WinPrintf("FAILURE:  iprintf().  Unexpected number of conversions.\n");
     WinPrintf("          Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     return (error_number);
}

int
CheckIVScanfError(int Conversion_Count)
{
     int error_number;

     if (Conversion_Count == 1)
     {
         return (I_ERR_NOERROR);
     }
     error_number = igeterrno();
     WinPrintf("FAILURE:  ivscanf().  Unexpected number of conversions.\n");
     WinPrintf("          Error = %s (%d).\n",
               igeterrstr(error_number),
               error_number);
     return (error_number);
}

int
IVPrintfWrapper(INST Id, char *Format_Ptr, ... )
{
     int  conversion_count;
     va_list   arguments;

     va_start(arguments, Format_Ptr);
     conversion_count = ivprintf(Id, Format_Ptr, arguments);
     va_end(arguments);
     return (conversion_count);
}

int
IVScanfWrapper(INST Id, char *Format_Ptr, ...)
{
     int  conversion_count;
     va_list   arguments;

     va_start(arguments, Format_Ptr);
     conversion_count = ivscanf(Id, Format_Ptr, arguments);
     va_end(arguments);
     return (conversion_count);
}

int
sample_ivscanf(void)
{
     int  error_number;
     INST id;
```

```
#if !defined(I_SICL_FMTIO)
     WinPrintf("Formatted I/O is not supported.\n");
     return (I_ERR_NOERROR);
#endif

     /* Open a device session. */

     id = iopen("vxisink");
     if (id == ((INST) 0))
     {
          error_number = igeterrno();
          WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
          return (error_number);
     }
     itimeout(id, 500);

     /* Test string formatting. */

     error_number = CheckIVPrintfError(IVPrintfWrapper(id, "%s\n", PrintString));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     error_number = CheckIVScanfError(IVScanfWrapper(id, "%s\n", &ScanString));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     WinPrintf("Printed string = \"%s\", Scanned string = \"%s\".\n",
               PrintString,
               ScanString);
     if (strcmp(PrintString, ScanString) != 0)
     {
          WinPrintf("FAILURE:  string data mismatch.\n");
          iclose(id);
          return (error_number);
     }
     iflush(id,I_BUF_READ);

     /* Test floating point formatting. */

     error_number = CheckIVPrintfError(IVPrintfWrapper(id, "%e\n", PrintDouble));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     error_number = CheckIVScanfError(IVScanfWrapper(id,"%e", &ScanDouble));
     if (error_number != I_ERR_NOERROR)
     {
          iclose(id);
          return (error_number);
     }
     WinPrintf("Printed value = %e, Scanned value = %e.\n",
               PrintDouble,
               ScanDouble);
     if (PrintDouble != ScanDouble)
     {
          WinPrintf("FAILURE:  floating point data mismatch.\n");
     }
     iclose(id);
```

```
    return (error_number);
}
```

ivxibusstatus

Description Gets VXIbus status.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ivxibusstatus(INST id, int request, unsigned long _far *result);
```

id	VXI interface session handle.
request	Status request.
result	Pointer to a location where the functions stores the requested status information.

Visual Basic Synopsis

Declare Sub **ivxibusstatus** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *request* As Integer, *result* As Long)

Remarks This function places the VXIbus interface status information specified by *request* in the location specified by *result*. It is valid only for VXI interface sessions.

The following are valid constants for *request*:

Constant	Description
I_VXI_BUS_CMDR_LADDR	Return the logical address of the commander of this EPC (0xFFFFFFFF = no commander exists, either because this EPC is a top-level commander or normal operation has not been established).
I_VXI_BUS_LADDR	Return the logical address of this EPC.

I_VXI_BUS_MAN_ID	Return the manufacturer's ID of this EPC.
I_VXI_BUS_MODEL_ID	Return the model ID of this EPC.
I_VXI_BUS_NORMOP	Return normal operation status of this EPC (1 = normal, 0 = other).
I_VXI_BUS_PROTOCOL	Return the protocol register value of this EPC.
I_VXI_BUS_SERVANT_AREA	Return the servant area size of this EPC.
I_VXI_BUS_SHM_ADDR_SPACE	Return this EPC's VXI memory space. Returns 24 for A24 space or 32 for A32 space.
I_VXI_BUS_SHM_PAGE	Return this EPC's VXI memory location, in pages. For A24 memory, page size is 256 bytes. For A32 memory, page size is 64K bytes.
I_VXI_BUS_SHM_SIZE	Returns this EPC's VXI memory size in pages. For A24 memory, page size is 256 bytes. For A32 memory, page size is 64K bytes.
I_VXI_BUS_TRIGGER	Return a bit mask of the currently asserted trigger lines (see **ivxitrigroute**).

I_VXI_BUS_TRIGSUPP	Return a bit mask of the triggers supported by this EPC. See **ivxigettrigroute.**
I_VXI_BUS_VXIMXI	Returns 1 if this device is an MXI controller. The EPC always returns 0.
I_VXI_BUS_XPORT	Return the READ PROTOCOL word serial command response value of this EPC.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, ivxitrigroute**

Example

```
/*
 * vxistat.c:  this example calls ivxibusstatus() to display VXIbus status
 *         information.
 */

#include "sicl.h"

#define DIM(x)  (sizeof(x)/sizeof(int))

int Requests[] =
{
      I_VXI_BUS_TRIGGER,
      I_VXI_BUS_LADDR,
      I_VXI_BUS_SERVANT_AREA,
      I_VXI_BUS_NORMOP,
      I_VXI_BUS_CMDR_LADDR,
      I_VXI_BUS_MAN_ID,
      I_VXI_BUS_MODEL_ID,
      I_VXI_BUS_PROTOCOL,
      I_VXI_BUS_SHM_SIZE,
      I_VXI_BUS_SHM_ADDR_SPACE,
      I_VXI_BUS_SHM_PAGE,
      I_VXI_BUS_VXIMXI,
      I_VXI_BUS_TRIGSUPP
};

char _far *Strings[] =
{
      "I_VXI_BUS_TRIGGER       ",
      "I_VXI_BUS_LADDR         ",
      "I_VXI_BUS_SERVANT_AREA  ",
      "I_VXI_BUS_NORMOP        ",
      "I_VXI_BUS_CMDR_LADDR    ",
      "I_VXI_BUS_MAN_ID        ",
```

```
    "I_VXI_BUS_MODEL_ID      ",
    "I_VXI_BUS_PROTOCOL      ",
    "I_VXI_BUS_SHM_SIZE      ",
    "I_VXI_BUS_SHM_ADDR_SPACE",
    "I_VXI_BUS_SHM_PAGE      ",
    "I_VXI_BUS_VXIMXI        ",
    "I_VXI_BUS_TRIGSUPP      "
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_ivxibusstatus(void)
{
    int        error_number;
    int        index;
    unsigned long  result;
    INST       id;

    /* Open a VXI interface session. */

    id = iopen("vxi");
    if (id == ((INST) 0))
    {
        error_number = igeterrno();
        WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
        return (error_number);
    }

    /* Request and print VXIbus status. */

    for (index = 0; index < DIM(Requests); index++)
    {
        error_number = ivxibusstatus(  id,
                            Requests[index],
                            &result);
        if (error_number != I_ERR_NOERROR)
        {
            WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
                    igeterrstr(error_number),
                    error_number);
            iclose(id);
            return (error_number);
        }
        WinPrintf("%s = 0x%08X.\n", Strings[index], result);
    }
    iclose(id);
    return (error_number);
}
```

ivxigettrigroute

Description Gets a current trigger routing for the VXI interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ivxigettrigroute(INST id, unsigned long intriggermask, unsigned
    long _far *outtriggermask);
```

id	VXI interface session handle.
intriggermask	Input triggermask.
outtriggermask	Pointer to a location where the function stores a trigger mask that describes the routing of the input trigger.

Visual Basic Synopsis

```
Declare Sub ivxigettrigroute Lib "sicl16.dll" (ByVal id As Integer,
ByVal which As Long, route As Long)
```

Remarks This function places a mask of the current trigger routing for triggers specified in *intriggermask* in the location specified by *outtriggermask*. The function is valid only for VXI interface sessions.

intriggermask contains a constant specifying a trigger whose routing should be queried. The following are valid constants for *intriggermask*:

intriggermask	Description
I_TRIG_ALL	All valid triggers.
I_TRIG_STD	Standard trigger.
I_TRIG_CLK0	Internal clock trigger 0.
I_TRIG_CLK1	Internal clock trigger 1.
I_TRIG_CLK2	Internal clock trigger 2.
I_TRIG_CLK10	10 MHz system clock.
I_TRIG_CLK100	100 MHz system clock.
I_TRIG_ECL0	ECL trigger 0.

I_TRIG_ECL1	ECL trigger 1.
I_TRIG_ECL2	ECL trigger 2.
I_TRIG_ECL3	ECL trigger 3.
I_TRIG_EXT0	External trigger 0.
I_TRIG_EXT1	External trigger 1.
I_TRIG_EXT2	External trigger 2.
I_TRIG_EXT3	External trigger 3.
I_TRIG_TTL0	TTL trigger 0.
I_TRIG_TTL1	TTL trigger 1.
I_TRIG_TTL2	TTL trigger 2.
I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.
I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

The value placed in the location specified by the *outtriggermask* pointer contains a bit mask of zero or more trigger bits corresponding to *intriggermask*'s routed output triggers.

Use **ivxitrigroute** to route triggers. Specifying an *intriggermask* of **I_TRIG_ALL** returns a mask of all valid triggers for this EPC.

Specifying an *intriggermask* of **I_TRIG_STD** returns a mask of triggers corresponding to the **I_TRIG_STD** constant.

Return Value

The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also

ivxitrigoff, ivxitrigon, ivxitrigroute, ixtrig

Example

```
/*
 * trigrout.c:  this example uses ivxitrigroute()/ivxigettrigroute() to
 *         define/query a trigger routing.
 */

#include "sicl.h"

unsigned long TriggerMasks[] =
{
     I_TRIG_TTL0,
     I_TRIG_TTL1,
     I_TRIG_TTL2,
     I_TRIG_TTL3,
```

```
        I_TRIG_TTL4,
        I_TRIG_TTL5,
        I_TRIG_TTL6,
        I_TRIG_TTL7,
        I_TRIG_ECL0,
        I_TRIG_ECL1,
        I_TRIG_ECL2,
        I_TRIG_ECL3,
        I_TRIG_EXT0,
        I_TRIG_EXT1,
        I_TRIG_EXT2,
        I_TRIG_EXT3,
        I_TRIG_CLK0,
        I_TRIG_CLK1,
        I_TRIG_CLK2,
        I_TRIG_CLK10,
        I_TRIG_CLK100
};

char *TriggerStrings[] =
{
        "I_TRIG_TTL0",
        "I_TRIG_TTL1",
        "I_TRIG_TTL2",
        "I_TRIG_TTL3",
        "I_TRIG_TTL4",
        "I_TRIG_TTL5",
        "I_TRIG_TTL6",
        "I_TRIG_TTL7",
        "I_TRIG_ECL0",
        "I_TRIG_ECL1",
        "I_TRIG_ECL2",
        "I_TRIG_ECL3",
        "I_TRIG_EXT0",
        "I_TRIG_EXT1",
        "I_TRIG_EXT2",
        "I_TRIG_EXT3",
        "I_TRIG_CLK0",
        "I_TRIG_CLK1",
        "I_TRIG_CLK2",
        "I_TRIG_CLK10",
        "I_TRIG_CLK100"
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_ivxitrigroute(void)
{
        int             error_number;
        int             index;
        unsigned long   trigger_mask;
        INST            id;

        /* Open a VXI interface session. */

        id = iopen("vxi");
        if (id == ((INST) 0))
        {
                error_number = igeterrno();
                WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                          igeterrstr(error_number),
                          error_number);
                return (error_number);
```

```
    }

    /* Query and print a list of valid triggers. */

    error_number = ivxigettrigroute(id, I_TRIG_ALL, &trigger_mask);
    if (error_number != I_ERR_NOERROR)
    {
         error_number = igeterrno();
         WinPrintf("FAILURE:  ivxigettrigroute().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         iclose(id);
         return (error_number);
    }
    WinPrintf("Valid triggers:\n");
    for (index = 0;
         index < (sizeof(TriggerMasks) / sizeof(unsigned long));
         index += 1)
    {
         if ((trigger_mask & TriggerMasks[index]) != 0)
         {
              WinPrintf("%s\n", TriggerStrings[index]);
         }
    }

    /* Route trigger_mask so that TTL trigger 1 will be asserted  */
    /* whenever external trigger 0 is asserted.                   */

    error_number = ivxitrigroute(id, I_TRIG_EXT0, I_TRIG_TTL1);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  ivxitrigroute().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         iclose(id);
         return (error_number);
    }

    /* Query and print the trigger routing for external trigger 0. */

    error_number = ivxigettrigroute(id, I_TRIG_EXT0, &trigger_mask);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  ivxigettrigroute().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         iclose(id);
         return (error_number);
    }
    WinPrintf("Triggers mapped to I_TRIG_EXT0:\n");
    for (index = 0;
         index < (sizeof(TriggerMasks) / sizeof(unsigned long));
         index += 1)
    {
         if ((trigger_mask & TriggerMasks[index]) != 0)
         {
              WinPrintf("%s\n", TriggerStrings[index]);
         }
    }
    iclose(id);
    return (error_number);
}
```

ivxirminfo

2

Description Gets VXI device information.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ivxirminfo(INST id, int ula, struct vxiinfo _far *information);
```

id	VXI session handle.
ula	Device unique logical address.
information	Pointer to a location where the function stores the device's VXI configuration information.

Visual Basic Synopsis

```
Declare Sub ivxirminfo Lib "sicl16.dll" Alias "vbvxirminfo"
(ByVal id As Integer, ByVal laddr As Integer, info As vxiinfo)
```

Remarks This function places the VXI configuration information of the device at unique logical address *ula* in the location specified by *information.*

The function ignores *id* when *ula* specifies a valid device on a VXI interface.

For VXI device sessions only, specifying a *ula* of −1 causes the function to return the configuration of the device session specified by *id.*

VXI configuration information is returned in the format of a **vxiinfo** structure. The **vxiinfo** structure is defined in **SICL.H** as:

```
struct vxiinfo
{
/* Device identification. */

short laddr;            /* Unique logical address. */
char name[16];          /* Symbolic name (primary) */
char manuf_name[16];          /* Manufacturer name.    */
char model_name[16];          /* Model name.           */
unsigned short man_id;        /* Manufacturer ID.      */
unsigned short model;         /* Model number.    */
unsigned short devclass;      /* Device class.    */
```

```
/* Self-test status. */

short selftest;                 /* Self test status:      */
                                      /* 1 == PASSED      */
                                      /* 0 == FAILED      */

/* Location of device. */

short cage_num;                 /* Card cage number.*/
short slot;                     /* Slot number:     */
                                      /* -1 == UNKNOWN    */
                                      /* -2 == MXI        */

/* Device information. */

unsigned short protocol;                /* Value of protocol register.*/
unsigned short x_protocol;          /* Value of extended protocol register */
unsigned short servant_area;        /* Value of servant area. */

/* Memory information. */

unsigned short addrspace;           /* Memory address space:       */
                                    /* 0  == None       */
                                    /* 24 == A24        */
                                    /* 32 == A32        */
  unsigned short memsize;           /* Amount of memory, in pages
             /* (pages are 256 bytes in A24, 64K in       32).*/
  unsigned short memstart;          /* Start of memory, in pages  (pages are 256 bytes in A24,
64K in A32).*/

/* Miscellaneous information. */

short slot0_laddr;              /* ULA of slot 0 controller (-1 if unknown). */
short cmdr_laddr;       /* ULA of commander (-1 if top level). */

/* Interrupt information. */

short int_handler[8];   /* Array of interrupt handler flags.*/
short interrupter[8];   /* Array of interrupter flags.      */
short fill[10];         /* Unused space.    */
);
```

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * vxirm.c:     this example uses ivxirminfo() to retrieve resource management
 *         configuration information for VXI devices.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ivxirminfo(void)
{
     int        error_number;
     struct vxiinfo  vxi_info;
     INST       id;
```

```
    /* Open a VXI interface session. */

    id = iopen("vxi");
    if (id == ((INST) 0))
    {
         error_number = igeterrno();
         WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         return (error_number);
    }

    /* Query and print info on the device at ULA 0. */

    error_number = ivxirminfo(id, 0, &vxi_info);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  ivxirminfo().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         iclose(id);
         return (error_number);
    }
    WinPrintf("Symbolic name     = \"%s\".\n", vxi_info.name);
    WinPrintf("Manufacturer name = \"%s\".\n", vxi_info.manuf_name);
    iclose(id);

    /* Open a VXI device session. */

    id = iopen("vxisink");
    if (id == ((INST) 0))
    {
         error_number = igeterrno();
         WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
         return (error_number);
    }

    /* Query and print info on the device. */

    error_number = ivxirminfo(id, -1, &vxi_info);
    if (error_number != I_ERR_NOERROR)
    {
         WinPrintf("FAILURE:  ivxirminfo().  Error = %s (%d).\n",
                   igeterrstr(error_number),
                   error_number);
    }
    else
    {
         WinPrintf("Symbolic name     = \"%s\".\n", vxi_info.name);
         WinPrintf("Manufacturer name = \"%s\".\n", vxi_info.manuf_name);
    }
    iclose(id);
    return (error_number);
}
```

ivxiservants

Description Gets a list of VXI servants.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ivxiservants(INST id, int listsize, int _far *list);
```

id	VXI interface session handle.
listsize	Size of servant list, in entries.
list	Pointer to a location where the function stores a list of the ULAs of this device's servant devices.

Visual Basic Synopsis

Declare Sub **ivxiservants** Lib "sicl16.dll" Alias "vbvxiservants" (ByVal *id* As Integer, ByVal *maxnum* As Integer, *list*() As Integer)

Remarks This function places a list of the unique logical addresses (ULA) of the servants of the VXI interface corresponding to *id* in the memory location specified by *list*. Specifying an *id* for a GPIB session or VXI device session generates an error.

Listsize specifies the maximum number of entries in *list*.

If the VXI interface has less than *listsize* servant devices, all unused entries are set to −1. If the interface has more than *listsize* servant devices, only the first *listsize* ULAs are placed in *list*.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen**

Example

```
/*
 * vxiserve.c:  this example uses ivxiservants() to query the list of VXI
 *         servant devices.
 */

#include "sicl.h"

#define LIST_SIZE    256

void
WinPrintf(char _far *Format_String, ...);

int
sample_ivxiservants(void)
{
      int  error_number;
      int  index;
      int  ula_list[LIST_SIZE];
      INST id;

      /* Open a VXI interface session. */

      id = iopen("vxi");
      if (id == ((INST) 0))
      {
            error_number = igeterrno();
            WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
            return (error_number);
      }

      /* Query and print a list of servant devices for this interface. */

      error_number = ivxiservants(id, LIST_SIZE, ula_list);
      if (error_number != I_ERR_NOERROR)
      {
            WinPrintf("FAILURE:  ivxiservants().  Error = %s (%d).\n",
                  igeterrstr(error_number),
                  error_number);
            iclose(id);
            return (error_number);
      }
      WinPrintf("VXI servant list:\n");
      for (index = 0; index < LIST_SIZE; index++)
      {
            if (ula_list[index] == -1)
            {
                  break;
            }
            WinPrintf("\tULA %d (0x%02X)\n",
                  ula_list[index],
                  ula_list[index]);
      }
      iclose(id);
      return (error_number);
}
```

ivxitrigoff

Description Deasserts VXIbus trigger lines.

C Synopsis

#include "sicl.h"

int SICLAPI
ivxitrigoff(INST *id*, **unsigned long** *triggermask*);

id	VXI interface session handle.
triggermask	VXIbus trigger line(s) to deassert.

Visual Basic Synopsis

Declare Sub **ivxitrigoff** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *which* As Long)

Remarks This function deasserts the VXIbus trigger lines specified in *triggermask* for the VXI interface session specified by *id*. *Triggermask* is a bit mask that is an OR'd combination of one or more of the following:

Constant	Description
I_TRIG_ALL	All valid triggers. (EPC-7 and VXLink only)
I_TRIG_ECL0	ECL trigger 0. (EPC-7 only)
I_TRIG_ECL1	ECL trigger 1. (EPC-7 only)
I_TRIG_EXT0	EXT trigger 0 ((EPC-7 only)). Has no effect unless **I_TRIG_EXT0** has been routed as an output of another trigger; see **ivxitrigroute**).
I_TRIB_EXT1	EXT trigger 1 (EPC-7 only)). Has no effect unless **I_TRIG_EXT1** has been routed as an output of another trigger; see **ivxitrigroute**).
I_TRIG_STD	Standard trigger. (EPC-7 and VXLink only)
I_TRIG_TTL0	TTL trigger 0. (EPC-7 and VXLink only)

I_TRIG_TTL1	TTL trigger 1. (EPC-7 and VXLink only)
I_TRIG_TTL2	TTL trigger 2. (EPC-7 and VXLink only)
I_TRIG_TTL3	TTL trigger 3. (EPC-7 and VXLink only)
I_TRIG_TTL4	TTL trigger 4. (EPC-7 and VXLink only)
I_TRIG_TTL5	TTL trigger 5. (EPC-7 and VXLink only)
I_TRIG_TTL6	TTL trigger 6. (EPC-7 and VXLink only)
I_TRIG_TTL7	TTL trigger 7. (EPC-7 and VXLink only)

Use **ivxigettrigroute** to get the trigger mask bits corresponding to the **I_TRIG_ALL** and **I_TRIG_STD** constants.

The trigger(s) corresponding to the **I_TRIG_STD** constant can be modified using **ivxitrigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Use **ixtrig** to assert a trigger line then immediately deassert it.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ivxigettrigroute, ivxitrigon, ivxitrigroute, ixtrig**

Example See **ivxitrigon**

ivxitrigon

Description Asserts VXIbus trigger lines.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ivxitrigon(INST id, unsigned long triggermask);
```

id	VXI interface session handle.
triggermask	VXIbus trigger line(s) to assert.

Visual Basic Synopsis

Declare Sub **ivxitrigon** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *which* As Long)

Remarks This function asserts the VXIBus trigger lines specified in *triggermask* for the VXI interface session specified by *id*. *Triggermask* is a bit mask that is an OR'd combination of one or more of the following:

Constant	Description
I_TRIG_ALL	All valid triggers. (EPC-7 and VXLink only)
I_TRIG_ECL0	ECL trigger 0. (EPC-7 only)
I_TRIG_ECL1	ECL trigger 1. (EPC-7 only)
I_TRIG_EXT0	EXT trigger 0 ((EPC-7 only)). Has no effect unless **I_TRIG_EXT0** has been routed as an output of another trigger; see **ivxitrigroute**).
I_TRIB_EXT1	EXT trigger 1 (EPC-7 only)). Has no effect unless **I_TRIG_EXT1** has been routed as an output of another trigger; see **ivxitrigroute**).
I_TRIG_STD	Standard trigger. (EPC-7 and VXLink only)
I_TRIG_TTL0	TTL trigger 0. (EPC-7 and VXLink only)

I_TRIG_TTL1	TTL trigger 1. (EPC-7 and VXLink only)
I_TRIG_TTL2	TTL trigger 2. (EPC-7 and VXLink only)
I_TRIG_TTL3	TTL trigger 3. (EPC-7 and VXLink only)
I_TRIG_TTL4	TTL trigger 4. (EPC-7 and VXLink only)
I_TRIG_TTL5	TTL trigger 5. (EPC-7 and VXLink only)
I_TRIG_TTL6	TTL trigger 6. (EPC-7 and VXLink only)
I_TRIG_TTL7	TTL trigger 7. (EPC-7 and VXLink only)

Use **ivxigettrigroute** to get the triggermask bits that correspond to the **I_TRIG_ALL** and **I_TRIG_STD** constants.

The trigger(s) corresponding to the **I_TRIG_STD** constant can be modified using **ivxitrigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Use **ixtrig** to assert a trigger line then immediately deassert it.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ivxigettrigroute, ivxitrigoff, ivxitrigroute, ixtrig**

Example

```
/*
 * ivxiton.c:  this example asserts, checks and then deasserts VXI TTL triggers
 *         using ivxitrigon(), ivxitrigoff(), and ivxibusstatus().
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ivxitrigon(void)
{
            int         error_number;
            INST        id;
            unsigned long     result;
```

```
/* Open a VXI interface session. */

id = iopen("vxi");
if (id == ((INST) 0))
{
   error_number = igeterrno();
   WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   return (error_number);
}

/* Assert and verify TTL trigger 0. */

error_number = ivxitrigon(id, I_TRIG_TTL0);
if (error_number != I_ERR_NOERROR)
{
   WinPrintf("FAILURE:  ivxitrigon().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   iclose(id);
   return (error_number);
}
error_number = ivxibusstatus(id, I_VXI_BUS_TRIGGER, &result);
if (error_number != I_ERR_NOERROR)
{
   WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   iclose(id);
   return (error_number);
}
if ((result & I_TRIG_TTL0) == 0)
{
   WinPrintf("FAILURE:  TTL trigger 0 not asserted!\n");
   iclose(id);
   return (error_number);
}
WinPrintf("TTL trigger 0 asserted.\n");

/* Deassert and verify TTL trigger 0. */

error_number = ivxitrigoff(id, I_TRIG_TTL0);
if (error_number != I_ERR_NOERROR)
{
   WinPrintf("FAILURE:  ivxitrigoff().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   iclose(id);
   return (error_number);
}
error_number = ivxibusstatus(id, I_VXI_BUS_TRIGGER, &result);
if (error_number != I_ERR_NOERROR)
{
   WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   iclose(id);
   return (error_number);
}
if ((result & I_TRIG_TTL0) == 0)
{
   WinPrintf("TTL trigger 0 deasserted.\n");
}
```

```
            else
            {
               WinPrintf("FAILURE:  TTL trigger 0 still asserted!\n");
            }
            iclose(id);
            return (error_number);
}
```

ivxitrigroute

Description Routes VXIbus trigger lines.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ivxitrigroute(INST id, unsigned long intrigger, unsigned long
    outtriggermask);
```

id	VXI interface session handle.
intrigger	Input trigger.
outtriggermask	Output trigger mask.

Visual Basic Synopsis

Declare Sub **ivxitrigroute** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *in_which* As Long, ByVal *out_which* As Long)

Remarks This function routes the VXIbus input trigger line *intrigger* to the VXIbus output trigger lines *outtriggermask* for the VXI interface of the session specified by *id*. Asserting an input trigger line causes assertion of all the routed output trigger lines.

Intrigger is a constant specifying the input trigger to route. *Outtriggermask* is an OR'd combination of constants specifying the routed trigger(s).

intrigger	*set outtriggermask*	Description
I_TRIG_STD	I_TRIG_ALL I_TRIG_EXT0 to EXT1 I_TRIG_STD I_TRIG_TTL0 to TTL7	Defines one or more triggers corresponding to the I_TRIG_STD constant. An *outtriggermask* containing the I_TRIG_EXT0 bit is valid only on an EPC-7, and only has an effect if I_TRIG_EXT0 is routed as an output trigger.
I_TRIG_EXT0	0x00000000	Unmaps EXT input trigger.
I_TRIG_EXT0	I_TRIG_TTL0 through I_TRIG_TTL7	Maps EXT input trigger to single TTl trigger.
0x00000000	I_TRIG_EXT1	Unmaps external output trigger.
I_TRIG_TTL0 through I_TRIG_TTL7	I_TRIG_EXT0	Maps a single TTL trigger to the external output trigger.

Valid combinations of *intrigger* and *outtriggermask* are:

intrigger	*outtriggermask*	Description
I_TRIG_STD	**I_TRIG_ALL** **I_TRIG_ECL0** to **ECL1** **I_TRIG_EXT0** **I_TRIG_STD** **I_TRIG_TTL0** to **TTL7**	Defines one or more triggers corresponding to the **I_TRIG_STD** constant. An *outtriggermask* containing the **I_TRIG_EXT0** bit is valid only on an EPC-7, and only has an effect if **I_TRIG_EXT0** is routed as an output trigger.
I_TRIG_TTL0 through **I_TRIG_TTL7**	**I_TRIG_EXT0**	Defines **I_TRIG_EXT0** as an output of another trigger. Valid only on an EPC-7.
I_TRIG_EXT0	**I_TRIG_TTL0** through **I_TRIG_TTL7**	Defines **I_TRIG_EXT0** as the input to one or more triggers. Valid only on an EPC-7.

This functionality is not present on an EPC-8.

If *intrigger* is **I_TRIG_STD**, then *outtriggermask* defines which triggers are affected when a subsequent **isetintr**, **ivxitrigon**, **ixtrig**, or **ivxitrigoff** function call executes with the **I_TRIG_STD** constant specified.

Calls to **ivxitrigroute** override previous routings. For example, routing **I_TRIG_STD** to **I_TRIG_TTL7** invalidates the default routing for **I_TRIG_STD**.

On an EPC-7, **I_TRIG_EXT0** must be routed as either an output from another trigger or as an input to exactly one trigger. It cannot be routed as an output trigger and an input trigger simultaneously. Also, **I_TRIG_EXT0** routing can never be disabled. At power-up, **I_TRIG_EXT0** is routed as an input to **I_TRIG_TTL0**.

On the VXLink interface, **I_TRIG_EXT0** can be either disabled or routed as an input to exactly one TTL trigger. **I_TRIG_EXT1** can be either disabled or routed as an output from exactly one TTL trigger.

Use **ivxigettrigroute** to get the trigger mask bits that correspond to the **I_TRIG_ALL** and **I_TRIG_STD** constants.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **isetintr, ivxigettrigroute, ivxitrigoff, ivxitrigon, ixtrig**

Example See **ivxigettrigroute**

ivxiwaitnormop

Description Waits for normal operation of a VXI interface.

C Synopsis

#include "sicl.h"

int SICLAPI
ivxiwaitnormop(INST *id***);**

id VXI session handle.

Visual Basic Synopsis

Declare Sub **ivxiwaitnormop** Lib "sicl16.dll" (ByVal *id* As Integer)

Remarks If the VXIbus interface specified by *id* has begun normal operations, the function returns immediately.

If the interface has not begun normal operations, the function waits until normal operation is established or a timeout occurs if a timeout limit has been set by **itimeout**.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iopen, itimeout**

Example

```
/*
 * normop.c: this example calls ivxiwaitnormop() to wait for the start of
 *              normal VXI operation.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ivxiwaitnormop(void)
{
            int     error_number;
            INST    id;
```

```
            /* Open a VXI interface session. */

            id = iopen("vxi");
            if (id == ((INST) 0))
            {
               error_number = igeterrno();
               WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
               return (error_number);
            }

            /* Wait (forever) for normal VXI operation. */

            error_number = ivxiwaitnormop(id);
            if (error_number != I_ERR_NOERROR)
            {
               WinPrintf("FAILURE:  ivxiwaitnormop().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
            }
            iclose(id);
            return (error_number);
}
```

ivxiws

Description Sends a word serial command to a VXI device.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ivxiws(INST id, unsigned short command, unsigned short _far
    *reply, unsigned short _far *error);
```

id	VXI device session handle.
command	Word serial command to send.
reply	Pointer to a location where the function stores the word serial response.
error	Pointer to a location where the function stores the response to a READ PROTOCOL ERROR word serial command.

Visual Basic Synopsis

Declare Sub **ivxiws** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *wscmd* As Integer, *wsresp* As Any, *rpe* As Any)

Remarks This function sends the word serial command specified by *command* to the VXI device session specified by *id*.

If *reply* is not null, a word serial response is read and stored in the location specified by *reply*.

If *error* is not null and a word serial protocol error is detected, a READ PROTOCOL ERROR word serial command is sent to the device and the response is placed in the location specified by *error*.

If a word serial protocol error is detected, the function returns **I_ERR_IO.**

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iclear, ilocal, iremote, itimeout.**

Example

```
/*
 * ivxiws.c: this example uses ivxiws() to send a word serial command to a
 *              device.
 */

#include "sicl.h"
#include "wscmds.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ivxiws(void)
{
             int          error_number;
             INST         id;
             unsigned short    ws_error;
             unsigned short    ws_reply;

             /* Open a VXI device session. */

             id = iopen("vxisink");
             if (id == ((INST) 0))
             {
                error_number = igeterrno();
                WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                          igeterrstr(error_number),
                          error_number);
                return (error_number);
             }

             /* Send a READ PROTOCOL word serial command to the device. */

             error_number = ivxiws(id, WSC_RDPROTO, &ws_reply, &ws_error);
             if (error_number != I_ERR_NOERROR)
             {
                WinPrintf("FAILURE:  ivxiws().  Error = %s (%d).\n",
                          igeterrstr(error_number),
                          error_number);
             }
             else
             {
                WinPrintf("Sent READ PROTOCOL to \"vxisink\".  Response =
0x%04X\n",
                          ws_reply);
             }
             iclose(id);
             return (error_number);
}
```

iwaithdlr

Description Waits for an SRQ or interrupt handler function to execute.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iwaithdlr(long timeout);
```

timeout Timeout interval, in milliseconds.

Visual Basic Synopsis

None

Remarks This function waits for *timeout* milliseconds for an SRQ or interrupt handler function to execute. If *timeout* is less than or equal to zero, processing suspends indefinitely until an SRQ or interrupt event handler completes execution. If *timeout* is greater than zero, processing suspends for up to the specified time.

This function ignores the state of event processing as set by **iintron** and **iintroff**.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **iintron, iintroff, ionintr, ionsrq, isetintr**

Example See **ionintr**.

iwblockcopy

Description Copies blocks of 16-bit words from one set of sequential memory locations to another.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iwblockcopy(INST id, unsigned short _far *src, unsigned short
    _far  *dest, unsigned long count, int swap);
```

id	Session handle.
src	Source pointer.
dest	Destination pointer.
count	Number of 16-bit words to copy.
swap	Byte swap flag.

Visual Basic Synopsis

Declare Sub **iwblockcopy** Lib "sicl16.dll" (ByVal *id* As Integer, *src* As Any, *dest* As Any, ByVal *cnt* As Long, ByVal *swap* As Integer)

Remarks This function copies 16-bit words from successive memory locations beginning at *src* into successive memory locations beginning at *dest*. *Count* specifies the number of 16-bit words to transfer. *Id* specifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Whether or not byte-swapping occurs depends upon the source and destination of the copy operation. The swap flag is ignored.

The following scenarios are possible when accessing EPC and VXIbus memory:

src	*dest*	Result
EPC	EPC	No byte-swapping
EPC	VXI	One byte-swap
VXI	EPC	One byte-swap
VXI	VXI	Two byte-swaps (equals no byte-swapping)

For 16-bit byte-swapping to execute properly, all 16-bit VXIbus accesses must be aligned on 16-bit boundaries.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibblockcopy, ilblockcopy, imap, iwpeek, iwpoke, iwpopfifo, iwpushfifo,**

Example

```
/*
 * iwblock.c:   this example uses iwblockcopy() to read a VXI register of the
 *              device configured as ULA 0.  The bit encoding of this register
 *             is defined by the VXI specification.  For this particular
 *             example, the program is using the Device Class bits.
 */

#include <windows.h>
#include "sicl.h"

#define NO_BYTE_SWAP 0
#define BYTE_SWAP    1

#define      VXI_REG_OFFSET    0xC000

char _far *Strings[] =
{
             "Memory",
             "Extended",
             "Message Based",
             "Register Based"
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_iwblockcopy(void)
{
             volatile char _far *   mapped_ptr;
             unsigned short         id_reg;
             int                error_number;
             INST               id;
```

```
/* Open a VXI interface session. */

id = iopen("vxi");
if (id == ((INST) 0))
{
   error_number = igeterrno();
   WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   return (error_number);
}

/* Map in A16 space */

mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
if (mapped_ptr == NULL)
{
   error_number = igeterrno();
   WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   iclose(id);
   return (error_number);
}

/* Copy the ID register of the device at ULA 0 and determine     */
/* the device's class.                                           */

error_number = iwblockcopy( id,
                   (unsigned short *)
                   (mapped_ptr + VXI_REG_OFFSET),
                   &id_reg,
                   1,
                   BYTE_SWAP);
if (error_number != I_ERR_NOERROR)
{
   WinPrintf("FAILURE:  iwblockcopy().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
}
else
{
   WinPrintf("Class of device at ULA 0 is %s.\n",
             Strings[id_reg >> 14]);
}
iclose(id);
return (error_number);
}
```

iwpeek

Description Reads a 16-bit word from a mapped address.

C Synopsis

#include "sicl.h"

unsigned short SICLAPI
iwpeek(volatile unsigned short _far *addr);

addr Address of a 16-bit word.

Visual Basic Synopsis

Declare Function **iwpeek** Lib "sicl16.dll" Alias "vbiwpeek" (ByVal *addr* As Long) As Integer

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call. Byte swapping is always performed.

For byte swapping to work properly, all 16-bit VXIbus accesses must be aligned on a 16-bit boundary.

Return Value The function returns the 16-bit word stored at *addr*.

See Also **ibpeek, ilpeek, imap, iwpoke**

Example

```
/*
 * iwpeek.c: this example uses iwpeek()/iwpoke() to read/write this EPC's
 *              slave memory via the VXIbus.
 */

#include <windows.h>
#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_iwpeek(void)
{
            volatile char _far *    mapped_ptr;
            int                 error_number;
            unsigned long           address_space;
            unsigned long           base_address;
            unsigned short          memory_data;
```

```
INST            id;

/* Open a VXI interface session. */

id = iopen("vxi");
if (id == ((INST) 0))
{
   error_number = igeterrno();
   WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   return (error_number);
}

/* Query the location of our slave memory. */

error_number = ivxibusstatus(    id,
                  I_VXI_BUS_SHM_ADDR_SPACE,
                  &address_space);
if (error_number != I_ERR_NOERROR)
{
   WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   iclose(id);
   return (error_number);
}
if (address_space == 0)
{
   WinPrintf("FAILURE:  the EPC's slave memory is not enabled.\n");
   iclose(id);
   return (error_number);
}
error_number = ivxibusstatus(    id,
                  I_VXI_BUS_SHM_PAGE,
                  &base_address);
if (error_number != I_ERR_NOERROR)
{
   WinPrintf("FAILURE:  ivxibusstatus().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   iclose(id);
   return (error_number);
}
iclose(id);

/* Open a VXI device session. */

id = iopen("vxisink");
if (id == ((INST) 0))
{
   error_number = igeterrno();
   WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
             igeterrstr(error_number),
             error_number);
   return (error_number);
}

/* Map in the first 64K of the EPC's slave memory. */

if (address_space == 24)
{
   mapped_ptr = imap(   id,
                  I_MAP_A24,
                  (unsigned int) (base_address >> 8),
```

```
                    1,
                    NULL);
        }
        else
        {
           mapped_ptr = imap(   id,
                    I_MAP_A32,
                    (unsigned int) base_address,
                    1,
                    NULL);
        }
        if (mapped_ptr == NULL)
        {
           error_number = igeterrno();
           WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                     igeterrstr(error_number),
                     error_number);
           iclose(id);
           return (error_number);
        }

        /* Read a 16-bit value from physical address 0 of EPC memory     */
        /* via the VXIbus, then write the value back.                     */

        memory_data = iwpeek((volatile unsigned short _far *) mapped_ptr);
        iwpoke((volatile unsigned short _far *) mapped_ptr, memory_data);
        iclose(id);
        return (I_ERR_NOERROR);
}
```

iwpoke

Description Writes a 16-bit word to a mapped address.

C Synopsis

```
#include "sicl.h"

void SICLAPI
iwpoke(volatile unsigned short _far *dest, unsigned short value);
```

dest	Destination address.
value	16-bit word to write.

Visual Basic Synopsis

```
Declare Sub iwpoke Lib "sicl16.dll" Alias "vbiwpoke" (ByVal addr
As Long, ByVal value As Integer)
```

Remarks The *addr* pointer should be a mapped pointer returned by a previous **imap** call. Byte swapping is always performed.

For byte-swapping to work properly, all 16-bit VXIbus accesses must be aligned on a 16-bit boundary.

Return Value The function returns no value.

See Also **ibpoke**, **ilpoke**, **imap**, **iwpeek**

Example See **iwpeek**

iwpopfifo

Description Copies 16-bit words from a single memory location (FIFO register) to sequential memory locations.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iwpopfifo(INST id, unsigned short _far *fifo, unsigned short
    _far *dest, unsigned long count, int swap);
```

id	Session handle.
fifo	FIFO pointer.
dest	Destination address.
count	Number of 16-bit words to copy.
swap	Byte swap flag.

Visual Basic Synopsis

Declare Sub **iwpopfifo** Lib "sicl16.dll" (ByVal *id* As Integer, *fifo* As Any, *dest* As Any, ByVal *cnt* As Long, ByVal *swap* As Integer)

Remarks This function copies *count* 16-bit words from *fifo* into sequential memory locations beginning at *dest*. *Count* specifies the number of 16-bit words to transfer. *Id* identifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Whether or not byte-swapping occurs depends upon the source and destination of the copy operation. The swap flag is ignored. The following scenarios are possible when accessing EPC and VXIbus memory:

src	*dest*	Result
EPC	EPC	No byte-swapping
EPC	VXI	One byte-swap
VXI	EPC	One byte-swap
VXI	VXI	Two byte-swaps (equals no byte-swapping)

For 16-bit byte-swapping to execute properly, all 16-bit VXIbus accesses must be aligned on 16-bit boundaries.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibpopfifo, ilpopfifo, imap, iwpushfifo**

Example

```
/*
 * iwpop.c:  this example uses iwpopfifo() to read from a hypothetical
 *             FIFO at address 0 in A16 space.
 */

#include <windows.h>
#include "sicl.h"

#define NO_BYTE_SWAP 0
#define BYTE_SWAP    1

void
WinPrintf(char _far *Format_String, ...);

int
sample_iwpopfifo(void)
{
            volatile char _far *   mapped_ptr;
            unsigned short         fifo_data[5];
            int                error_number;
            INST               id;

            /* Open a VXI interface session. */

            id = iopen("vxi");
            if (id == ((INST) 0))
            {
               error_number = igeterrno();
               WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
               return (error_number);
            }
```

```
            /* Map in A16 space */

            mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
            if (mapped_ptr == NULL)
            {
               error_number = igeterrno();
               WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
               iclose(id);
               return (error_number);
            }

            /* Read the FIFO 5 times, storing the values into fifo_data[]. */

            error_number = iwpopfifo(    id,
                               (unsigned short *) mapped_ptr,
                               fifo_data,
                               sizeof(fifo_data) / sizeof(unsigned short),
                               BYTE_SWAP);
            if (error_number != I_ERR_NOERROR)
            {
               WinPrintf("FAILURE:  iwpopfifo().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
            }
            iclose(id);
            return (error_number);
}
```

iwpushfifo

Description Copies 16-bits words from sequential memory locations to a single memory location (FIFO register).

C Synopsis

```
#include "sicl.h"

int SICLAPI
iwpushfifo(INST id, unsigned short _far *src, unsigned short
    _far *fifo, unsigned long count);
```

id	Session handle.
src	Source address.
fifo	FIFO pointer.
count	Number of 16-bit words to copy.
swap	Byte swap flag.

Visual Basic Synopsis

Declare Sub **iwpushfifo** Lib "sicl16.dll" (ByVal *id* As Integer, *src* As Any, *fifo* As Any, ByVal *cnt* As Long, ByVal *swap* As Integer)

Remarks This function copies *count* 16-bit words from the sequential memory locations beginning at *src* into the FIFO at *fifo*. *Count* specifies the number of 16-bit words to transfer. *Id* specifies the interface to use for the transfer.

The function does not detect bus errors caused by its use.

This function supports copies from any address (mapped bus address or local EPC address) to any address (mapped bus address or local EPC address).

Whether or not byte-swapping occurs depends upon the source and destination of the copy operation. The swap flag is ignored.

The following scenarios are possible when accessing EPC and VXIbus memory:

src	*dest*	Result
EPC	EPC	No byte-swapping
EPC	VXI	One byte-swap
VXI	EPC	One byte-swap
VXI	VXI	Two byte-swaps (equals no byte-swapping)

For 16-bit byte-swapping to execute properly, all 16-bit VXIbus accesses must be aligned on 16-bit boundaries.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ibpushfifo, ilpushfifo, imap, iwpopfifo**

Example

```
/*
 * iwpush.c: this example uses iwpushfifo() to write to a hypothetical
 *             FIFO at address 0 in A16 space.
 */

#include <windows.h>
#include "sicl.h"

#define NO_BYTE_SWAP 0
#define BYTE_SWAP    1

unsigned short fifo_data[] =
{
            0x5361, 0x6D70, 0x6C65, 0x4461, 0x7461
};

void
WinPrintf(char _far *Format_String, ...);

int
sample_iwpushfifo(void)
{
            volatile char _far *   mapped_ptr;
            int                    error_number;
            INST                   id;

            /* Open a VXI interface session. */

            id = iopen("vxi");
            if (id == ((INST) 0))
            {
               error_number = igeterrno();
               WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
               return (error_number);
```

```
        }

        /* Map in A16 space */

        mapped_ptr = imap(id, I_MAP_A16, 0, 0, NULL);
        if (mapped_ptr == NULL)
        {
           error_number = igeterrno();
           WinPrintf("FAILURE:  imap().  Error = %s (%d).\n",
                     igeterrstr(error_number),
                     error_number);
           iclose(id);
           return (error_number);
        }

        /* Write the FIFO 5 times, storing the values from fifo_data[]. */

        error_number = iwpushfifo(  id,
                           fifo_data,
                           (unsigned short *) mapped_ptr,
                           sizeof(fifo_data) / sizeof(unsigned short),
                           BYTE_SWAP);
        if (error_number != I_ERR_NOERROR)
        {
           WinPrintf("FAILURE:  iwpushfifo().  Error = %s (%d).\n",
                     igeterrstr(error_number),
                     error_number);
        }
        iclose(id);
        return (error_number);
}
```

iwrite

Description Writes data to a device or interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
iwrite(INST id, char _far *buf, unsigned long bufsize, int end,
    unsigned long _far *actualcnt);
```

id	Interface session handle.
buf	Pointer to the data buffer.
bufsize	Length, in bytes, of data buffer.
end	END indicator flag.
actualcnt	Pointer to a location where the function stores the actual number of bytes written.

Visual Basic Synopsis

Declare Sub **iwrite** Lib "sicl16.dll" (ByVal *id* As Integer, *buf* As Any, ByVal *datalen* As Long, ByVal *endi* As Integer, *actual* As Long)

Remarks This function writes the *bufsize* bytes at *buf* to the device or interface of the session specified by *id*. It performs no buffering, formatting or data conversion.

Writing ends when *bufsize* bytes are written or a timeout occurs. This function blocks until one of these two conditions is met.

If *end* is non-zero, the function writes an END indicator with the last data byte. If *end* is zero, the function does not write an END indicator with the last data byte.

If *actualcnt* is not null, the function stores the number of data bytes written in the referenced memory location.

For VXI device sessions, the function generates BYTE AVAILABLE word serial commands. The function supports only message based VXI devices; other VXI devices generate an error.

For VXI interface sessions, the function generates an **I_ERR_NOTSUPP** error.

For GPIB device sessions, the function first causes all devices to unlisten. Then, it issues the interface's talk address, followed by the device's listen address. Finally, the function writes the data.

For GPIB interface sessions, the function writes bytes directly to the interface without performing any addressing.

To avoid unpredictable results, do not mix buffered output function calls (**ifwrite, iprintf, ipromptf, ivprintf, ivpromptf**) and unbuffered output function calls (**iwrite**) within the same session.

Return Value The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also **ifwrite, iprintf, ipromptf, iread, itimeout, ivprintf, ivpromptf**

Example

```
/*
 * iwrite.c:  This example calls iwrite() to write to an instrument.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

#define      BUFFER_SIZE  3
#define      EOI          1

char DataBuffer[] = "RST";

int
sample_iwrite(void)
{
             int          error_number;
             unsigned long     actual_count;
             INST         id;

             /* Open a VXI device session. */

             id = iopen("vxisink");
```

```
            if (id == ((INST) 0))
            {
               error_number = igeterrno();
               WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
               return (error_number);
            }

            /* Write a buffer of data to the device. */

            error_number = iwrite(id, DataBuffer, BUFFER_SIZE, EOI,
&actual_count);
            if (error_number != I_ERR_NOERROR)
            {
               WinPrintf("FAILURE:  iwrite().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
            }
            else
            {
               WinPrintf("%d bytes written to \"vxisink\".\n",
                         BUFFER_SIZE);
            }
            iclose(id);
            return (error_number);
}
```

ixtrig

Description Asserts and deasserts one or more triggers on an interface.

C Synopsis

```
#include "sicl.h"

int SICLAPI
ixtrig(INST id, unsigned long triggermask);
```

id	Session handle.
triggermask	Trigger mask to assert.

Visual Basic Synopsis

Declare Sub **ixtrig** Lib "sicl16.dll" (ByVal *id* As Integer, ByVal *which* As Long)

Remarks For GPIB interface sessions, the function issues a Group Execute Trigger (GET) command without performing any addressing. The user should use **igpibsendcmd** to set up those listeners to receive the trigger. The *triggermask* argument must be **I_TRIG_STD** or **I_TRIG_ALL**.

For VXI interface sessions, the function asserts and immediately deasserts the VXIbus triggers specified by the *triggermask* argument. *Triggermask* is a bit mask that is an OR'd combination of one or more of the following:

Constant	Description
I_TRIG_ALL	All valid triggers.
I_TRIG_ECL0	ECL trigger 0.
I_TRIG_ECL1	ECL trigger 1.
I_TRIG_EXT0	EXT trigger 0 (valid only on an EPC-7). Has no effect unless **I_TRIG_EXT0** has been routed as an output of another trigger; see **ivxitrigroute**).
I_TRIG_STD	Standard trigger.
I_TRIG_TTL0	TTL trigger 0.
I_TRIG_TTL1	TTL trigger 1.
I_TRIG_TTL2	TTL trigger 2.

I_TRIG_TTL3	TTL trigger 3.
I_TRIG_TTL4	TTL trigger 4.
I_TRIG_TTL5	TTL trigger 5.
I_TRIG_TTL6	TTL trigger 6.
I_TRIG_TTL7	TTL trigger 7.

Use **ivxigettrigroute** to get the VXIbus trigger mask bits corresponding to the **I_TRIG_ALL** and **I_TRIG_STD** constants.

The VXIbus triggers corresponding to the **I_TRIG_STD** constant can be modified using **ivxitrigroute**. By default, **I_TRIG_STD** corresponds to **I_TRIG_TTL0**.

Return Value

The function returns **I_ERR_NOERROR** upon successful completion. Any other return value indicates a failure.

See Also

itimeout, itrigger, ivixgettrigroute, ivxitrigoff, ivxitrigon, ivxitrigroute

Example

```
/*
 * ixtrig.c:  this example uses ixtrig() to send an unaddressed GET on GPIB.
 */

#include "sicl.h"

void
WinPrintf(char _far *Format_String, ...);

int
sample_ixtrig(void)
{
            int     error_number;
            INST    id;

            /* Open a GPIB interface session. */

            id = iopen("gpib");
            if (id == ((INST) 0))
            {
               error_number = igeterrno();
               WinPrintf("FAILURE:  iopen().  Error = %s (%d).\n",
                         igeterrstr(error_number),
                         error_number);
               return (error_number);
            }

            /* Send an unaddressed GET command. */

            error_number = ixtrig(id, I_TRIG_STD);
            if (error_number != I_ERR_NOERROR)
            {
```

```
        WinPrintf("FAILURE:  ixtrig().  Error = %s (%d).\n",
                igeterrstr(error_number),
                error_number);
    }
    iclose(id);
    return (error_number);
}
```

_siclcleanup

Description Releases Windows 3.1 I/O resources before terminating.

C Synopsis

```
#include "sicl.h"

int _export SICLAPI
_siclcleanup(void);
```

Visual Basic Synopsis

Declare Sub **siclcleanup** Lib "sicl16.dll" Alias "_siclcleanup" ()

Remarks This function tells Windows 3.1 that a program is done with all SICL I/O resources. The function must be called before a Windows 3.1 SICL application terminates.

Return Value This function returns zero (0) if successful, or a non-zero error number if an error occurs.

NOTES

3. Advanced Topics

3

This chapter discusses topics of interest to advanced application programmers. Topics include:

- Byte Ordering and Data Representation
- Handler Operations Under Windows
- SRQ, Interrupt, and Error Handler Execution
- VXI TTL Trigger Interrupts on an EPC-7
- Common SICL proglrms with Windows 3.1
- Avoiding Nested I/O
- Using **_siclcleanup** Before Exiting

3.1 Byte Ordering and Data Representation

Byte ordering adds complexity to the VXIbus interface. Many VXIbus devices use the data formats of Motorola microprocessors. Others, including RadiSys EPC controllers, use the data format of Intel microprocessors. Although the Motorola and Intel microprocessors use the same data types, the hardware representations of these data types differ.

Figure 3-1 shows how the same sequence of bytes in memory is interpreted by Intel and Motorola microprocessors. Memory value 11 is at the lowest address and memory value 88 is at the highest address. The data widths shown correspond to the data operand sizes found on both microprocessors.

Memory Value	Intel Order	Data Width	Motorola Order
11	11	8 bits	11
22	2211	16 bits	1122
33			
44	44332211	32 bits	11223344
55			
66			
77			
88	8877665544332211	64 bits	1122334455667788

Figure 3-1. Byte Order Example

3.1.1 Byte Swapping Functions

The SICL **iswap** function converts 16-bit, 32-bit, and 64-bit data between Intel and Motorola byte orders (8-bit data does not require conversion).

The SICL 16-bit peek and poke functions (**iwpeek** and **iwpoke**) and 32-bit peek and poke functions (**ilpeek** and **ilpoke**) always perform byte-swapping. The SICL peek functions assume the data at the specified address is in Motorola byte order, and byte-swaps the data to Intel byte order after reading it. Conversely, the SICL poke functions assume the specified data is in Intel byte order, and byte-swaps the data to Motorola byte order before writing it to the specified address.

The SICL 16-bit block transfer functions (**iwblockcopy**, **iwpopfifo**, and **iwpushfifo**) and 32-bit block transfer functions (**ilblockcopy**, **ilpopfifo**, and **ilpushfifo**) conditionally perform byte-swapping. The SICL block transfer functions assume that all EPC addresses use Intel byte order and all VXIbus addresses use Motorola byte order.

3.1.2 Correcting Data Structure Byte Ordering

The SICL 16-bit and 32-bit peek and poke (**ilpeek**, **iwpeek**, **ilpoke**, and **iwpoke**) and block transfer functions (**ilblockcopy** and **iwblockcopy**) do not solve all byte ordering problems. Even if byte-swapping occurs during a SICL block transfer function, byte ordering problems occur when data is copied between Motorola and Intel memory using a different data width than the width of the operand itself. This situation occurs when a data structure containing mixed-type fields is copied in a single operation. The following code fragment illustrates how to use the **iswap** function to correct the byte order in the local copy of the data structure:


```
struct DataStructure
{
        char        field8;
        short       field16;
        long        field32;
        double      field64;
} data;

/* Copy the data structure to local memory from the VMEbus. */

ibblockcopy(ID, VXIADDR, &data, sizeof(struct DataStructure));

/* Byte-swap the individual structure fields (data.field8 is an
8-bit field, so it is already correct).
*/

iswap(&data.field16,sizeof(short),sizeof(short));
iswap(&data.field32,sizeof(long),sizeof(long));
iswap(&data.field64,sizeof(double),sizeof(double));
```

In the above example, the data structure was copied from VXIbus memory one byte at a time. To copy data from EPC memory to Motorola-ordered memory, byte-swap the fields of the structure in local memory (using the above byte swapping functions) and copy the data using the SICL **ibblockcopy** function.

It is sometimes more efficient to copy blocks of data using data transfer width greater than the expected data width. If you use a greater data transfer width to copy data structures containing mixed-type fields to/from Motorola-order memory, do not use the SICL function byte-swapping feature. Swap the data structure fields individually.

3.2 Handler Operations Under Windows

SRQ, interrupt, and error handlers execute as part of a Windows application's foreground thread, not as part of an interrupt thread. This feature implies that a SICL handler can safely call all "C" library and Windows support functions. Also, most SICL functions may be called from a SICL handler.

3.3 SRQ Handler Execution

These conditions must be true before an application's SICL SRQ handlers can execute:

- The application must call **ionsrq** to install an SRQ handler.
- An SRQ must occur.
- The application must call **iwaithdlr** or enable asynchronous event processing. Asynchronous event handling is enabled by default. If disabled, it can be re-enabled by calling **iintron**.

SICL discards all SRQ events that occur before the application installs an SRQ handler.

When an application installs an SRQ handler and enables asynchronous event processing, the SRQ handler processes SRQ events as soon as they are received.

When an application installs an SRQ handler and does not enable asynchronous event processing, SICL queues SRQ events as they are received. The SRQ handler will process the queued events when the application enables asynchronous event processing or calls **iwaithdlr**. If the application removes the installed SRQ handler before processing the queued events, the handler discards the events.

Under Windows, an installed SRQ handler executes as part of the application's foreground thread, with virtual interrupts in a state defined by the application, and using the application's stack.

3.4 Interrupt Handler Execution

These conditions must be true before an application's SICL interrupt handlers can execute:

- The application must use **ionintr** to install an interrupt handler.
- The application must use **isetintr** to enable interrupt reception.
- An interrupt must occur.
- The application must call **iwaithdlr** or enable asynchronous event processing. Asynchronous event processing is enabled by default. If disabled, it can be re-enabled by calling **iintron**.

SICL discards all interrupt events that occur before the application installs an interrupt handler and enables interrupt reception.

When an application installs an interrupt handler, enables interrupt reception, and enables asynchronous event processing, the interrupt handler processes interrupts as soon as they are received.

When an application installs an interrupt handler, enables interrupt reception, and does not enable asynchronous event processing, SICL queues the interrupts as they are received. The interrupt handler will process the interrupts when the application enables asynchronous event processing or calls **iwaithdlr**. If the application removes the interrupt handler before processing the queued interrupts, the handler discards the interrupts.

Under Windows, an installed interrupt handler executes as part of the application's foreground thread, with virtual interrupts in a state defined by the application, and using the application's stack.

3.5 Error Handler Execution

These conditions must be true before an application's SICL error handler can execute:

- The application must use **ionerror** to install the error handler.
- A SICL error must occur.

SICL discards all errors that occur before the application installs an error handler.

When an application has installed an error handler, and an error occurs, the error handler processes the error.

Enabling or disabling asynchronous event processing does not affect error handler execution.

3.6 VXI TTL Trigger Interrupts on an EPC-7

Receiving and processing VXI TTL trigger interrupts on an EPC-7 requires software intervention.

EPC-7 hardware generates a VXI TTL trigger interrupt when all of the following conditions are true:

- A bit in the TTL trigger interrupt enable register is set. The SICL function **isetintr** clears the register then sets one or more of the register's bits when it enables the reception of **I_INTR_TRIG** interrupts for a VXI interface session.
- The corresponding bit in the TTL trigger latch register is clear.
- The corresponding TTL trigger line is asserted for at least 30 nanoseconds.

The main complication to this scenario is that a bit in the TTL trigger latch register cannot be cleared until the corresponding TTL trigger line is deasserted. In order to clear a bit in the register, the register must be read while the corresponding TTL trigger line is deasserted. TTL trigger line assertion is not necessarily under EPC control.

The operation of the EPC-7 TTL trigger latch register has three potential side effects for software:

- If a TTL trigger interrupt remains enabled after receiving the initial interrupt and clearing the trigger latch register, the CPU can be monopolized by redundant TTL trigger interrupts.

- If a TTL trigger latch register bit is not cleared before enabling the corresponding TTL trigger interrupt, it is possible to receive an interrupt for a TTL trigger that was asserted, latched, and deasserted long before the TTL trigger interrupt was enabled.

- If a TTL trigger latch register bit is not cleared after receiving the corresponding TTL trigger interrupt, the EPC will not latch subsequent TTL trigger line asserts and, therefore, will miss subsequent TTL trigger interrupts.

To avoid the first side effect, the SICL implementation globally disables a TTL trigger interrupt upon reception. In addition, the SICL implementation provides two non-standard SICL VXI interface drivers, **DOCMD_VXI_CLEARLATCH** and **DOCMD_VXI_SETTRIGINTR**.

The **DOCMD_VXI_CLEARLATCH** SICL VXI interface driver command waits for specific bits in the EPC's TTL trigger latch register to become deasserted.

The **DOCMD_VXI_CLEARLATCH** command takes two unsigned 4-byte parameters: a trigger mask specifying the TTL trigger latch bits to wait upon and a timeout specifying the number of milliseconds to wait. The trigger mask parameter is an OR'd combination of the following constants:

Constant	Description
I_TRIG_TTL0	TTL trigger 0
I_TRIG_TTL1	TTL trigger 1
I_TRIG_TTL2	TTL trigger 2
I_TRIG_TTL3	TTL trigger 3
I_TRIG_TTL4	TTL trigger 4
I_TRIG_TTL5	TTL trigger 5
I_TRIG_TTL6	TTL trigger 6
I_TRIG_TTL7	TTL trigger 7

The **DOCMD_VXI_SETTRIGINTR** SICL VXI interface driver command enables one or more TTL trigger interrupts without disabling currently enabled TTL trigger interrupts.

The **DOCMD_VXI_SETTRIGINTR** command takes a single 4-byte parameter: a trigger mask specifying the TTL trigger interrupts to enable. It is an OR'd combination of the following contents:

Constant	Description
I_TRIG_TTL0	TTL trigger 0
I_TRIG_TTL1	TTL trigger 1
I_TRIG_TTL2	TTL trigger 2
I_TRIG_TTL3	TTL trigger 3
I_TRIG_TTL4	TTL trigger 4
I_TRIG_TTL5	TTL trigger 5
I_TRIG_TTL6	TTL trigger 6
I_TRIG_TTL7	TTL trigger 7

To avoid the side effect of receiving extraneous TTL trigger interrupts, execute the **DOCMD_VXI_CLEARLATCH** command before calling **isetintr** to enable **I_INTR_TRIG** interrupts for a VXI interface session. For example:

```
#include "radvxi.h"
#include "sicl.h"

int
EnableTTLTriggerInterrupts(INST Id, unsigned long TriggerMask)
{
     int error;
     unsigned long docmd_data[2];

     /*
      * Wait up to 10 seconds for the corresponding TTL
      * trigger latch register bits to clear, then enable
      * the TTL trigger interrupts.
      */

     docmd_data[0]=TrigggerMask;
     docmd_data[1]=10000;
     if((error=icmd(Id,
                    DOCMD_VXI_CLEARLATCH,
                    sizeof(docmd_data),
                    sizeof(unsigned long),
                    (void _far *)docmd_data))!=I_ERR_NOERROR)
     {
               return (error);
     }
```

```
        return (isetintr(Id, I_INTR_TRIG, (long) TriggerMask));
}
```

To avoid the side effect of missing multiple **I_INTR_TRIG** interrupts from the same TTL trigger, execute the **DOCMD_VXI_CLEARLATCH** and **DOCMD_VXI_SETTRIGINTR** commands immediately after receiving an **I_INTR_TRIG** interrupt, preferably as part of the SICL handler function itself.

For example:


```
#include "radvxi.h"
#include "sicl.h"

void
TTLTriggerInterruptHandler(INST Id, long Data1, long Data2)
{
     unsigned long docmd_data[2];

    /*
     * Wait "forever" for the corresp. TTL trigger latch
     * register bit to clear, then re-enable the TTL
     * trigger interrupt.
     */

     docmd_data[0]=(unsigned long)Data2;
     docmd_data[1]=0xFFFFFFFF;
     icmd(Id,
          DOCMD_VXI_CLEARLATCH,
          sizeof(docmd_data),
          sizeof(unsigned long),
          (void _far *)docmd_data);
      icmd(Id,
          DOCMD_VXI_SETTRIGINTR,
          sizeof)unsigned long),
          sizeof(unsigned long),
          (void _far *)docmd_data);

     /*
      * Execute other SICL handler tasks...
      */
 }
```

3.7 Common SICL problems with Windows 3.1

The following are descriptions of general problems that may occur.

General Protection Fault occurs when interrupt, SRQ or error handler called

Check that the interrupt or error handler routine was declare with the **SICLCALLBACK** modifier. Also, make sure that compiler options to generate prolog code for exported functions were selected. If you are using he QuickWin feature of Microsoft compilers, you must also use the **_loadds** modifier in the handler declaration.

General Protection Fault when calling SICL formatted I/O routine

Verify that all pointer parameters passed to SICL formatted I/O routines are non-null, are declared as **_far**, and that the compiler large memory model option is selected.

I_ERR_NESTED_IO occurs

A SICL I/O function call has been made before a previous call completed. In order to allow other Windows 3.1 applications to execute while a SICL application is running, SICL may temporarily suspend execution in the middle of a SICL call while waiting for a slow transaction to complete. Without this feature, your Windows system would be "locked up" until the transaction competes. However, because Windows is an event-/message-driven operating system, it is possible that the SICL application would receive a message instructing it to initiate another SICL call before the first one completes. This will result in a SICL error. Your program must be designed so that this situation does not occur.

3.8 Avoiding Nested I/O

The **I_ERR_NESTED** I/O error is generated by SICL whenever an attempt is made to call a SICL I/O function before a previous call to another SICL I/O function is complete. This error can occur in Windows 3.1 event-driven programs where SICL functions are called in response to events such as menu selections or button clicks. To avoid this problem, you should disable menu items, buttons or other controls that cause SICL calls to be made before calling a SICL function. Note that all of the sample programs that make SICL calls in response to events do this.

3.9 Using _siclcleanup Before Exiting

Make sure that you call **_siclcleanup** before exiting any function you create. This ensures that all Windows 3.1 I/O resources are released before the function terminates.

NOTES

4. I/O Formatting

This chapter discusses input and output format strings used in formatted I/O functions.

4.1 Output Format

A formatted output string controls how to format and convert optional **iprintf, ipromptf, isprintf, isvprintf, ivprintf,** and **ivpromptf** argument parameters.

A formatted output string contains ordinary characters, escape character sequences, and format specifications. Ordinary characters and escape character sequences are written as they are encountered.

Valid escape character sequences include:

Sequence	Description
\n	Write the ASCII line-feed character. The END indicator is also automatically sent, but can be disabled using the **–t** type characters.
\r	Write the ASCII carriage return character.
\\	Write the backslash (\) character.
\t	Write the ASCII tab character.
\###	Write the ASCII character specified by the three digit octal value ###.
*	Write the ASCII double-quote (") character.

Format specifications always begin with the percent sign (%) and are processed left to right. The first format specification causes the first argument parameter to be converted and written. The second format specification causes conversion and writing of the second argument, and so forth.

To avoid unpredictable results, there must be an argument for each format specification. If there are more arguments than format specifications, the excess arguments are ignored.

Format Specification Fields

There are six format specification fields. Each field is a character, a series of characters, or a number that specifies how to convert and write the associated *argument*. A format specification has these fields:

%[flags] [width] ["."precision] [","array_size] [length] type

Field	Description
type	Required characters that determine how to interpret the associated argument parameter (character, string, number, or pointer.)
flags	Optional characters that control the justification of characters and the printing of signs, blanks, decimal points. It also controls the printing of binary, octal and hexadecimal prefixes. More than one *flag* can appear in a format specification.
width	Optional characters that specify the minimum number of characters to write.
precision	Optional field that specifies the minimum number of digits to write for numeric formats. For string formats, *precision* specifies the maximum number of characters to write.
array_size	Optional field that specifies the number of elements in a numeric array.
length	Optional field that specifies an argument length modifier.

The simplest format contains only the percent sign % and a *type* field character. The optional fields that appear before the *type* field character control other formatting aspects. Any character that follows the % sign that is not a valid format specification field is interpreted as data.

Type Field

The *type* field is the only required format specification field and determines whether the associated argument is interpreted as a character, string, number, or pointer. It also controls writing of the END indicator when a linefeed character is written.

The following lists the valid *type* fields and describes how the associated argument parameter is interpreted:

Character	Type	Description
d	**int**	Signed decimal integer.
i	**int**	Signed decimal integer.
u	**int**	Unsigned decimal integer.
o	**int**	Unsigned octal integer.
x	**int**	Unsigned hexadecimal integer, using lower case letters.
X	**int**	Unsigned hexadecimal integer, using upper case letters.
f	**double**	Signed value having the form [−]*dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number. The number of digits after the decimal point depends on the *precision* field value.
e	**double**	Signed value having the form [-]*d.dddd***e**[*sign*]*ddd*, where *d* is a single decimal digit, *dddd* is one or more decimal digits, *ddd* is exactly three decimal digits, and *sign* is + or −.
E	**double**	Same as **e**, but the argument parameter uses "**E**" instead of "**e**".

g	**double**	Signed value in the **f** or **e** format, whichever is more compact for the given value and *precision*. The **e** format is used only when the exponent of the value is less than –4 or greater than or equal to the *precision* value. Trailing zeros and decimal point are written only if necessary.
G	**double**	Same as **g**; *argument* uses "**G**" instead of "**g**".
c	**int**	Single character.
C	**int**	Single character with the END indicator appended.
s	**Pointer**	Pointer to a null-terminated string. The null character or the *precision* value determines the length of the formatted string.
S	**Pointer**	Pointer to a null-terminates string that is written as an IEEE 488.2 STRING RESPONSE DATA block. The string is enclosed in double quotes ("). Double quotes within the string are double quoted ("").
n	**Pointer to integer**	Pointer to the number of characters converted and written to the buffer. This value is stored in the integer whose address is given as the argument.
b	**Pointer to data block**	Pointer to a block of data that is written as an IEEE 488.2 DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA block. *Flags* must contain a long specifying the maximum the number of elements (specified by the *size* **w**, **i**, **z**, or **Z** or default) in the data block or an asterisk. An asterisk specifies that the next two arguments contain the number of bytes to write and a pointer to the data block, respectively. The number of bytes to write is an unsigned long type. *Width* and *precision* are not allowed.
B	**Pointer to data block**	Same as **b**, except that the data block is written as an IEEE 488.2 INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA. This format writes the END indicator.

–t	**N/A**	Turns off sending of the END indicator when an ASCII line feed character is written from within the format string. The flag does not affect transmission of the END indicator for conversion with *types* **s**, **S**, **c**, and **C**.
+t	**N/A**	Turns on sending of the END indicator when an ASCII line feed character is written from within the format string. The flag does not affect transmission of the END indicator for conversion with *types* **s**, **S**, **c**, and **C**.

Flags Field

The *flags* field is optional and controls the justification of characters and the writing of signs, blanks, and decimal points. It also controls the writing of binary, octal, and hexadecimal prefixes, and modifies the meaning of the *type* field character. More than one *flags* field can be used in a format specification. The following describes the *flags* field and the defaults when that *flags* field is not specified:

Flags	Definition	Default
–	Left-justify the result within the given field width.	Right justify.
+	Prefix data with a sign (+ or –) if the data is of a signed type. Can be used with *flags* **@1**, **@2**, or **@3**. Not valid with *flags* **@H**, **@Q**, or **@B**.	Only negative values are prefixed.
blank	Prefix with a blank if the value is signed and positive; the blank is ignored if both the "blank" and "+" flags appear. Can be used with *flags* **@1**, **@2**, or **@3**, but not valid with *flags* **@H**, **@Q**, or **@B**	No blank appears.
0	If *width* is prefixed with **0**, pad with zeros until the minimum width is reached. If "**0**" and "–" are specified, the **0** is ignored. If **0** is specified with an integer format (**i**, **u**, **x**, **X**, **o**, **d**), the **0** is ignored.	No padding

#	When used with *types* **o**, **x**, or **X**, prefixes any non-zero output value with 0, 0x, or 0X, respectively.	No blank appears.
	When used with *types* **e**, **E**, or **f**, always forces the output value to contain a decimal point.	Decimal point appears only if digits follow it.
	When used with *types* **g** or **G**, forces the output value to always contain a decimal point and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it. Trailing zeros are truncated.
	Ignored when used with *types* **c**, **d**, **i**, **u**, or **s**.	
@1	Converts the *type* to an integer with no decimal point (NR1 compatible). Valid only with types **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.
@2	Converts the *type* to a number with at least one digit to be right of the decimal point (NR2 compatible). Valid only with the **d**, **f**, **e**, **E**, **g**, and **G** types.	Format data based on *type* only.
@3	Converts the *type* to a floating point number with exponential notations (NR3 compatible). Valid only with types **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.
@H	Create an IEEE 488.2 HEXADECIMAL NUMERIC RESPONSE DATA number (e.g. #H4A81). Valid only with *types* **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.
@Q	Create an IEEE 488.2 OCTAL NUMERIC RESPONSE DATA number (e.g. #Q17774). Valid only with *types* **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.
@B	Create an IEEE 488.2 BINARY NUMERIC RESPONSE DATA number (e.g. #B11011000). Valid only with *types* **d**, **f**, **e**, **E**, **g**, and **G**.	Format data based on *type* only.

Width Field

The *width* field is optional and contains a non-negative decimal integer that specifies the minimum number of characters written. If the number of characters to write is less than the specified *width*, blanks are added to the left or right of the value, depending on whether the – flag is specified, until the minimum width is reached. If *width* is prefixed with the "0" flag, zeros are added until the minimum with is reached.

The *width* field never causes a value to be truncated. If the number of characters to write is greater than the specified width or *width* is not given, all characters of the value are written (subject to *precision*).

If *width* is an asterisk (*), the next *argument* from the argument list is treated as an **int** and supplies the width value. The value to format immediately follows the *precision* value in the argument list. A nonexistent or small field does not cause truncation. If the result of the conversion is wider than the field width, the field expands to contain the conversion result.

Precision Field

The *precision* field is optional and contains a non-negative decimal integer, preceded by a period, that specifies the number of characters to write. Unlike the *width* field, *precision* can cause truncation of the output value, or rounding in the case of a floating point number.

If *precision* is an asterisk (*), the next *argument* from the argument list is treated as an **int** and supplies the precision value. The value to format immediately follows the *precision* value in the argument list. The following describes how *precision* values affect the various *types* (defaults are actions when *precision* is omitted with the *type*.)

Type	Meaning	Default
d, i, u, o, x, X	Specifies the minimum number of digits to write. If the number of digits in the argument is less than *precision*, the output is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	Default is 1.
e, E	Specifies the number of digits to write after the decimal point. The last written digit is rounded.	Default is 6. If *precision* is 0 or the period appears without a number following it, no decimal point is written.
f	Specifies the number of digits to write after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default is 6. If *precision* is 0 or the period appears without a number following it, no decimal point is written.
g, G	Specifies the maximum number of significant digits to write.	Six significant digits are written with any trailing zeros truncated.
c, C	No effect	Character is written.
s, S	Specifies the maximum number of character to write. Characters in excess of *precision* are not written	Characters are written until a null character is encountered.

Array_size Field

The *array_size* field is optional and contains a non-negative decimal integer, preceded by a comma, that specifies the number of elements in a numeric array.

If *array_size* is an asterisk (*), the next *argument* from the argument list is treated as an **int** and supplies the *array_size* value. An *array_size* field is only valid for integer (**d**) and floating point (**f**) types.

A formatted array is written as a comma-separated list with *array_size* entries separated by *array_size* -1 commas.

Length Field

The *length* field is optional and modifies the corresponding *argument* parameter modifier. The following defines the valid *length* entries:

Character	Description
h	Use with *types* **d**, **i**, **o**, **x**, and **X** to specify that the argument is a **short int** or with *type* **u** to specify a **short unsigned int**. If used with type **p** , it indicates a 16-bit pointer (offset only).
l	Use with *types* **d**, **i**, **o**, **x**, and **X** to specify that the argument is a long int. Use with the *type* **u** to specify a **long unsigned int**. Use with *types* **e**, **E**, **f**, **g**, and **G** to specify a **double** rather than a **float**. If used with *type* **p** , it indicates a 32-bit pointer. Use with *types* **b** and **B** to specify that the argument is a pointer to an array of **long unsigned ints** (32-bits). The data block is sent as an array of 32-bit words. The longwords are byte swapped and padded as necessary so that they conform to IEEE 488.2.
L	Use with *types* **e**, **E**, **f**, **g**, and **G** to specify a **long double**.
w	Use with *types* **b** and **B** to specify that the argument is a pointer to an array of **unsigned shorts** (16-bits). The data block is sent as an array of 16-bit words. *Flags* must be a long and specifies the number of words in the data block. The words are byte swapped and padded as necessary so that they conform to IEEE 488.2.

z Use with *types* **b** and **B** to specify that the argument is a pointer to an array of **floats**. The data block is sent as an array of 32-bit IEEE-754 floating point numbers. If the internal floating point representation of the computer is not IEEE-754 compliant, the numbers are converted before being written.

Z Use with *types* **b** and **B** to specify that the argument is a pointer to an array of **doubles**. The data block is sent as an array of 64-bit IEEE-754 floating point numbers. If the internal floating point representation of the computer is not IEEE-754 compliant, the numbers are converted before being written.

4.2 Input Format

A formatted input string controls how to format and convert input data for optional **ipromptf, iscanf, isscanf, isvscanf, ivpromptf,** and **ivscanf** argument parameters.

A formatted input string contain one or more of the following:

- The white-space characters blank (" "), tab (\t), or newline (\n). A white-space character causes the input function to read, but not store, all consecutive white-space characters up to the next non-white-space character. One white-space character in the format string matches any number (including 0) and combination of white-space characters in the input.

- Non-white-space characters, except the percent sign (%). A non-white-space character causes the input function to read, but not store, a matching non-white-space character. If the read character does not match the format character, the input function terminates.

- Format specifications. Format specifications begin with the percent sign (%) and cause the input function to read and convert input characters into values of a specified type. The value is assigned to an argument in the *argument* list.

Format specifications always begin with the percent sign (%) and are read left to right. Characters outside the format specification are expected to match the sequence of characters from the input data. The matching characters from the input data are scanned but not stored. If a scanned character does not match the format specification, the input function terminates.

The first format specification causes the first field from the input data to be converted and written to the location pointed to by the parameter. The second format specification causes conversion of the second field from the input data to be converted and written to the location pointed to by the second argument parameter, and so forth. There must be enough format specifications and arguments for the field being read for the results to be predictable. Excess format specifications and argument parameters are ignored.

Format Specification Fields

There are five format specification fields. Each field is a character, a series of characters, or number signifying a format option. The following defines the form of a format specification:

%[*] [*width*] ["," *array_size*] [*length*] *type*

Field	Description
type	Required field that determines whether the associated input field is interpreted as a character, string, number, or pointer.
*	Optional field that suppresses assignment of the next input field. The field is scanned but not stored.
width	Optional character that specifies the maximum number of characters to read.
array_size	Optional field that specifies the maximum number of elements in a numeric array.
length	Optional field that specifies an argument size modifier.

The simplest format contains only the percent sign (%) and a *type* field character. The option fields that appear before the *type* field character control other formatting aspects.

Type Field

The *type* field is the only required format field and determines whether the read input is interpreted as a character, string, number, or pointer. It also controls whether the read input terminates with a END indicator.

The following describes the *type* field characters:

Character	Expected Input Type	Argument Type
d	Decimal integer in either IEEE 488.2 DECIMAL NUMERIC PROGRAM DATA (NRf) or NON-DECIMAL NUMERIC PROGRAM DATA (#H, #Q, and #B) format.	Pointer to **int**.
i	Decimal, octal, or hexadecimal integer.	Pointer to **int**.
u	Unsigned decimal integer	Pointer to **unsigned int**.
o	Octal integer	Pointer to **int**.
x,X	Hexadecimal integer	Pointer to **int**.
e, E, f, g, G	Floating-point value in either IEEE 488.2 DECIMAL NUMERIC PROGRAM DATA (NRf) or NON-DECIMAL NUMERIC PROGRAM DATA (#H, #Q, and #B) format. The value consists of an optional sign (+ or –), a series of one or more decimal digits containing a decimal point, and an optional exponent (**e** or **E**) followed by an optionally signed integer value.	Pointer to **float**.
c	Character. White-space characters that are ordinarily skipped are read when **c** is specified. To read the next non-white-space character use "**%1s**".	Pointer to a **char**.

s	Null-terminated string where leading white-space characters are ignored and all ordinary characters are read until a white-space character is read. *Flags* can contain either an integer or #. When *flags* is an integer, it specifies the maximum string size. The string size must be large enough to hold the characters and a NULL character. When *flags* contains a #, it specifies that the next argument parameter contains a pointer to the maximum size of the string. If maximum number of characters is read before a white-space character, all additional characters are read and discarded until a white-space character is found.	Pointer to a string.
S	Null-terminated string that conforms to IEEE 488.2 STRING RESPONSE DATA. Leading white-space before the required double quote is ignored, then all characters up to the next double quote are read. Two double quote characters are converted to a single quote. The beginning and ending double quotes are not inserted into the argument. *Flags* is the same as **s.**	Pointer to a string.
n	No input read.	Pointer to **int**, into which is stored the number of characters read so far.

b	Data block that conforms to IEEE 488.2 ARBITRARY BLOCK PROGRAM DATA. *Width* must contain a long that specifies the number of elements in the data block or an #. If *width* contains #, two argument parameters are used. The first contains a pointer to a long containing the size of the second argument parameter, which is a pointer to the array.	Pointer to data block.
t	END indicator terminated string. *Flags* is the same as **s**. The stored string is null terminate. If the maximum number of characters is read before an END indicator is read, all additional characters are read and discarded until an END indicator is read.	Pointer to a string.

For 32-bit systems, **int** and **long** are the same.

To read characters not delimited by white-space characters, a set of characters in brackets ([ ]) can be substituted for the **s** type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. Use a caret (^) to reverse the effect.

To store a string without storing the terminating null character (**\0**), use the specification **%nc**, where n is a decimal integer specifying the number of characters to store.

A formatted input function can stop converting a field for a variety of reasons:

- The specified width has been reached.
- The next character cannot be converted as specified.
- The next character conflicts with a character in the format specification string that it is supposed to match.
- The next character fails to appear in a given character set.

After reading stops, the next input field is considered to begin at the first unread character. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent operations.

An input field is defined as all characters up to the first white-space character, or up to the first character that can not be converted as specified, or until *width* is reached.

Width Field

The *width* field is an optional field containing a positive decimal integer that controls the maximum number of characters read. No more than *width* characters are converted and stored at the corresponding argument parameter. Fewer than *width* characters may be read if a white-space character or a character that can not be converted is read before *width* is reached.

Array_Size Field

The *array_size* field is optional and contains a non-negative decimal integer, preceded by a comma, that specifies the number of elements in a numeric array.

If *array_size* is a pound sign (#), the next argument parameter in the argument list is treated as a pointer to an *int* representing the array size.

An *array_size* field is only valid for integers (**d**) and floating point (**f**) types.

The values received must be a comma-separated list (white space is ignored).

Length Field

The *length* field is optional and is an argument modifier. The following defines the valid *length* entries:

Character	Description
h	Use with *types* **d**, **i**, **o**, **x**, and **X** to specify that the argument is a **short int** or with *type* **u** to specify a **short unsigned int**. If used with *type* **p** , it indicates a 16-bit pointer (offset only).

l Use with *types* **d**, **i**, **o**, **x**, and **X** to specify that the argument is a **long int**. Use with the *type* **u** to specify a **long unsigned int**. Use with *types* **e**, **E**, **f**, **g**, and **G** to specify a **double** rather than a **float**. If used with *type* **p**, it indicates a 32-bit pointer.

Use with *type* **b** to specify that the argument is a pointer to an array of **long unsigned ints** (32-bits). The data block is sent as an array of 32-bit words. *Flags* must contain an integer or #. When *flags* contains a long, it specifies the maximum number of longwords to read. When *flags* contains #, it specifies that the next argument parameters contains a pointer to a long containing the size of the following argument parameters. For *types* **s**, **S**, **t**, and **B**, *flags* must contain a # or a *width* must be specified for types. The longwords are byte swapped and padded as necessary so that they conform to IEEE 488.2.

L Use with *types* **e**, **E**, **f**, **g**, and **G** to specify a **long double**.

w Use with *type* **b** to specify that the argument is a pointer to an array of **unsigned shorts** (16-bits). The data block is sent as an array of 16-bit words. *Flags* must contain a long or #. When *flags* contains a long, it specifies the maximum number of words to read. When *flags* contains #, it specifies that the next argument parameters contains a pointer to a long containing the size of the following argument parameters. The words are byte swapped and padded as necessary so they conform to IEEE 488.2.

z Use with *type* **b** to specify that the argument is a pointer to an array of **floats**. The data block is read as an array of 32-bit IEEE-754 floating point numbers. *Flags* must contain a long or #. When *flags* contains a long, it specifies the maximum number of **floats** to read. When *flags* contains #, it specifies that the next argument parameter contains a pointer to a long containing the size of the following argument parameter.

Z Use with *type* **b** to specify that the argument is a pointer to an array of **doubles**. The data block is read as an array of 64-bit IEEE-754 floating point numbers. *Flags* must contain an integer or #. When *flags* contains an integer, it specifies the maximum number of **doubles** to read. When *flags* contains #, it specifies that the next argument parameter contains a pointer to a long containing the size of the following argument parameter.

NOTES

5. SICL Errors

This chapter contains a listing of return values that are generated by SICL function calls.

When you install a default SICL error handler such as **I_ERROR_EXIT** or **I_ERROR_NOEXIT** with an **ionerror** call, a SICL internal error message will be logged. To view these messages, start the Message Viewer utility in the HP SICL group. You may want to "iconify" the utility during execution of your program. However, you must always start it before you execute a program in order for messages to be logged.

You may also use **ionerror** to install your own custom error handler. You error handler can call **igeterrstr** with the given error code, and the corresponding error message string will be returned.

5.1 SICL Errors

Accompanying each error below is a description of the error.

Error Type	NAME	DESCRIPTION
Good Complete	I_ERR_NOERROR	No error - successful completion.
Invalid parameters	I_ERR_PARAM	Invalid parameter.
	I_ERR_BADID	The specified INST *id* is invalid.
	I_ERR_BADFMT	Invalid format for **iprintf** or **iscanf**.
	I_ERR_BADMAP	Invalid map request.
Open errors	I_ERR_SYNTAX	Syntax error occurred parsing address.
	I_ERR_BADADDR	Bad address (device/interface doesn't exist).
	I_ERR_SYMNAME	Invalid symbolic name given.
	I_ERR_INVLADDR	Invalid address given.
Unsupported or unallowed operations	I_ERR_NOPERM	Permission denied. Access rights violated.
	I_ERR_NOTSUPP	Not supported operation. The request is not valid on this session or interface type.
	I_ERR_NOTIMPL	Not supported on implementation. The request is valid, just not supported on this implementation.
Unavailable errors	I_ERR_NOINTF	Interface is not active.
	I_ERR_NODEV	Device is not active or available, or doesn't exist.
	I_ERR_NOCMDR	Commander is not active or available, or doesn't exist.
	I_ERR_NOCONN	No connection to remote.
Data I/O errors	I_ERR_IO	Generic I/O error.
	I_ERR_DATA	Data integrity violation (CRC, Checksum, etc.)
	I_ERR_TIMEOUT	Timeout occurred.

Locking Errors	**I_ERR_LOCKED**	Locked by another user (see **isetlockwait** intrinsic).
	I_ERR_NOLOCK	An **iunlock** was specified when device wasn't locked.
OS and Resource errors	**I_ERR_OS**	Generic O.S. error.
	I_ERR_NORSRC	Out of system resources.
	I_ERR_BUSY	Interface is in use by a non-SICL entity.
	I_ERR_OVERFLOW	Arithmetic overflow.
	I_ERR_BADCONFIG	An invalid configuration was identified.
Miscellaneous errors	**I_ERR_ABORTED**	SICL call aborted by **iabort** or other external means.

NOTES

5

6. Support and Service

6.1 In North America

6.1.1 Technical Support

RadiSys maintains a technical support phone line at (503) 646-1800 that is staffed weekdays (except holidays) between 8 AM and 5 PM Pacific time. If you have a problem outside these hours, you can leave a message on voice-mail using the same phone number. You can also request help via electronic mail or by FAX addressed to RadiSys Technical Support. The RadiSys FAX number is (503) 646-1850. The RadiSys E-mail address on the Internet is *support@radisys.com*. If you are sending E-mail or a FAX, please include information on both the hardware and software being used and a detailed description of the problem, specifically how the problem can be reproduced. We will respond by E-mail, phone or FAX by the next business day.

Technical Support Services are designed for customers who have purchased their products from RadiSys or a sales representative. If your RadiSys product is part of a piece of OEM equipment, or was integrated by someone else as part of a system, support will be better provided by the OEM or system vendor that did the integration and understands the final product and environment.

6.1.2 Bulletin Board

RadiSys operates an electronic bulletin board (BBS) 24 hours per day to provide access to the latest drivers, software updates and other information. The bulletin board is not monitored regularly, so if you need a fast response please use the telephone or FAX numbers listed above.

The BBS operates at up to 14400 baud. Connect using standard settings of eight data bits, no parity, and one stop bit (8, N, 1). The telephone number is (503) 646-8290.

6.2 Other Countries

Contact the sales organization from which you purchased your RadiSys product for service and support.

Index

NOTES