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

EPC and RadiSys are registered trademarks and EPConnect is a trademark of RadiSys Corporation.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation and Windows is a trademark of Microsoft Corporation.

IBM and PC/AT are trademarks of International Business Machines Corporation.

May 1994

Copyright © 1994 by RadiSys Corporation

All rights reserved.

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

1. Introducing SICL for Windows 1-1

1.1 How This Manual is Organized 1-2
1.2 What is SICL For Windows? 1-2
1.2.1 Conformance to the SICL Standard 1-3
1.2.2 Portability 1-3
1.2.3 Transparency 1-3
1.2.4 SICL for Windows Architecture 1-4
1.2.5 SURM 1-5
1.2.6 SICL 1-5
1.2.7 SICL VXI and GPIB Interface Drivers 1-5
1.2.8 OLRM 1-5
1.2.9 Bus Management Library and BusManager VxD 1-6
1.3 Programming, Compiling and Linking 1-6
1.3.1 SICL.H Header File 1-6
1.3.2 SRQ Interrupt and Error Handler Declarations 1-7
1.3.3 Compiling and Linking SICL for Windows Applications 1-8
1.3.4 Considerations when using Visual Basic 1-9
1.4 What to do Next 1-10

2. Function Descriptions 2-1

2.1 Functions by Category 2-1
2.1.1 Session Handling 2-2
2.1.2 Unformatted I/O 2-3
2.1.3 Formatted I/O 2-4
2.1.4 Asynchronous Event Control 2-6
2.1.5 Memory Mapping 2-6
2.1.6 Memory Mapped I/O 2-7
2.1.7 Error Handling 2-8
2.1.8 Locking 2-9
Actions of Locking 2-10
Locking in a multi-user environment 2-11
2.1.9 Timeouts 2-11
2.1.10 Device and Interface Control 2-12
2.1.11 VXI Interface 2-13
2.1.12 GPIB Interface 2-14
2.1.13 Version Control 2-15
2.1.14 Microsoft Windows Control 2-15
2.2 Functions by Name 2-15
iabort 2-16
ibblockcopy 2-17

ibeswap 2-20
ibpeek 2-21
ibpoke 2-23
ibpopfifo 2-25
ibpushfifo 2-27
icauseerr 2-29
iclear 2-30
iclose 2-32
icmd 2-34
iflush 2-36
ifread 2-39
ifwrite 2-43
igetaddr 2-46
igetdata 2-48
igetdevaddr 2-51
igeterrno 2-53
igeterrstr 2-54
igetintfsess 2-55
igetintftype 2-57
igetlockwait 2-59
igetlu 2-61
igetluinfo 2-62
igetlulist 2-63
igetonerror 2-65
igetonintr 2-66
igetonsrq 2-67
igetsesstype 2-68
igettermchr 2-71
igettimeout 2-73
igpibatnctl 2-75
igpibbusaddr 2-77
igpibbusstatus 2-78
igpibgett1delay 2-82
igpibllo 2-83
igpibpassctl 2-85
igpibppoll 2-88
igpibppollconfig 2-91
igpibppollresp 2-92
igpibrenctl 2-93
igpibsendcmd 2-95
igpibsett1delay 2-98

ihint 2-99
iintroff 2-101
iintron 2-102
ilblockcopy 2-103
ileswap 2-107
ilocal 2-108
ilock 2-110
ilpeek 2-113
ilpoke 2-116
ilpopfifo 2-117
ilpushfifo 2-120
imap 2-123
imapinfo 2-127
ionerror 2-130
ionintr 2-133
ionsrq 2-138
iopen 2-142
iprintf 2-147
i 2-150
iread 2-153
ireadstb 2-157
iremote 2-159
iscanf 2-161
isetbuf 2-165
isetdata 2-169
isetintr 2-170
isetlockwait 2-174
isetstb 2-175
isetubuf 2-176
isprintf 2-181
isscanf 2-182
isvprintf 2-184
isvscanf 2-186
iswap 2-188
itermchr 2-191
itimeout 2-192
itrigger 2-194
iunlock 2-196
iunmap 2-197
iversion 2-198
ivprintf 2-199
i 2-202

ivscanf 2-205
ivxibusstatus 2-210
ivxigettrigroute 2-214
ivxirminfo 2-218
ivxiservants 2-221
ivxitrigoff 2-223
ivxitrigon 2-225
ivxitrigroute 2-229
ivxiwaitnormop 2-233
ivxiws 2-235
iwaithdlr 2-237
iwblockcopy 2-238
iwpeek 2-241
iwpoke 2-244
iwpopfifo 2-245
iwpushfifo 2-248
iwrite 2-251
ixtrig 2-254
_siclcleanup 2-257

3. Advanced Topics 3-1
- 3.1 Byte Ordering and Data Representation 3-1
  - 3.1.1 Byte Swapping Functions 3-2
  - 3.1.2 Correcting Data Structure Byte Ordering 3-3
- 3.2 Handler Operations Under Windows 3-4
- 3.3 SRQ Handler Execution 3-4
- 3.4 Interrupt Handler Execution 3-5
- 3.5 Error Handler Execution 3-6
- 3.6 VXI TTL Trigger Interrupts on an EPC-7 3-6
- 3.7 Common SICL problems with Windows 3.1 3-10
- 3.8 Avoiding Nested I/O 3-11
- 3.9 Using _siclcleanup Before Exiting 3-11

4. I/O Formatting 4-1
- 4.1 Output Format 4-1
- 4.2 Input Format 4-10

5. SICL Errors 5-1
- 5.1 SICL Errors 5-2

6. Support and Service 6-1
- 6.1 In North America 6-1
  - 6.1.1 Technical Support 6-1
- 6.1.2 Bulletin Board 6-1

6.2 Other Countries 6-2

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 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.

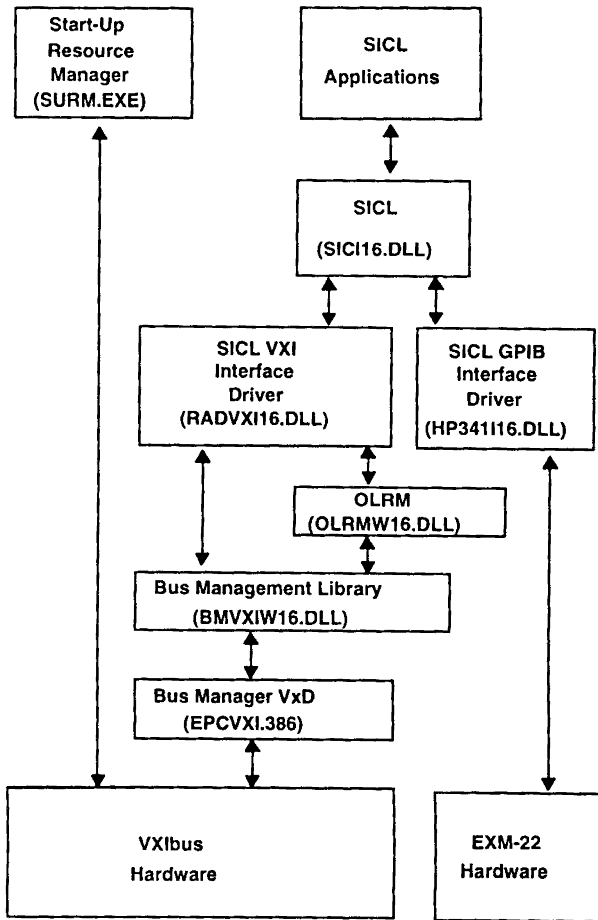

Figure 1-1. SICL for Windows Software Architecture.

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

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.

2.1.3 Formatted I/O

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	**igpibsettldelay**	**itrigger**
ifread	**ilocal**	**ivprintf**
ifwrite	**ilock**	**ivpromptf**
igpibatnctl	**imap**	**ivscanf**
igpibgettldelay	**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**

2

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**

2

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);
}
```

2

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);
}
```

2

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;
          }
     }
```

2

```
    /* 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**

2

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**

2

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"
};
```

2

```
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)
{
```

2

```
    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);
}
```

2

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**

2

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)
```

2

```
    {
        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**

2

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**

2

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);
```

2

```
    }
    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	**igpibsettldelay**	**isetubuf**
ifread	**ilocal**	**itrigger**
ifwrite	**ilock**	**ivprintf**
igpibatnctl	**imap**	**ivpromptf**
igpibgettldelay	**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;
```

2

```
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);
}
```

2

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**

2

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);
}
```

2

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);
}
```

2

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. */
```

```
    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);
}
```

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.

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**

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**

2

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**

2

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);
```

2

```
    }
    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**.

2

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

2

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**

2

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**

2

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**.

2

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	**igpibsettldelay**	**itrigger**
ifread	**ilocal**	**ivprintf**
ifwrite	**ilock**	**ivpromptf**
igpibatnctl	**imap**	**ivscanf**
igpibgettldelay	**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);
}
```

2

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.

Writeformat 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 *arguments* 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)
```

2

```
{
     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;
```

2

```
#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);
}
```

2

ivxirminfo

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;
```

2

```
    /* 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**

2

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).

2

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

3

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

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

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

32-bit systems, 4-14

A

active controller status, passing, 2-85
address space
 deleting, 2-197, 2-198
 getting, 2-127
 mapping, 2-123
address string
 device session, 2-143
 interface session, 2-142
advanced application programming topics, 3-1
application data structure, 2-48, 2-169
application development
 compiling, paths, 1-8
 portability, 1-3
architecture, EPConnect software, 1-4
array_size field, 4-9, 4-15
assert, interface triggers, 2-254, 2-257
asynchronous event control, 2-6
asynchronous event processing, 3-4, 3-5, 3-6
 enabling, 3-5
ATN line, controlling, 2-75

B

buffers. see I/O buffers
Bus Manager, 1-6
 foundation of EPConnect, 1-6
 portability issues, 1-6
byte
 controller's status, setting, 2-175
byte order (EPC), 2-20
Byte ordering, 3-1
byte swapping, 2-107
byte swapping functions, 3-2
byte-swapping, 3-3
 with greater data transfer widths, 3-3

C

command bytes, writing, 2-95
compiler errors, 1-7
compiling under C++, 1-7
compiling, applications, 1-8
configuring, parallel poll response, 2-91
constants
 interface type, 2-57
controller, set status byte, 2-175
copying
 ilblockcopy, 2-103
 iwblockcopy, 2-238
 long word, 2-103
 word, 2-238
creating buffers, 2-36

D

data structure
 application, 2-48, 2-169
 byte ordering, 3-3
 session, getting, 2-169
data widths, 3-1
deassert, interface triggers, 2-254, 2-257
defining, trigger routes, 2-214
description
 SICL header file, 1-7

device
address, getting, 2-51
clearing, 2-30
error, getting, 2-53
formatted I/O, reading, 2-150, 2-161, 2-182, 2-186, 2-202, 2-205
formatted I/O, writing, 2-147, 2-150, 2-181, 2-184, 2-199, 2-202
information, 2-218
locking, 2-110
putting in local mode, 2-108
putting in remote mode, 2-159
reading data, 2-39, 2-153
reading status byte, 2-157
send word serial command, 2-235
session, opening, 2-142
SRQ handler, installing, 2-138
trigger, sending, 2-194
unlocking, 2-196
writing data, 2-43, 2-251
device and interface control functions, 2-12
device session
address string, 2-143

E

ECL, triggers, 2-223, 2-225
END indicator, 4-1
EPC, 1-2
EPC-7, 3-6
TTL trigger latch register, 3-7
EPC-7 VXI interface sessions, 2-171
EPC-7, VXI trigger interrupt operation, 2-134
EPConnect, software, 1-4
error generation, when locked, 2-174
error handler
execution, 3-6
igetonerror, 2-65
ionerror, 2-130
query, 2-65
error handlers
installing, 2-130
error handling functions, 2-8
error number
setting, 2-29
error string, getting, 2-54, 2-55
event processing
disabling, 2-101
enabling, 2-102
extraneous TTL trigger interrupts, 3-8

F

fifo
long word copying to, 2-120
long word, copying, 2-117
word copying to, 2-248
word, copying, 2-245
flags field, 4-5
format specification fields, 4-2, 4-11
Format specifications, 4-2
formatted I/O
buffer flushing, 2-36
iflush, 2-36
isetbuf, 2-165
isetubuf, 2-176
reading, 2-150, 2-202
setting buffer size, 2-165, 2-176
writing, 2-150, 2-202
formatted I/O functions, 2-4, 4-1
formatted input string, 4-10
formatted output string, 4-1
formatting and conversion operations, 2-4

G

getting started, 1-10
GNU CC for LynxOS, 1-2
GPIB
active controller status, passing, 2-85
ATN line, controlling, 2-75

LLO mode, 2-83
parallel poll response, configuring, 2-91
parallel poll, execute, 2-82, 2-88, 2-92, 2-98
REN line, controlling, 2-93
status, getting, 2-77, 2-78
write command bytes, 2-95
GPIB instruments, 1-2
GPIB interface functions, 2-14, 2-15
GPIB interface sessions, 2-44
Group Execute Trigger (GET) command, 2-254

H

handlers
error, 2-130
error, execution, 3-6
interrupt, 2-133
interrupt execution, 3-5
SRQ, 2-138
SRQ, execution, 3-4
hang, when locked, 2-174
header file
description, 1-7

I

I/O buffers
creating, 2-165, 2-176
flushing, 2-36
ibblockcopy (function), 2-16, 2-17
ibeswap (function), 2-20
ibpeek (function), 2-21
ibpoke (function), 2-23
ibpopfifo (function), 2-25
ibpushfifo (function), 2-27
icauseerr (function), 2-29
iclear (function), 2-30
iclose (function), 2-32
iflush (function), 2-36
ifread (function), 2-39
ifwrite (function), 2-43
igetaddr (function), 2-46
igetdata (function), 2-48
igetdevaddr (function), 2-51
igeterrno (function), 2-53
igeterrstr (function), 2-54
igetintfsess (function), 2-55
igetintftype (function), 2-57
igetlockwait (function), 2-59
igetlu (function), 2-61
igetluinfo (function), 2-62
igetlulist (function), 2-63
igetonerror (function), 2-65
igetonintr (function), 2-66
igetonsrq (function), 2-67
igetsesstype (function), 2-68
igettermchr (function), 2-71
igettimeout (function), 2-73
igpibatnctl (function), 2-75
igpibbusstatus (function), 2-77, 2-78
igpibllo (function), 2-83
igpibpassctl (function), 2-85
igpibppoll (function), 2-82, 2-88, 2-92, 2-98
igpibppollconfig (function), 2-91
igpibrenctl (function), 2-93
igpibsendcmd (function), 2-95
ihint (function), 2-99
iintroff (function), 2-101
iintron, 3-5
iintron (function), 2-102
ilblockcopy (function), 2-103
ileswap (function), 2-107
ilocal (function), 2-108
ilock (function), 2-110
ilpeek (function), 2-113
ilpoke (function), 2-116
ilpopfifo (function), 2-117
ilpushfifo (function), 2-120
imap (function), 2-123
imapinfo (function), 2-127

input format strings, 4-1
installing
    error handler, 2-130
    SRQ handler, 2-138
Intel and Motorola byte orders, 3-2
Intel, byte ordering, 3-1
interface
    address space, getting, 2-127
    clearing, 2-30
    constants, type, 2-57
    formatted I/O, reading, 2-150, 2-161, 2-186, 2-202, 2-205
    formatted I/O, writing, 2-147, 2-150, 2-181, 2-184, 2-199, 2-202
    locking, 2-110
    reading data, 2-39, 2-153
    session address string, 2-142
    session type, getting, 2-57
    session, opening, 2-142
    trigger, sending, 2-194
    triggers, assert or deassert, 2-254, 2-257
    unlocking, 2-196
    writing data, 2-43, 2-251
interrupt
    disabling event processing, 2-101
    enabling, 2-170
    enabling event processing, 2-102
    handler execution, 3-5
    types, valid, 2-170
    wait for execution, 2-237
interrupt handler
    getting, 2-66
    installation, 3-5
    installing, 2-133
interrupt reception
    enabling, 3-5
    reception, 3-5
interrupts
    disabling, 2-170
    enabling, 2-170
ionerror (function), 2-130
ionintr, 3-5
ionintr (function), 2-133
ionsrq (function), 2-138
iopen (function), 2-142
iprintf (function), 2-147
ipromptf (function), 2-150
iread (function), 2-153
ireadstb (function), 2-157
iremote (function), 2-159
iscanf (function), 2-161
isetbuf (function), 2-165
isetdata (function), 2-169
isetintr, 3-6
isetintr (function), 2-170
isetlockwait (function), 2-174
isetstb (function), 2-175
isetubuf (function), 2-176
isprintf (function), 2-181
isscanf (function), 2-182
isvprintf (function), 2-184
isvscanf (function), 2-186
iswap (function), 2-188
itermchr (function), 2-191
itimeout (function), 2-192
itrigger (function), 2-194
iunlock (function), 2-196
iunmap (function), 2-197, 2-198
ivprintf (function), 2-199
ivpromptf (function), 2-202
ivscanf (function), 2-205
ivxibusstatus (function), 2-210
ivxigettrigroute (function), 2-214
ivxirminfo (function), 2-218
ivxiservants (function), 2-221
ivxitrigoff (function), 2-223
ivxitrigon (function), 2-225
ivxitrigroute (function), 2-229
ivxiwaitnormop (function), 2-233
ivxiws (function), 2-235
iwaithdlr, 3-5

iwaithdlr (function), 2-237
iwblockcopy (function), 2-238
iwpeek (function), 2-241
iwpoke (function), 2-244
iwpopfifo (function), 2-245
iwpushfifo (function), 2-248
iwrite (function), 2-251
ixtrig (function), 2-254, 2-257

L

length field, 4-9, 4-15
local mode, device, put in, 2-108
locking
    device, 2-110
    functions affected, 2-9, 2-111
    generate error, 2-174
    hang, 2-174
    ilock, 2-110
    interface, 2-110
    nesting, 2-110
    suspend, 2-174
locking functions, 2-9
lock-wait flag, getting, 2-59
logical unit, 2-142
logical unit, getting, 2-63
long word
    copying, 2-103
    copying from fifo, 2-117
    copying to fifo, 2-120
    reading, 2-113
    writing, 2-116

M

memory
    mapping, 2-123
    mapping constants, 2-128
    unmapping, 2-197, 2-198
memory mapped I/O functions, 2-7
memory mapping functions, 2-6
memory mapping, delete, 2-197, 2-198
Motorola, byte ordering, 3-1

N

normal operation, VXIbus, 2-233

O

opening, a session, 2-32, 2-142
Optimize I/O performance, 2-99
output format strings, 4-1

P

parallel poll, execute, 2-82, 2-88, 2-92, 2-98
portability, application, 1-3
precision field, 4-7
primary address, 2-143

R

RadiSys EPC controllers, 3-1
read buffer, size setting, 2-165, 2-176
read termination, reasons, 2-40, 2-154
read/write buffer operations, 2-4
read/write buffers, flushing, 2-36
read/write, formatted I/O, 2-150, 2-202
reading
    data with blocking, 2-39, 2-153
    formatted I/O, 2-161, 2-182, 2-186, 2-205
    long word, 2-113
    status byte, 2-157
    word, 2-241
register, 3-6
remote mode, device, put in, 2-159
REN line, controlling, 2-93
required format characters, 4-2
routing, trigger lines, 2-229

S

SAMPLE PROGRAMS, 1-10
secondary address, 2-143
send, word serial command, 2-235
servants, VXIbus, list of, 2-221
service request (SRQ), 2-6

session
    address string, getting, 2-46
    closing, 2-32
    data structure, getting, 2-48, 2-169
    installing interrupt handler, 2-133
    interface type, getting, 2-57
    interrupt handler, getting, 2-66
    lock-wait flag, getting, 2-59
    logical unit, getting, 2-62
    opening, 2-142
    SRQ handler, getting, 2-67
    termination character, getting, 2-71
    timeout, getting, 2-73
    timeout, setting, 2-192
    type
        constants, 2-68
    type, getting, 2-68
    ULA, getting, 2-61
Session handling, 2-2
setting
    error number, 2-29
    termination character, 2-191
SICL, 3-5, 3-7
    16-bit block transfer functions, 3-2
    16-bit peek and poke functions, 3-2
    32-bit block transfer functions, 3-2
    32-bit peek and poke functions, 3-2
    block transfer functions, 3-3
    standard, compliance, 1-3
SICL functions
    capabilities, 1-3
SICL interface
    independence, 1-5
SICL.H, 2-62
    structure, 1-7
siclcleanup, 2-257
size, setting buffer, 2-165, 2-176
software
    EPConnect, 1-4
SRQ
    disabling event processing, 2-101
    enabling event processing, 2-102
    handler execution, 3-4
    handler, getting, 2-67
    handler, installing, 2-138
    wait for execution, 2-237
starting, 1-10
status
    VXIbus, getting, 2-210
status byte
    reading, 2-157
    setting controller's, 2-175
status, GPIB
    constants, 2-78
    getting, 2-77, 2-78
string, error, getting, 2-54, 2-55
SURM
    capabilities, 1-5
    name generation, 2-144
symbolic names, 2-142
    defined, 2-144
symbolic naming, 1-3

T

Technical Support, 6-1
    electronic bulletin board (BBS), 6-1
    E-mail, 6-1
    E-mail address, 6-1
    FAX, 6-1
terminating Windows, 2-257
termination character
    getting, 2-71
    setting, 2-191
timeout
    functions, affected, 2-11, 2-192
    session, getting, 2-73
    session, setting, 2-192
timeout functions, 2-11
trigger
    constants, 2-134
    lines, asserting, 2-225
    lines, deasserting, 2-223

lines, routing, 2-229
route, getting, 2-214
routes, defining, 2-214
sending, 2-194
trigger lines
asserting, 2-225
deasserting, 2-223
triggers
interface, assert or deassert, 2-254, 2-257
TTL trigger latch register, 3-6
TTL, triggers, 2-223, 2-225
type field, 4-3, 4-11
type, session, getting, 2-68
types, interrupt, valid, 2-170

U

ULA, getting, 2-61
unformatted I/O functions, 2-3
unlocking
device, 2-196
interface, 2-196

V

va_list parameter, 2-5
valid escape character sequences, 4-1
VXI interface functions, 2-13
VXI interface session, 3-6
VXI interface sessions, 2-44
VXI TTL trigger interrupts, 3-6
VXIbus
device information, getting, 2-218
memory mapping, 2-123
memory unmapping, 2-197, 2-198
normal operation, wait for, 2-233
route trigger lines, 2-229
send word serial command, 2-235
servants, list of, 2-221
status constants, 2-210
status, getting, 2-210
trigger lines, asserting, 2-225
trigger lines, deasserting, 2-223
trigger routing, getting, 2-214
VXIbus devices, 3-1
VXIbus instruments, 1-2

W

wait, SRQ or interrupt execution, 2-237
width field, 4-7, 4-15
Windows
terminating, 2-257
word
copying, 2-238
copying from fifo, 2-245
copying to fifo, 2-248
reading, 2-241
writing, 2-244
word serial command, send, 2-235
write buffer, setting size, 2-165, 2-176
writing
data with blocking, 2-43, 2-251
ifwrite, 2-43
iwrite, 2-251
long word, 2-116
word, 2-244
writing, formatted I/O, 2-147, 2-181, 2-184, 2-199
iprintf, 2-147
isprintf, 2-181
isvprintf, 2-184
ivprintf, 2-199

NOTES