HEWLETT
PACKARD

Reference
Manual

HP Standard Instrument Control Library

HP Standard Instrument Control Library

Reference Manual

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP makes no warranties of any kind with regard to this document, whether express or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information.

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend.

U.S. Government Restricted Rights. The Software and Documentation have been developed entirely at private expense. They are delivered and licensed as "commercial computer software" as defined in DFARS 252.227-7013 (Oct 1988), DFARS 252.211-7015 (May 1991) or DFARS 252.227-7014 (Jun 1995), as a "commercial item" as defined in FAR 2.101(a), or as "Restricted computer software" as defined in FAR 52.227-19 (Jun 1987) (or any equivalent agency regulation or contract clause), whichever is applicable. You have only those rights provided for such Software and Documentation by the applicable FAR or DFARS clause or the HP standard software agreement for the product involved.

Microsoft, Windows NT, and Windows 95 are U.S. registered trademarks of Microsoft Corporation.

Printing History

This is the fifth edition of the *HP Standard Instrument Control Library Reference Manual.*

April 1994 — First Edition

January 1995 — Second Edition

September 1995 — Third Edition

May 1996 — Fourth Edition

September 1996 — Fifth Edition

Contents

1. Introduction

2. HP SICL Language Reference

1

Introduction

Introduction

Welcome to the *HP Standard Instrument Control Library (SICL) Reference Manual*. This manual provides the function syntax and description of each SICL function.

See the *HP I/O Libraries Installation and Configuration Guide* for HP-UX or Windows for detailed information on SICL installation and configuration.

This first chapter provides an overview of SICL. In addition, this manual contains the following chapters and appendices:

- **Chapter 2 - HP SICL Language Reference** defines all of the supported SICL functions. The SICL functions are provided in alphabetical order to make them easier to look-up and reference.
- **Appendix A - HP SICL Error Codes** lists all the error codes for SICL.
- **Appendix B - HP SICL Function Summary** contains a table of SICL functions with supported features.

This manual also contains a **Glossary** of terms and their definitions, as well as an **Index**.

HP SICL Overview

SICL is a modular instrument communications library that works with a variety of computer architectures, I/O interfaces, and operating systems. Applications written in C/C++ or Visual BASIC using this library can be ported at the source code level from one system to another without, or with very few, changes.

SICL uses standard, commonly used functions to communicate over a wide variety of interfaces. For example, a program written to communicate with a particular instrument on a given interface can also communicate with an equivalent instrument on a different type of interface. This is possible because the commands are independent of the specific communications interface. SICL also provides commands to take advantage of the unique features of each type of interface, thus giving you, the programmer, complete control over I/O communications.

Users

SICL is intended for instrument I/O and C/C++ or Visual BASIC programmers developing applications on either HP-UX version 10.20 or later, Microsoft® Windows 95®, or Microsoft Windows NT® operating system. If you will use the SICL library, you should become familiar with all of the SICL functions that are defined in this manual before writing any applications that use SICL.

Features

SICL has several features that distinguish it from other I/O libraries:

- Portability
- Centralized error handling
- Formatted I/O
- Device, interface, and commander communications sessions

Each of these key features is explained in the following subsections.

Portability

SICL can be considered portable at two levels. The first level is that SICL is a C library that can be called by C, ANSI C, C++, and Visual BASIC applications. As such, it can be ported at the source code level to other systems.

The second level of portability is found in the types of functions that SICL provides. The first type are the core (interface-independent) functions. These functions work on all types of devices and interfaces. The second type are the interface-specific functions. These functions accomplish tasks that are specific to an interface.

If applications are written using only core functions, these applications can be used to talk to equivalent devices on different types of interfaces. Note that programming with interface-specific functions makes a program less portable across interface types.

Centralized Error Handling

In SICL, an application can install an error handling function that will be called whenever a SICL function encounters an error. By eliminating the need to check the value returned from SICL functions, you can considerably reduce the amount of code in an application. Also, I/O errors can be handled in a uniform way.

Formatted I/O

SICL provides formatted I/O that is similar to the C `stdio` mechanism. However, SICL is designed specifically for instrument communication and is optimized for IEEE 488.2 compatible instruments.

Device, Interface, and Commander Sessions

SICL introduces the concept of a **device session**. Typically a device is an instrument, but it could be a computer or another peripheral (such as a printer or plotter). Device sessions insulate the user from interface-specific functions. The user can directly access the device without worrying about the type of interface connecting it. (For example, on GPIB, the user does not have to address a device to listen before sending data to it). This insulation makes applications more robust and portable across interfaces. Device sessions should be used for most applications.

Interface sessions allow the user to access the specified interface in a raw fashion. There is a full set of interface-specific functions for programming features that are specific to a particular interface. This gives the user full control of the activities on the chosen interface. Most of these interface-specific functions are not available with device sessions.

Commander sessions allow the user to communicate with the controller of the system (that is, allowing the computer to act like a device on the interface).

Other Documentation

The following documentation is also helpful when using SICL:

- *HP I/O Libraries Installation and Configuration Guide* explains how to install and configure the HP Virtual Instrument Software Architecture (VISA) library and SICL on HP-UX or Microsoft Windows.
- *HP SICL User's Guide* explains how to program SICL applications on HP-UX or Windows.
- *HP SICL Quick Reference Guide for C Programmers* helps you find SICL function syntax information quickly if you are programming in C/C++.
- *HP SICL Quick Reference Guide for Visual BASIC Programmers* helps you find SICL function syntax information quickly if you are programming in Visual BASIC.
- *HP SICL Online Help* is provided in the form of manual pages (man pages) and online help on HP-UX, and in the form of Windows Help on Microsoft Windows.
- *HP SICL Example Programs* are provided online to help you develop your SICL applications more easily.

The following VXIbus Consortium specifications may also be helpful when using SICL over LAN:

- *TCP/IP Instrument Protocol Specification* - VXI-11, Rev. 1.0
- *TCP/IP-VXIbus Interface Specification* - VXI-11.1, Rev. 1.0
- *TCP/IP-IEEE 488.1 Interface Specification* - VXI-11.2, Rev. 1.0
- *TCP/IP-IEEE 488.2 Instrument Interface Specification* - VXI-11.3, Rev. 1.0

2

HP SICL Language Reference

HP SICL Language Reference

This chapter defines all of the supported SICL functions. The functions are listed in alphabetical order to make them easier for you to look-up and reference. In this chapter, the entry for each SICL function includes:

- C syntax and Visual BASIC syntax (if the function is supported on Visual BASIC).
- Complete description.
- Return value(s).
- Related SICL functions that you may want to see also.

NOTE

This edition of this manual supports and shows the syntax structure for programming SICL applications in Visual BASIC version 4.0 or later.

If you have SICL applications written in an earlier Visual BASIC version than version 4.0 (for example, version 3.0), you can easily port your SICL applications to Visual BASIC version 4.0. See Appendix C, "Porting to Visual BASIC 4.0," in the *HP SICL User's Guide for Windows*.

Along with this chapter, you may also want to refer to:

- Appendix A, which lists all the SICL error codes.
- Appendix B, which summarizes the supported features of each core and interface-specific SICL function.

Session Identifiers

SICL uses session identifiers to refer to specific SICL sessions. The **iopen** function will create a SICL session and return a session identifier to you. A session identifier is needed for most SICL functions.

Note that for the C and C++ languages, SICL defines the variable type `INST`. C and C++ programs should declare session identifiers to be of type `INST`. For example:

```
INST id;
```

Visual BASIC programs should declare session identifiers to be of type Integer. For example:

```
DIM id As Integer
```

Device, Interface, and Commander Sessions

Some SICL functions are supported on device sessions, some on interface sessions, some on commander sessions, and some on all three. The listing for each function in this chapter indicates which sessions support that function.

Functions Affected by Locks

In addition, some functions are affected by locks (refer to the **ilock** function). This means that if the device or interface that the session refers to is locked by another process, this function will block and wait for the device or interface to be unlocked before it will succeed, or it will return immediately with the error `I_ERR_LOCKED`. Refer to the **isetlockwait** function.

Functions Affected by Timeouts

Likewise, some functions are affected by timeouts (refer to the **itimeout** function). This means that if the device or interface that the session refers to is currently busy, this function will wait for the amount of time specified by **itimeout** to succeed. If it cannot, it will return the error `I_ERR_TIMEOUT`.

Per-Process Functions

Functions that do not support sessions and are not affected by **ilock** or **itimeout** are *per-process* functions. The SICL function **ionerror** is an example of this. The **ionerror** function installs an error handler for the process. As such, it handles errors for all sessions in the process regardless of the type of session.

IBLOCKCOPY

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ibblockcopy (id, src, dest, cnt);
INST id;
unsigned char  *src;
unsigned char  *dest;
unsigned long cnt;

int iwblockcopy (id, src, dest, cnt, swap);
INST id;
unsigned char  *src;
unsigned char  *dest;
unsigned long cnt;
int swap;

int ilblockcopy (id, src, dest, cnt, swap);
INST id;
unsigned char  *src;
unsigned char  *dest;
unsigned long cnt;
int swap;
```

Visual BASIC Syntax

```
Function ibblockcopy
(ByVal id As Integer, ByVal src As Long,
 ByVal dest As Long, ByVal cnt As Long)

Function iwblockcopy
(ByVal id As Integer, ByVal src As Long,
 ByVal dest As Long, ByVal cnt As Long,
 ByVal swap As Integer)

Function ilblockcopy
(ByVal id As Integer, ByVal src As Long,
 ByVal dest As Long, ByVal cnt As Long,
 ByVal swap As Integer)
```

NOTE

Not supported over LAN.

Description

The three forms of **iblockcopy** assume three different types of data: byte, word, and long word (8 bit, 16 bit, and 32 bit). The **iblockcopy** functions copy data from memory on one device to memory on another device. They can transfer entire blocks of data.

The *id* parameter, although specified, is normally ignored except to determine an interface-specific transfer mechanism such as DMA. To prevent using an interface-specific mechanism, pass a zero (0) for this parameter. The *src* argument is the starting memory address for the source data. The *dest* argument is the starting memory address for the destination data. The *cnt* argument is the number of transfers (bytes, words, or long words) to perform. The *swap* argument is the byte swapping flag. If *swap* is zero, no swapping occurs. If *swap* is non-zero the function swaps bytes (if necessary) to change byte ordering from the internal format of the controller to/from the VXI (big-endian) byte ordering.

NOTE

If a bus error occurs, unexpected results may occur.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IPEEK", "IPOKE", "IPOPFIFO", "IPUSHFIFO"

ICAUSEERR

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

void icauseerr (id, errcode, flag);
INST id;
int errcode;
int flag;
```

Visual BASIC Syntax

```
Sub icauseerr
(ByVal id As Integer, ByVal errcode As Integer,
 ByVal flag As Integer)
```

Description

Occasionally it is necessary for an application to simulate a SICL error. The `icauseerr` function performs that function. This function causes SICL to act as if the error specified by *errcode* (see Appendix A for a list of errors) has occurred on the session specified by *id*. If *flag* is 1, the error handler associated with this process is called (if present); otherwise it is not.

On operating systems that support multiple threads, the error is per-thread, and the error handler will be called in the context of this thread.

See Also

"IONERROR", "IGETONERROR", "IGETERRNO", "IGETERRSTR"

ICLEAR

Supported sessions: .... device, interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iclear (id);
INST id;
```

Visual BASIC Syntax

```
Function iclear
(ByVal id As Integer)
```

Description

Use the `iclear` function to clear a device or interface. If *id* refers to a device session, this function sends a *device clear* command. If *id* refers to an interface, this function sends an *interface clear* command.

The `iclear` function also discards the data in both the read and the write formatted I/O buffers. This discard is equivalent to performing the following `iflush` call (in addition to the device or interface clear function):

```
iflush (id, I_BUF_DISCARD_READ | I_BUF_DISCARD_WRITE);
```

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IFLUSH", and the interface-specific chapter in the *HP SICL User's Guide* for details of implementation.

ICLOSE

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int iclose (id);
INST id;
```

Visual BASIC Syntax

```
Function iclose
(ByVal id As Integer)
```

Description

Once you no longer need a session, close it using the `iclose` function. This function closes a SICL session. After calling this function, the value in the *id* parameter is no longer a valid session identifier and cannot be used again.

> **NOTE**
>
> Do not call `iclose` from an SRQ or interrupt handler, because it may cause unpredictable behavior.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IOPEN"

IFLUSH

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iflush (id, mask);
INST id;
int mask;
```

Visual BASIC Syntax

```
Function iflush
(ByVal id As Integer, ByVal mask As Integer)
```

Description

This function is used to manually flush the read and/or write buffers used by formatted I/O. The *mask* may be one or a combination of the following flags:

`I_BUF_READ`	Indicates the read buffer (`iscanf`). If data is present, it will be discarded until the end of data (that is, if the END indicator is not currently in the buffer, reads will be performed until it is read).
`I_BUF_WRITE`	Indicates the write buffer (`iprintf`). If data is present, it will be discarded.
`I_BUF_WRITE_END`	Flushes the write buffer of formatted I/O operations and sets the END indicator on the last byte (for example, sets EOI on HP-IB).
`I_BUF_DISCARD_READ`	Discards the read buffer (does not perform I/O to the device).
`I_BUF_DISCARD_WRITE`	Discards the write buffer (does not perform I/O to the device).

The `I_BUF_READ` and `I_BUF_WRITE` flags may be used together (by OR-ing them together), and the `I_BUF_DISCARD_READ` and `I_BUF_DISCARD_WRITE` flags may be used together. Other combinations are invalid.

If **iclear** is called to perform either a device or interface clear, then both the read and the write buffers are discarded. Performing an **iclear** is equivalent to performing the following **iflush** call (in addition to the device or interface clear function):

```
iflush (id, I_BUF_DISCARD_READ | I_BUF_DISCARD_WRITE);
```

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IPRINTF", "ISCANF", "IPROMPTF", "IFWRITE", "IFREAD", "ISETBUF", "ISETUBUF", "ICLEAR"

IFREAD

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ifread (id, buf, bufsize, reason, actualcnt);
INST id;
char  *buf;
unsigned long bufsize;
int  *reason;
unsigned long  *actualcnt;
```

Visual BASIC Syntax

```
Function ifread
(ByVal id As Integer, buf As String,
 ByVal bufsize As Long, reason As Integer,
 actual As Long)
```

Description

This function reads a block of data from the device via the formatted I/O read buffer (the same buffer used by **iscanf**). The *buf* argument is a pointer to the location where the block of data can be stored. The *bufsize* argument is an unsigned long integer containing the size, in bytes, of the buffer specified in *buf*.

The *reason* argument is a pointer to an integer that, upon exiting **ifread**, contains the reason why the read terminated. If the *reason* parameter contains a zero (0), then no termination reason is returned. The *reason* argument is a bit mask, and one or more reasons can be returned.

Values for *reason* include:

`I_TERM_MAXCNT`	*bufsize* characters read.
`I_TERM_END`	*END* indicator received on last character.
`I_TERM_CHR`	Termination character enabled and received.

The *actualcnt* argument is a pointer to an unsigned long integer which, upon exit, contains the actual number of bytes read from the formatted I/O read buffer.

If a termination condition occurs, the **`ifread`** will terminate. However, if there is nothing in the formatted I/O read buffer to terminate the read, then **`ifread`** will read from the device, fill the buffer again, and so forth.

This function obeys the **`itermchr`** termination character, if any, for the specified session. The read terminates only on one of the following conditions:

- It reads *bufsize* number of bytes.
- It finds a byte with the *END* indicator attached.
- It finds the current termination character in the read buffer (set with **`itermchr`**).
- An error occurs.

This function acts identically to the **`iread`** function, except the data is not read directly from the device. Instead the data is read from the formatted I/O read buffer. The advantage to this function over **`iread`** is that it can be intermixed with calls to **`iscanf`**, while **`iread`** may not.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **`Err`** variable is set if an error occurs.

See Also

"IPRINTF", "ISCANF", "IPROMPTF", "IFWRITE", "ISETBUF", "ISETUBUF", "IFLUSH", "ITERMCHR"

IFWRITE

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ifwrite (id, buf, datalen, end, actualcnt);
INST id;
char  *buf;
unsigned long datalen;
int end;
unsigned long  *actualcnt;
```

Visual BASIC Syntax

```
Function ifwrite
(ByVal id As Integer, ByVal buf As String,
 ByVal datalen As Long, ByVal endi As Integer,
 actual As Long)
```

Description

This function is used to send a block of data to the device via the formatted I/O write buffer (the same buffer used by **iprintf**). The *id* argument specifies the session to send the data to when the formatted I/O write buffer is flushed. The *buf* argument is a pointer to the data that is to be sent to the specified interface or device. The *datalen* argument is an unsigned long integer containing the length of the data block in bytes.

If the *end* argument is non-zero, this function will send the *END* indicator with the last byte of the data block. Otherwise, if *end* is set to zero, no *END* indicator will be sent.

The *actualcnt* argument is a pointer to an unsigned long integer. Upon exit, it will contain the actual number of bytes written to the specified device. A NULL pointer can be passed for this argument, and it will be ignored.

This function acts identically to the **iwrite** function, except the data is not written directly to the device. Instead the data is written to the formatted I/O write buffer (the same buffer used by **iprintf**). The formatted I/O write buffer is then flushed to the device at normal times, such as when the buffer is full, or when **iflush** is called. The advantage to this function over **iwrite** is that it can be intermixed with calls to **iprintf**, while **iwrite** cannot.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IPRINTF", "ISCANF", "IPROMPTF", "IFREAD", "ISETBUF", "ISETUBUF", "IFLUSH", "ITERMCHR", "IWRITE", "IREAD"

IGETADDR

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igetaddr (id, addr);
INST id;
char * *addr;
```

> **NOTE**
>
> Not supported on Visual BASIC.

Description

The **igetaddr** function returns a pointer to the address string which was passed to the **iopen** call for the session *id*.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IOPEN"

IGETDATA

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igetdata (id, data);
INST id;
void * *data;
```

NOTE

Not supported on Visual BASIC.

Description

The **igetdata** function retrieves the pointer to the data structure stored by **isetdata** associated with a session.

The **isetdata/igetdata** functions provide a good method of passing data to event handlers, such as error, interrupt, or SRQ handlers.

For example, you could set up a data structure in the main procedure and retrieve the same data structure in a handler routine. You could set a device command string in this structure so that an error handler could re-set the state of the device on errors.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"ISETDATA"

IGETDEVADDR

Supported sessions: .... device

C Syntax

```
#include <sicl.h>

int igetdevaddr (id, prim, sec);
INST id;
int  *prim;
int  *sec;
```

Visual BASIC Syntax

```
Function igetdevaddr
(ByVal id As Integer, prim As Integer,
 sec As Integer)
```

Description

The **igetdevaddr** function returns the device address of the device associated with a given session. This function returns the primary device address in *prim*. The *sec* parameter contains the secondary address of the device or -1 if no secondary address exists.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IOPEN"

IGETERRNO

C Syntax

```
#include <sicl.h>

int igeterrno ();
```

Visual BASIC Syntax

```
Function igeterrno ()
```

Description

All functions (except a few listed below) return a zero if no error occurred (I_ERR_NOERROR), or a non-zero error code if an error occurs (see Appendix A). This value can be used directly. The **igeterrno** function will return the last error that occurred in one of the library functions.

Also, if an error handler is installed, the library calls the error handler when an error occurs.

The following functions do not return the error code in the return value. Instead, they simply indicate whether an error occurred.

```
iopen
iprintf
isprintf
ivprintf
isvprintf
iscanf
isscanf
ivscanf
isvscanf
ipromptf
ivpromptf
imap
i?peek
i?poke
```

For these functions (and any of the other functions), when an error is indicated, read the error code by using the **igeterrno** function, or read the associated error message by using the **igeterrstr** function.

Return Value

This function returns the error code from the last failed SICL call. If a SICL function is completed successfully, this function returns undefined results.

On operating systems that support multiple **threads**, the error number is per-thread. This means that the error number returned is for the last failed SICL function for this **thread** (not necessarily for the session).

See Also

"IONERROR", "IGETONERROR", "IGETERRSTR", "ICAUSEERR"

IGETERRSTR

C Syntax

```
#include <sicl.h>

char  *igeterrstr (errorcode);
int errorcode;
```

Visual BASIC Syntax

```
Function igeterrstr
(ByVal errcode As Integer, myerrstr As String)
```

Description

SICL has a set of defined error messages that correspond to error codes (see Appendix A) that can be generated in SICL functions. To get these error messages from error codes, use the **igeterrstr** function.

Return Value

Pass this function the error code you want, and this function will return a human-readable string.

See Also

"IONERROR", "IGETONERROR", "IGETERRNO", "ICAUSEERR"

IGETGATEWAYTYPE

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igetgatewaytype (id, gwtype);
INST id;
int *gwtype;
```

Visual BASIC Syntax

```
Function igetgatewaytype
(ByVal id As Integer, pdata As Integer) As Integer
```

NOTE

LAN is *not* supported with 16-bit SICL on Windows 95.

Description

The **igetgatewaytype** function returns in *gwtype* the gateway type associated with a given session *id*.

This function returns one of the following values in *gwtype*:

I_INTF_LAN	The session is using a LAN gateway to access the remote interface.
I_INTF_NONE	The session is not using a gateway.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

The "Using HP SICL with LAN" chapter of the *HP SICL User's Guide*.

IGETINTFSESS

Supported sessions: .... device, commander

C Syntax

```
#include <sicl.h>

INST igetintfsess (id);
INST id;
```

Visual BASIC Syntax

```
Function igetintfsess
(ByVal id As Integer)
```

Description

The **igetintfsess** function takes the device session specified by *id* and returns a new session *id* that refers to an interface session associated with the interface that the device is on.

Most SICL applications will take advantage of the benefits of device sessions and not want to bother with interface sessions. Since some functions only work on device sessions and others only work on interface sessions, occasionally it is necessary to perform functions on an interface session, when only a device session is available for use. An example is to perform an interface clear (see **iclear**) from within an SRQ handler (see **ionsrq**).

In addition, multiple calls to **igetintfsess** with the same *id* will return the same interface session each time. This makes this function useful as a filter, taking a device session in and returning an interface session.

SICL will close the interface session when the device or commander session is closed. Therefore, do *not* close this session.

Return Value

If no errors occur, this function returns a valid session *id*; otherwise it returns zero (0).

See Also

"IOPEN"

IGETINTFTYPE

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igetintftype (id, pdata);
INST id;
int  *pdata;
```

Visual BASIC Syntax

```
Function igetintftype
(ByVal id As Integer, pdata As Integer)
```

Description

The **igetintftype** function returns a value indicating the type of interface associated with a session. This function returns one of the following values in *pdata*:

I_INTF_GPIB	This session is associated with a GPIB interface.
I_INTF_GPIO	This session is associated with a GPIO interface.
I_INTF_LAN	This session is associated with a LAN interface.
I_INTF_RS232	This session is associated with an RS-232 (Serial) interface.
I_INTF_VXI	This session is associated with a VXI interface.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IOPEN"

IGETLOCKWAIT

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igetlockwait (id, flag);
INST id;
int  *flag;
```

Visual BASIC Syntax

```
Function igetlockwait
(ByVal id As Integer, flag As Integer)
```

Description

To get the current status of the lockwait flag, use the **igetlockwait** function. This function stores a one (1) in the variable pointed to by *flag* if the wait mode is enabled, or a zero (0) if a no-wait, error-producing mode is enabled.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"ILOCK", "IUNLOCK", "ISETLOCKWAIT"

IGETLU

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igetlu (id, lu);
INST id;
int  *lu;
```

Visual BASIC Syntax

```
Function igetlu
(ByVal id As Integer, lu As Integer)
```

Description

The `igetlu` function returns in *lu* the logical unit (interface address) of the device or interface associated with a given session *id*.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IOPEN", "IGETLUINFO"

IGETLUINFO

C Syntax

```
#include <sicl.h>

int igetluinfo (lu, luinfo);
int lu;
struct lu_info  *luinfo;
```

Visual BASIC Syntax

```
Function igetluinfo
(ByVal lu As Integer, result As lu_info)
```

Description

The igetluinfo function is used to get information about the interface associated with the *lu* (logical unit). For C programs, the *lu_info* structure has the following syntax:

```
struct lu_info {
   ...
   long logical_unit;      /* same as value passed into igetluinfo */
   char symname[32];   /* symbolic name assigned to interface */
   char cardname[32];  /* name of interface card */
   long intftype;       /* same value returned by igetintftype */
   ...
};
```

For Visual BASIC programs, the *lu_info* structure has the following syntax:

```
Type lu_info
  logical_unit As Long
  symname As String
  cardname As String
  filler1 As Long
  intftype As Long
          .
          .
          .
End Type
```

Notice that, in a given implementation, the exact structure and contents of the *lu_info* structure is implementation-dependent. The structure can contain any amount of non-standard, implementation-dependent fields. However, the structure must always contain the above fields. If you are programming in C, please refer to the `sicl.h` file to get the exact *lu_info* syntax. If you are programming in Visual BASIC, please refer to the `SICL.BAS` or `SICL4.BAS` file for the exact syntax.

Note that `igetluinfo` will return information for valid local interfaces only, *not* remote interfaces being accessed via LAN.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IOPEN", "IGETLU", "IGETLULIST"

IGETLULIST

C Syntax

```
#include <sicl.h>

int igetlulist (lulist);
int  *  *lulist;
```

Visual BASIC Syntax

```
Function igetlulist
(list() As Integer)
```

Description

The **igetlulist** function stores in *lulist* the logical unit (interface address) of each valid interface configured for SICL. The last element in the list is set to -1.

This function can be used with **igetluinfo** to retrieve information about all local interfaces.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IOPEN", "IGETLUINFO", "IGETLU"

IGETONERROR

C Syntax

```
#include <sicl.h>

int igetonerror (proc);
void ( *  *proc)(INST, int);
```

NOTE

Not supported on Visual BASIC.

NOTE

For WIN16 programs on Microsoft Windows platforms, the variable used to store a handler's address must be declared `(_far _pascal * _far *proc)`.

Description

The `igetonerror` function returns the current error handler setting. This function stores the address of the currently installed error handler into the variable pointed to by *proc*. If no error handler exists, it will store a zero (0).

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IONERROR", "IGETERRNO", "IGETERRSTR", "ICAUSEERR"

IGETONINTR

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igetonintr (id, proc);
INST id;
void ( *  *proc)(INST, long, long);
```

NOTE

Not supported on Visual BASIC.

NOTE

For WIN16 programs on Microsoft Windows platforms, the variable used to store a handler's address must be declared `(_far _pascal * _far *proc)`.

Description

The `igetonintr` function stores the address of the current interrupt handler in *proc*. If no interrupt handler is currently installed, *proc* is set to zero (0).

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IONINTR", "IWAITHDLR", "IINTROFF", "IINTRON"

IGETONSRQ

Supported sessions: .... device, interface

C Syntax

```
#include <sicl.h>

int igetonsrq (id, proc);
INST id;
void ( *  *proc)(INST);
```

NOTE

Not supported on Visual BASIC.

NOTE

For WIN16 programs on Microsoft Windows platforms, the variable used to store a handler's address must be declared `(_far _pascal * _far *proc)`.

Description

The `igetonsrq` function stores the address of the current SRQ handler in *proc*. If there is no SRQ handler installed, *proc* will be set to zero (0).

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IONSRQ", "IWAITHDLR", "IINTROFF", "IINTRON"

IGETSESSTYPE

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igetsesstype (id, pdata);
INST id;
int  *pdata;
```

Visual BASIC Syntax

```
Function igetsesstype
(ByVal id As Integer, pdata As Integer)
```

Description

The **igetsesstype** function returns in *pdata* a value indicating the type of session associated with a given session *id*.

This function returns one of the following values in *pdata*:

I_SESS_CMDR	The session associated with *id* is a commander session.
I_SESS_DEV	The session associated with *id* is a device session.
I_SESS_INTF	The session associated with *id* is an interface session.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IOPEN"

IGETTERMCHR

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igettermchr (id, tchr);
INST id;
int *tchr;
```

Visual BASIC Syntax

```
Function igettermchr
(ByVal id As Integer, tchr As Integer)
```

Description

This function sets the variable referenced by *tchr* to the termination character for the session specified by *id*. If no termination character is enabled for the session, then the variable referenced by *tchr* is set to -1.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"ITERMCHR"

IGETTIMEOUT

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int igettimeout (id, tval);
INST id;
long  *tval;
```

Visual BASIC Syntax

```
Function igettimeout
(ByVal id As Integer, tval As Long)
```

Description

The **igettimeout** function stores the current timeout value in *tval*. If no timeout value has been set, *tval* will be set to zero (0).

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"ITIMEOUT"

IGPIBATNCTL

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibatnctl (id, atnval);
INST id;
int atnval;
```

Visual BASIC Syntax

```
Function igpibatnctl
(ByVal id As Integer, ByVal atnval As Integer)
```

Description

The **igpibatnctl** function controls the state of the ATN (Attention) line. If *atnval* is non-zero, then ATN is set. If *atnval* is 0, then ATN is cleared.

This function is used primarily to allow GPIB devices to communicate without the controller participating. For example, after addressing one device to talk and another to listen, ATN can be cleared with **igpibatnctl** to allow the two devices to transfer data.

NOTE

This function will not work with **iwrite** to send GPIB command data onto the bus. The **iwrite** function on a GPIB interface session always clears the ATN line before sending the buffer. To send GPIB command data, use the **igpibsendcmd** function.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IGPIBSENDCMD", "IGPIBRENCTL", "IWRITE"

IGPIBBUSADDR

Supported sessions: . . . . interface
Affected by functions: . . . . **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibbusaddr (id, busaddr);
INST id;
int busaddr;
```

Visual BASIC Syntax

```
Function igpibbusaddr
(ByVal id As Integer, ByVal busaddr As Integer)
```

Description

This function changes the interface bus address to *busaddr* for the GPIB interface associated with the session *id*.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IGPIBBUSSTATUS"

IGPIBBUSSTATUS

Supported sessions: ....interface

C Syntax

```
#include <sicl.h>

int igpibbusstatus (id, request, result);
INST id;
int request;
int  *result;
```

Visual BASIC Syntax

```
Function igpibbusstatus
(ByVal id As Integer, ByVal request As Integer,
 result As Integer)
```

Description

The **igpibbusstatus** function returns the status of the GPIB interface. This function takes one of the following parameters in the *request* parameter and returns the status in the *result* parameter.

I_GPIB_BUS_REM	Returns a 1 if the interface is in remote mode, 0 otherwise.
I_GPIB_BUS_SRQ	Returns a 1 if the SRQ line is asserted, 0 otherwise.
I_GPIB_BUS_NDAC	Returns a 1 if the NDAC line is asserted, 0 otherwise.
I_GPIB_BUS_SYSCTLR	Returns a 1 if the interface is the system controller, 0 otherwise.
I_GPIB_BUS_ACTCTLR	Returns a 1 if the interface is the active controller, 0 otherwise.
I_GPIB_BUS_TALKER	Returns a 1 if the interface is addressed to talk, 0 otherwise.

`I_GPIB_BUS_LISTENER` Returns a 1 if the interface is addressed to listen, 0 otherwise.

`I_GPIB_BUS_ADDR` Returns the bus address (0-30) of this interface on the GPIB bus.

`I_GPIB_BUS_LINES` Returns the state of various GPIB lines. The result is a bit mask with the following bits being significant (bit 0 is the least-significant-bit):

Bit 0:	1 if SRQ line is asserted.
Bit 1:	1 if NDAC line is asserted.
Bit 2:	1 if ATN line is asserted.
Bit 3:	1 if DAV line is asserted.
Bit 4:	1 if NRFD line is asserted.
Bit 5:	1 if EOI line is asserted.
Bit 6:	1 if IFC line is asserted.
Bit 7:	1 if REN line is asserted.
Bit 8:	1 if in REMote state.
Bit 9:	1 if in LLO (local lockout) mode.
Bit 10:	1 if currently the active controller.
Bit 11:	1 if addressed to talk.
Bit 12:	1 if addressed to listen.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IGPIBPASSCTL", "IGPIBSENDCMD"

IGPIBGETT1DELAY

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibgett1delay (id, delay);
INST id;
int *delay;
```

Visual BASIC Syntax

```
Function igpibgett1delay
(ByVal id As Integer, delay As Integer)
```

Description

This function retrieves the current setting of t1 delay on the GPIB interface associated with session *id*. The value returned is the time of t1 delay in nanoseconds.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IGPIBSETT1DELAY"

IGPIBLLO

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibllo (id);
INST id;
```

Visual BASIC Syntax

```
Function igpibllo
(ByVal id As Integer)
```

Description

The **igpibllo** function puts all GPIB devices on the given bus in local lockout mode. The *id* specifies a GPIB interface session. This function sends the GPIB LLO command to all devices connected to the specified GPIB interface. Local Lockout prevents you from returning to local mode by pressing a device's front panel keys.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IREMOTE", "ILOCAL"

IGPIBPASSCTL

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibpassctl (id, busaddr);
INST id;
int busaddr;
```

Visual BASIC Syntax

```
Function igpibpassctl
(ByVal id As Integer, ByVal busaddr As Integer)
```

Description

The **igpibpassctl** function passes control from this GPIB interface to another GPIB device specified in *busaddr*. The *busaddr* parameter must be between 0 and 30. Note that this will also cause an I_INTR_INTFDEACT interrupt, if enabled.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IONINTR", "ISETINTR"

IGPIBPPOLL

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibppoll (id, result)
INST id;
unsigned int *result;
```

Visual BASIC Syntax

```
Function igpibppoll
(ByVal id As Integer, result As Integer)
```

Description

The **igpibppoll** function performs a parallel poll on the bus and returns the (8-bit) result in the lower byte of *result*.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IGPIBPPOLLCONFIG", "IGPIBPPOLLRESP"

IGPIBPPOLLCONFIG

Supported sessions: .... device, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibppollconfig (id, cval);
INST id;
unsigned int cval;
```

Visual BASIC Syntax

```
Function igpibppollconfig
(ByVal id As Integer, ByVal cval As Integer)
```

Description

For device sessions, the **igpibppollconfig** function enables or disables the parallel poll responses. If *cval* is greater than or equal to 0, then the device specified by *id* is enabled in generating parallel poll responses. In this case, the lower 4 bits of *cval* correspond to:

`bit 3`	Set the sense of the PPOLL response. A 1 in this bit means that an affirmative response means service request. A 0 in this bit means that an affirmative response means no service request.
`bit 2-0`	A value from 0-7 specifying the GPIB line to respond on for PPOLL's.

If *cval* is equal to -1, then the device specified by *id* is disabled from generating parallel poll responses.

For commander sessions, the **igpibppollconfig** function enables/disables parallel poll responses for this device (that is, how we respond when our controller PPOLL's us).

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **`Err`** variable is set if an error occurs.

See Also

"IGPIBPPOLL", "IGPIBPPOLLRESP"

IGPIBPPOLLRESP

Supported sessions: .... commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibppollresp (id, sval);
INST id;
int sval;
```

Visual BASIC Syntax

```
Function igpibppollresp
(ByVal id As Integer, ByVal sval As Integer)
```

Description

The `igpibppollresp` function sets the state of the PPOLL bit (the state of the PPOLL bit when the controller PPOLL's us).

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IGPIBPPOLL", "IGPIBPPOLLCONFIG"

IGPIBRENCTL

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibrenctl (id, ren);
INST id;
int ren;
```

Visual BASIC Syntax

```
Function igpibrenctl
(ByVal id As Integer, ByVal ren As Integer)
```

Description

The **igpibrenctl** function controls the state of the REN (Remote Enable) line. If *ren* is non-zero, then REN is set. If *ren* is 0, then REN is cleared.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IGPIBATNCTL"

IGPIBSENDCMD

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibsendcmd (id, buf, length);
INST id;
char  *buf;
int length;
```

Visual BASIC Syntax

```
Function igpibsendcmd
(ByVal id As Integer, ByVal buf As String,
 ByVal length As Integer)
```

Description

The **igpibsendcmd** function sets the ATN line and then sends bytes to the GPIB interface. This function sends *length* number of bytes from *buf* to the GPIB interface. Note that the **igpibsendcmd** function leaves the ATN line set.

If the interface is not active controller, this function will return an error.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IGPIBATNCTL", "IWRITE"

IGPIBSETT1DELAY

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpibsett1delay (id, delay);
INST id;
int delay;
```

Visual BASIC Syntax

```
Function igpibsett1delay
(ByVal id As Integer, ByVal delay As Integer)
```

Description

This function sets the t1 delay on the GPIB interface associated with session *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 find out the actual value used by calling the `igpibgett1delay` function.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IGPIBGETT1DELAY"

IGPIOCTRL

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpioctrl (id, request, setting);
INST id;
int request;
unsigned long setting;
```

Visual BASIC Syntax

```
Function igpioctrl
(ByVal id As Integer, ByVal request As Integer,
 ByVal setting As Long)
```

NOTE

GPIO is *not* supported over LAN.

Description

The `igpioctrl` function is used to control various lines and modes of the GPIO interface. This function takes *request* and sets the interface to the specified *setting*. The *request* parameter can be one of the following:

`I_GPIO_AUTO_HDSK` If the *setting* parameter is non-zero, then the interface uses auto-handshake mode (the default). This gives the best performance for `iread` and `iwrite` operations. If the *setting* parameter is zero (0), then auto-handshake mode is canceled. This is *required* for programs that implement their own handshake using `I_GPIO_SET_PCTL`.

`I_GPIO_AUX`

The *setting* parameter is a mask containing the state of all auxiliary control lines. A 1 bit asserts the corresponding line; a 0 (zero) bit clears the corresponding line.

When configured in Enhanced Mode, the HP E2074/5 interface has 16 auxiliary control lines. In HP 98622 Compatibility Mode, it has none. Attempting to use `I_GPIO_AUX` in HP 98622 Compatibility Mode results in the error: **`Operation not supported`**.

`I_GPIO_CHK_PSTS`

If the *setting* parameter is non-zero, then the PSTS line is checked before each block of data is transferred. If the *setting* parameter is zero (0), then the PSTS line is ignored during data transfers. If the PSTS line is checked and false, SICL reports the error: **`Device not active or available`**.

`I_GPIO_CTRL`

The *setting* parameter is a mask containing the state of all control lines. A 1 bit asserts the corresponding line; a 0 (zero) bit clears the corresponding line.

The HP E2074/5 interface has two control lines, so only the two least-significant bits have meaning for that interface. These can be represented by the following. All other bits in the *setting* mask are ignored.

`I_GPIO_CTRL_CTL0` The CTL0 line.
`I_GPIO_CTRL_CTL1` The CTL1 line.

`I_GPIO_DATA`

The *setting* parameter is a mask containing the state of all data out lines. A 1 bit asserts the corresponding line; a 0 (zero) bit clears the corresponding line. The HP E2074/5 interface has either 8 or 16 data out lines, depending on the setting specified by **`igpiosetwidth`**.

Note that this function changes the data lines asynchronously, without any type of handshake. It is intended for programs that implement their own handshake explicitly.

`I_GPIO_READ_EOI` If the *setting* parameter is `I_GPIO_EOI_NONE`, then END pattern matching is disabled for read operations. Any other *setting* enables END pattern matching with the specified value. If the current data width is 16 bits, then the lower 16 bits of *setting* are used. If the current data width is 8 bits, then only the lower 8 bits of *setting* are used.

`I_GPIO_SET_PCTL` If the *setting* parameter is non-zero, then a GPIO handshake is initiated by setting the PCTL line. Auto-handshake mode must be disabled to allow explicit control of the PCTL line. Attempting to use `I_GPIO_SET_PCTL` in auto-handshake mode results in the error: `Operation not supported`.

`I_GPIO_PCTL_DELAY` The *setting* parameter selects a PCTL delay value from a set of eight "click stops" numbered 0 through 7. A *setting* of 0 selects 200 ns; a setting of 7 selects 50 μs. For a complete list of delay values, see the *HP E2074/5 GPIO Interface Installation Guide.*

Changes made by this function can remain in the interface hardware after your program ends. On HP-UX and Windows NT, the *setting* remains until the computer is rebooted. On Windows 95, it remains until `hp074i16.dll` is reloaded.

`I_GPIO_POLARITY`

The *setting* parameter determines the logical polarity of various interface lines according to the following bit map. A 0 sets active-low polarity; a 1 sets active-high polarity.

Bit 4	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Data Out	Data In	PSTS	PFLG	PCTL
Value=16	Value=8	Value=4	Value=2	Value=1

Changes made by this function can remain in the interface hardware after your program ends. On HP-UX and Windows NT, the *setting* remains until the computer is rebooted. On Windows 95, it remains until `hp074i16.dll` is reloaded.

`I_GPIO_READ_CLK`

The *setting* parameter determines when the data input registers are latched. It is recommended that you represent *setting* as a hex number. In that representation, the first hex digit corresponds to the upper (most-significant) input byte, and the second hex digit corresponds to the lower input byte. The clocking choices are: 0=Read, 1=Busy, 2=Ready. For an explanation of the data-in clocking, see the *HP E2074/5 GPIO Interface Installation Guide.*

Changes made by this function can remain in the interface hardware after your program ends. On HP-UX and Windows NT, the *setting* remains until the computer is rebooted. On Windows 95, it remains until `hp074i16.dll` is reloaded.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IGPIOSTAT", "IGPIOSETWIDTH"

IGPIOGETWIDTH

Supported sessions: .... interface

C Syntax

```
#include <sicl.h>

int igpiogetwidth (id, width);
INST id;
int *width;
```

Visual BASIC Syntax

```
Function igpiogetwidth
(ByVal id As Integer, width As Integer)
```

NOTE

GPIO is *not* supported over LAN.

Description

The **igpiogetwidth** function returns the current data width (in bits) of a GPIO interface. For the HP E2074/5 interface, *width* will be either 8 or 16.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IGPIOSETWIDTH"

IGPIOSETWIDTH

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int igpiosetwidth (id, width);
INST id;
int width;
```

Visual BASIC Syntax

```
Function igpiosetwidth
(ByVal id As Integer, ByVal width As Integer)
```

NOTE

GPIO is *not* supported over LAN.

Description

The **igpiosetwidth** function is used to set the data width (in bits) of a GPIO interface. For the HP E2074/5 interface, the acceptable values for *width* are 8 and 16.

While in 16-bit width mode, all **iread** calls will return an even number of bytes, and all **iwrite** calls must send an even number of bytes.

16-bit words are placed on the data lines using "big-endian" byte order (most significant bit appears on data line D_15). Data alignment is automatically adjusted for the native byte order of the computer. This is a programming concern only if your program does its own packing of bytes into words. The following program segment is an **iwrite** example. The analogous situation exists for **iread**.

```
/* System automatically handles byte order */
unsigned short words[5];

/* Programmer assumes responsibility for byte order */
unsigned char  bytes[10];

/* Using the GPIO interface in 16-bit mode */
igpiosetwidth(id, 16);

/* This call is platform-independent */
iwrite(id, words, 10, ... );

/* This call is NOT platform-independent */
iwrite(id, bytes, 10, ... );

/* This sequence is platform-independent */
ibeswap(bytes, 10, 2);
iwrite(id, bytes, 10, ... );
```

There are several notable details about GPIO width. The "count" parameters for `iread` and `iwrite` always specify bytes, even when the interface has a 16-bit width. For example, to send 100 *words*, specify 200 *bytes*. The `itermchr` function always specifies an 8-bit character. If a 16-bit width is set, only the lower 8 bits are used when checking for an `itermchr` match.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IGPIOGETWIDTH"

IGPIOSTAT

Supported sessions: .... interface

C Syntax

```
#include <sicl.h>

int igpiostat (id, request, result);
INST id;
int request;
unsigned long *result;
```

Visual BASIC Syntax

```
Function igpiostat
(ByVal id As Integer, ByVal request As Integer,
 ByVal result As Long)
```

> **NOTE**
>
> GPIO is *not* supported over LAN.

Description

The `igpiostat` function is used to determine the current state of various GPIO modes and lines. The *request* parameter can be one of the following:

`I_GPIO_CTRL` The *result* is a mask representing the state of all control lines.

The HP E2074/5 interface has two control lines, so only the two least-significant bits have meaning for that interface. These can be represented by the following. All other bits in the *result* mask are 0 (zero).

`I_GPIO_CTRL_CTL0`	The CTL0 line.
`I_GPIO_CTRL_CTL1`	The CTL1 line.

`I_GPIO_DATA`

The *result* is a mask representing the state of all data input latches. The HP E2074/5 interface has either 8 or 16 data in lines, depending on the setting specified by **`igpiosetwidth`**.

Note that this function reads the data lines asynchronously, without any type of handshake. It is intended for programs that implement their own handshake explicitly.

An **`igpiostat`** function from one process will proceed even if another process has a lock on the interface. Ordinarily, this does not alter or disrupt any hardware states. Reading the data in lines is one exception. A data read causes an “input” indication on the I/O line (pin 20). In rare cases, that change might be unexpected, or undesirable, to the session that owns the lock.

I_GPIO_INFO The *result* is a mask representing the following information about the device and the HP E2074/5 interface:

- I_GPIO_PSTS State of the PSTS line.
- I_GPIO_EIR State of the EIR line.
- I_GPIO_READY True if ready for a handshake. (Exact meaning depends on the current handshake mode.)
- I_GPIO_AUTO_HDSK True if auto-handshake mode is enabled. False if auto-handshake mode is disabled.
- I_GPIO_CHK_PSTS True if the PSTS line is to be checked before each block of data is transferred. False if PSTS is to be ignored during data transfers.
- I_GPIO_ENH_MODE True if the HP E2074/5 data ports are configured in Enhanced (bi-directional) Mode. False if the ports are configured in HP 98622 Compatibility Mode.

I_GPIO_READ_EOI The *result* is the value of the current END pattern being used for read operations. If the *result* is I_GPIO_EOI_NONE, then no END pattern matching is being used. Any other *result* is the value of the END pattern.

I_GPIO_STAT The *result* is a mask representing the state of all status lines.

The HP E2074/5 interface has two status lines, so only the two least-significant bits have meaning for that interface. These can be represented by the following. All other bits in the *result* mask are 0 (zero).

I_GPIO_STAT_STI0 The STI0 line.
I_GPIO_STAT_STI1 The STI1 line.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IGPIOCTRL", "IGPIOSETWIDTH"

IHINT

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int ihint (id, hint);
INST id;
int hint;
```

Visual BASIC Syntax

```
Function ihint
(ByVal id As Integer, ByVal hint As Integer)
```

Description

There are three common ways a driver can implement I/O communications: Direct Memory Access (DMA), Polling (POLL), and Interrupt Driven (INTR). Note, however, that some systems may not implement all of these transfer methods.

The SICL software permits you to "recommend" your preferred method of communication. To do this, use the **ihint** function. The *hint* argument can be one of the following values:

I_HINT_DONTCARE	No preference.
I_HINT_USEDMA	Use DMA if possible and feasible. Otherwise use POLL.
I_HINT_USEPOLL	Use POLL if possible and feasible. Otherwise use DMA or INTR.
I_HINT_USEINTR	Use INTR if possible and feasible. Otherwise use DMA or POLL.
I_HINT_SYSTEM	The driver should use whatever mechanism is best suited for improving overall system performance.
I_HINT_IO	The driver should use whatever mechanism is best suited for improving I/O performance.

Keep the following in mind as you make your suggestions to the driver:

- DMA tends to be very fast at sending data but requires more time to set up a transfer. It is best for sending large amounts of data in a single request. Not all architectures and interfaces support DMA.
- Polling tends to be fast at sending data and has a small set up time. However, if the interface only accepts data at a slow rate, polling wastes a lot of CPU time. Polling is best for sending smaller amounts of data to fast interfaces.
- Interrupt driven transfers tend to be slower than polling. It also has a small set up time. The advantage to interrupts is that the CPU can perform other functions while waiting for data transfers to complete. This mechanism is best for sending small to medium amounts of data to slow interfaces or interfaces with an inconsistent speed.

NOTE

The parameter passed in `ihint` is only a suggestion to the driver software. The driver will still make its own determination of which technique it will use. The choice has no effect on the operation of any intrinsics, just on the performance characteristics of that operation.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IREAD", "IWRITE", "IFREAD", "IFWRITE", "IPRINTF", "ISCANF"

IINTROFF

C Syntax

```
#include <sicl.h>

int iintroff ();
```

NOTE

Not supported on Visual BASIC.

Description

The **iintroff** function disables SICL's asynchronous events for a process. This means that all installed handlers for any sessions in a process will be held off until the process enables them with **iintron**.

By default, asynchronous events are enabled. However, the library will not generate any events until the appropriate handlers are installed. To install handlers, refer to the **ionsrq** and **ionintr** functions.

NOTE

The **iintroff/iintron** functions do not affect the **isetintr** values or the handlers in any way.

Default is on.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IONINTR", "IGETONINTR", "IONSRQ", "IGETONSRQ", "IWAITHDLR", "IINTRON"

IINTRON

C Syntax

```
#include <sicl.h>

int iintron ();
```

NOTE

Not supported on Visual BASIC.

Description

The **iintron** function enables all asynchronous handlers for all sessions in the process.

NOTE

The **iintroff/iintron** functions do not affect the **isetintr** values or the handlers in any way.

Default is on.

Calls to **iintroff/iintron** can be nested, meaning that there must be an equal number of on's and off's. This means that simply calling the **iintron** function may not actually enable interrupts again. For example, note how the following code enables and disables events.

```
iintroff();
/* Events Disabled */
iintron();
/* Events Enabled */
iintroff();
/* Events Disabled */
iintroff();
/* Events Disabled */
iintron();
/* Events STILL Disabled */
iintron();
/* Events NOW Enabled */
```

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IONINTR", "IGETONINTR", "IONSRQ", "IGETONSRQ", "IWAITHDLR", "IINTROFF", "ISETINTR"

ILANGETTIMEOUT

Supported sessions: .... interface

C Syntax

```
#include <sicl.h>

int ilangettimeout (id, tval);
INST id;
long *tval;
```

Visual BASIC Syntax

```
Function ilangettimeout
(ByVal id As Integer, tval As Long) As Integer
```

> **NOTE**
>
> LAN is *not* supported with 16-bit SICL on Windows 95.

Description

The `ilangettimeout` function stores the current LAN timeout value in *tval*. If the LAN timeout value has not been set via `ilantimeout`, then *tval* will contain the LAN timeout value calculated by the system.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"ILANTIMEOUT", and the "LAN and Timeouts" section of the "Using HP SICL with LAN" chapter of the *HP SICL User's Guide*.

ILANTIMEOUT

Supported sessions: .... interface

C Syntax

```
#include <sicl.h>

int ilantimeout (id, tval);
INST id;
long tval;
```

Visual BASIC Syntax

```
Function ilantimeout
(ByVal id As Integer, ByVal tval As Long) As Integer
```

NOTE

LAN is *not* supported with 16-bit SICL on Windows 95.

Description

The **ilantimeout** function is used to set the length of time that the application (LAN client) will wait for a response from the LAN server. Once an application has manually set the LAN timeout via this function, the software will no longer attempt to determine the LAN timeout which should be used. Instead, the software will simply use the value set via this function.

In this function, *tval* defines the timeout in milliseconds. A value of zero (0) disables timeouts. The value 1 has special significance, causing the LAN client to not wait for a response from the LAN server. However, the value 1 should be used in special circumstances only and should be used with extreme caution. See the following subsection, "Using the No-Wait Value," for more information.

NOTE

The `ilantimeout` function is per process. Thus, when `ilantimeout` is called, all sessions which are going out over the network are affected.

NOTE

Not all computer systems can guarantee an accuracy of one millisecond on timeouts. Some computer clock systems only provide a resolution of 1/50th or 1/60th of a second. Other computers have a resolution of only 1 second. Note that the time value is *always* rounded up to the next unit of resolution.

This function does not affect the SICL timeout value set via the `itimeout` function. The LAN server will attempt the I/O operation for the amount of time specified via `itimeout` before returning a response.

NOTE

If the SICL timeout used by the server is greater than the LAN timeout used by the client, the client may timeout prior to the server, while the server continues to service the request. This use of the two timeout values is not recommended, since under this situation the server may send an unwanted response to the client.

Using the No-Wait Value

A *tval* value of 1 has special significance to **ilantimeout**, causing the LAN client to not wait for a response from the LAN server. For a very limited number of cases, it may make sense to use this no-wait value. One such scenario is when the performance of paired writes and reads over a wide-area network (WAN) with long latency times is critical, and losing status information from the write can be tolerated. Having the write (and only the write) call not wait for a response allows the read call to proceed immediately, potentially cutting the time required to perform the paired WAN write/read in half.

NOTE

This value should be used with great caution. If **ilantimeout** is set to **1** and then is not reset for a subsequent call, the system may deadlock due to responses being buffered which are never read, filling the buffers on both the LAN client and server.

To use the no-wait value, do the following:

- Prior to the **iwrite** call (or any formatted I/O call that will write data) which you do not wish to block waiting for the returned status from the server, call **ilantimeout** with a timeout value of 1.
- Make the **iwrite** call. The **iwrite** call will return as soon as the message is sent, not waiting for a reply. The **iwrite** call's return value will be I_ERR_TIMEOUT, and the reported count will be 0 (even though the data will be written, assuming no errors).

  Note that the server will send a reply to the write, even though the client will simply discard it. There is no way to directly determine the success or failure of the write, although a subsequent, functioning read call can be a good sign.
- Reset the client side timeout to a reasonable value for your network by calling **ilantimeout** again with a value sufficiently large enough to allow a read reply to be received. It is recommended that you use a value which provides some margin for error. Note that the timeout specified to **ilantimeout** is in milliseconds (rounded up to the nearest second).

- Make the blocking **iread** call (or formatted I/O call that will read data). Since **ilantimeout** has been set to a value other than **1** (preferably not 0), the **iread** call will wait for a response from the server for the specified time (rounded up to the nearest second).

NOTE

If the no-wait value is used in a multi-threaded application and multiple threads are attempting I/O over the LAN, the I/O operations using the no-wait option will wait for access to the LAN for 2 minutes. If another thread is using the LAN interface for greater than 2 minutes, the no-wait operation will timeout.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"ILANGETTIMEOUT", and the "LAN and Timeouts" section of the "Using HP SICL with LAN" chapter of the *HP SICL User's Guide*.

ILOCAL

Supported sessions: .... device
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ilocal (id);
INST id;
```

Visual BASIC Syntax

```
Function ilocal
(ByVal id As Integer)
```

Description

Use the **ilocal** function to put a device into Local Mode. Putting a device in Local Mode enables the device's front panel interface.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IREMOTE", and the interface-specific chapter of the *HP SICL User's Guide* for details of implementation.

ILOCK

Supported sessions: .... device, interface, commander
Affected by functions: .... `itimeout`

C Syntax

```
#include <sicl.h>

int ilock (id);
INST id;
```

Visual BASIC Syntax

```
Function ilock
(ByVal id As Integer)
```

NOTE

Locks are not supported for LAN interface sessions, such as those opened with:

```
   lan_intf = iopen("lan");
```

Description

To lock a session, ensuring exclusive use of a resource, use the **ilock** function.

The *id* parameter refers either to a device, interface, or commander session. If it refers to an interface, then the entire interface is locked; other interfaces are not affected by this session. If the *id* refers to a device or commander, then only that device or commander is locked, and only that session may access that device or commander. However, other devices either on that interface or on other interfaces may be accessed as usual.

Locks are implemented on a per-session basis. If a session within a given process locks a device or interface, then that device or interface is only accessible from that session. It is not accessible from any other session in this process, or in any other process.

Attempting to call a SICL function that obeys locks on a device or interface that is locked will cause the call either to hang until the device or interface is unlocked, to timeout, or to return with the error I_ERR_LOCKED (see isetlockwait).

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 use this interface.

Locking a **commander** (from a commander session) restricts other commander sessions from accessing this controller; however, interface sessions may continue to use this interface.

NOTE

Locking an interface *does* lock out all device session accesses on that interface, such as iwrite (*dev2*, ...), as well as all other SICL interface session accesses on that interface.

The following C example will cause the device session to hang:

```
intf = iopen ("hpib");
dev = iopen ("hpib,7");
      .
      .
      .
ilock (intf);
ilock (dev);                    /* this will succeed */
iwrite (dev, "*CLS", 4, 1, 0);  /* this will hang */
```

The following Visual BASIC example will cause the device session to hang:

```
intf = iopen("hpib")
dev = iopen("hpib,7")
      .
      .
      .
call ilock (intf)
call ilock(dev)                              ' this will succeed
call iwrite(dev, "*CLS", 4, 1, 0&)          ' this will hang
```

Locks can be nested. So every **ilock** requires a matching **iunlock**.

NOTE

If **iclose** is called (either implicitly by exiting the process, or explicitly) for a session that currently has a lock, the lock will be released.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IUNLOCK", "ISETLOCKWAIT", "IGETLOCKWAIT"

IMAP

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

char  *imap (id, map_space, pagestart, pagecnt, suggested);
INST id;
int map_space;
unsigned int pagestart;
unsigned int pagecnt;
char  *suggested;
```

Visual BASIC Syntax

```
Function imap
(ByVal id As Integer, ByVal mapspace As Integer,
 ByVal pagestart As Integer, ByVal pagecnt As Integer,
 ByVal suggested As Long) As Long
```

NOTE

Not supported over LAN.

Description

The **imap** function maps a memory space into your process space. The SICL **i?peek** and **i?poke** functions can then be used to read and write to VXI address space.

The *id* argument specifies a VXI interface or device. The *pagestart* argument indicates the page number within the given memory space where the memory mapping starts. The *pagecnt* argument indicates how many pages to use. For Visual BASIC, you must specify 1 for the *pagecnt* argument.

The *map_space* argument will contain one of the following values:

I_MAP_A16 Map in VXI A16 address space (64 Kbyte pages).

I_MAP_A24 Map in VXI A24 address space (64 Kbyte pages).

I_MAP_A32 Map in VXI A32 address space (64 Kbyte pages).

I_MAP_VXIDEV Map in VXI device registers. (Device session only, 64 bytes.)

I_MAP_EXTEND Map in VXI Device Extended Memory address space in A24 or A32 address space. See individual device manuals for details regarding extended memory address space. (Device session only.)

I_MAP_SHARED Map in VXI A24/A32 memory that is physically located on this device (sometimes called local shared memory). If the hardware supports it (that is, the local shared VXI memory is dual-ported), this map should be through the local system bus and not through the VXI memory. This mapping mechanism provides an alternate way of accessing local VXI memory without having to go through the normal VXI memory system. The value of *pagestart* is the offset (in 64 Kbyte pages) into the shared memory. The value of *pagecnt* is the amount of memory (in 64 Kbyte pages) to map.

NOTE

The E1489 MXIbus Controller Interface can generate 32-bit data reads and writes to VXIbus devices with D32 capability. To use 32-bit transfers with the E1489, use I_MAP_A16_D32, I_MAP_A24_D32, and I_MAP_A32_D32 in place of I_MAP_A16, I_MAP_A24, and I_MAP_A32 when mapping to D32 devices.

The *suggested* argument, if non-NULL, contains a suggested address to begin mapping memory. However, the function may not always use this suggested address. For Visual BASIC, you must pass a 0 (zero) for this argument.

After memory is mapped, it may be accessed directly. Since this function returns a C pointer, you can also use C pointer arithmetic to manipulate the pointer and access memory directly. Note that accidentally accessing non-existent memory will cause bus errors. See the "Using HP SICL with VXI" chapter in the *HP SICL User's Guide for HP-UX* for an example of trapping bus errors. Or see your operating system's programming information for help in trapping bus errors. You will probably find this information under the command **signal** in your operating system's manuals. Note that Visual BASIC programs can perform pointer arithmetic within a single page.

> **NOTE**
>
> Due to hardware constraints on a given device or interface, not all address spaces may be implemented. In addition, there may be a maximum number of pages that can be simultaneously mapped. If a request is made that cannot be granted due to hardware constraints, the process will hang until the desired resources become available. To avoid this, use the **isetlockwait** command with the *flag* parameter set to 0, and thus generate an error instead of waiting for the resources to become available. You may also use the **imapinfo** function to determine hardware constraints before making an **imap** call.

Remember to **iunmap** a memory space when you no longer need it. The resources may be needed by another process.

Return Value

For C programs, this function returns a zero (0) if successful, or a non-zero number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IUNMAP", "IMAPINFO"

IMAPINFO

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int imapinfo (id, map_space, numwindows, winsize);
INST id;
int map_space;
int  *numwindows;
int  *winsize;
```

Visual BASIC Syntax

```
Function imapinfo
(ByVal id As Integer, ByVal mapspace As Integer,
 numwindows As Integer, winsize As Integer)
```

NOTE

Not supported over LAN.

Description

To determine hardware constraints on memory mappings imposed by a particular interface, use the **imapinfo** function.

The *id* argument specifies a VXI interface. The *map_space* argument specifies the address space. Valid values for *map_space* are:

I_MAP_A16	VXI A16 address space (64 Kbyte pages).
I_MAP_A24	VXI A24 address space (64 Kbyte pages).
I_MAP_A32	VXI A32 address space (64 Kbyte pages).

The *numwindows* argument is filled in with the total number of windows available in the address space.

The *winsize* argument is filled in with the size of the windows in pages.

Hardware design constraints may prevent some devices or interfaces from implementing all of the various address spaces. Also there may be a limit to the number of pages that can simultaneously be mapped for usage. In addition, some resources may already be in use and locked by another process. If resource constraints prevent a mapping request, the **`imap`** function will hang, waiting for the resources to become available.

Remember to unmap a memory space when you no longer need it. The resources may be needed by another process.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **`Err`** variable is set if an error occurs.

See Also

"IMAP", "IUNMAP"

IONERROR

C Syntax

```
#include <sicl.h>

int ionerror(proc);
void ( *proc)(id, error);
 INST id;
 int error;
```

NOTE

For WIN16 programs on Microsoft Windows platforms, handler functions used with `ionerror`, `ionintr`, and `ionsrq` must be exported and declared as `_far _pascal`.

NOTE

For Visual BASIC, error handlers are installed using the Visual BASIC `On Error` statement. See the section titled "Using Error Handlers in Visual BASIC" in the "Programming with HP SICL" chapter of the *HP SICL User's Guide for Windows* for more information on error handling with Visual BASIC.

Description

The **ionerror** function is used to install a SICL error handler. Many of the SICL functions can generate an error. When a SICL function errors, it typically returns a special value such as a NULL pointer, zero, or a non-zero error code. A process can specify a procedure to execute when a SICL error occurs. This allows your process to ignore the return value and simply permit the error handler to detect errors and do the appropriate action.

The error handler procedure executes immediately before the SICL function that generated the error completes its operation. There is only one error handler for a given process which handles all errors that occur with any session established by that process.

On operating systems that support multiple **threads**, the error handler is still per-process. However, the error handler will be called in the context of the thread that caused the error.

Error handlers are called with the following arguments:

```
void  proc (id, error);
INST id;
int error;
```

The *id* argument indicates the session that generated the error.

The *error* argument indicates the error that occurred. See Appendix A for a complete description of the error codes.

NOTE

The `INST` *id* that is passed to the error handler is the same `INST` *id* that was passed to the function that generated the error. Therefore, if an error occurred because of an invalid `INST` *id*, the `INST` *id* passed to the error handler is also invalid. Also, if **iopen** generates an error before a session has been established, the error handler will be passed a zero (0) `INST` *id*.

Two special reserved values of *proc* can be passed to the `ionerror` procedure:

`I_ERROR_EXIT` This value installs a special error handler which logs a diagnostic message and terminates the process.

`I_ERROR_NO_EXIT` This value also installs a special error handler which logs a diagnostic message but does not terminate the process.

If a zero (0) is passed as the value of *proc*, it will remove the error handler.

Note that the error procedure could perform a *setjmp/longjmp* or an escape using the *try/recover* clauses.

Example for using *setjmp/longjmp*:

```
#include <sicl.h>

INST id;
jmp_buf env;
...
void proc (INST,int) {
   /* Error occurred, perform a longjmp */
   longjmp (env, 1);
}

void xyzzy () {
   if (setjmp (env) == 0) {
      /* Normal code */
      ionerror (proc);

      /* Do actions that could cause errors */
      iwrite (.......);
      iread (........);
      ...etc...

      ionerror (0);
   } else {
      /* Error Code */
      ionerror (0);
      ... do error processing ...
      if (igeterrno () ==...)
         ... etc ...;
   }
}
```

Or, using *try/recover/escape*:

```
#include <sicl.h>

INST id;
...
void  proc (INST id, int error) {
   /* Error occurred, perform an escape */
   escape (id);
}

void xyzzy () {
   try {
      /* Normal code */
      ionerror (proc);

      /* Do actions that could cause errors */
      iwrite (.......);
      iread (........);
      ...etc...

      ionerror (0);
   } recover {
      /* Error Code */
      ionerror (0);
      ... do error processing ...
      if (igeterrno () == ...)
         ... etc ...;
   }
}
```

ırn Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

Also

“IGETONERROR”, “IGETERRNO”, “IGETERRSTR”, “ICAUSEERR”

IONINTR

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int ionintr (id, proc);
INST id;
void ( *proc)(id, reason, secval);
 INST id;
 long reason;
 long secval;
```

NOTE

Not supported on Visual BASIC.

NOTE

For WIN16 programs on Microsoft Windows platforms, handler functions used with `ionerror`, `ionintr`, and `ionsrq` must be exported and declared as `_far _pascal`.

Description

The library can notify a process when an interrupt occurs by using the **ionintr** function. This function installs the procedure *proc* as an interrupt handler.

After you install the interrupt handler with **ionintr**, use the **isetintr** function to enable notification of the interrupt event or events.

The library calls the *proc* procedure whenever an enabled interrupt occurs. It calls *proc* with the following parameters:

```
void  proc (id, reason, secval);
INST id;
long reason;
long secval;
```

Where:

id	The INST that refers to the session that installed the interrupt handler.
reason	Contains a value which corresponds to the reason for the interrupt. These values correspond to the **isetintr** function parameter *intnum*. See a listing of the values below.
secval	Contains a secondary value which depends on the type of interrupt which occurred. For I_INTR_TRIG, it contains a bit mask corresponding to the trigger lines which fired. For interface-dependent and device-dependent interrupts, it contains an appropriate value for that interrupt.

The *reason* parameter specifies the cause for the interrupt. Valid *reason* values for all interface sessions are:

I_INTR_INTFACT	Interface became active.
I_INTR_INTFDEACT	Interface became deactivated.
I_INTR_TRIG	A Trigger occurred. The *secval* parameter contains a bit-mask specifying which triggers caused the interrupt. See the **ixtrig** function's *which* parameter for a list of valid values.
I_INTR_*	Individual interfaces may use other interface-interrupt conditions.

Valid *reason* values for all device sessions are:

`I_INTR_*`	Individual interfaces may include other interface-interrupt conditions.

To remove the interrupt handler, pass a zero (0) in the *proc* parameter. By default, no interrupt handler is installed.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"ISETINTR", "IGETONINTR", "IWAITHDLR", "IINTROFF", "IINTRON", and the section titled "Asynchronous Events and HP-UX Signals" in the "Programming with HP SICL" chapter of the *HP SICL User's Guide for HP-UX* for protecting I/O calls against interrupts.

IONSRQ

Supported sessions: .... device, interface

C Syntax

```
#include <sicl.h>

int ionsrq (id, proc);
INST id;
void ( *proc)(id);
 INST id;
```

NOTE

For WIN16 programs on Microsoft Windows platforms, handler functions used with `ionerror`, `ionintr`, and `ionsrq` must be exported and declared as `_far _pascal`.

NOTE

Not supported on Visual BASIC.

Description

Use the **ionsrq** function to notify an application when an SRQ occurs. This function installs the procedure *proc* as an SRQ handler.

An SRQ handler is called any time its corresponding interface generates an SRQ. If an interface device driver receives an SRQ and cannot determine the generating device (for example, on HP-IB), it passes the SRQ to *all* SRQ handlers assigned to the interface. Therefore, an SRQ handler cannot assume that its corresponding device actually generated an SRQ. An SRQ handler should use the **ireadstb** function to determine whether its corresponding device generated the SRQ. It calls *proc* with the following parameters:

```
void  proc (id);
INST id;
```

To remove an SRQ handler, pass a zero (0) as the *proc* parameter.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IGETONSRQ", "IWAITHDLR", "IINTROFF", "IINTRON", "IREADSTB"

IOPEN

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

INST iopen (addr);
char  *addr
```

Visual BASIC Syntax

```
Function iopen
(ByVal addr As String)
```

Description

Before using any of the SICL functions, the application program must establish a session with the desired interface or device. Create a session by using the **iopen** function.

This function creates a session and returns a session identifier. Note that the session identifier should only be passed as a parameter to other SICL functions. It is not designed to be updated manually by you.

The *addr* parameter contains the device, interface, or commander address.

An application may have multiple sessions open at the same time by creating multiple session identifiers with the **iopen** function.

> **NOTE**
>
> If an error handler has been installed (see **ionerror**), and an **iopen** generates an error before a session has been established, the handler will be called with the session identifier set to zero (0). Caution must be used if using the session identifier in an error handler.
>
> Also, it is possible for an **iopen** to succeed on a device that does not exist. In this case, other functions (such as **iread**) will fail with a nonexistent device error.

Creating A Device Session

To create a device session, specify a particular interface name followed by the device's address in the *addr* parameter. For more information on addressing devices, see the section on "Addressing Device Sessions" in the "Programming with HP SICL" chapter of the *HP SICL User's Guide*.

C example:

```
INST dmm;
dmm = iopen("hpib,15");
```

Visual BASIC example:

```
DIM dmm As Integer
dmm = iopen("hpib,15")
```

Creating An Interface Session

To create an interface session, specify a particular interface in the *addr* parameter. For more information on addressing interfaces, see the section on "Addressing Interface Sessions" in the "Programming with HP SICL" chapter of the *HP SICL User's Guide*.

C example:

```
INST hpib;
hpib = iopen("hpib");
```

Visual BASIC example:

```
DIM hpib As Integer
hpib = iopen("hpib")
```

Creating A Commander Session

To create a commander session, use the keyword **cmdr** in the *addr* parameter. For more information on commander sessions, see the section on "Addressing Commander Sessions" in the "Programming with HP SICL" chapter of the *HP SICL User's Guide*.

C example:

```
INST cmdr;
cmdr = iopen("hpib,cmdr");
```

Visual BASIC example:

```
DIM cmdr As Integer
cmdr = iopen("hpib,cmdr")
```

Return Value

The **iopen** function returns a zero (0) *id* value if an error occurs; otherwise a valid session *id* is returned.

See Also

"ICLOSE"

IPEEK

C Syntax

```
#include <sicl.h>

unsigned char ibpeek (addr);
unsigned char  *addr;

unsigned short iwpeek (addr);
unsigned short  *addr;

unsigned long ilpeek (addr);
unsigned long  *addr;
```

Visual BASIC Syntax

```
Function ibpeek
(ByVal addr As Long) As Byte

Function iwpeek
(ByVal addr As Long) As Integer

Function ilpeek
(ByVal addr As Long) As Long
```

NOTE

Not supported over LAN.

Description

The **i?peek** functions will read the value stored at *addr* from memory and return the result. The **i?peek** functions are generally used in conjunction with the SICL **imap** function to read data from VXI address space.

NOTE

The **`iwpeek`** and **`ilpeek`** functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering.

Also, if a bus error occurs, unexpected results may occur.

See Also

"IPOKE", "IMAP"

IPOKE

C Syntax

```
#include <sicl.h>

void ibpoke (addr, val);
unsigned char  *addr;
unsigned char val;

void iwpoke (addr, val);
unsigned short  *addr;
unsigned short val;

void ilpoke (addr, val);
unsigned long  *addr;
unsigned long val;
```

Visual BASIC Syntax

```
Sub ibpoke
(ByVal addr As Long, ByVal value As Integer)

Sub iwpoke
(ByVal addr As Long, ByVal value As Integer)

Sub ilpoke
(ByVal addr As Long, ByVal value As Long)
```

NOTE

Not supported over LAN.

Description

The **i?poke** functions will write to memory. The **i?poke** functions are generally used in conjunction with the SICL **imap** function to write to VXI address space.

The *addr* is a valid memory address. The *val* is a valid data value.

NOTE

The **iwpoke** and **ilpoke** functions perform byte swapping (if necessary) so that VXI memory accesses follow correct VXI byte ordering.

Also, if a bus error occurs, unexpected results may occur.

See Also

"IPEEK", "IMAP"

IPOPFIFO

C Syntax

```
#include <sicl.h>

int ibpopfifo (id, fifo, dest, cnt);
INST id;
unsigned char  *fifo;
unsigned char  *dest;
unsigned long cnt;

int iwpopfifo (id, fifo, dest, cnt, swap);
INST id;
unsigned char  *fifo;
unsigned char  *dest;
unsigned long cnt;
int swap;

int ilpopfifo (id, fifo, dest, cnt, swap);
INST id;
unsigned char  *fifo;
unsigned char  *dest;
unsigned long cnt;
int swap;
```

Visual BASIC Syntax

```
Function ibpopfifo
(ByVal id As Integer, ByVal fifo As Long,
 ByVal dest As Long, ByVal cnt As Long)

Function iwpopfifo
(ByVal id As Integer, ByVal fifo As Long,
 ByVal dest As Long, ByVal cnt As Long,
 ByVal swap As Integer)

Function ilpopfifo
(ByVal id As Integer, ByVal fifo As Long,
 ByVal dest As Long, ByVal cnt As Long,
 ByVal swap As Integer)
```

NOTE

Not supported over LAN.

Description

The `i?popfifo` functions read data from a FIFO and puts it in memory. Use `b` for byte, `w` for word, and `l` for long word (8-bit, 16-bit, and 32-bit, respectively). These functions increment the write address, to write successive memory locations, while reading from a single memory (FIFO) location. Thus, these functions can transfer entire blocks of data.

The *id*, although specified, is normally ignored except to determine an interface-specific transfer mechanism such as DMA. To prevent using an interface-specific mechanism, pass a zero (0) in this parameter. The *dest* argument is the starting memory address for the destination data. The *fifo* argument is the memory address for the source FIFO register data. The *cnt* argument is the number of transfers (bytes, words, or longwords) to perform. The *swap* argument is the byte swapping flag. If *swap* is zero, no swapping occurs. If *swap* is non-zero, the function swaps bytes (if necessary) to change byte ordering from the internal format of the controller to/from the VXI (big-endian) byte ordering.

NOTE

If a bus error occurs, unexpected results may occur.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IPEEK", "IPOKE", "IPUSHFIFO", "IMAP"

IPRINTF

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iprintf (id, format [,arg1] [,arg2] [,...]);
int isprintf (buf, format [,arg1] [,arg2] [,...]);
int ivprintf (id, format, va_list ap);
int isvprintf (buf, format, va_list ap);
INST id;
char  *buf;
const char  *format;
param arg1, arg2, ...;
va_list ap;
```

NOTE

For WIN16 programs on Microsoft Windows platforms, if compiling with tiny, small, or medium models, make sure all pointer/address parameters are passed as `_far`.

Visual BASIC Syntax

```
Function ivprintf
(ByVal id As Integer, ByVal fmt As String,
 ByVal ap As Any)
```

Description

These functions convert data under the control of the *format* string. The *format* string specifies how the argument is converted before it is output. If the first argument is an `INST`, the data is sent to the device to which the `INST` refers. If the first argument is a character buffer, the data is placed in the buffer.

The *format* string contains regular characters and special conversion sequences. The `iprintf` function sends the regular characters (not a `%` character) in the *format* string directly to the device. Conversion specifications are introduced by the `%` character. Conversion specifications control the type, the conversion, and the formatting of the *arg* parameters.

NOTE

The formatted I/O functions, `iprintf` and `ipromptf`, can re-address the bus multiple times during execution. This behavior may cause problems with instruments which do not comply with IEEE 488.2.

Re-addressing occurs under the following circumstances:

- After the internal buffer fills. (See `isetbuf`.)
- When a `\n` is found in the *format* string in C/C++, or when a `Chr$(10)` is found in the *format* string in Visual BASIC.
- When a `%C` is found in the *format* string.

This behavior affects only non-IEEE 488.2 devices on the GPIB interface.

Use the special characters and conversion commands explained later in this section to create the *format* string's contents.

Restrictions Using `ivprintf` in Visual BASIC

The following restrictions apply when using **`ivprintf`** with Visual BASIC.

- Format Conversion Commands:

  Only one format conversion command can be specified in a format string for **`ivprintf`** (a format conversion command begins with the % character). For example, the following is invalid:

  ```
  nargs% = ivprintf(id, "%lf%d" + Chr$(10), ...)
  ```

  Instead, you must call **`ivprintf`** once for each format conversion command, as shown in the following example:

  ```
  nargs% = ivprintf(id, "%lf" + Chr$(10), dbl_value)
  nargs% = ivprintf(id, "%d" + Chr$(10), int_value)
  ```

- Writing Numeric Arrays:

  For Visual BASIC, when writing from a numeric array with **`ivprintf`**, you must specify the first element of a numeric array as the *ap* parameter to **`ivprintf`**. This passes the address of the first array element to **`ivprintf`**. For example:

  ```
  Dim flt_array(50) As Double
  nargs% = ivprintf(id, "%,50f", dbl_array(0))
  ```

  This code declares an array of 50 floating point numbers and then calls **`ivprintf`** to write from the array.

  For more information on passing numeric arrays as arguments with Visual BASIC, see the "Arrays" section of the "Calling Procedures in DLLs" chapter of the *Visual BASIC Programmer's Guide.*

- Writing Strings:

  The %S format string is not supported for **`ivprintf`** on Visual BASIC.

Special Characters for C/C++

Special characters in C/C++ consist of a backslash (\) followed by another character. The special characters are:

`\n`	Send the ASCII LF character with the END indicator set.
`\r`	Send the ASCII CR character.
`\\`	Send the backslash (\) character.
`\t`	Send the ASCII TAB character.
`\###`	Send the ASCII character specified by the octal value **###**.
`\v`	Send the ASCII VERTICAL TAB character.
`\f`	Send the ASCII FORM FEED character.
`\"`	Send the ASCII double-quote (") character.

Special Characters for Visual BASIC

Special characters in Visual BASIC are specified with the `CHR$()` function. These special characters are added to the format string by using the `+` string concatenation operator in Visual BASIC. For example:

```
nargs=ivprintf(id, "*RST"+CHR$(10), 0&)
```

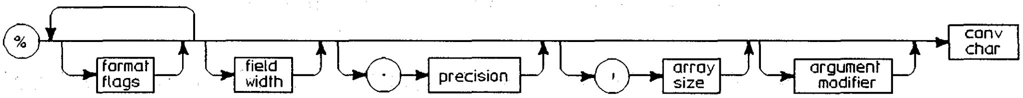
The special characters are:

`Chr$(10)`	Send the ASCII LF character with the END indicator set.
`Chr$(13)`	Send the ASCII CR character.
`\`	Sends the backslash (\) character.[1]
`Chr$(9)`	Send the ASCII TAB character.
`Chr$(11)`	Send the ASCII VERTICAL TAB character.
`Chr$(12)`	Send the ASCII FORM FEED character.
`Chr$(34)`	Send the ASCII double-quote (") character.

[1] In Visual BASIC, the backslash character can be specified in a format string directly, instead of being "escaped" by prepending it with another backslash.

Format Conversion Commands

An `iprintf` format conversion command begins with a % character. After the % character, the optional modifiers appear in this order: format flags, field width, a period and precision, a comma and array size (comma operator), and an argument modifier. The command ends with a conversion character.



Syntax for `iprintf`
Format Conversion Commands

The modifiers in a conversion command are:

format flags Zero or more flags (in any order) that modify the meaning of the conversion character. See the following subsection, "List of *format flags*" for the specific flags you may use.

field width An optional minimum *field width* is an integer (such as `"%8d"`). If the formatted data has fewer characters than field width, it will be padded. The padded character is dependent on various flags. In C/C++, an asterisk (`*`) may appear for the integer, in which case it will take another *arg* to satisfy this conversion command. The next *arg* will be an integer that will be the *field width* (for example, `iprintf (`*id*`, "%*d", 8, num)`).

. precision — The precision operator is an integer preceded by a period (such as `"%.6d"`). The optional precision for conversion characters `e`, `E`, and `f` specifies the number of digits to the right of the decimal point. For the `d, i, o, u, x,` and `X` conversion characters, it specifies the minimum number of digits to appear. For the `s` and `S` conversion characters, the precision specifies the maximum number of characters to be read from your *arg* string. In C/C++, an asterisk (*) may appear in the place of the integer, in which case it will take another *arg* to satisfy this conversion command. The next *arg* will be an integer that will be the *precision* (for example, `iprintf (`*id*`, "%.*d", 6, num)`).

, array size — The comma operator is an integer preceded by a comma (such as `"%,10d"`). The optional comma operator is only valid for conversion characters `d` and `f`. This is a comma followed by a number. This indicates that a list of comma-separated numbers is to be generated. The argument is an array of the specified type instead of the type (that is, an array of integers instead of an integer). In C/C++, an asterisk (*) may appear for the number, in which case it will take another *arg* to satisfy this conversion command. The next *arg* will be an integer that is the number of elements in the array.

argument modifier — The meaning of the modifiers `h, l, w, z,` and `Z` is dependent on the conversion character (such as `"%wd"`).

conv char — A conversion character is a character that specifies the type of *arg* and the conversion to be applied. This is the only required element of a conversion command. See the following subsection, "List of *conv chars*" for the specific conversion characters you may use.

Examples of Format Conversion Commands

The following are some examples of conversion commands used in the *format* string and the output that would result from them. (The output data is arbitrary.)

Conversion Command	Output	Description
`%@Hd`	`#H3A41`	format flag
`%10s`	`       str`	field width
`%-10s`	`str`	format flag (left justify) & field width
`%.6f`	`21.560000`	precision
`%,3d`	`18,31,34`	comma operator
`%6ld`	`   132`	field width & argument modifier (long)
`%.6ld`	`000132`	precision & argument modifier (long)
`%.6ld`	`000132`	precision & argument modifier (long)
`%@1d`	`61`	format flag (IEEE 488.2 NR1)
`%@2d`	`61.000000`	format flag (IEEE 488.2 NR2)
`%@3d`	`6.100000E+01`	format flag (IEEE 488.2 NR3)

List of *format flags*

The *format flags* you can use in conversion commands are:

`@1` Convert to an NR1 number (an IEEE 488.2 format integer with no decimal point). Valid only for `%d` and `%f`. Note that `%f` values will be truncated to the integer value.

`@2` Convert to an NR2 number (an IEEE 488.2 format floating point number with at least one digit to the right of the decimal point). Valid only for `%d` and `%f`.

`@3` Convert to an NR3 number (an IEEE 488.2 format number expressed in exponential notation). Valid only for `%d` and `%f`.

`@H` Convert to an IEEE 488.2 format hexadecimal number in the form #Hxxxx. Valid only for `%d` and `%f`. Note that `%f` values will be truncated to the integer value.

`@Q` Convert to an IEEE 488.2 format octal number in the form #Qxxxx. Valid only for `%d` and `%f`. Note that `%f` values will be truncated to the integer value.

`@B` Convert to an IEEE 488.2 format binary number in the form #Bxxxx. Valid only for `%d` and `%f`. Note that `%f` values will be truncated to the integer value.

`-` Left justify the result.

`+` Prefix the result with a sign (`+` or `-`) if the output is a signed type.

`space` Prefix the result with a blank ( ) if the output is signed and positive. Ignored if both blank and `+` are specified.

`#` Use alternate form. For the `o` conversion, it prints a leading zero. For `x` or `X`, a non-zero will have 0x or 0X as a prefix. For `e, E, f, g,` and `G`, the result will always have one digit on the right of the decimal point.

`0` Will cause the left pad character to be a zero (0) for all numeric conversion types.

List of *conv chars*

The *conv chars* (conversion characters) you can use in conversion commands are:

`d` Corresponding *arg* is an integer. If no flags are given, send the number in IEEE 488.2 NR1 (integer) format. If flags indicate an NR2 (floating point) or NR3 (floating point) format, convert the argument to a floating point number. This argument supports all six flag modifier formatting options: NR1 - `@1`, NR2 - `@2`, NR3 - `@3`, `@H`, `@Q`, or `@B`. If the `l` argument modifier is present, the *arg* must be a long integer. If the `h` argument modifier is present, the *arg* must be a short integer for C/C++, or an Integer for Visual BASIC.

`f` Corresponding *arg* is a double for C/C++, or a Double for Visual BASIC. If no flags are given, send the number in IEEE 488.2 NR2 (floating point) format. If flags indicate that NR1 format is to be used, the *arg* will be truncated to an integer. This argument supports all six flag modifier formatting options: NR1 - `@1`, NR2 - `@2`, NR3 - `@3`, `@H`, `@Q`, or `@B`. If the `l` argument modifier is present, the *arg* must be a double. If the `L` argument modifier is present, the *arg* must be a long double for C/C++ (not supported for Visual BASIC).

b In C/C++, corresponding *arg* is a pointer to an arbitrary block of data. (Not supported in Visual BASIC.) The data is sent as IEEE 488.2 Definite Length Arbitrary Block Response Data. The field width must be present and will specify the number of elements in the data block. An asterisk (*) can be used in place of the integer, which indicates that two *arg*s are used. The first is a long used to specify the number of elements. The second is the pointer to the data block. No byte swapping is performed.

If the w argument modifier is present, the block of data is an array of unsigned short integers. The data block is sent to the device as an array of words (16 bits). The *field width* value now corresponds to the number of short integers, not bytes. Each word will be appropriately byte swapped and padded so that they are converted from the internal computer format to the standard IEEE 488.2 format.

If the l argument modifier is present, the block of data is an array of unsigned long integers. The data block is sent to the device as an array of longwords (32 bits). The *field width* value now corresponds to the number of long integers, not bytes. Each word will be appropriately byte swapped and padded so that they are converted from the internal computer format to the standard IEEE 488.2 format.

If the z argument modifier is present, the block of data is an array of floats. The data is sent to the device as an array of 32-bit IEEE 754 format floating point numbers. The *field width* is the number of floats.

If the Z argument modifier is present, the block of data is an array of doubles. The data is sent to the device as an array of 64-bit IEEE 754 format floating point numbers. The *field width* is the number of doubles.

B Same as b in C/C++, except that the data block is sent as IEEE 488.2 Indefinite Length Arbitrary Block Response Data. (Not supported in Visual BASIC.) Note that this format involves sending a newline with an END indicator on the last byte of the data block.

`c` In C/C++, corresponding *arg* is a character. (Not supported in Visual BASIC.)

`C` In C/C++, corresponding *arg* is a character. Send with END indicator. (Not supported in Visual BASIC.)

`t` In C/C++, control sending the END indicator with each LF character in the *format* string. (Not supported in Visual BASIC.) A `+` flag indicates to send an END with each succeeding LF character (default), a `-` flag indicates to not send END. If no `+` or `-` flag appears, an error is generated.

`s` Corresponding *arg* is a pointer to a null-terminated string that is sent as a string.

`S` In C/C++, corresponding *arg* is a pointer to a null-terminated string that is sent as an IEEE 488.2 string response data block. (Not supported in Visual BASIC.) An IEEE 488.2 string response data block consists of a leading double quote (") followed by non-double quote characters and terminated with a double quote.

`%` Send the ASCII percent (`%`) character.

`i` Corresponding *arg* is an integer. Same as `d` except that the six `flag` modifier formatting options: NR1 - `@1`, NR2 - `@2`, NR3 - `@3`, `@H`, `@Q`, or `@B` are ignored.

`o,u,x,X` Corresponding *arg* will be treated as an unsigned integer. The argument is converted to an unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x,X). The letters `abcdef` are used with x, and the letters `ABCDEF` are used with X. The precision specifies the minimum number of characters to appear. If the value can be represented with fewer than precision digits, leading zeros are added. If the precision is set to zero and the value is zero, no characters are printed.

`e,E` Corresponding *arg* is a double in C/C++, or a Double in Visual BASIC. The argument is converted to exponential format (that is, `[-]d.dddde+/-dd`). The precision specifies the number of digits to the right of the decimal point. If no precision is specified, then six digits will be converted. The letter e will be used with `e` and the letter E will be used with `E`.

`g,G` Corresponding *arg* is a double in C/C++, or a Double in Visual BASIC. The argument is converted to exponential (e with `g`, or E with `G`) or floating point format depending on the value of the *arg* and the precision. The exponential style will be used if the resulting exponent is less than -4 or greater than the precision; otherwise it will be printed as a float.

`n` Corresponding *arg* is a pointer to an integer in C/C++, or an Integer for Visual BASIC. The number of bytes written to the device for the entire **`iprintf`** call is written to the *arg*. No argument is converted.

`F` On HP-UX or Windows NT, corresponding *arg* is a pointer to a FILE descriptor. (Not supported on Windows 95.) The data will be read from the file that the FILE descriptor points to and written to the device. The FILE descriptor must be opened for reading. No flags or modifiers are allowed with this conversion character.

Return Value

This function returns the total number of arguments converted by the *format* string.

Buffers and Errors

Since **`iprintf`** does not return an error code and data is buffered before it is sent, it cannot be assumed that the device received any data after the **`iprintf`** has completed.

The best way to detect errors is to install your own error handler. This handler can decide the best action to take depending on the error that has occurred.

If an error has occurred during an **`iprintf`** with no error handler installed, the only way you can be informed that an error has occurred is to use **`igeterrno`** right after the **`iprintf`** call.

Remember that **`iprintf`** can be called many times without any data being flushed to the session. There are only three (3) conditions where the write formatted I/O buffer is flushed. Those conditions are:

- If a newline is encountered in the *format* string.
- If the buffer is filled.
- If **`iflush`** is called with the `I_BUF_WRITE` value.

If an error occurs while writing data, such as a timeout, the buffer will be flushed (that is, the data will be lost) and, if an error handler is installed, it will be called, or the error number will be set to the appropriate value.

See Also

"ISCANF", "IPROMPTF", "IFLUSH", "ISETBUF", "ISETUBUF", "IFREAD", "IFWRITE"

IPROMPTF

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ipromptf (id, writefmt, readfmt[, arg1][, arg2][, ...]);
int ivpromptf (id, writefmt, readfmt, ap);
INST id;
const char  *writefmt;
const char  *readfmt;
param arg1,arg2,...;
va_list ap;
```

NOTE

Not supported on Visual BASIC.

NOTE

For WIN16 programs on Microsoft Windows platforms, if compiling with tiny, small, or medium models, make sure all pointer/address parameters are passed as `_far`.

Description

The **ipromptf** function is used to perform a formatted write immediately followed by a formatted read. This function is a combination of the **iprintf** and **iscanf** functions. First, it flushes the read buffer. It then formats a string using the *writefmt* string and the first *n* arguments necessary to implement the prompt string. The write buffer is then flushed to the device. It then uses the *readfmt* string to read data from the device and to format it appropriately.

The *writefmt* string is identical to the format string used for the **iprintf** function.

The *readfmt* string is identical to the format string used for the **iscanf** function. It uses the arguments immediately following those needed to satisfy the *writefmt* string.

This function returns the total number of arguments used by both the read and write format strings.

See Also

"IPRINTF", "ISCANF", "IFLUSH", "ISETBUF", "ISETUBUF", "IFREAD", "IFWRITE"

IPUSHFIFO

C Syntax

```
#include <sicl.h>

int ibpushfifo (id, src, fifo, cnt);
INST id;
unsigned char  *src;
unsigned char  *fifo;
unsigned long cnt;

int iwpushfifo (id, src, fifo, cnt, swap);
INST id;
unsigned short  *src;
unsigned short  *fifo;
unsigned long cnt;
int swap;

int ilpushfifo (id, src, fifo, cnt, swap);
INST id;
unsigned long  *src;
unsigned long  *fifo;
unsigned long cnt;
int swap;
```

Visual BASIC Syntax

```
Function ibpushfifo
(ByVal id As Integer, ByVal src As Long,
 ByVal fifo As Long, ByVal cnt As Long)

Function iwpushfifo
(ByVal id As Integer, ByVal src As Long,
 ByVal fifo As Long, ByVal cnt As Long,
 ByVal swap As Integer)

Function ilpushfifo
(ByVal id As Integer, ByVal src As Long,
 ByVal fifo As Long, ByVal cnt As Long,
 ByVal swap As Integer)
```

NOTE

Not supported over LAN.

Description

The `i?pushfifo` functions copy data from memory on one device to a FIFO on another device. Use `b` for byte, `w` for word, and `l` for long word (8-bit, 16-bit, and 32-bit, respectively). These functions increment the read address, to read successive memory locations, while writing to a single memory (FIFO) location. Thus, they can transfer entire blocks of data.

The *id*, although specified, is normally ignored except to determine an interface-specific transfer mechanism such as DMA. To prevent using an interface-specific mechanism, pass a zero (0) in this parameter. The *src* argument is the starting memory address for the source data. The *fifo* argument is the memory address for the destination FIFO register data. The *cnt* argument is the number of transfers (bytes, words, or longwords) to perform. The *swap* argument is the byte swapping flag. If *swap* is zero, no swapping occurs. If *swap* is non-zero the function swaps bytes (if necessary) to change byte ordering from the internal format of the controller to/from the VXI (big-endian) byte ordering.

NOTE

If a bus error occurs, unexpected results may occur.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IPOPFIFO", "IPOKE", "IPEEK", "IMAP"

IREAD

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iread (id, buf, bufsize, reason, actualcnt);
INST id;
char  *buf;
unsigned long bufsize;
int  *reason;
unsigned long  *actualcnt;
```

Visual BASIC Syntax

```
Function iread
(ByVal id As Integer, buf As String,
 ByVal bufsize As Long, reason As Integer,
 actual As Long)
```

Description

This function reads raw data from the device or interface specified by *id*. The *buf* argument is a pointer to the location where the block of data can be stored. The *bufsize* argument is an unsigned long integer containing the size, in bytes, of the buffer specified in *buf*.

The *reason* argument is a pointer to an integer that, on exiting the **iread** call, contains the reason why the read terminated. If the *reason* parameter contains a zero (0), then no termination reason is returned. Reasons include:

I_TERM_MAXCNT	*bufsize* characters read.
I_TERM_END	*END* indicator received on last character.
I_TERM_CHR	Termination character enabled and received.

The *actualcnt* argument is a pointer to an unsigned long integer. Upon exit, this contains the actual number of bytes read from the device or interface. If the *actualcnt* parameter is NULL, then the number of bytes read will not be returned.

If you want to pass a NULL *reason* or *actualcnt* parameter to **iread** in Visual BASIC, you should pass the expression **0&**.

For LAN, if the client times out prior to the server, the *actualcnt* returned will be 0, even though the server may have read some data from the device or interface.

This function reads data from the specified device or interface and stores it in *buf* up to the maximum number of bytes allowed by *bufsize.* The read terminates only on one of the following conditions:

- It reads *bufsize* number of bytes.
- It receives a byte with the *END* indicator attached.
- It receives the current termination character (set with **itermchr**).
- An error occurs.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IWRITE", "ITERMCHR", "IFREAD", "IFWRITE"

IREADSTB

Supported sessions: .... device
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ireadstb (id, stb);
INST id;
unsigned char  *stb;
```

Visual BASIC Syntax

```
Function ireadstb
(ByVal id As Integer, stb As String)
```

Description

The **ireadstb** function reads the status byte from the device specified by *id*. The *stb* argument is a pointer to a variable which will contain the status byte upon exit.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IONSRQ", "ISETSTB"

IREMOTE

Supported sessions: .... device
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iremote (id);
INST id;
```

Visual BASIC Syntax

```
Function iremote
(ByVal id As Integer)
```

Description

Use the **iremote** function to put a device into remote mode. Putting a device in remote mode disables the device's front panel interface.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"ILOCAL", and the interface-specific chapter in the *HP SICL User's Guide* for details of implementation.

ISCANF

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iscanf (id, format [,arg1][,arg2][,...]);
int isscanf (buf, format [,arg1][,arg2][,...]);
int ivscanf (id, format, va_list ap);
int isvscanf (buf, format, va_list ap);
INST id;
char  *buf;
const char  *format;
ptr arg1, arg2, ...;
va_list ap;
```

NOTE

For WIN16 programs on Microsoft Windows platforms, if compiling with tiny, small, or medium models, make sure all pointer/address parameters are passed as `_far`.

Visual BASIC Syntax

```
Function ivscanf
(ByVal id As Integer, ByVal fmt As String,
 ByRef ap As Any)
```

Description

These functions read formatted data, convert it, and store the results into your *args*. These functions read bytes from the specified device, or from *buf*, and convert them using conversion rules contained in the *format* string. The number of *args* converted is returned.

The *format* string contains:

- White-space characters, which are spaces, tabs, or special characters.
- An ordinary character (not %), which must match the next non-white-space character read from the device.
- Format conversion commands.

Use the white-space characters and conversion commands explained later in this section to create the *format* string's contents.

Notes on Using `iscanf`

- Using `itermchr` with `iscanf`:

  The `iscanf` function only terminates reading on an END indicator. The `itermchr` function has no effect on the termination of an `iscanf` read.

- Using `iscanf` with Certain Instruments:

  The `iscanf` function cannot be used easily with instruments that do not send an END indicator.

- Buffer Management with `iscanf`:

  By default, `iscanf` does *not* flush its internal buffer after each call. This means data left from one call of `iscanf` can be read with the next call to `iscanf`. One side effect of this is that successive calls to `iscanf` may yield unexpected results. For example, reading the following data:

  ```
  "1.25\r\n"
  "1.35\r\n"
  "1.45\r\n"
  ```

  With:

  ```
  iscanf(id, "%lf", &res1);   // Will read the 1.25
  iscanf(id, "%lf", &res2);   // Will read the \r\n
  iscanf(id, "%lf", &res3);   // Will read the 1.35
  ```

There are four ways to get the desired results:

- □ Use the newline and carriage return characters at the end of the format string to match the input data. This is the recommended approach. For example:

```
iscanf(id, "%lf%\r\n", &res1);
iscanf(id, "%lf%\r\n", &res2);
iscanf(id, "%lf%\r\n", &res3);
```

- □ Use **isetbuf** with a negative buffer size. This will create a buffer the size of the absolute value of *bufsize*. This also sets a flag that tells **iscanf** to flush its buffer after every **iscanf** call.

```
isetbuf(id, I_BUF_READ, -128);
```

- □ Do explicit calls to **iflush** to flush the read buffer.

```
iscanf(id, "%lf", &res1);
iflush(id, I_BUF_READ);
iscanf(id, "%lf", &res2);
iflush(id, I_BUF_READ);
iscanf(id, "%lf", &res3);
iflush(id, I_BUF_READ);
```

- □ Use the **%*t** conversion to read to the end of the buffer and discard the characters read, if the last character has an END indicator.

```
iscanf(id, "%lf%*t", &res1);
iscanf(id, "%lf%*t", &res2);
iscanf(id, "%lf%*t", &res3);
```

Restrictions Using ivscanf in Visual BASIC

The following restrictions apply when using **ivscanf** with Visual BASIC.

- Format Conversion Commands:

  Only one format conversion command can be specified in a format string for **ivscanf** (a format conversion command begins with the % character). For example, the following is invalid:

```
nargs% = ivscanf(id, "%,50lf%,50d", ...)
```

  Instead, you must call **ivscanf** once for each format conversion command, as shown in the following example:

```
nargs% = ivscanf(id, "%,50lf", dbl_array(0))
nargs% = ivscanf(id, "%,50d", int_array(0))
```

- Reading in Numeric Arrays:

  For Visual BASIC, when reading into a numeric array with `ivscanf`, you must specify the first element of a numeric array as the *ap* parameter to `ivscanf`. This passes the address of the first array element to `ivscanf`. For example:

  ```
  Dim preamble(50) As Double
  nargs% = ivscanf(id, "%,50lf", preamble(0))
  ```

  This code declares an array of 50 floating point numbers and then calls `ivscanf` to read into the array.

  For more information on passing numeric arrays as arguments with Visual BASIC, see the "Arrays" section of the "Calling Procedures in DLLs" chapter of the *Visual BASIC Programmer's Guide.*

- Reading in Strings:

  For Visual BASIC, when reading in a string value with `ivscanf`, you must pass a fixed length string as the *ap* parameter to `ivscanf`. For more information on fixed length strings with Visual BASIC, see the "String Types" section of the "Variables, Constants, and Data Types" chapter of the *Visual BASIC Programmer's Guide.*

White-Space Characters for C/C++

White-space characters are spaces, tabs, or special characters. For C/C++, the white-space characters consist of a backslash (\) followed by another character. The white-space characters are:

`\t`	The ASCII TAB character
`\v`	The ASCII VERTICAL TAB character
`\f`	The ASCII FORM FEED character
space	The ASCII space character

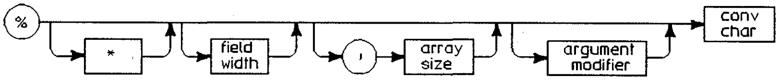
White-Space Characters for Visual BASIC

White-space characters are spaces, tabs, or special characters. For Visual BASIC, the white-space characters are specified with the `Chr$()` function. The white-space characters are:

`Chr$(9)`	The ASCII TAB character
`Chr$(11)`	The ASCII VERTICAL TAB character
`Chr$(12)`	The ASCII FORM FEED character
space	The ASCII space character

Format Conversion Commands

An `iscanf` format conversion command begins with a % character. After the % character, the optional modifiers appear in this order: an assignment suppression character (*), field width, a comma and array size (comma operator), and an argument modifier. The command ends with a conversion character.



Syntax for `iscanf` Format Conversion Commands

The modifiers in a conversion command are:

*
: An optional, assignment suppression character (*). This provides a way to describe an input field to be skipped. An input field is defined as a string of non-white-space characters that extends either to the next inappropriate character, or until the *field width* (if specified) is exhausted.

field width
: An optional integer representing the *field width*. In C/C++, if a pound sign (#) appears instead of the integer, then the next *arg* is a pointer to the *field width*. This *arg* is a pointer to an integer for `%c`, `%s`, `%t`, and `%S`. This *arg* is a pointer to a long for `%b`. The *field width* is not allowed for `%d` or `%f`.

, array size — An optional comma operator is an integer preceded by a comma. It reads a list of comma-separated numbers. The comma operator is in the form of **,dd**, where **dd** is the number of array elements to read. In C/C++, a pound sign (**#**) can be substituted for the number, in which case the next argument is a pointer to an integer that is the number of elements in the array.

The function will set this to the number of elements read. This operator is only valid with the conversion characters **d** and **f**. The argument must be an array of the type specified.

argument modifier — The meaning of the optional argument modifiers **h, l, w, z,** and **Z** is dependent on the conversion character.

conv char — A conversion character is a character that specifies the type of *arg* and the conversion to be applied. This is the only required element of a conversion command. See the following subsection, "List of *conv chars*" for the specific conversion characters you may use.

NOTE

Unlike C's **scanf** function, SICL's **iscanf** functions do not treat the newline (**\n**) and carriage return (**\r**) characters as white-space. Therefore, they are treated as ordinary characters and must match input characters. (Note that this does *not* apply in Visual BASIC.)

The conversion commands direct the assignment of the next *arg*. The **iscanf** function places the converted input in the corresponding variable, unless the * assignment suppression character causes it to use no *arg* and to ignore the input.

This function ignores all white-space characters in the input stream.

Examples of Format Conversion Commands

The following are examples of conversion commands used in the *format* string and typical input data that would satisfy the conversion commands.

Conversion Command	Input Data	Description
`%*s`	`onestring`	suppression (no assignment)
`%*s %s`	`two strings`	suppression (two) assignment (strings)
`%,3d`	`21,12,61`	comma operator
`%hd`	`64`	argument modifier (short)
`%10s`	`onestring`	field width
`%10c`	`onestring`	field width
`%10t`	`two strings`	field width (10 chars read into 1 arg)

List of *conv chars*

The *conv chars* (conversion characters) are:

`d` Corresponding *arg* must be a pointer to an integer for C/C++, or an Integer in Visual BASIC. The library reads characters until an entire number is read. It will convert IEEE 488.2 HEX, OCT, BIN, and NRf format numbers. If the `l` (ell) argument modifier is used, the argument must be a pointer to a long integer in C/C++, or it must be a Long in Visual BASIC. If the `h` argument modifier is used, the argument must be a pointer to a short integer for C/C++, or an Integer for Visual BASIC.

`i` Corresponding *arg* must be a pointer to an integer in C/C++, or an Integer in Visual BASIC. The library reads characters until an entire number is read. If the number has a leading zero (0), the number will be converted as an octal number. If the data has a leading 0x or 0X, the number will be converted as a hexidecimal number. If the `l` (ell) argument modifier is used, the argument must be a pointer to a long integer in C/C++, or it must be a Long for Visual BASIC. If the `h` argument modifier is used, the argument must be a pointer to a short integer for C/C++, or an Integer for Visual BASIC.

`f` Corresponding *arg* must be a pointer to a float in C/C++, or a Single in Visual BASIC. The library reads characters until an entire number is read. It will convert IEEE 488.2 HEX, OCT, BIN, and NRf format numbers. If the `l` (ell) argument modifier is used, the argument must be a pointer to a double for C/C++, or it must be a Double for Visual BASIC. If the `L` argument modifier is used, the argument must be a pointer to a long double for C/C++ (not supported for Visual BASIC).

`e,g` Corresponding *arg* must be a pointer to a float for C/C++, or a Single for Visual BASIC. The library reads characters until an entire number is read. If the `l` (ell) argument modifier is used, the argument must be a pointer to a double for C/C++, or a Double for Visual BASIC. If the `L` argument modifier is used, the argument must be a pointer to a long double for C/C++ (not supported for Visual BASIC).

`c` Corresponding *arg* is a pointer to a character sequence for C/C++, or a fixed length String for Visual BASIC. Reads the number of characters specified by field width (default is 1) from the device into the buffer pointed to by *arg*. White-space is not ignored with `%c`. No null character is added to the end of the string.

`s` Corresponding *arg* is a pointer to a string for C/C++, or a fixed length String for Visual BASIC. All leading white-space characters are ignored, then all characters from the device are read into a string until a white-space character is read. An optional *field width* indicates the maximum length of the string. Note that you should specify the maximum field width of the buffer being used to prevent overflows.

`S` Corresponding *arg* is a pointer to a string for C/C++, or a fixed length String for Visual BASIC. This data is received as an IEEE 488.2 string response data block. The resultant string will not have the enclosing double quotes in it. An optional *field width* indicates the maximum length of the string. Note that you should specify the maximum field width of the buffer being used to prevent overflows.

`t` Corresponding *arg* is a pointer to a string for C/C++, or a fixed length String for Visual BASIC. Read all characters from the device into a string until an END indicator is read. An optional *field width* indicates the maximum length of the string. All characters read beyond the maximum length are ignored until the END indicator is received. Note that you should specify the maximum field width of the buffer being used to prevent overflows.

`b` Corresponding *arg* is a pointer to a buffer. This conversion code reads an array of data from the device. The data must be in IEEE 488.2 Arbitrary Block Program Data format. Note that, depending on the structure of the data, data may be read until an END indicator is read.

The *field width* must be present to specify the maximum number of elements the buffer can hold. For C/C++ programs, the *field width* can be a pound sign (#). If the *field width* is a pound sign, then two arguments are used to fulfill this conversion type. The first argument is a pointer to a long that will be used as the *field width*. The second will be the pointer to the buffer that will hold the data. After this conversion is satisfied, the *field width* pointer is assigned the number of elements read into the buffer. This is a convenient way to determine the actual number of elements read into the buffer.

If there is more data than will fit into the buffer, the extra data is lost.

If no argument modifier is specified, the array is assumed to be an array of bytes.

If the `w` argument modifier is specified, then the array is assumed to be an array of short integers (16 bits). The data read from the device is byte swapped and padded as necessary to convert from IEEE 488.2 byte ordering (big endian) to the native ordering of the controller. The *field width* is the number of words.

If the `l` (ell) argument modifier is specified, then the array is assumed to be an array of long integers (32 bits). The data read from the device is byte swapped and padded as necessary to convert from IEEE 488.2 byte ordering (big endian) to the native ordering of the controller. The *field width* is the number of long words.

If the z argument modifier is specified, then the array is assumed to be an array of floats. The data read from the device is an array of 32 bit IEEE-754 floating point numbers. The *field width* is the number of floats.

If the Z argument modifier is specified, then the array is assumed to be an array of doubles. The data read from the device is an array of 64 bit IEEE-754 floating point numbers. The *field width* is the number of doubles.

o Corresponding *arg* must be a pointer to an unsigned integer for C/C++, or an Integer for Visual BASIC. The library reads characters until the entire octal number is read. If the l (ell) argument modifier is used, the argument must be a pointer to an unsigned long integer for C/C++, or a Long for Visual BASIC. If the h argument modifier is used, the argument must be a pointer to an unsigned short integer for C/C++, or the argument must be an Integer for Visual BASIC.

u Corresponding *arg* must be a pointer to an unsigned integer for C/C++, or an Integer for Visual BASIC. The library reads characters until an entire number is read. It will accept any valid decimal number. If the l (ell) argument modifier is used, the argument must be a pointer to an unsigned long integer for C/C++, or a Long for Visual BASIC. If the h argument modifier is used, the argument must be a pointer to an unsigned short integer for C/C++, or the argument must be an Integer for Visual BASIC.

x Corresponding *arg* must be a pointer to an unsigned integer for C/C++, or an Integer for Visual BASIC. The library reads characters until an entire number is read. It will accept any valid hexadecimal number. If the l (ell) argument modifier is used, the argument must be a pointer to an unsigned long integer for C/C++, or a Long for Visual BASIC. If the h argument modifier is used, the argument must be a pointer to an unsigned short integer for C/C++, or it must be an Integer for Visual BASIC.

`[` Corresponding *arg* must be a character pointer for C/C++, or a fixed length character String for Visual BASIC. The `[` conversion type matches a non-empty sequence of characters from a set of expected characters. The characters between the `[` and the `]` are the scanlist. The scanset is the set of characters that match the scanlist, unless the circumflex (^) is specified. If the circumflex is specified, then the scanset is the set of characters that do not match the scanlist. The circumflex must be the first character after the `[`, otherwise it will be added to the scanlist.

The `-` can be used to build a scanlist. It means to include all characters between the two characters in which it appears (for example, `%[a-z]` means to match all the lower case letters between and including `a` and `z`). If the `-` appears at the beginning or the end of conversion string, `-` is added to the scanlist.

`n` Corresponding *arg* is a pointer to an integer for C/C++, or it is an Integer for Visual BASIC. The number of bytes currently converted from the device is placed into the *arg*. No argument is converted.

`F` Supported on HP-UX only. (Not supported on Windows 95 or Windows NT.) Corresponding *arg* is a pointer to a FILE descriptor. The input data read from the device is written to the file referred to by the FILE descriptor until the END indicator is received. The file must be opened for writing. No other modifiers or flags are valid with this conversion character.

Data Conversions

The following table lists the types of data that each of the numeric formats accept.

`d` IEEE 488.2 HEX, OCT, BIN, and NRf formats (for example, #HA, #Q12, #B1010, 10, 10.00, and 1.00E+01).

`f` IEEE 488.2 HEX, OCT, BIN, and NRf formats (for example, #HA, #Q12, #B1010, 10, 10.00, and 1.00E+01).

`i` Integer. Data with a leading 0 will be converted as octal; data with leading 0x or 0X will be converted as hexidecimal.

`u` Unsigned integer. Same as `i` except value is unsigned.

`o` Unsigned integer. Data will be converted as octal.

`x,X` Unsigned integer. Data will be converted as hexidecimal.

`e,g` Floating. Integers, floating point, and exponential numbers will be converted into floating point numbers (default is float).

Note that the conversion types `i` and `d` are not the same. This is also true for `f` and `e,g`.

Return Value

This function returns the total number of arguments converted by the format string.

See Also

"IPRINTF", "IPROMPTF", "IFLUSH", "ISETBUF", "ISETUBUF", "IFREAD", "IFWRITE"

ISERIALBREAK

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iserialbreak (id);
INST id;
```

Visual BASIC Syntax

```
Function iserialbreak
(ByVal id As Integer)
```

Description

The **iserialbreak** function is used to send a BREAK on the interface specified by *id*.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

ISERIALCTRL

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iserialctrl (id, request, setting);
INST id;
int request;
unsigned long setting;
```

Visual BASIC Syntax

```
Function iserialctrl
(ByVal id As Integer, ByVal request As Integer,
 ByVal setting As Long)
```

Description

The **iserialctrl** function is used to set up the serial interface for data exchange. This function takes *request* (one of the following values) and sets the interface to the setting. The following are valid values for *request*:

I_SERIAL_BAUD

The *setting* parameter will be the new speed of the interface. The value should be a valid baud rate for the interface (for example, 300, 1200, 9600). The baud rate is represented as an unsigned long integer, in bits per second. If the value is not a recognizable baud rate, an err_param error is returned. The following are the supported baud rates: 50, 110, 300, 600, 1200, 2400, 4800, 7200, 9600, 19200, 38400, and 57600.

I_SERIAL_PARITY

The following values are acceptable values for *setting*:

I_SERIAL_PAR_EVEN	Even parity
I_SERIAL_PAR_ODD	Odd parity
I_SERIAL_PAR_NONE	No parity bit is used
I_SERIAL_PAR_MARK	Parity is always one
I_SERIAL_PAR_SPACE	Parity is always zero

I_SERIAL_STOP

The following are acceptable values for *setting*:

I_SERIAL_STOP_1	1 stop bit
I_SERIAL_STOP_2	2 stop bits

I_SERIAL_WIDTH

The following are acceptable values for *setting*:

I_SERIAL_CHAR_5	5 bit characters
I_SERIAL_CHAR_6	6 bit characters
I_SERIAL_CHAR_7	7 bit characters
I_SERIAL_CHAR_8	8 bit characters

I_SERIAL_READ_BUFSZ

This is used to set the size of the read buffer. The *setting* parameter is used as the size of buffer to use. This value must be in the range of 1 and 32767.

I_SERIAL_DUPLEX

The following are acceptable values for *setting*:

I_SERIAL_DUPLEX_FULL	Use full duplex
I_SERIAL_DUPLEX_HALF	Use half duplex

I_SERIAL_FLOW_CTRL

The *setting* parameter must be set to one of the following values. If no flow control is to be used, set *setting* to zero (0). The following are the supported types of flow control:

I_SERIAL_FLOW_NONE	No handshaking
I_SERIAL_FLOW_XON	Software handshaking
I_SERIAL_FLOW_RTS_CTS	Hardware handshaking
I_SERIAL_FLOW_DTR_DSR	Hardware handshaking

`I_SERIAL_READ_EOI`

Used to set the type of END Indicator to use for reads.

In order for `iscanf` to work as specified, data must be terminated with an END indicator. The RS-232 interface has no standard way of doing this. SICL gives you two different methods of indicating EOI.

The first method is to use a character. The character can have a value between 0 and 0xff. Whenever this value is encountered in a read (`iread`, `iscanf`, or `ipromptf`), the read will terminate and the term reason will include `I_TERM_END`. The default for serial is the newline character (`\n`).

The second method is to use bit 7 (if numbered 0-7) of the data as the END indicator. The data would be bits 0 through 6 and, when bit 7 is set, that means EOI. The following values are valid for the *setting* parameter:

- `I_SERIAL_EOI_CHR|(n)` - A character is used to indicate EOI, where `n` is the character. This is the default type, and `\n` is used.
- `I_SERIAL_EOI_NONE` - No EOI indicator.
- `I_SERIAL_EOI_BIT8` - Use the eighth bit of the data to indicate EOI. On the last byte, the eighth bit will be masked off, and the result will be placed into the buffer.

`I_SERIAL_WRITE_EOI`

The *setting* parameter will contain the value of the type of END Indicator to use for reads. The following are valid values to use:

- `I_SERIAL_EOI_NONE` - No EOI indicator. This is the default for `I_SERIAL_WRITE` (`iprintf`).
- `I_SERIAL_EOI_BIT8` - Use the eighth bit of the data to indicate EOI. On the last byte, the eighth bit will be masked off, and the result will be placed into the buffer.

`I_SERIAL_RESET` This will reset the serial interface. The following actions will occur: any pending writes will be aborted, the data in the input buffer will be discarded, and any error conditions will be reset. This differs from **`iclear`** in that no BREAK will be sent.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **`Err`** variable is set if an error occurs.

See Also

"ISERIALSTAT"

ISERIALMCLCTRL

Supported sessions: ....interface
Affected by functions: ....ilock, itimeout

C Syntax

```
#include <sicl.h>

int iserialmclctrl (id, sline, state);
INST id;
int sline;
int state;
```

Visual BASIC Syntax

```
Function iserialmclctrl
(ByVal id As Integer, ByVal sline As Integer,
 ByVal state As Integer)
```

Description

The **iserialmclctrl** function is used to control the Modem Control Lines. The *sline* parameter sends one of the following values:

I_SERIAL_RTS	Ready To Send line
I_SERIAL_DTR	Data Terminal Ready line

If the *state* value is non-zero, the Modem Control Line will be asserted; otherwise it will be cleared.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"ISERIALMCLSTAT", "IONINTR", "ISETINTR"

ISERIALMCLSTAT

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iserialmclstat (id, sline, state);
INST id;
int sline;
int  *state;
```

Visual BASIC Syntax

```
Function iserialmclstat
(ByVal id As Integer, ByVal sline As Integer,
 state As Integer)
```

Description

The **iserialmclstat** function is used to determine the current state of the Modem Control Lines. The *sline* parameter sends one of the following values:

I_SERIAL_RTS	Ready To Send line
I_SERIAL_DTR	Data Terminal Ready line

If the value returned in *state* is non-zero, the Modem Control Line is asserted; otherwise it is clear.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"ISERIALMCLCTRL"

ISERIALSTAT

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iserialstat (id, request, result);
INST id;
int request;
unsigned long  *result;
```

Visual BASIC Syntax

```
Function iserialstat
(ByVal id As Integer, ByVal request As Integer,
 result As Long)
```

Description

The **iserialstat** function is used to find the status of the serial interface. This function takes one of the following values passed in *request* and returns the status in the *result* parameter:

I_SERIAL_BAUD	The *result* parameter will be set to the speed of the interface.
I_SERIAL_PARITY	The *result* parameter will be set to one of the following values:

I_SERIAL_PAR_EVEN	Even parity
I_SERIAL_PAR_ODD	Odd parity
I_SERIAL_PAR_NONE	No parity bit is used
I_SERIAL_PAR_MARK	Parity is always one
I_SERIAL_PAR_SPACE	Parity is always zero

I_SERIAL_STOP	The *result* parameter will be set to one of the following values:

I_SERIAL_STOP_1	1 stop bits
I_SERIAL_STOP_2	2 stop bits

I_SERIAL_WIDTH

The *result* parameter will be set to one of the following values:

I_SERIAL_CHAR_5	5 bit characters
I_SERIAL_CHAR_6	6 bit characters
I_SERIAL_CHAR_7	7 bit characters
I_SERIAL_CHAR_8	8 bit characters

I_SERIAL_DUPLEX

The *result* parameter will be set to one of the following values:

I_SERIAL_DUPLEX_FULL	Use full duplex
I_SERIAL_DUPLEX_HALF	Use half duplex

I_SERIAL_MSL

The *result* parameter will be set to the bit wise OR of all of the Modem Status Lines that are currently being asserted. The value of the *result* parameter will be the logical OR of all of the serial lines currently being asserted. The serial lines are both the Modem Control Lines and the Modem Status Lines. The following are the supported serial lines:

- I_SERIAL_DCD - Data Carrier Detect.
- I_SERIAL_DSR - Data Set Ready.
- I_SERIAL_CTS - Clear To Send.
- I_SERIAL_RI - Ring Indicator.
- I_SERIAL_TERI - Trailing Edge of RI.
- I_SERIAL_D_DCD - The DCD line has changed since the last time this status has been checked.
- I_SERIAL_D_DSR - The DSR line has changed since the last time this status has been checked.
- I_SERIAL_D_CTS - The CTS line has changed since the last time this status has been checked.

`I_SERIAL_STAT`

This is a read destructive status. That means reading this request resets the condition.

The *result* parameter will be set the bit wise OR of the following conditions:

- `I_SERIAL_DAV` - Data is available.
- `I_SERIAL_PARITY` - Parity error has occurred since the last time the status was checked.
- `I_SERIAL_OVERFLOW` - Overflow error has occurred since the last time the status was checked.
- `I_SERIAL_FRAMING` - Framing error has occurred since the last time the status was checked.
- `I_SERIAL_BREAK` - Break has been received since the last time the status was checked.
- `I_SERIAL_TEMT` - Transmitter empty.

`I_SERIAL_READ_BUFSZ`

The *result* parameter will be set to the current size of the read buffer.

`I_SERIAL_READ_DAV`

The *result* parameter will be set to the current amount of data available for reading.

`I_SERIAL_FLOW_CTRL`

The *result* parameter will be set to the value of the current type of flow control that the interface is using. If no flow control is being used, *result* will be set to zero (0). The following are the supported types of flow control:

`I_SERIAL_FLOW_NONE`	No handshaking
`I_SERIAL_FLOW_XON`	Software handshaking
`I_SERIAL_FLOW_RTS_CTS`	Hardware handshaking
`I_SERIAL_FLOW_DTR_DSR`	Hardware handshaking

`I_SERIAL_READ_EOI` The *result* parameter will be set to the value of the current type of END indicator that is being used for reads. The following values can be returned:

- `I_SERIAL_EOI_CHR|(`*n*`)` - A character is used to indicate EOI, where *n* is the character. These two values are logically OR-ed together. To find the value of the character, AND *result* with 0xff. The default is a **`\n`**.
- `I_SERIAL_EOI_NONE` - No EOI indicator. This is the default for `I_SERIAL_READ` (**`iscanf`**).
- `I_SERIAL_EOI_BIT8` - Use the eighth bit of the data to indicate EOI. This last byte will mask off this bit and use the rest for the data that is put in your buffer.

`I_SERIAL_WRITE_EOI` The *result* parameter will be set to the value of the current type of END indicator that is being used for reads. The following values can be returned:

- `I_SERIAL_EOI_NONE` - No EOI indicator. This is the default for `I_SERIAL_WRITE` (**`iprintf`**).
- `I_SERIAL_EOI_BIT8` - Use the eighth bit of the data to indicate EOI. This last byte will mask off this bit and use the rest for the data that is put in your buffer.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **`Err`** variable is set if an error occurs.

See Also

"ISERIALCTRL"

ISETBUF

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int isetbuf (id, mask, size);
INST id;
int mask;
int size;
```

NOTE

Not supported on Visual BASIC.

Description

This function is used to set the size and actions of the read and/or write buffers of formatted I/O. The *mask* can be one or the bit-wise OR of both of the following flags:

`I_BUF_READ`	Specifies the read buffer.
`I_BUF_WRITE`	Specifies the write buffer.

The *size* argument specifies the size of the read or write buffer (or both) in bytes. Setting a size of zero (0) disables buffering. This means that for write buffers, each byte goes directly to the device. For read buffers, the driver reads each byte directly from the device.

Setting a size greater than zero creates a buffer of the specified size. For write buffers, the buffer flushes (writes to the device) whenever the buffer fills up and for each newline character in the format string. (However, note that the buffer is *not* flushed by newline characters in the argument list.) For read buffers, the buffer is never flushed (that is, it holds any leftover data for the next `iscanf/ipromptf` call). This is the default action.

Setting a size less than zero creates a buffer of the absolute value of the specified size. For write buffers, the buffer flushes (writes to the device) whenever the buffer fills up, for each newline character in the format string, or at the completion of every `iprintf` call. For read buffers, the buffer flushes (erases its contents) at the end of every `iscanf` (or `ipromptf`) function.

NOTE

Calling `isetbuf` flushes any data in the buffer(s) specified in the *mask* parameter.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IPRINTF", "ISCANF", "IPROMPTF", "IFWRITE", "IFREAD", "IFLUSH", "ISETUBUF"

ISETDATA

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int isetdata (id, data);
INST id;
void  *data;
```

NOTE

Not supported on Visual BASIC.

Description

The `isetdata` function stores a pointer to a data structure and associates it with a session (or `INST` *id*).

You can use these user-defined data structures to associate device-specific data with a session such as device name, configuration, instrument settings, and so forth.

You are responsible for the management of the buffer (that is, if the buffer needs to be allocated or deallocated, you must do it).

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IGETDATA"

ISETINTR

Supported sessions: ....device, interface, commander

C Syntax

```
#include <sicl.h>

int isetintr (id, intnum, secval);
INST id;
int intnum;
long secval;
```

> **NOTE**
>
> Not supported on Visual BASIC.

Description

The `isetintr` function is used to enable interrupt handling for a particular event. Installing an interrupt handler only allows you to receive enabled interrupts. By default, all interrupt events are disabled.

The *intnum* parameter specifies the possible causes for interrupts. A valid *intnum* value for *any* type of session is:

`I_INTR_OFF` Turns off all interrupt conditions previously enabled with calls to `isetintr`.

A valid *intnum* value for *all* device sessions (except for GPIB and GPIO, which have no device-specific interrupts) is:

`I_INTR_*` Individual interfaces may include other interface-interrupt conditions. See the following information on each interface for more details.

Valid *intnum* values for *all* interface sessions are:

`I_INTR_INTFACT` Interrupt when the interface becomes active. Enable if *secval*! = 0; disable if *secval* = 0.

`I_INTR_INTFDEACT` Interrupt when the interface becomes deactivated. Enable if *secval*! = 0; disable if *secval* = 0.

`I_INTR_TRIG` Interrupt when a trigger occurs. The *secval* parameter contains a bit-mask specifying which triggers can cause an interrupt. See the **ixtrig** function's *which* parameter for a list of valid values.

`I_INTR_*` Individual interfaces may include other interface-interrupt conditions. See the following information on each interface for more details.

Valid *intnum* values for *all* commander sessions (except RS-232 and GPIO, which do not support commander sessions) are:

`I_INTR_STB` Interrupt when the commander reads the status byte from this controller. Enable if *secval*! = 0; disable if *secval* = 0.

`I_INTR_DEVCLR` Interrupt when the commander sends a device clear to this controller (on the given interface). Enable if *secval*! = 0; disable if *secval* = 0.

Interrupts on GPIB

GPIB Device Session Interrupts

There are no device-specific interrupts for the GPIB interface.

GPIB Interface Session Interrupts

The interface-specific interrupt for the GPIB interface is:

`I_INTR_GPIB_IFC` Interrupt when an interface clear occurs. Enable when *secval*! = 0; disable when *secval* = 0. This interrupt will be generated regardless of whether this interface is the system controller or not (that is, regardless of whether this interface generated the IFC, or another device on the interface generated the IFC).

The following are generic interrupts for the GPIB interface:

`I_INTR_INTFACT` Interrupt occurs whenever this controller becomes the active controller.

`I_INTR_INTFDEACT` Interrupt occurs whenever this controller passes control to another GPIB device. (For example, the **`igpibpassctl`** function has been called.)

GPIB Commander Session Interrupts

The following are commander-specific interrupts for GPIB:

`I_INTR_GPIB_PPOLLCONFIG` This interrupt occurs whenever there is a change to the PPOLL configuration. This interrupt is enabled using **`isetintr`** by specifying a *secval* greater than 0. If *secval*=0, this interrupt is disabled.

`I_INTR_GPIB_REMLOC` This interrupt occurs whenever a remote or local message is received and addressed to listen. This interrupt is enabled using **`isetintr`** by specifying a *secval* greater than 0. If *secval*=0, this interrupt is disabled.

`I_INTR_GPIB_GET` This interrupt occurs whenever the GET message is received and addressed to listen. This interrupt is enabled using **`isetintr`** by specifying a *secval* greater than 0. If *secval*=0, this interrupt is disabled.

`I_INTR_GPIB_TLAC` This interrupt occurs whenever this device has been addressed to talk or untalk, or the device has been addressed to listen or unlisten. When the interrupt handler is called, the *secval* value is set to a bit mask. Bit 0 is for listen, and bit 1 is for talk. If:

- Bit 0 = 1, then this device is addressed to listen.
- Bit 0 = 0, then this device is not addressed to listen.
- Bit 1 = 1, then this device is addressed to talk.
- Bit 1 = 0, then this device is not addressed to talk.

This interrupt is enabled using **`isetintr`** by specifying a *secval* greater than 0. If *secval*=0, this interrupt is disabled.

Interrupts on GPIO

GPIO Device Session Interrupts

GPIO does not support device sessions. Therefore, there are no device session interrupts for GPIO.

GPIO Interface Session Interrupts

The interface-specific interrupts for the GPIO interface are:

`I_INTR_GPIO_EIR` This interrupt occurs whenever the EIR line is asserted by the peripheral device. Enabled when *secval*!=0, disabled when *secval=0.*

`I_INTR_GPIO_RDY` This interrupt occurs whenever the interface becomes ready for the next handshake. (The exact meaning of "ready" depends on the configured handshake mode.) Enabled when *secval*!=0, disabled when *secval=0.*

NOTE

The GPIO interface is always active. Therefore, the interrupts for `I_INTR_INTFACT` and `I_INTR_INTFDEACT` will never occur.

GPIO Commander Session Interrupts

GPIO does not support commander sessions. Therefore, there are no commander session interrupts for GPIO.

Interrupts on RS-232 (Serial)

RS-232 Device Session Interrupts

The device-specific interrupt for the RS-232 interface is:

`I_INTR_SERIAL_DAV` This interrupt occurs whenever the receive buffer in the driver goes from the empty to the non-empty state.

RS-232 Interface Session Interrupts

The interface-specific interrupts for the RS-232 interface are:

`I_INTR_SERIAL_MSL` This interrupt occurs whenever one of the specified modem status lines changes states. The *secval* argument in `ionintr` is the logical OR of the Modem Status Lines to monitor. In the interrupt handler, the *sec* argument will be the logical OR of the MSL line(s) that caused the interrupt handler to be invoked.

Note that most implementations of the ring indicator interrupt only deliver the interrupt when the state goes from high to low (that is, a trailing edge). This differs from the other MSLs in that it's not simply just a state change that causes the interrupts.

The status lines that can cause this interrupt are DCD, CTS, DSR, and RI.

`I_INTR_SERIAL_BREAK` This interrupt occurs whenever a BREAK is received.

`I_INTR_SERIAL_ERROR` This interrupt occurs whenever a parity, overflow, or framing error happens. The *secval* argument in `ionintr` is the logical OR of one or more of the following values to enable the appropriate interrupt. In the interrupt handler, the *sec* argument will be the logical OR of these values that indicate which error(s) occurred:

- `I_SERIAL_PARERR` - Parity Error
- `I_SERIAL_OVERFLOW`- Buffer Overflow Error
- `I_SERIAL_FRAMING` - Framing Error

`I_INTR_SERIAL_DAV` This interrupt occurs whenever the receive buffer in the driver goes from the empty to the non-empty state.

`I_INTR_SERIAL_TEMT` This interrupt occurs whenever the transmit buffer in the driver goes from the non-empty to the empty state.

The following are generic interrupts for the RS-232 interface:

`I_INTR_INTFACT` This interrupt occurs when the Data Carrier Detect (DCD) line is asserted.

`I_INTR_INTFDEACT` This interrupt occurs when the Data Carrier Detect (DCD) line is cleared.

RS-232 Commander Session Interrupts

RS-232 does not support commander sessions. Therefore, there are no commander session interrupts for RS-232.

Interrupts on VXI

VXI Device Session Interrupts

The device-specific interrupt for the VXI interface is:

`I_INTR_VXI_SIGNAL` A specified device wrote to the VXI signal register (or a VME interrupt arrived from a VXI device that is in the servant list), and the signal was an event you defined. This interrupt is enabled using **`isetintr`** by specifying a *secval*!=0. If *secval*=0, then this is disabled. The value written into the signal register is returned in the *secval* parameter of the interrupt handler.

VXI Interface Session Interrupts

The following are interface-specific interrupts for the VXI interface:

`I_INTR_VXI_SYSRESET` A VXI SYSRESET occurred. This interrupt is enabled using **`isetintr`** by specifying a *secval*!=0. If *secval*=0, then this is disabled.

`I_INTR_VXI_VME` A VME interrupt occurred from a non-VXI device, or a VXI device that is not a servant of this interface. This interrupt is enabled using **`isetintr`** by specifying a *secval*!=0. If *secval*=0, then this is disabled.

`I_INTR_VXI_UKNSIG` A write to the VXI signal register was performed by a device that is not a servant of this controller. This interrupt condition is enabled using **`isetintr`** by specifying a *secval*!=0. If *secval*=0, then this is disabled. The value written into the signal register is returned in the *secval* parameter of the interrupt handler.

`I_INTR_VXI_VMESYSFAIL` The VME SYSFAIL line has been asserted.

`I_INTR_VME_IRQ1` VME IRQ1 has been asserted.

`I_INTR_VME_IRQ2` VME IRQ2 has been asserted.

`I_INTR_VME_IRQ3` VME IRQ3 has been asserted.

`I_INTR_VME_IRQ4` VME IRQ4 has been asserted.

`I_INTR_VME_IRQ5` VME IRQ5 has been asserted.

`I_INTR_VME_IRQ6` VME IRQ6 has been asserted.

`I_INTR_VME_IRQ7` VME IRQ7 has been asserted.

`I_INTR_ANY_SIG` A write has occurred to the SIGNAL register value.

The following are generic interrupts for the VXI interface:

`I_INTR_INTFACT` This interrupt occurs whenever the interface receives a BNO (Begin Normal Operation) message.

`I_INTR_INTFDEACT` This interrupt occurs whenever the interface receives an ANO (Abort Normal Operation) or ENO (End Normal Operation) message.

VXI Commander Session Interrupts

The commander-specific interrupt for VXI is:

`I_INTR_VXI_LLOCK` A lock/clear lock word-serial command has arrived. This interrupt is enabled using **isetintr** by specifying a *secval*!=0. If *secval*=0, then this is disabled. If a lock occurred, the *secval* in the handler is passed a 1; if an unlock, the *secval* in the handler is passed 0.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IONINTR", "IGETONINTR", "IWAITHDLR", "IINTROFF", "IINTRON", "IXTRIG", and the section titled "Asynchronous Events and HP-UX Signals" in the "Programming with HP SICL" chapter of the *HP SICL User's Guide for HP-UX* for protecting I/O calls against interrupts.

ISETLOCKWAIT

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int isetlockwait (id, flag);
INST id;
int flag;
```

Visual BASIC Syntax

```
Function isetlockwait
(ByVal id As Integer, ByVal flag As Integer)
```

Description

The `isetlockwait` function determines whether library functions wait for a device to become unlocked or return an error when attempting to operate on a locked device. The error that is returned is `I_ERR_LOCKED`.

If *flag* is non-zero, then all operations on a device or interface locked by another session will wait for the lock to be removed. This is the default case.

If *flag* is zero (0), then all operations on a device or interface locked by another session will return an error (`I_ERR_LOCKED`). This will disable the timeout value set up by the `itimeout` function.

NOTE

If a request is made that cannot be granted due to hardware constraints, the process will hang until the desired resources become available. To avoid this, use the `isetlockwait` command with the *flag* parameter set to 0, and thus generate an error instead of waiting for the resources to become available.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"ILOCK", "IUNLOCK", "IGETLOCKWAIT"

ISETSTB

Supported sessions: .... commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int isetstb (id, stb);
INST id;
unsigned char stb;
```

Visual BASIC Syntax

```
Function isetstb
(ByVal id As Integer, ByVal stb As Byte)
```

Description

The **isetstb** function allows the status byte value for this controller to be changed. This function is only valid for commander sessions.

Bit 6 in the *stb* (status byte) has special meaning. If bit 6 is set, then an SRQ notification is given to the remote controller, if its identity is known. If bit 6 is not set, then the SRQ notification is canceled. The exact mechanism for sending the SRQ notification is dependent on the interface.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IREADSTB", "IONSRQ"

ISETUBUF

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int isetubuf (id, mask, size, buf);
INST id;
int mask;
int size;
char  *buf;
```

NOTE

Not supported on Visual BASIC.

Description

The **isetubuf** function is used to supply the buffer(s) used for formatted I/O. With this function you can specify the size and the address of the formatted I/O buffer.

This function is used to set the size and actions of the read and/or write buffers of formatted I/O. The *mask* may be one, but NOT both of the following flags:

I_BUF_READ	Specifies the read buffer.
I_BUF_WRITE	Specifies the write buffer.

Setting a *size* greater than zero creates a buffer of the specified size. For write buffers, the buffer flushes (writes to the device) whenever the buffer fills up and for each newline character in the format string. For read buffers, the buffer is never flushed (that is, it holds any leftover data for the next **iscanf/ipromptf** call). This is the default action.

Setting a size less than zero creates a buffer of the absolute value of the specified size. For write buffers, the buffer flushes (writes to the device) whenever the buffer fills up, for each newline character in the format string, or at the completion of every **iprintf** call. For read buffers, the buffer flushes (erases its contents) at the end of every **iscanf** (or **ipromptf**) function.

> **NOTE**
>
> Calling **isetubuf** flushes the buffer specified in the *mask* parameter.

> **NOTE**
>
> Once a buffer is allocated to **isetubuf**, do not use the buffer for any other use. In addition, once a buffer is allocated to **isetubuf** (either for a read or write buffer), don't use the same buffer for any other session or for the opposite type of buffer on the same session (write or read, respectively).

In order to free a buffer allocated to a session, make a call to **isetbuf,** which will cause the user-defined buffer to be replaced by a system-defined buffer allocated for this session. The user-defined buffer may then be either re-used, or freed by the program.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IPRINTF", "ISCANF", "IPROMPTF", "IFWRITE", "IFREAD", "ISETBUF", "IFLUSH"

ISWAP

C Syntax

```
#include <sicl.h>

int iswap (addr, length, datasize);
int ibeswap (addr, length, datasize);
int ileswap (addr, length, datasize);
char *addr;
unsigned long length;
int datasize;
```

Visual BASIC Syntax

```
Function iswap
(ByVal addr As Long, ByVal length As Long,
 ByVal datasize As Integer)

Function ibeswap
(ByVal addr As Long, ByVal length As Long,
 ByVal datasize As Integer)

Function ileswap
(ByVal addr As Long, ByVal length As Long,
 ByVal datasize As Integer)
```

Description

These functions provide an architecture-independent way of byte swapping data received from a remote device or data that is to be sent to a remote device. This data may be received/sent using the **iwrite/iread** calls, or the **ifwrite/ifread** calls.

The **iswap** function will always swap the data.

The **ibeswap** function assumes the data is in big-endian byte ordering (big-endian byte ordering is where the most significant byte of data is stored at the least significant address) and converts the data to whatever byte ordering is native on this controller's architecture. Or it takes the data that is byte ordered for this controller's architecture and converts the data to big-endian byte ordering. (Notice that these two conversions are identical.)

The **ileswap** function assumes the data is in little-endian byte ordering (little-endian byte ordering is where the most significant byte of data is stored at the most significant address) and converts the data to whatever byte ordering is native on this controller's architecture. Or it takes the data that is byte ordered for this controller's architecture and converts the data to little-endian byte ordering. (Notice that these two conversions are identical.)

NOTE

Depending on the native byte ordering of the controller in use (either little-endian, or big-endian), that either the **ibeswap** or **ileswap** functions will always be a no-op, and the other will always swap bytes, as appropriate.

In all three functions, the *addr* parameter specifies a pointer to the data. The *length* parameter provides the length of the data in bytes. The *datasize* must be one of the values 1, 2, 4, or 8. It specifies the size of the data in bytes and the size of the byte swapping to perform:

- 1 = byte data and no swapping is performed.
- 2 = 16-bit word data and bytes are swapped on word boundaries.
- 4 = 32-bit longword data and bytes are swapped on longword boundaries.
- 8 = 64-bit data and bytes are swapped on 8-byte boundaries.

The *length* parameter must be an integer multiple of *datasize*. If not, unexpected results will occur.

IEEE 488.2 specifies the default data transfer format to transfer data in big-endian format. Non-488.2 devices may send data in either big-endian or little-endian format.

NOTE

These functions do not depend on a SICL session *id*. Therefore, they may be used to perform non-SICL related task (namely, file I/O).

The following constants are available for use by your application to determine which byte ordering is native to this controller's architecture.

`I_ORDER_LE`	This constant is defined if the native controller is little-endian.
`I_ORDER_BE`	This constant is defined if the native controller is big-endian.

These constants may be used in #if or #ifdef statements to determine the byte ordering requirements of this controller's architecture. This information can then be used with the known byte ordering of the devices being used to determine the swapping that needs to be performed.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IPOKE", "IPEEK", "ISCANF", "IPRINTF"

ITERMCHR

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int itermchr (id, tchr);
INST id;
int tchr;
```

Visual BASIC Syntax

```
Function itermchr
(ByVal id As Integer, ByVal tchr As Integer)
```

Description

By default, a successful **iread** only terminates when it reads *bufsize* number of characters, or it reads a byte with the END indicator. The **itermchr** function permits you to define a termination character condition.

The *tchr* argument is the character specifying the termination character. If *tchr* is between 0 and 255, then **iread** terminates when it reads the specified character. If *tchr* is -1, then no termination character exists, and any previous termination character is removed.

Calling **itermchr** affects all further calls to **iread** and **ifread** until you make another call to **itermchr**. The default termination character is -1, meaning no termination character is defined.

> **NOTE**
>
> The **iscanf** function only terminates reading on an END indicator. The **itermchr** function has no effect on the termination of an **iscanf** read.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IREAD", "IFREAD", "IGETTERMCHR"

ITIMEOUT

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int itimeout (id, tval);
INST id;
long tval;
```

Visual BASIC Syntax

```
Function itimeout
(ByVal id As Integer, ByVal tval As Long)
```

Description

The `itimeout` function is used to set the maximum time to wait for an I/O operation to complete. In this function, *tval* defines the timeout in milliseconds. A value of zero (0) disables timeouts.

> **NOTE**
>
> Not all computer systems can guarantee an accuracy of one millisecond on timeouts. Some computer clock systems only provide a resolution of 1/50th or 1/60th of a second. Other computers have a resolution of only 1 second. Note that the time value is *always* rounded up to the next unit of resolution.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IGETTIMEOUT"

ITRIGGER

Supported sessions: .... device, interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int itrigger (id);
INST id;
```

Visual BASIC Syntax

```
Function itrigger
(ByVal id As Integer)
```

Description

The **itrigger** function is used to send a trigger to a device.

Triggers on GPIB

GPIB Device Session Triggers

The **itrigger** function performs an addressed GPIB group execute trigger (GET).

GPIB Interface Session Triggers

The **itrigger** function performs an unaddressed GPIB group execute trigger (GET). The **itrigger** command on a GPIB interface session should be used in conjunction with **igpibsendcmd**.

Triggers on GPIO

GPIO Interface Session Triggers

The **itrigger** function performs the same function as calling **ixtrig** with the **I_TRIG_STD** value passed to it: it pulses the CTL0 control line.

Triggers on RS-232 (Serial)

RS-232 Device Session Triggers

The **itrigger** function sends the 488.2 ***TRG\n** command to the serial device.

RS-232 Interface Session Triggers

The **itrigger** function performs the same function as calling **ixtrig** with the I_TRIG_STD value passed to it: it pulses the DTR modem control line.

VXI Triggers

VXI Device Session Triggers

The **itrigger** function sends a word-serial trigger command to the specified device.

> **NOTE**
>
> The **itrigger** function is only supported on message-based device sessions with VXI.

VXI Interface Session Triggers

The **itrigger** function performs the same function as calling **ixtrig** with the I_TRIG_STD value passed to it: it causes one or more VXI trigger lines to fire. Which trigger lines are fired is determined by the **ivxitrigroute** function.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IXTRIG", and the interface-specific chapter in the *HP SICL User's Guide* for more information on trigger actions.

IUNLOCK

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int iunlock (id);
INST id;
```

Visual BASIC Syntax

```
Function iunlock
(ByVal id As Integer)
```

Description

The **iunlock** function unlocks a device or interface that has been previously locked. If you attempt to perform an operation on a device or interface that is locked by another session, the call will hang until the device or interface is unlocked.

Calls to **ilock**/**iunlock** may be nested, meaning that there must be an equal number of unlocks for each lock. This means that simply calling the **iunlock** function may not actually unlock a device or interface again. For example, note how the following C code locks and unlocks devices:

```
ilock(id);
/* Device locked */
iunlock(id);
/* Device unlocked */
ilock(id);
/* Device locked */
ilock(id);
/* Device locked */
iunlock(id);
/* Device still locked */
iunlock(id);
/* Device unlocked */
```

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"ILOCK", "ISETLOCKWAIT", "IGETLOCKWAIT"

IUNMAP

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int iunmap (id, addr, map_space, pagestart, pagecnt);
INST id;
char  *addr;
int map_space;
unsigned int pagestart;
unsigned int pagecnt;
```

Visual BASIC Syntax

```
Function iunmap
(ByVal id As Integer, ByVal addr As Long,
 ByVal mapspace As Integer, ByVal pagestart As Integer,
 ByVal pagecnt As Integer)
```

NOTE

Not supported over LAN.

Description

The **iunmap** function unmaps a mapped memory space. The *id* specifies a VXI interface or device session. The *addr* argument contains the address value returned from the **imap** call. The *pagestart* argument indicates the page within the given memory space where the memory mapping starts. The *pagecnt* argument indicates how many pages to free.

The *map_space* argument contains the following legal values:

`I_MAP_A16`	Map in VXI A16 address space.
`I_MAP_A24`	Map in VXI A24 address space.
`I_MAP_A32`	Map in VXI A32 address space.
`I_MAP_VXIDEV`	Map in VXI device registers. (Device session only.)
`I_MAP_EXTEND`	Map in VXI A16 address space. (Device session only.)
`I_MAP_SHARED`	Map in VXI A24/A32 memory that is physically located on this device (sometimes called local shared memory).

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IMAP"

IVERSION

C Syntax

```
#include <sicl.h>

int iversion (siclversion, implversion);
int  *siclversion;
int  *implversion;
```

Visual BASIC Syntax

```
Function iversion
(ByVal id As Integer, siclversion As Integer,
 implversion As Integer)
```

Description

The **iversion** function stores in *siclversion* the current SICL revision number times ten that the application is currently linked with. The SICL version number is a constant defined in **sicl.h** for C, and in **SICL.BAS** or **SICL4.BAS** for Visual BASIC, as **I_SICL_REVISION**. This function stores in *implversion* an implementation specific revision number (the version number of this implementation of the SICL library).

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

IVXIBUSSTATUS

Supported sessions: .... interface

C Syntax

```
#include <sicl.h>

int ivxibusstatus (id, request, result);
INST id;
int request;
unsigned long  *result;
```

Visual BASIC Syntax

```
Function ivxibusstatus
(ByVal id As Integer, ByVal request As Integer,
 result As Long)
```

Description

The **ivxibusstatus** function returns the status of the VXI interface. This function takes one of the following parameters in the *request* parameter and returns the status in the *result* parameter.

I_VXI_BUS_TRIGGER	Returns a bit-mask corresponding to the trigger lines which are currently being driven active by a device on the VXI bus.
I_VXI_BUS_LADDR	Returns the logical address of the VXI interface (viewed as a device on the VXI bus).
I_VXI_BUS_SERVANT_AREA	Returns the servant area size of this device.
I_VXI_BUS_NORMOP	Returns 1 if in normal operation, and a 0 otherwise.
I_VXI_BUS_CMDR_LADDR	Returns the logical address of this device's commander, or -1 if no commander is present (either this device is the top level commander, or normal operation has not been established.
I_VXI_BUS_MAN_ID	Returns the manufacturer's ID of this device.
I_VXI_BUS_MODEL_ID	Returns the model ID of this device.

`I_VXI_BUS_PROTOCOL`	Returns the value stored in this device's protocol register.
`I_VXI_BUS_XPROT`	Returns the value that this device will use to respond to a *read protocol* word-serial command.
`I_VXI_BUS_SHM_SIZE`	Returns the size of VXI memory available on this device. For A24 memory, this value represents 256 byte pages. For A32 memory, this value represents 64 Kbyte pages. Interpret as an unsigned integer for this command.
`I_VXI_BUS_SHM_ADDR_SPACE`	Returns either `24` or `32` depending on whether the device's VXI memory is located in A24 or A32 memory space.
`I_VXI_BUS_SHM_PAGE`	Returns the location of the device's VXI memory. For A24 memory, the *result* is in 256 byte pages. For A32 memory, the *result* is in 64 Kbyte pages.
`I_VXI_BUS_VXIMXI`	Returns `0` if this device is a VXI device. Returns `1` if this device is a MXI device.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IVXITRIGON", "IVXITRIGOFF"

IVXIGETTRIGROUTE

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ivxigettrigroute (id, which, route);
INST id;
unsigned long which;
unsigned long  *route;
```

Visual BASIC Syntax

```
Function ivxigettrigroute
(ByVal id As Integer, ByVal which As Long,
 route As Long)
```

Description

The **ivxigettrigroute** function returns in *route* the current routing of the *which* parameter. See the **ivxitrigroute** function for more details on routing and the meaning of *route*.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IVXITRIGON", "IVXITRIGOFF", "IVXITRIGROUTE", "IXTRIG"

IVXIRMINFO

Supported sessions: .... device, interface, commander

C Syntax

```
#include <sicl.h>

int ivxirminfo (id, laddr, info);
INST id;
int laddr;
struct vxiinfo  *info;
```

Visual BASIC Syntax

```
Function ivxirminfo
(ByVal id As Integer, ByVal laddr As Integer,
 info As vxiinfo)
```

Description

The **ivxirminfo** function returns information about a VXI device from the VXI Resource Manager. The *id* is the **INST** for any open VXI session. The *laddr* parameter contains the logical address of the VXI device. The *info* parameter points to a structure of type **struct vxiinfo**. The function fills in the structure with the relevant data.

The structure **struct vxiinfo** (defined in the file **sicl.h**) is listed on the following pages.

For C programs, the `vxiinfo` structure has the following syntax:

```
struct vxiinfo {
   /* Device Identification */
   short laddr;                 /* 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;               /* 1=PASSED 0=FAILED */

   /* Location of Device */
   short cage_num;               /* Card Cage Number */
   short slot;                 /* Slot #, -1 is unknown, -2 is MXI */

   /* Device Information */
   unsigned short protocol;    /* Value of protocol register */
   unsigned short x_protocol;  /* Value from Read Protocol command */
   unsigned short servant_area; /* Value of servant area */

   /* Memory Information */
   /* page size is 256 bytes for A24 and 64K bytes for A32*/
   unsigned short addrspace;    /* 24=A24, 32=A32, 0=none */
   unsigned short memsize;       /* Amount of memory in pages */
   unsigned short memstart;     /* Start of memory in pages */

   /* Misc. Information */
   short slot0_laddr;          /* LU of slot 0 device, -1 if unknown */
   short cmdr_laddr;             /* LU of commander, -1 if top level*/

   /* Interrupt Information */
   short int_handler[8];       /* List of interrupt handlers */
   short interrupter[8];      /* List of interrupters */

   short file[10];             /* Unused */

}
```

This static data is set up by the VXI resource manager.

For Visual BASIC programs, the **vxiinfo** structure has the following syntax:

```
Type vxiinfo
  laddr As Integer
  name As String * 16
  manuf_name As String * 16
  model_name As String * 16
  man_id As Integer
  model As Integer
  devclass As Integer
  selftest As Integer
  cage_num As Integer
  slot As Integer
  protocol As Integer
  x_protocol As Integer
  servant_area As Integer
  addrspace As Integer
  memsize As Integer
  memstart As Integer
  slot0_laddr As Integer
  cmdr_laddr As Integer
  int_handler(0 To 7)  As Integer
  interrupter(0 To 7) As Integer
  fill(0 To 9) As Integer
End Type
```

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

See the platform-specific manual for the section on the Resource Manager.

IVXISERVANTS

Supported sessions: .... interface

C Syntax

```
#include <sicl.h>

int ivxiservants (id, maxnum, list);
INST id;
int maxnum;
int  *list;
```

Visual BASIC Syntax

```
Function ivxiservants
(ByVal id As Integer, ByVal maxnum As Integer,
 list() As Integer)
```

Description

The **ivxiservants** function returns a list of VXI servants. This function returns the first *maxnum* servants of this controller. The *list* parameter points to an array of integers that holds at least *maxnum* integers. This function fills in the array from beginning to end with the list of active VXI servants. All unneeded elements of the array are filled with -1.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global Err variable is set if an error occurs.

IVXITRIGOFF

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ivxitrigoff (id, which);
INST id;
unsigned long which;
```

Visual BASIC Syntax

```
Function ivxitrigoff
(ByVal id As Integer, ByVal which As Long)
```

Description

The **ivxitrigoff** function de-asserts trigger lines and leaves them deactivated. The *which* parameter uses all of the same values as the **ixtrig** command, namely:

I_TRIG_ALL	All standard triggers for this interface (that is, the bitwise OR of all valid triggers)
I_TRIG_TTL0	TTL Trigger Line 0
I_TRIG_TTL1	TTL Trigger Line 1
I_TRIG_TTL2	TTL Trigger Line 2
I_TRIG_TTL3	TTL Trigger Line 3
I_TRIG_TTL4	TTL Trigger Line 4
I_TRIG_TTL5	TTL Trigger Line 5
I_TRIG_TTL6	TTL Trigger Line 6
I_TRIG_TTL7	TTL Trigger Line 7
I_TRIG_ECL0	ECL Trigger Line 0
I_TRIG_ECL1	ECL Trigger Line 1
I_TRIG_ECL2	ECL Trigger Line 2
I_TRIG_ECL3	ECL Trigger Line 3
I_TRIG_EXT0	External BNC or SMB Trigger Connector 0
I_TRIG_EXT1	External BNC or SMB Trigger Connector 1

Any combination of values may be used in *which* by performing a bit-wise OR of the desired values.

NOTE

To simply fire trigger lines (assert then de-assert the lines), use **ixtrig** instead of **ivxitrigon** and **ivxitrigoff**.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IVXITRIGON", "IVXITRIGROUTE", "IVXIGETTRIGROUTE", "IXTRIG"

IVXITRIGON

Supported sessions: .... interface
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ivxitrigon (id, which);
INST id;
unsigned long which;
```

Visual BASIC Syntax

```
Function ivxitrigon
(ByVal id As Integer, ByVal which As Long)
```

Description

The **ivxitrigon** function asserts trigger lines and leaves them activated. The *which* parameter uses all of the same values as the **ixtrig** command, namely:

I_TRIG_ALL	All standard triggers for this interface (that is, the bitwise OR of all valid triggers)
I_TRIG_TTL0	TTL Trigger Line 0
I_TRIG_TTL1	TTL Trigger Line 1
I_TRIG_TTL2	TTL Trigger Line 2
I_TRIG_TTL3	TTL Trigger Line 3
I_TRIG_TTL4	TTL Trigger Line 4
I_TRIG_TTL5	TTL Trigger Line 5
I_TRIG_TTL6	TTL Trigger Line 6
I_TRIG_TTL7	TTL Trigger Line 7
I_TRIG_ECL0	ECL Trigger Line 0
I_TRIG_ECL1	ECL Trigger Line 1
I_TRIG_ECL2	ECL Trigger Line 2
I_TRIG_ECL3	ECL Trigger Line 3
I_TRIG_EXT0	External BNC or SMB Trigger Connector 0
I_TRIG_EXT1	External BNC or SMB Trigger Connector 1

Any combination of values may be used in *which* by performing a bit-wise OR of the desired values.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IVXITRIGOFF", "IVXITRIGROUTE", "IVXIGETTRIGROUTE", "IXTRIG"

IVXITRIGROUTE

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ivxitrigroute (id, in_which, out_which);
INST id;
unsigned long in_which;
unsigned long out_which;
```

Visual BASIC Syntax

```
Function ivxitrigroute
(ByVal id As Integer, ByVal in_which As Long,
 ByVal out_which As Long)
```

Description

The `ivxitrigroute` function routes VXI trigger lines. With some VXI interfaces, it is possible to route one trigger input to several trigger outputs.

The *in_which* parameter may contain only one of the valid trigger values. The *out_which* may contain zero, one, or several of the following valid trigger values:

`I_TRIG_ALL`	All standard triggers for this interface (that is, the bit-wise OR of all valid triggers) (*out_which* ONLY)
`I_TRIG_TTL0`	TTL Trigger Line 0
`I_TRIG_TTL1`	TTL Trigger Line 1
`I_TRIG_TTL2`	TTL Trigger Line 2
`I_TRIG_TTL3`	TTL Trigger Line 3
`I_TRIG_TTL4`	TTL Trigger Line 4
`I_TRIG_TTL5`	TTL Trigger Line 5
`I_TRIG_TTL6`	TTL Trigger Line 6
`I_TRIG_TTL7`	TTL Trigger Line 7
`I_TRIG_ECL0`	ECL Trigger Line 0
`I_TRIG_ECL1`	ECL Trigger Line 1
`I_TRIG_ECL2`	ECL Trigger Line 2
`I_TRIG_ECL3`	ECL Trigger Line 3

`I_TRIG_EXT0`	External BNC or SMB Trigger Connector 0
`I_TRIG_EXT1`	External BNC or SMB Trigger Connector 1

The *in_which* parameter may also contain:

`I_TRIG_CLK0`	Internal clocks provided by the controller (implementation-specific)
`I_TRIG_CLK1`	Internal clocks provided by the controller (implementation-specific)
`I_TRIG_CLK2`	Internal clocks provided by the controller (implementation-specific)

This function routes the trigger line in the *in_which* parameter to the trigger lines contained in the *out_which* parameter. In other words, when the line contained in *in_which* fires, all of the lines contained in *out_which* are also fired.

For example, the following command causes EXT0 to fire whenever TTL3 fires:

```
ivxitrigroute(id, I_TRIG_TTL3, I_TRIG_EXT0);
```

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IVXITRIGON", "IVXITRIGOFF", "IVXIGETTRIGROUTE", "IXTRIG"

IVXIWAITNORMOP

Supported sessions: .... device, interface, commander
Affected by functions: .... itimeout

C Syntax

```
#include <sicl.h>

int ivxiwaitnormop (id);
INST id;
```

Visual BASIC Syntax

```
Function ivxiwaitnormop
(ByVal id As Integer)
```

Description

The **ivxiwaitnormop** function is used to suspend the process until the interface or device is active (that is, establishes normal operation). See the **iwaithdlr** function for other methods of waiting for an interface to become ready to operate.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"IWAITHDLR", "IONINTR", "ISETINTR", "ICLEAR"

IVXIWS

Supported sessions: .... device
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ivxiws(id,wscmd,wsresp,rpe);
INST id;
unsigned short wscmd;
unsigned short  *wsresp;
unsigned short  *rpe;
```

Visual BASIC Syntax

```
Function ivxiws
(ByVal id As Integer, ByVal wscmd As Integer,
 wsresp As Integer, rpe As Integer)
```

Description

The **ivxiws** function sends a word-serial command to a VXI message-based device. The *wscmd* contains the word-serial command. If *wsresp* contains zero (0), then this function does not read a word-serial response. If *wsresp* is non-zero, then the function reads a word-serial response and stores it in that location. If **ivxiws** executes successfully, *rpe* does not contain valid data. If the word-serial command errors, *rpe* contains the Read Protocol Error response, the **ivxiws** function returns **I_ERR_IO**, and the *wsresp* parameter contains invalid data.

NOTE

The **ivxiws** function will always try to read the response data if the *wsresp* parameter is non-zero. If you send a word serial command that does not return response data, and the *wsresp* argument is non-zero, this function will hang or timeout (see **itimeout**) waiting for the response.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **`Err`** variable is set if an error occurs.

See Also

"ITIMEOUT"

IWAITHDLR

C Syntax

```
#include <sicl.h>

int iwaithdlr (timeout);
long timeout;
```

NOTE

Not supported on Visual BASIC.

Description

The **iwaithdlr** function causes the process to suspend until either an enabled SRQ or interrupt condition occurs and the related handler executes. Once the handler completes its operation, this function returns and processing continues.

If *timeout* is non-zero, then **iwaithdlr** terminates and generates an error if no handler executes before the given time expires. If *timeout* is zero, then **iwaithdlr** waits indefinitely for the handler to execute.

Specify *timeout* in milliseconds.

NOTE

Not all computer systems can guarantee an accuracy of one millisecond on timeouts. Some computer clock systems only provide a resolution of 1/50th or 1/60th of a second. Other computers have a resolution of only 1 second. Note that the time value is *always* rounded up to the next unit of resolution.

The **iwaithdlr** function will implicitly enable interrupts. In other words, if you have called **iintroff**, **iwaithdlr** will re-enable interrupts, then disable them again before returning.

NOTE

Interrupts should be disabled if you are using **iwaithdlr**. Use **iintroff** to disable interrupts.

The reason for disabling interrupts is because there is a race condition between the **isetintr** and **iwaithdlr**, and, if you only expect one interrupt, it might come before the **iwaithdlr** executes.

The interrupts will still be disabled after the **iwaithdlr** function has completed.

For example:

```
...
iintroff ();
ionintr (hpib, act_isr);
isetintr (hpib, I_INTR_INTFACT, 1);
...
igpibpassctl (hpib, ba);
iwaithdlr (0);
iintron ();
...
```

In a multi-threaded application, **iwaithdlr** will enable interrupts for the whole process. If two threads call **iintroff**, and one of them then calls **iwaithdlr**, interrupts will be enabled and both threads can receive interrupt events. Note that this is not a defect, since your application must handle the enabling/disabling of interrupts and keep track of when all threads are ready to receive interrupts.

Return Value

This function returns zero (0) if successful, or a non-zero error number if an error occurs.

See Also

"IONINTR", "IGETONINTR", "IONSRQ", "IGETONSRQ", "IINTROFF", "IINTRON"

IWRITE

Supported sessions: .... device, interface, commander
Affected by functions: .... **ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int iwrite (id, buf, datalen, endi, actualcnt);
INST id;
char  *buf;
unsigned long datalen;
int endi;
unsigned long  *actualcnt;
```

Visual BASIC Syntax

```
Function iwrite
(ByVal id As Integer, ByVal buf As String,
 ByVal datalen As Long, ByVal endi As Integer,
 actual As Long)
```

Description

The `iwrite` function is used to send a block of data to an interface or device. This function writes the data specified in *buf* to the session specified in *id*. The *buf* argument is a pointer to the data to send to the specified interface or device. The *datalen* argument is an unsigned long integer containing the length of the data block in bytes.

If the *endi* argument is non-zero, this function will send the END indicator with the last byte of the data block. Otherwise, if *endi* is set to zero, no END indicator will be sent.

The *actualcnt* argument is a pointer to an unsigned long integer which, upon exit, will contain the actual number of bytes written to the specified interface or device. A NULL pointer can be passed for this argument and no value will be written.

If you want to pass a NULL *actualcnt* parameter to `iwrite` in Visual BASIC, you should pass the expression `0&`.

For LAN, if the client times out prior to the server, the *actualcnt* returned will be 0, even though the server may have written some data to the device or interface.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

See Also

"IREAD", "IFREAD", "IFWRITE"

IXTRIG

Supported sessions: ....interface
Affected by functions: ....**ilock**, **itimeout**

C Syntax

```
#include <sicl.h>

int ixtrig (id, which);
INST id;
unsigned long which;
```

Visual BASIC Syntax

```
Function ixtrig
(ByVal id As Integer, ByVal which As Long)
```

Description

The **ixtrig** function is used to send an extended trigger to an interface. The argument *which* can be:

I_TRIG_STD	Standard trigger operation for all interfaces. The exact operation of I_TRIG_STD depends on the particular interface. See the following subsections for interface-specific information.
I_TRIG_ALL	All standard triggers for this interface (that is, the bit-wise OR of all supported triggers).
I_TRIG_TTL0	TTL Trigger Line 0
I_TRIG_TTL1	TTL Trigger Line 1
I_TRIG_TTL2	TTL Trigger Line 2
I_TRIG_TTL3	TTL Trigger Line 3
I_TRIG_TTL4	TTL Trigger Line 4
I_TRIG_TTL5	TTL Trigger Line 5
I_TRIG_TTL6	TTL Trigger Line 6
I_TRIG_TTL7	TTL Trigger Line 7
I_TRIG_ECL0	ECL Trigger Line 0
I_TRIG_ECL1	ECL Trigger Line 1
I_TRIG_ECL2	ECL Trigger Line 2
I_TRIG_ECL3	ECL Trigger Line 3

I_TRIG_EXT0	External BNC or SMB Trigger Connector 0
I_TRIG_EXT1	External BNC or SMB Trigger Connector 1
I_TRIG_EXT2	External BNC or SMB Trigger Connector 2
I_TRIG_EXT3	External BNC or SMB Trigger Connector 3

Triggers on GPIB

When used on a GPIB interface session, passing the **I_TRIG_STD** value to the **ixtrig** function causes an unaddressed GPIB group execute trigger (GET). The **ixtrig** command on a GPIB interface session should be used in conjunction with the **igpibsendcmd**. There are no other valid values for the **ixtrig** function.

Triggers on GPIO

The **ixtrig** function will pulse either the CTL0 or CTL1 control line. The following values can be used:

I_TRIG_STD	CTL0
I_TRIG_GPIO_CTL0	CTL0
I_TRIG_GPIO_CTL1	CTL1

Triggers on RS-232 (Serial)

The **ixtrig** function will pulse either the DTR or RTS modem control lines. The following values can be used:

I_TRIG_STD	Data Terminal Ready (DTR)
I_TRIG_SERIAL_DTR	Data Terminal Ready (DTR)
I_TRIG_SERIAL_RTS	Ready To Send (RTS)

Triggers on VXI

When used on a VXI interface session, passing the **I_TRIG_STD** value to the **ixtrig** function causes one or more VXI trigger lines to fire. Which trigger lines are fired is determined by the **ivxitrigroute** function. The **I_TRIG_STD** value has no default value. Therefore, if it is not defined before it is used, no action will be taken.

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global **Err** variable is set if an error occurs.

See Also

"ITRIGGER", "IVXITRIGON", "IVXITRIGOFF"

_SICLCLEANUP

C Syntax

```
#include <sicl.h>

int _siclcleanup(void);
```

Visual BASIC Syntax

```
Function siclcleanup () As Integer
```

Description

This routine is called when a program is done with all SICL I/O resources. The routine must be called before a WIN16 SICL program terminates on Windows 95. Calling this routine is not required on Windows NT or HP-UX. Calling this routine is also not required for WIN32 SICL programs on Windows 95.

Note that Visual BASIC programs call this routine without the initial underscore (_).

Return Value

For C programs, this function returns zero (0) if successful, or a non-zero error number if an error occurs.

For Visual BASIC programs, no error number is returned. Instead, the global `Err` variable is set if an error occurs.

A

HP SICL Error Codes

HP SICL Error Codes

The following table contains the error codes for the SICL software.

SICL Error Codes and Messages

Error Code	Error String	Description
I_ERR_ABORTED	Externally aborted	A SICL call was aborted by external means.
I_ERR_BADADDR	Bad address	The device/interface address passed to iopen doesn't exist. Verify that the interface name is the one assigned in the I/O Setup utility (hwconfig.cf file) for HP-UX, or with the I/O Config utility for Windows.
I_ERR_BADCONFIG	Invalid configuration	An invalid configuration was identified when calling iopen.
I_ERR_BADFMT	Invalid format	Invalid format string specified for iprintf or iscanf.
I_ERR_BADID	Invalid INST	The specified INST *id* does not have a corresponding iopen.
I_ERR_BADMAP	Invalid map request	The imap call has an invalid map request.
I_ERR_BUSY	Interface is in use by non-SICL process	The specified interface is busy.
I_ERR_DATA	Data integrity violation	The use of CRC, Checksum, and so forth imply invalid data.
I_ERR_INTERNAL	Internal error occurred	SICL internal error.
I_ERR_INTERRUPT	Process interrupt occurred	A process interrupt (signal) has occurred in your application.
I_ERR_INVLADDR	Invalid address	The address specified in iopen is not a valid address (for example, "hpib,57").
I_ERR_IO	Generic I/O error	An I/O error has occurred for this communication session.
I_ERR_LOCKED	Locked by another user	Resource is locked by another session (see isetlockwait).

SICL Error Codes and Messages (continued)

Error Code	Error String	Description
I_ERR_NESTEDIO	Nested I/O	Attempt to call another SICL function when current SICL function has not completed (WIN16). More than one I/O operation is prohibited.
I_ERR_NOCMDR	Commander session is not active or available	Tried to specify a commander session when it is not active, available, or does not exist.
I_ERR_NOCONN	No connection	Communication session has never been established, or connection to remote has been dropped.
I_ERR_NODEV	Device is not active or available	Tried to specify a device session when it is not active, available, or does not exist.
I_ERR_NOERROR	No Error	No SICL error returned; function return value is zero (0).
I_ERR_NOINTF	Interface is not active	Tried to specify an interface session when it is not active, available, or does not exist.
I_ERR_NOLOCK	Interface not locked	An **iunlock** was specified when device wasn't locked.
I_ERR_NOPERM	Permission denied	Access rights violated.
I_ERR_NORSRC	Out of resources	No more system resources available.
I_ERR_NOTIMPL	Operation not implemented	Call not supported on this implementation. The request is valid, but not supported on this implementation.
I_ERR_NOTSUPP	Operation not supported	Operation not supported on this implementation.
I_ERR_OS	Generic O.S. error	SICL encountered an operating system error.
I_ERR_OVERFLOW	Arithmetic overflow	Arithmetic overflow. The space allocated for data may be smaller than the data read.
I_ERR_PARAM	Invalid parameter	The constant or parameter passed is not valid for this call.
I_ERR_SYMNAME	Invalid symbolic name	Symbolic name passed to **iopen** not recognized.
I_ERR_SYNTAX	Syntax error	Syntax error occurred parsing address passed to **iopen**. Make sure that you have formatted the string properly. White space is not allowed.
I_ERR_TIMEOUT	Timeout occured	A timeout occurred on the read/write operation. The device may be busy, in a bad state, or you may need a longer timeout value for that device. Check also that you passed the correct address to **iopen**.
I_ERR_VERSION	Version incompatibility	The **iopen** call has encountered a SICL library that is newer than the drivers. Need to update drivers.

B

HP SICL Function Summary

HP SICL Function Summary

The following tables summarize the supported features for each SICL function. The first table lists the core (interface-independent) SICL functions. The core SICL functions work on all types of devices and interfaces. The tables after that list the interface-specific SICL functions (that is, the SICL functions that are specific to HP-IB/GPIB, GPIO, LAN, RS-232/Serial, and VXI interfaces, respectively).

Each table notes if the particular SICL function that is listed:

- Supports device (DEV), interface (INTF), and/or commander (CMDR) session(s).
- Is affected by the `ilock` (LOCK) and/or the `itimeout` (TIMEOUT) function(s).

Also, the first table titled "Core SICL Functions" and the last table titled "VXI-Specific SICL Functions" have the additional column, LAN CLIENT TIMEOUT. The SICL functions that have X's in this column may timeout over LAN, even those functions which cannot timeout over local interfaces.

Core SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT	LAN CLIENT TIMEOUT
IBLOCKCOPY						
ICAUSEERR	X	X	X			
ICLEAR	X	X		X	X	X
ICLOSE	X	X	X			X
IFLUSH	X	X	X	X	X	X
IFREAD	X	X	X	X	X	X
IFWRITE	X	X	X	X	X	X
IGETADDR	X	X	X			
IGETDATA	X	X	X			
IGETDEVADDR	X					
IGETERRNO						
IGETERRSTR						
IGETINTFSESS	X		X			X
IGETINTFTYPE	X	X	X			
IGETLOCKWAIT	X	X	X			
IGETLU	X	X	X			
IGETLUINFO						
IGETLULIST						
IGETONERROR	X	X	X			
IGETONINTR	X	X	X			
IGETONSRQ	X	X				
IGETSESSTYPE	X	X	X			
IGETTERMCHR	X	X	X			
IGETTIMEOUT	X	X	X			
IHINT	X	X	X			
IINTROFF						
IINTRON						
ILOCAL	X			X	X	X

Core SICL Functions (continued)

Function	DEV	INTF	CMDR	LOCK	TIMEOUT	LAN CLIENT TIMEOUT
ILOCK	X	X	X		X	X
IONERROR						
IONINTR	X	X	X			X
IONSRQ	X	X				X
IOPEN	X	X	X			X
IPOPFIFO						
IPRINTF	X	X	X	X	X	X
IPROMPTF	X	X	X	X	X	X
IPUSHFIFO						
IREAD	X	X	X	X	X	X
IREADSTB	X			X	X	X
IREMOTE	X			X	X	X
ISCANF	X	X	X	X	X	X
ISETBUF	X	X	X			X
ISETDATA	X	X	X			
ISETINTR	X	X	X			X
ISETLOCKWAIT	X	X	X			
ISETSTB			X	X	X	X
ISETUBUF	X	X	X			X
ISWAP						
ITERMCHR	X	X	X			
ITIMEOUT	X	X	X			
ITRIGGER	X	X		X	X	X
IUNLOCK	X	X	X			X
IVERSION						X
IWAITHDLR						
IWRITE	X	X	X	X	X	X
IXTRIG		X		X	X	X

HP-IB/GPIB-Specific SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT
IGPIBATNCTL		X		X	X
IGPIBBUSADDR		X		X	X
IGPIBBUSSTATUS		X		X	X
IGPIBGETT1DELAY		X		X	X
IGPIBLLO		X		X	X
IGPIBPASSCTL		X		X	X
IGPIBPPOLL		X		X	X
IGPIBPPOLLCONFIG	X		X	X	X
IGPIBPPOLLRESP			X	X	X
IGPIBRENCTL		X		X	X
IGPIBSENDCMD		X		X	X
IGPIBSETT1DELAY		X		X	X

GPIO-Specific SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT
IGPIOCTRL		X		X	X
IGPIOGETWIDTH		X			
IGPIOSETWIDTH		X		X	X
IGPIOSTAT		X			

LAN-Specific SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT
IGETGATEWAYTYPE	X	X	X		
ILANGETTIMEOUT		X			
ILANTIMEOOUT		X			

RS-232/Serial-Specific SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT
ISERIALBREAK		X		X	X
ISERIALCTRL		X		X	X
ISERIALMCLCTRL		X		X	X
ISERIALMCLSTAT		X		X	X
ISERIALSTAT		X		X	X

VXI-Specific SICL Functions

Function	DEV	INTF	CMDR	LOCK	TIMEOUT	LAN CLIENT TIMEOUT
IMAP	X	X	X	X	X	
IMAPINFO	X	X	X			
IPEEK						
IPOKE						
IUNMAP	X	X	X			
IVXIBUSSTATUS		X		X	X	X
IVXIGETTRIGROUTE		X		X	X	X
IVXIRMINFO	X	X	X			X
IVXISERVANTS		X				X
IVXITRIGOFF		X		X	X	X
IVXITRIGON		X		X	X	X
IVXITRIGROUTE		X		X	X	X
IVXIWAITNORMOP	X	X	X		X	X
IVXIWS	X			X	X	X

Glossary

Glossary

address
A string uniquely identifying a particular interface or a device on that interface.

bus error
An action that occurs when access to a given address fails either because no register exists at the given address, or the register at the address refuses to respond.

bus error handler
Programming code executed when a bus error occurs.

commander session
A session that communicates to the controller of this system.

controller
A computer used to communicate with a remote device such as an instrument. In the communications between the controller and the device the controller is in charge of, and controls the flow of communication (that is, does the addressing and/or other bus management).

controller role
A computer acting as a controller communicating with a device.

device
A unit that receives commands from a controller. Typically a device is an instrument but could also be a computer acting in a non-controller (commander) role, or another peripheral such as a printer or plotter.

device driver
A segment of software code that communicates with a device. It may either communicate directly with a device by reading and writing registers, or it may communicate through an interface driver.

device session
A session that communicates as a controller specifically with a single device, such as an instrument.

handler
A software routine used to respond to an asynchronous event such as an error or an interrupt.

instrument
A device that accepts commands and performs a test or measurement function.

interface
A connection and communication media between devices and controllers, including mechanical, electrical, and protocol connections.

interface driver
A software segment that communicates with an interface. It also handles commands used to perform communications on an interface.

interface session
A session that communicates and controls parameters affecting an entire interface.

interrupt
An asynchronous event requiring attention out of the normal flow of control of a program.

lock
A state that prohibits other users from accessing a resource, such as a device or interface.

logical unit
A logical unit is a number associated with an interface. A logical unit, in SICL, uniquely identifies an interface. Each interface on the controller must have a unique logical unit.

mapping
An operation that returns a pointer to a specified section of an address space as well as makes the specified range of addresses accessible to the requester.

non-controller role
A computer acting as a device communicating with a controller.

process

An operating system object containing one or more threads of execution that share a data space. A multi-process system is a computer system that allows multiple programs to execute simultaneously, each in a separate process environment. A single-process system is a computer system that allows only a single program to execute at a given point in time.

register

An address location that controls or monitors hardware.

session

An instance of a communications channel with a device, interface, or commander. A session is established when the channel is opened with the `iopen` function and is closed with a corresponding call to `iclose`.

SRQ

Service Request. An asynchronous request (an interrupt) from a remote device indicating that the device requires servicing.

status byte

A byte of information returned from a remote device showing the current state and status of the device.

symbolic name

A name corresponding to a single interface or device. This name uniquely identifies the interface or device on this controller. If there is more than one interface or device on the controller, each interface or devie must have a unique symbolic name.

thread

An operating system object that consists of a flow of control within a process. A single process may have multiple threads that each have access to the same data space within the process. However, each thread has its own stack and all threads may execute concurrently with each other (either on multiple processors, or by time-sharing a single processor). Note that multi-threaded applications are only supported with 32-bit SICL.

Index

HEWLETT®
PACKARD

Printed in U.S.A. E0996

E2090-90033